Automatic Compilation of Logical Specifications into Efficient Programs

Donald Cohen[1]
U.S.C. Information Sciences Institute
4676 Admiralty Way
Marina del Rey, Ca. 90292

Abstract

We describe an automatic programmer, or "compiler" which accepts as input a predicate calculus specification of a set to generate or a condition to test, along with a description of the underlying representation of the data. This compiler searches a space of possible algorithms for the one that is expected to be most efficient. We describe the knowledge that is and is not available to this compiler, and its corresponding capabilities and limitations. This compiler is now regularly used to produce large programs.

1. Introduction

This work is motivated by a desire to help programmers do their job better, i.e., more easily and quickly create and modify programs that are more efficient, correct and understandable. Our approach follows the well-travelled route of supplying a "higher level language" which allows a programmer to say more of what he wants the machine to do and less of the details of how it is to be done. This leads to programs that are shorter, easier to understand and modify, and contain fewer bugs. However, higher level languages tend to degrade efficiency, since their compilers fail to make many optimizations that a human might make. In fact, some optimizations cannot even be expressed in the higher level language.

Our higher level language, AP5, is an extension of lisp in which programs can be written with much less commitment to particular algorithms or data representations. This is a benefit to the degree (which we believe is quite large) that programmers spend their effort dealing with these issues. One way to avoid thinking about data representation is to use a single powerful representation for all data. To some extent this is the approach of APL [Pakin 68], PROLOG [Clocksin 84] and Relational Databases [Ullman 82]. This unfortunately results in a large performance penalty. AP5, SETL [Schonberg 81] and MRS [Genesereth 81] provide the illusion of a uniform relational representation, but avoid the penalty by representing different relations with different data structures. AP5 goes further by accepting "specifications" that contain compound well-formed formulas (wffs). Its compiler assumes the responsibility of finding good algorithms to implement these specifications. Another advantage of multiple representations, though not the focus of this paper, is that new things can be regarded as data, e.g., the + function may be regarded as a relation which is true of infinitely many tuples. AP5 compiles uses of such relations into computations rather than data structure accesses.[2]

The compilation process is not entirely done by the machine. The programmer must provide a small amount of "annotation", which is not part of the actual specification. The most important annotations tell AP5 how to represent the data. If the predefined representations are inadequate, he can add new ones. The compiler searches a space of algorithms appropriate for the given representations. Its job is to find the most efficient one that meets the specification. Of course, the variablity of the representations makes this job much more difficult. This paper describes how the compiler does its job.

The following programming methodology seems to work well for AP5: First the programmer writes a specification. Next he adds annotations that select general (but inefficient) data representations. The result is compiled into a prototype that can be tested on small examples. (Even specifications contain bugs!) Finally he optimizes the prototype by changing the most critical annotations.

2. Example

Suppose we want a program to find the nephews of an input person. The data is a set of people along with the sex and parents of each. For simplicity we define a nephew as the son of a sibling (not including the son of a brother-in-law or sister-in-law), and we define siblings to include half-brothers and half-sisters. Figure 2-1 shows a sample lisp program for this task, and figure 2-2 shows a corresponding AP5 program.

The AP5 program contains two wffs, one as a definition for the Sibling relation and the other as a specification of the objects desired as output. Wffs are represented in prefix notation. The quantifiers ∀ and ∃ are represented by the symbols All and Exists. The connectives, ∧, ∨, ~, etc. are represented by the symbols "and", "or", "not", etc.

Comparison of these two programs reveals that the lisp program specifies both a data representation and an algorithm while the AP5 program only specifies the result to

[1]This research was supported by the Defense Advanced Research Projects Agency under contract No. MDA903 81 C 0335. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of DARPA, the U.S. Government, or any person or agency connected with them.

[2]Computations can also express the intent of recursive definitions, which AP5 prohibits because they allow multiple interpretations.

```
(Defun list-nephews (person)
  ; Algorithm:
  ;  (1) get the parents of person,
  ;  (2) get their children, excluding the
  ;      input person (siblings)
  ;  (3) get their children (nephews and nieces)
  ;  (4) filter out the nieces
  ;  (5) remove duplicate nephews
  ; Representation:
  ;  persons are symbols (with property lists)
  ;  (get x 'parents) is a list of x's parents
  ;  (get x 'children) is a list of x's children
  ;  (get x 'sex) is x's sex
  (remove-duplicates
    (loop for parent in (get person 'parents)
      nconc
      (loop for sibling in (get parent 'children)
         unless (eq person sibling) nconc
         (loop for nephew in
               (get sibling 'children)
               when (eq (get nephew 'sex) 'male)
               collect nephew)))))
```

Figure 2-1: Lisp program to find nephews

```
(DeclareRel Sex 2)      ; a binary relation,
                        ; e.g., (Sex Sam Male)
(DeclareRel Parent 2)   ; (Parent child parent)
(DefineRel Sibling (x y)
   (and (not (eq x y))
        (Exists (parent)
                (and (Parent x parent)
                     (Parent y parent)))))
(Defun list-nephews (person)
  (loop for nephew s.t.
     (and (Sex nephew 'male)
          (Exists (sibling)
            (and (Sibling person sibling)
                 (Parent nephew sibling))))
     collect nephew))
```

Figure 2-2: AP5 program to find nephews

be computed. The AP5 program can be written and understood without thinking about either representation or algorithm, while the lisp program cannot be written or understood without understanding both. When similar representations for the Sex and Parent relations were selected from the library, the AP5 compiler generated the same algorithm used in the lisp program.[3] However, other annotations might cause it to produce a very different program. For instance if there were only ten males in the world and the typical person had a thousand children, a better algorithm might start by enumerating the males. Such statistical information comprises most of the annotations other than relation representations.

3. What the Compiler Does and Doesn't Do

The compiler expects to be told how to test and generate primitive relations such as Parent and Sex. It combines these small pieces of functionality into programs that test and generate compound wffs (those with connectives and quantifiers). The compiler can be thought of as three parts: a simplifier, a compiler for tests and a compiler for generators. The second and third of these embody knowledge of how to test and generate various sorts of compound wffs. Before that knowledge is applied, the wff is simplified. We will not describe the simplifier in detail. It does the sorts of things that other simplifiers do: removing double negations, repeated conjuncts and disjuncts and vacuous quantifiers, detecting simple tautologies and contradictions, etc. Like all simplifiers it could be improved, but for present purposes the reader should assume that it produces reasonable results. The compilers for tests and generators expect their inputs to be simplified.

Not surprisingly, the compiler is somewhat limited, as compared to human programmers:

- It does not understand general lisp code. A human programmer (but not AP5) might use his understanding of lisp to optimize
  ```
  (loop for x s.t. (P x)
        until (lisp-predicate)
        do (lisp-code x))
  ```
  by generating x's in an order that reduces the number of iterations.
- It produces only "uniform" algorithms. A program written by a human to find a prime between two inputs might use these inputs to choose among several internal subprograms. AP5 does not write such programs. It believes that the cost of any algorithm is independent of the particular inputs.
- It does not understand the problem domain. If a prime number is defined as an integer greater than one with no integer divisor between one and itself, AP5 will not know to quit looking for divisors after the square root.
- It cannot be given special purpose algorithms for compound wffs. We all know a good algorithm for generating a range of integers, but AP5 cannot find a terminating algorithm for generating

  $\{n \mid \text{integer}(n) \wedge \text{lower} \leq n \wedge n \leq \text{upper}\}$[4]

Some of these limitations are discussed further in the section on future work. We now turn to a detailed description of what AP5 *can* do and how.

4. Testing

The description of a representation provides the compiler with an algorithm for testing relations with that representation and a cost estimate for that algorithm. It is trivial to compile programs that test conjunctions, disjunctions, negations, etc. and to estimate their costs. The cost estimates (and size estimates) of conjuncts and

[3] Almost - the only difference was that it removed duplicates from the intermediate set of siblings. I was too lazy to make this optimization in the lisp program, although I would expect the result to be twice as fast. Also, it turns out that the AP5 algorithm for removing duplicates is linear in the size of the input, while remove-duplicates (at least in the commonlisp I use) is quadratic.

[4] One way around this problem is to introduce the relation Integer-Between. We can then tell AP5 how to generate this relation and never admit that it has anything to do with the "≤" relation. We can also replace the specification with a lower level program.

disjuncts can be used to further improve the order in which the conjuncts or disjuncts are tested.

Wffs of the form (Exists *vars wff*) are tested by trying to generate an example:

`(loop for` *vars* `s.t.` *wff* `thereis t)`

As we will see, not all wffs can be generated, so some wffs cannot be tested. Wffs of the form (All *vars wff*) are tested as if they had been written (not (Exists *vars* (not *wff*))).

5. Generating

The largest and most interesting part of the compiler is devoted to generating tuples that satisfy a wff. By generating we mean enumerating in any order all such tuples without producing any duplicates. Of course, external mechanisms might decide to quit before they have all been produced, e.g., the algorithm for testing existential wffs quits when the first example is generated. This section attempts to:

- describe what AP5 can and cannot generate
- convince the reader (but not prove) that this covers all sets (with a few reasonable exceptions) that can be generated knowing only about logic and the primitive relations
- convince the reader that AP5 usually finds *efficient* algorithms and point out known exceptions

It is *not* assumed that the set of objects is enumerable.[5] Thus some generation tasks cannot be accomplished. For example, {x | true} cannot be generated. Likewise, for any set of variables, *vars*, and wff, *wff*, it is not possible to generate both {*vars* | *wff*}, and {*vars* | ~*wff*}. Even finite sets can fail to be generated due to the lack of a suitable algorithm. Suppose elements of the set S are identified by a special property on their property lists. This enables us to test whether an object is in S, but without a way to find all objects in the world with property lists, there may still be no way to generate S. Of course, it is also possible simply to neglect to tell AP5 how to generate a primitive wff.

For the remainder of this section we assume that AP5 is given a set of variables, V, and a simplified wff, W. If not all of the variables of V are free in W, AP5 complains at compile time. Clearly, if any set of values for V satisfies W, then uncountably many sets of values do: any value can be used for variables that are not free in W. We therefore are only sacrificing the ability to generate {V | W} when it turns out to be empty. Such programs are not necessary, and are probably not what the programmer intends anyway.

The generator compiler consists of one compiler for each logical construct, e.g., conjunction or existential quantification. Each compiler recursively finds the best ways to generate or test subwffs and combines these into a program for its construct. Each of the following sections will describe how a construct is generated, the coverage and efficiency of that algorithm, and how the cost and number of results are estimated.

[5] The set of objects is meant to include all potential lisp objects, e.g., all numbers and lists. Even if this set could be enumerated in principle, nobody would do it. Since we are interested in running programs it seems appropriate to consider the set of all objects not to be enumerable.

5.1. Primitive wffs

The description of a representation provides the compiler with a set of algorithms for generating relations of that representation, along with estimates of their costs. Each such algorithm requires values for certain positions as inputs and generates the others as outputs. It can thus be characterized by a list of "i"s and "o"s, standing for inputs and outputs. For example, an algorithm characterized by (i o i o o) accepts as input two values, v1 and v2, and generates x,y,z triples such that R(v1, x, v2, y, z). If the Employer relation were represented as a list of (person, employer) pairs there would be an algorithm characterized by (o o) which simply enumerated the pairs in the list.

Suppose <args> is a list of k arguments to a k-ary relation, R, and V is a list of the variables in <args>. AP5 can generate {V | R<args>} iff R has a primitive generating algorithm with a pattern that contains "o"s in all of the positions where <args> contains variables. For example, the pattern above would identify an algorithm that could be used to generate {x,y | R(1,x,2,y,3)}, but not {x,y | R(x,1,2,y,3)}.

AP5 uses the cheapest generator that is sufficient. It also compiles in code to filter out tuples which fail to match the pattern. Such filters are needed in two cases. One is that the algorithm generates positions that are actually supplied as inputs, e.g., the program to find John's employers might look something like

```
(loop for pair in Employer-Tuple-List
      when (eq (car pair) 'John)
      collect (cdr pair))
```

The other case is that some variable was used more than once. For example the compiled program to find people who employ themselves would be something like

```
(loop for pair in Employer-Tuple-List
      when (eq (car pair) (cdr pair))
      collect (cdr pair))
```

There is currently no way to supply an algorithm for generating {x | R(x,x)}.

Estimates of the number of tuples matching a pattern can be provided by annotation. A simple heuristic uses those values to estimate the number of tuples matching more specific patterns (changing "o"s to "i"s). A default value is used if no such estimate exists.

5.2. Negations

The current version of AP5 cannot generate negations of primitive wffs. This is not a serious problem in practice, since people tend to name relations in such a way that the negations are not enumerable, e.g., we talk about the relation Parent, with finitely many positive tuples, rather than the relation Non-Parent with finitely many negative tuples. It would be easy to allow generator algorithms for negations of relations should the need arise. For negations of compound wffs, the negation is simply pushed inward (by the simplifier).

5.3. Disjunctions

In order to generate $\{V \mid w_1 \vee w_2 \vee \ldots \vee w_n\}$, AP5 generates $\{V \mid w_i\}$ for each $1 \leq i \leq n$ and removes duplications of tuples that satisfy multiple disjuncts.[6]

If a set described by a disjunction is countable then the set described by each disjunct is countable. Therefore the only way for this algorithm to fail to generate a disjunctive countable set is for it to fail to generate some countable disjunct. Either there is some algorithm for generating this disjunct that AP5 simply doesn't know or else the disjunct could actually be simplified out, e.g.,

{x | [x is even] ∨
[x is an even Godel number of a non-theorem]}

One algorithm for generating any set that can be tested is to filter a generable superset. However, this does not allow AP5 to generate any new disjunctions since any such superset of the disjunctive set could just as well be used to generate each disjunct.

On the other hand, it would be faster to generate the superset once for the entire disjunction. AP5 does not currently make this optimization. The simplifier might achieve this in some cases, e.g., by replacing
{x | [P(x) ∧ Q(x)] ∨ [P(x) ∧ R(x)]} with
{x | P(x) ∧ [Q(x) ∨ R(x)]}, but suppose we want a list of people who either have parents or children, i.e.,
{x | ∃ y [Child(x,y) ∨ Child(y,x)]}. The compiler will create two loops which it also ought to consider merging.

AP5 pessimistically estimates the number of tuples satisfying a disjunction as the sum of the numbers of tuples satisfying its disjuncts. The estimated generating cost is the sum of the costs of generating the disjuncts.

5.4. Existential Quantification

Suppose we want to generate {V | ∃ U w}, where U and V are lists of variables. We can assume that no variable appears more than once in V,U (the concatenation of the lists) and that every such variable is free in w.[7] AP5 generates {V | ∃ U w} by generating {V,U | w}. The values for U are discarded and those for V returned, after removing duplicate tuples.

There are some representations for which better algorithms exist, e.g., if R(x,y) is represented as an AList where the CDR of each entry is a list of y values related to the x value in the CAR, there is no need to look at all the elements of the CDR. However, this is an example of an algorithm for a compound wff that depends on the representation of a component relation, and AP5 cannot at present be given such algorithms.

There are countable sets described by wffs of the form {V | ∃ U w} that cannot be generated by this algorithm: {V,U | w} might contain uncountably many tuples which all share the same few values for V. However algorithms for actually generating such sets always seem to rely on domain knowledge or on transformations of the wff that could be done by the simplifier.

Again, it is instructive to imagine that we have S, a superset of {V | ∃ U w} which we can generate and filter by ∃ U w. This might be easier than generating {V,U | w}, since the values for V are already supplied. While finding an appropriate S requires domain knowlege in general, an example where logical knowledge suffices is when w is actually a conjunction of S and another wff, w1, i.e., we are generating {V | ∃ U [S(V) ∧ w1]}. But in this case S must not use any of the variables of U and the wff can be simplified to {V | S(V) ∧ ∃ U w1}, for which, as we will see, AP5 will find the algorithm we described.

The estimated cost of generating {V | ∃ U w} is the cost of generating {V,U | w}. The estimated number of tuples is the size of {V,U | w} divided by the size of {U | w}.

5.5. Conjunctions

AP5 can generate $\{V \mid w_1 \wedge w_2 \wedge \ldots \wedge w_n\}$ iff it can:

1. choose some conjunct to generate first - we will assume without loss of generality that it is the first conjunct (we can always reorder the conjuncts). Let V1 be a list of variables in w_1, and V2 be a list of the others
2. generate {V1 | w1}
3. generate $\{V2 \mid w_2 \wedge \ldots \wedge w_n\}$ (given bindings for the variables in V1)

If either V1 or V2 is empty, the corresponding generation is just a test. The point is that we can use the bindings for the variables of V1 that were obtained from w1 in order to find bindings for the variables of V2.

Example: Suppose we want {x | P(x) ∧ Q(x)}. Clearly, V1 will have to be {x} and V2 the empty set. We still have to decide whether to generate P and test Q or vice versa. If only one can be generated there's no choice. If neither can be generated, there's no solution. If both can be generated, the choice can be made on efficiency grounds. The cost of generating P and testing Q is easily computed from the cost of generating P, the cost of testing Q (once), and the number of tuples expected in P.

AP5 also considers the possibility of generating Q, storing the answers in a local cache which is more efficiently tested than the original representation of Q, then generating elements of P and testing them with the cache. This strategy is better if the cost of testing each element of P with the original representation of Q exceeds the cost of generating Q once, building the cache and testing each element of P with the cache.[8]

---

[6] AP5 caches previously returned tuples in a hashtable. It knows that tuples of the first disjunct need not be tested and those of the last need not be stored.

[7] This is because (1) the simplifier deletes variables from U that are not free in w, (2) variables in U are not free in [∃ U w], and (3) AP5 complains if V contains variables not free in [∃ U w] (this includes duplicated variables).

[8] To really compare costs one must know how much of the set will actually be generated. AP5 assumes the whole set is needed, but the computation is organized to return the first values as soon as possible.

In general, when a conjunct is to be used many times, it may be worthwhile to make a local cache that is optimized for the kind of access that is needed. AP5 currently only considers building temporary caches for testing an entire conjunct. This misses some opportunities for optimization, a deficiency we hope to correct. As an example, suppose we want a list of people who either have parents or have no children, where the Child relation is stored as a list of (parent, child) pairs. There are uncountably many objects with parents or no children, so AP5 tests each person separately, searching the entire Child relation. It might be better to first build separate caches for the objects with children and those with parents. Then enumerate people and filter them with the caches.

Example: Suppose we want $\{x,y,z \mid P(x,y) \wedge Q(y,z)\}$. In this case V1 must be either $\{x,y\}$ or $\{y,z\}$. If $\{x,y \mid P(x,y)\}$ can be generated, then it's only necessary to generate $\{z \mid Q(y,z)\}$. The alternative is to generate $\{y,z \mid Q(y,z)\}$ and $\{x \mid P(x,y)\}$. In either case the resulting program will look like a pair of nested loops. Again, if the inner loop is expensive it may be worthwhile to build a local cache.

Example: Suppose we want $\{x,y \mid P(x) \wedge Q(y)\}$. Obviously we have to be able to generate both $\{x \mid P(x)\}$ and $\{y \mid Q(y)\}$. The nested loop tends to be more efficient if the inner loop generates the conjunct that takes less time per output.[9] Again, it may be worthwhile to build a local cache for the inner loop.

It's easy to find countable sets described by conjunctions that cannot be generated with the algorithm above: imagine two uncountable sets with a countable intersection. The conjunction could be generated if we had a generable superset S of the intersection. One example mentioned earlier in the context of domain knowledge is
$\{x \mid \text{integer}(x) \wedge \text{lower} \leq x \wedge x \leq \text{upper}\}$.
If logical knowledge is sufficient to recognize such a case, then it would seem to be the responsibility of the simplifier, e.g., if A is an infinite set with an infinite complement and B is a finite set,
$\{x \mid [A(x) \vee B(x)] \wedge [\sim A(x) \vee B(x)]\} = \{x \mid B(x)\}$

AP5 assumes pessimistically that the tuples that satisfy the conjuncts will be highly correlated, e.g., that if there are 100 elements of P and 1000 elements of Q, there are nearly 100 elements of the intersection. Given the estimates of the sizes of these sets, the costs of generating them (and testing them), and some algorithm as described above, it's fairly easy to estimate the cost of generating the conjunction. AP5 could also use annotations estimating correlations among sets to improve its size estimates. This would allow it to apply the strongest filters earliest. Suppose sets P, Q and R each have the same size and the same cost for generating and testing. If P has a large intersection with each of Q and R, but Q and R have a small intersection, the best way to intersect all three is to intersect Q and R first and leave P for last. Unfortunately this data seems too much to ask of the user.

Some conjunctions can be generated in many different ways. Much effort has been spent optimizing the search for the best algorithm, but space does not permit a description of how this search is performed. The potential for exponential explosion has not been a problem in practice. Compilation of queries with conjunctions tends to be more expensive than other wffs, but not enough so to discourage their use.

5.6. Universal Quantification

AP5 cannot generate $\{V \mid \forall\ U\ w\}$. A degenerate case that AP5 could compile arises when $\{V,U \mid w\}$ can be generated: $\{V \mid \forall\ U\ w\}$ is trivially empty (since any set of values for V would require w to have too many tuples of U to generate). Again, we think it's acceptable not to compile expressions with constant values since they are probably errors and could always be written in a better way.

If $\{V \mid \forall\ U\ w\}$ is countable, $\{V,U \mid w\}$ must contain many U's for a few V's. AP5 cannot verify that a V is in the set by checking all the U's, because there are too many. Another approach is to determine that there are no values of U for which V fails to satisfy w. The remaining problem of generating candidate V's could be solved if $\{V \mid \forall\ U\ w\}$ were known to be a subset of some generable set, S.

One case where this strategy would seem possible is where w has the form $[S(V) \wedge \sim P]$, i.e., S does not depend on U, S is a superset of the set we want, and the universal property can be checked by an algorithm that finds the counterexamples. In this case, we are trying to generate $\{V \mid \forall\ U\ [S(V) \wedge \sim P]\}$, which can be simplified to $\{V \mid S(V) \wedge \forall\ U\ \sim P\}$, which can be generated by the algorithm for conjunctions.

As an example, suppose we have a relation Grade which is true of the 3-tuples, $\langle x,y,z \rangle$ such that student x got grade y in course z. One possible query with a universal quantifier is a request for a list of all the straight-A students, i.e., the students all of whose grades are A's. Notice that AP5 couldn't possibly generate the *objects* all of whose grades are A's, since this would include all objects without any grades, which is almost all objects in the world. The point is that universal queries tend to specify a generable range, and that AP5 can use this range to compile an algorithm that generates the range and tests the universal condition. In AP5 the straight-A students could be generated by this program:

```
(loop for x s.t.
   (and (Student x)
        (All (y z)
             (implies (Grade x y z) (= y 'A))))
   collect x)
```

6. Related work

[Smith 85] discusses the problem of optimizing conjunctions, which is the AP5 compiler's hardest problem. The space of algorithms considered is quite similar to the one used in AP5 but the cost model is much more simplistic: every conjunct is assumed to be either ungenerable or generable in constant time per tuple. Smith also deals

[9] The actual analysis shows that the comparison should be done on the generation time divided by *one less than* the size, since each relation has to be generated at least once either way.

extensively with the issue of searching for the best ordering, which we have not discussed here, other than remarking that AP5 seems to solve it in practice.

The largest body of work related to AP5 compilation deals with database query optimization [Ullman 82]. The biggest difference is that database systems do not allow user defined data representations. The representations available can only represent finite relations, so general computations cannot be treated like relations. Another major difference is that AP5 makes the assumption, typical of most programming, that it's dealing with data that fits in the address space. Database algorithms assume the opposite and therefore do not consider some of AP5's algorithms. For instance AP5's algorithm for eliminating duplicates requires that the set of previously generated tuples fit in the address space. Other differences between databases and AP5 are similar to those described by Smith in his comparison of databases with his own work.

7. Future Plans

Several failings of the AP5 compiler which we hope to correct have already been mentioned. In addition, some of the limitations listed in section 3 can be attacked. For one thing, it would be easy to accept generating algorithms for arbitrary wffs. The hard problem is recognizing when another wff can be transformed to make use of that algorithm. A trivial example is that we would like to recognize that an algorithm for generating a particular conjuction applies when more conjuncts are added. Fortunately, this problem does not have to be solved completely in order to gain significant advantages.

AP5 will never have as much domain knowledge as humans, but some kinds of domain knowledge are readily available and offer immediate advantage. One candidate is type information. Suppose we represent Parent(x,y) by putting y on the parent property of x and x on the child property of y. Then {x,y | Parent(x,y)} cannot be generated. However, if we knew that the Parent relation could only relate people, we could instead try compiling {x,y | Parent(x,y) $\wedge$ Person(x) $\wedge$ Person(y)}, which would succeed if the set of people could be generated. The same type information could be used to optimize this to {x,y | Parent(x,y) $\wedge$ Person(x)}. Similarly, if the set of people cannot be directly generated, it might have a generable supertype.

We would ultimately like AP5 to choose representations for relations. One problem is that we usually want to run part of the program before the whole program is written. This requires representations to be chosen for the first part. Suppose, for example, that AP5 decides to represent the Parent relation with the parent and child properties. If a later addition to the program needs the set of (parent, child) pairs it will be too late to recover this data. New annotations might reserve (or forfeit) the right to make such requests. Of course, the global optimization problem is also more difficult, and requires more data, such as the relative frequency of different requests and their time constraints.

8. Conclusion

We have described how AP5 compiles logical specifications into efficient lisp programs, given a small amount of annotation. We have also described the limitations of the compiler. Despite these limitations AP5 has proven very useful. The effect is to automate much of the work of programming. AP5 is currently used by a small number of people on a regular basis. One indication of success is that we tend not to think about what the AP5 compiler does. We simply assume that our specifications are being compiled into acceptably efficient programs. The only reasons for looking at the algorithms chosen by the compiler are curiosity and performance bugs, which can usually be fixed by changing annotations.

References

[Clocksin 84] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1984. This book is chosen as a representative of a large Prolog literature.

[Genesereth 81] Michael R. Genesereth, Russell Greiner and David E. Smith, *MRS Manual*, Stanford Heuristic Programming Project, 1981. Memo HPP-80-24

[Pakin 68] Sandra Pakin, *APL\360 reference manual*, Science Research Associates, Chicago, 1968.

[Schonberg 81] Schonberg, E., Schwartz, J.T. & Sharir, M., "An automatic technique for selection of data representations in SETL programs," *ACM Transactions on Programming Languages and Systems* 3, (2), April 1981, 126-143.

[Smith 85] David E. Smith and Michael R. Genesereth, "Ordering Conjunctive Queries," *Artificial Intelligence* 26, (2), May 1985, 171-215.

[Ullman 82] Jeffrey D. Ullman, *Principles of Database Systems*, Computer Science Press, Rockville, Maryland, 1982. This book is chosen as a representative of a large database literature.