Automatic Composition of Data Structures to Represent Relations

Donald Cohen, Neil Campbell

USC / Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695

Abstract

This paper describes a very general class of composite data structures, and a language in which programmers can describe which to use for each purpose. These data structures are composed of primitive data structures drawn from a relatively small library that can be extended by the programmer. We describe the semantics of composite data structures in terms of a relational model. Programmers can use relational programming languages to express computations independent of data representation decisions, and then use our data structure language to specify representations for the relations in their programs. Finally, we describe how a compiler translates operations on relations represented by composite data structures. It uses a cost model to choose among alternative algorithms.

1 Introduction

For some years now our group has been working to support a style of programming in which the programmer first thinks about the desired behavior of a program and only later, when he is satisfied with the behavior, thinks about performance. In order to promote this style, programs are separated into "specifications", which describe the desired behavior independent of performance considerations, and "annotations" which affect only performance. Data representation decisions in particular are regarded as performance concerns. We therefore wish to make it trivial to choose or change data representations.

We also wish to minimize any penalty in performance for separating specifications from annotations. Our specification language models data in terms of objects and relations. It provides operations on relations such as adding tuples, iterating over tuples, and testing whether a tuple is in the relation. The representations used for those objects and relations are determined by annotations. We want our relational languages to offer operations on relations that can be compiled, given appropriate representation annotations, into code comparable to what the programmer would have written. We also wish to make available, through annotations, all of the interesting data structures that appear in real programs. For instance, the operation "make 3 be the only value in relation R", if R were represented by a variable, might compile into "set V to 3".

Our first approach to satisfying these goals, AP5[2], allowed a programmer to choose representations for relations from a library of representations. The library could be extended with new representations, but this was much more difficult than choosing one already in the library. For this reason programmers tended to use the best representation in the library for the situation at hand, even when they knew of a better representation that was not in the library. Our solution to this problem can be viewed either as making the library much larger or much easier to extend. It provides a small specification language with which programmers can describe many relation representations as compositions of building blocks. We will call this the representation specification language, to distinguish it from the relational specification language in which data is modelled in terms of objects and relations. As before, it is much easier to use building blocks that are already in the library than to add new ones to the library. However a very small library of building blocks can be combined into a very large set of compound representations.

Describing data representations in our representation specification language is not much easier or harder than writing data declarations in conventional programming languages. The leverage for the programmer is that a compiler can automatically translate those descriptions into programs that perform operations on relations with the given representations. In
a conventional language the programmer would have to write these himself, and would have to rewrite them if he decided to change the data representations. In some cases there is a large set of possible algorithms for an operation. Like any query optimizer, our compiler bases its choice on a cost model. This is provided in part by the building blocks and in part by additional programmer annotations.

Example: Consider a program that deals with groups of parts combined into assemblies, where these are in turn combined into larger assemblies, etc. The objects in this domain would include parts and assemblies. The relations would include the subassembly relation, indicating that one assembly is a subassembly of another. A programmer might make the following representation decision(s):

Assemblies are represented by records. These contain a "children" field which holds a pointer to a linked list of subassemblies. Assembly records also contain a "parent" field which holds a pointer to the parent assembly (there must be exactly one).

The information expressed informally above can be expressed formally as follows. The syntax is described in detail in section 4.

```
/* relation subassembly(super, sub) */
/* label representation slots children */

field-map(children) super sub-list
list-set sub sub-list
field-map(parent) sub one-super
singleton-set super

Such terms as "field" and "list" correspond to elements of the building block library. The first two lines (after the comment lines) mean that the children field of each superassembly points to a list of subassemblies, and the last two lines mean that the parent field of each subassembly points to its superassembly. This description is used as input to a compiler that implements operations of the relational specification language, such as testing whether assembly $Y$ is a subassembly of assembly $X$. In this case there are two possible algorithms. One is to compare $X$ with the parent field of $Y$. The other is to iterate through the list in the children field of $X$ comparing the elements to $Y$. In order to make a rational choice, the compiler must estimate the cost of each of these algorithms. It can then implement the operation with the lower cost algorithm.

Section 2 describes, on an intuitive level, the kinds of data structures supported. Section 3 describes the kinds of building blocks that are used. Section 4 describes the composition language. It also provides a number of examples and discusses the semantics of compositions. Section 5 describes compilation of relational interfaces. This includes descriptions of the operations provided by building blocks and examples of compilers for some common relational operations. It also describes some of the choices the compilers must make, and the cost model. We finish with brief sections on related work, current status, future work, and conclusions.

2 Intuitive description of compositions

Composite representations offer the following features that appear in the data structures of real programs:

- the ability to build nested data structures, e.g., lists of arrays of hash tables,
- the ability to store data redundantly (in order to improve speed at the cost of space), and
- the ability to share structure between redundant representations (in order to save space).\(^1\)

An example of a nested data structure is the field of the assembly record that contains a list of other assemblies. As another example, a text editor might associate with each buffer an array indexed by line number, where each element is another array indexed by character position containing in each element the character at that position in the given line of the given buffer. We have in mind here a particular meaning for nesting. In the example above, given a buffer we can find an array. This array represents the relation between line numbers, columns, and characters for that buffer. From that array, given a line number we can find another array. This array represents the relation between columns and characters for the buffer and line number that were used to find it. Similarly, the list of subassemblies contains only those related to the assembly represented by the record containing the list.

Nesting is certainly the most important capability provided by composite representations. It alone would

---

\(^1\)Since readers tend to ask at this point about recursion, here's the answer. Recursive data structures appear in two ways. One is that the data structures used as building blocks are recursive, e.g., a list or tree used to represent a set of objects. The other is that recursion is encoded in the relations. For instance the subassembly relation may be regarded as an instance of recursion, since subassemblies may themselves have subassemblies. However the relation itself only represents the single level relationship between one assembly and another.
provide a useful form of composition. The others appear less frequently in real programs. Of these, redundancy is probably more important than sharing. By redundancy we simply mean that there is more than one way to obtain the same information. The relation between assemblies provides an example. There are two different ways to determine whether one assembly is a subassembly of another. Similarly, in order to add or remove such a relationship it is necessary to update two different data structures.

Structure sharing is simply storing several pointers to the same data structures. This saves space when the data structures to be stored are larger than the pointers needed to refer to them. It's also faster to obtain the same information. The relation between assemblies provides an example. There are two different data structures. This saves space when one shared data structure than several that are stored separately. As an example of structure sharing, suppose we are writing a program to schedule travel. Suppose we have a relation between people, places, and times, e.g., some person is scheduled to be at some place at some time. (The example is not meant to imply any cardinality constraints, i.e., a person might be scheduled to be in several places at the same time.)

Figure 1: Example of structure sharing

Suppose this relation were redundantly represented as illustrated in figure 1. First, each representation of a time contains a pointer to a table associating places with lists of people. Second, each representation of a place contains a pointer to a table associating times with lists of people. For any time and place that are related, we would then have two lists of people, one stored in the table for the place and another in the table for the time. Of course, these lists would always represent the same set of people, namely the set of people related to that particular place and time, so there is no need to keep two copies. Instead both tables could hold pointers to the same list.

A nested data structure can be described as a sequence of primitives where each element contains (or contains pointers to) the next, e.g., a table containing lists of arrays corresponds to the sequence table, list, array. From this perspective the example above looks like two sequences sharing a tail. The symmetric case where two sequences share a head is also possible. For instance the same relation might be represented as a single table indexed by person, and containing for each person two more tables. One of these would associate times with lists of places and the other would associate places with lists of times. As we will see, it is also possible to share both heads and tails, and even middle segments.

3 Building blocks

Composite data structures are made out of primitive data structures, which we call building blocks. These include such common data structures as arrays, linked lists, hash tables, records, etc. We do not claim to supply an exhaustive set of such building blocks. Rather a library contains the most common ones and a programmer can, with some effort, add more. The important point is that the number of building blocks is much smaller than the number of ways in which they can be composed. We expect that new building blocks will be invented much more slowly than new compositions of old ones.

There are three different kinds of building blocks, which we call tuples, sets, and maps. A tuple represents a (mathematical) tuple of fixed length known at compile time, containing objects or data structures of fixed types also known at compile time. A set is very much like a relation, in that it represents a (mathematical) set of (mathematical) tuples of objects. A map is similar to a set in that it represents a set of tuples, but some of the elements of those tuples are objects and others are sets or maps. Maps, in effect, relate tuples to other relations, e.g., person objects (1-tuples) might be related to relations between times and places. In order for a map to be useful it must provide access to the relation associated with an input tuple.

Maps provide nesting. The relation in the text editor example, as illustrated in figure 2, would be represented by a map from buffer objects to maps from lines (or line numbers) to maps from columns to sets each containing a single character object. In this case each map has a single object as the domain, but this is not always the case. For example, if it were important to be able to quickly iterate over the characters in a given column (and there were a limited number of lines

---

2 The typewriter typeface is used to distinguish these implementation data structures from the mathematical objects of the same names which they closely resemble.
and columns), we might have used a two dimensional array indexed on both line and column. Similarly a set representing a relation between lines and columns might be implemented by a two dimensional bit array.

Tuples are used to factor out one aspect shared by many maps and sets. We sometimes want to store multiple domain objects or multiple range building blocks in a map or set for which the representation does not offer any interesting indexing. Rather than forcing the map or set to worry about these multiple things we allow it to store single things, tuples, which encapsulate the multiple things. Since there are actually several different representations for tuples, this also allows us to factor the tuple representation out from that of the map or set. It further allows us to share pointers to tuples.

4 Specifying compositions

A relation is represented by a set of building blocks. It is convenient to abstract out the tuples by regarding them as parts of the maps and sets that they help to represent. This leaves just maps and sets, each of which refer to objects which are in tuples of the relation being represented. In addition, the maps refer to other maps and sets. In figure 2 the columns group building blocks by their use. The first column contains the single map used to associate buffers with the maps in the second column. Each of these (one for each buffer in the relation) associates the lines in the buffer with maps in the next column. Notice that figure 1 is not organized in this way. In each of the first two columns the different maps represent totally different kinds of relations. Our representation specification language exploits the regularity of figure 2: all of the building blocks in a given column are described by a single building block description.

In general, a relation representation is specified by a set of building block descriptions, where each description contains the following information:

- **slots**: an ordered set of relation slots
- **representation**: a representation choice
- **parameters**: representation specific parameters
- **children**: an ordered set of “child” descriptions (present only in map descriptions)

The elements of the tuples of a relation are labelled by what we call the slots of the relation. For instance, the appointment scheduling relation has three slots, which we might label the "person", "time" and "place". The subassembly relation has two slots, which we might label "super" and "sub". The text editor relation has four slots, which we might label "buffer", "line", "column" and "character".

The slots of a building block description specifies which slots of the relation are represented in the corresponding building blocks. For instance, in the text editor example, the description of the maps used to map lines to relations between columns and characters has as its set of slots the singleton set containing the line slot. The description of the singleton sets of characters has as its set of slots the singleton set containing the character slot. For descriptions of building blocks that represent multiple slots, these must be ordered. The interpretation of the ordering is dependent on the representation choice. For instance, in a two dimensional array, it might mean that the first slot indexes the rows of the array and the second indexes the columns.

The representation identifies how the corresponding building blocks are to be represented. This is just the name of a representation in the library. The library associates these names with a set of standard functions that provide information or produce code to implement standard operations on building blocks of that representation. We describe these functions in section 5.

The parameters might include such things as the size of the hash table to be allocated, the name of the field of a record in which some data is to be stored, or the ordering relation used to sort a list.

The children of a map description indicates that each corresponding map associates tuples of its slots with an ordered set of building blocks corresponding to the children. For instance, the description of the maps that relate lines to relations between columns and characters has one child description, which is the description of the maps that relate columns to relations (sets) of characters. Again, if there is more than one child, the interpretation of the ordering depends on the representation choice. Shared pointers to a single
data structure are specified by having several building block descriptions with common children.

4.1 Examples

For purposes of exposition we offer both a minimal textual syntax and a graphical depiction for specifications of relation representations. In the graphical depiction each building block description is a box containing the representation and slots, with arrows pointing to the children. The textual syntax uses a separate line for each building block description, containing an optional list of parameters in parentheses, a slot specification and a child specification. A slot specification is either a list of slot labels or an at-sign followed by the label of a tuple building block description, where the tuple description with that label contains a slot specification that is a list of slot labels and an empty child specification. Similarly, a child specification is either a list of building block descriptions or an at-sign followed by the label of a tuple building block description, where the tuple description with that label contains an empty slot specification and a child specification that is a list of labels of building block descriptions. Descriptions of sets and maps must have non-empty slot specifications while child specifications for sets must be empty and for maps must be non-empty.

Representation names will include one of the words "map", "set" or "tuple" to indicate the type of building blocks they represent.

```
Field-map super  List-set sub

Field-map sub  Singleton-set
```

```
/* relation subassembly(super, sub) */
/* label representation slots children */
field-map(children) super  sub-list
sub-list list-set sub
field-map(parent) super  one-super
one-super singleton-set super
```

Figure 3: Subassembly representation

Figure 3 describes the representation of the relation between assemblies and their subassemblies. The representation field-map describes a map which associates an object represented by a record with another building block. A pointer to the associated building block is stored in a field of that record. The parameter, (children), is the name of the field. This representation requires that the slot specification contain only one slot, where the objects that fill that slot are represented by records. Such restrictions are checked at compile time by code supplied as part of the building blocks. In this case there are two different descriptions using field-map on two different fields of assemblies. The first maps a superassembly to a building block corresponding to the description with the label sub-list. This is a list of subassemblies. The other maps a subassembly to a building block corresponding to the description with the label one-super. This is a degenerate case of a set which represents a singleton set by its sole element, in this case the superassembly.

```
/* relation Schedule(person, time, place) */
/* label representation slots children */
tuple hash-map person 0 tuple
  hash-map record-tuple
time-map hash-map time
time-map place-map
place-map hash-map place
time-set list-set place
time-set list-set time
```

Figure 4: Tuples of building blocks

Figure 4 illustrates the use of a tuple to store building blocks. In this case we initially index on people. Each person is thereby associated with a record containing a pair of hash tables. The first maps times to lists of places and the second maps places to lists of times.

As mentioned earlier, AP5 provided a library of representations for relations. It is instructive to see how all of those representations can be specified. Of course, this library provided representations that were either frequently desired or sufficiently general to provide tolerable performance in a wide variety of situations. While it's important to be able to provide these, the real point of composing building blocks is to tailor representations to particular applications. Some of the representations in the library correspond to building blocks, e.g., arrays, hash tables, fields of records,
etc. However these representations tended to come in pairs. The more general mapped objects to lists of related objects, while the other was specialized to subassembly representation, except that bot fields map objects to single objects. These now correspond etc. However these representations tended to come by pairs of pointers between structures (similar to the presented by lists. This would now be expressed as in Figure 5:

/* relation Schedule (person, time, place) */
/* label representation slots children */
  list-set   @ tuple
  tuple      list-tuple person,
             time, place

Figure 5: List of tuples representation

There were two other interesting representations that were capable of representing relations of any arity. One was a discrimination net where the discriminations were made on the basis of different slots of the relation. For instance, the relation Schedule (person, time, place), was represented as a map from people to maps from times to sets of places. The maps all used one particular representation which was a hash table optimized for the case of a small number of keys. Our composed relations were originally conceived as a generalization of this representation. Figure 6 shows how that can now be specified.

/* relation Schedule (person, time, place) */
/* label representation slots children */
  small-hash-map person child1
  small-hash-map time  child2
  small-hash-set place

Figure 6: Discrimination net representation

The other representation used shared tuples. This was a set of hash tables, each indexed by a user chosen slot (specified as a parameter of the representation).

Thus there might be a hash table mapping places to lists of tuples of the relation, and another hash table mapping times to lists of tuples. Figure 7 shows how that can now be specified.

4.2 Semantic restrictions

We now formalize what each building block represents and explore some consequences. For simplicity we will again abstract out the tuples. Above we described what it meant for one building block description to be the child of another. We will say that one building block description is the parent of another if the second is a child of the first. Since we are abstracting out tuples this includes the case where the first is a map that maps its tuples of objects to tuples of building blocks which include the second. We now extend the notion of parent and child from building block descriptions to the building blocks themselves. One building block is the parent of another if the first is a map that maps some tuple of objects to the second. The number of children of a building block is therefore the number of tuples that it is associating with other building blocks times the number of children that its description has. We further use the terms ancestor and descendant in the obvious ways. We call a building block without parents a root, and one without children a leaf. (Leaves are the same thing as sets.)

We defined above the slots of a building block description. The slots* of a building block description is the union of the slots of that building block description and all of its ancestors. Recall that when we abstract out tuples we include in the slots of a building block description any slots that are stored by the tuples that it uses to store its slots.

Given a set of slot labels, \( \{L_1, \ldots, L_n\} \), we define a partial tuple over that set as an assignment of values
to some subset of the labels. We define the *merge* of two such partial tuples as the union of the two sets of assignments. The case where two partial tuples to be merged assign different values to the same label will not arise below.

Finally we define a *residue* of a relation, $R$, with respect to a partial tuple, $T_1$, over the slots of $R$. This is another relation, $R'$, over the slots of $R$ that are not assigned by $T_1$. $R'$ contains a tuple, $T_2$, iff the result of merging $T_1$ and $T_2$ is in $R$. For instance, the residue of Schedule with respect to a given person and time is the unary relation (think of it as a set) containing the places related by Schedule to the given person and time. The residue of Schedule with respect to a given person, place and time is a 0-ary relation, in essence a proposition, which is true (contains a single tuple of zero slots) if the given person, place and time is in the Schedule relation and otherwise false (contains no tuples). The residue of any relation with respect to an empty partial tuple is the relation itself.

Each building block (still ignoring tuples) represents a residue of the relation. Any root building block represents the entire relation. That is, by following pointers in that building block one can find out about complete tuples. In that same sense, if a building block, $B$, represents the residue of relation $R$ with respect to tuple $T_1$, and maps a tuple, $T_2$ to a child, $C$, then $C$ represents the residue of $R$ with respect to the result of merging $T_1$ and $T_2$.

An immediate consequence of the requirement above is that all children of a given building block must represent the same residue. It also follows that all parents of a given building block must have the same slots*. Any building block (other than a tuple) with the same slots* as its parent(s) is devoid of information and therefore of no value. Our requirements prohibit "partial representations". In particular every leaf building block must have all of the slots of the relation in its slots*. Thus we cannot represent Schedule by just a list of times. Of course, one might define another relation as the set of times that appear in Schedule, and represent that relation by a list of times.

5 Compilation

Since different relational languages may provide different interfaces to relations, we concentrate here on a few examples that seem likely to apply (with perhaps minor modification) to almost any relational language. We describe in some detail iteration over a relation (which includes testing as a special case) and in less detail adding a tuple to a relation. A compiler for a relational language uses data from several different sources when compiling an operation on a relation:

- the declaration of the relation (its representation specification)
- parameters of the operation describing semantic variants
- other annotations
- the library of building blocks

The parameters of operations indicate such things as the fact that a tuple to be added is known to not already be present, which kind of iteration over a relation is desired (described below), or the fact that only one tuple is needed. Other annotations come from the source program. The most relevant of these for our purposes are estimates of relation sizes. For instance the fact that most assemblies have many subassemblies but only one superassembly could influence the choice of testing algorithm. The library of building blocks provides standard interfaces for building block representations. These do things like generating code for particular operations on building blocks and estimating the cost of that code. In general these interfaces are optional. In many cases default methods can combine interfaces that are supplied to build one that is not. At worst, the lack of support for some operations will simply make it impossible to compile certain operations on relations that use the building block. In order to follow the examples below it is necessary to understand the following building block interfaces.

*Test/Fetch:* This provides code that tests whether a tuple of objects is stored in a building block. In the case of a *map* it may also return the building blocks associated with the tuple. (This is actually a special case of iteration described below.)

*Add/Add-new:* This provides code that adds a tuple to a *set* or associates a tuple with a set of building blocks in a *map*. (It does not apply to *tuples*.) If the tuple is already there, this is a noop for a *set* and it replaces the associated building blocks in a *map*. The Add-new variation assumes that the tuple is not already present and is therefore faster in some cases.

*Test-add:* This is a combination of test and add. It returns two pieces of code, one to do a test and the other to do an add. The "test" is allowed to make some temporary data available to the add, which may allow it to be faster than a normal add. For instance, a hash table test would compute the hash table index and this could be used by the add.
Iterate: This provides code for iterating over the tuples and associated building blocks of a set or map. There may be several ways of iterating over the same kind of building block. These are characterized in terms of which slots of the set or map are inputs and which are to be generated. If all the slots are inputs this is very much like a test, since it results in one iteration if the tuple of inputs is in the relation and otherwise none. As an example, suppose that we have an array indexed by time and place, where each element is a list of people. There are four possible iterations. The one in which both slots are outputs would iterate over the entire array, for each (non-empty) element reporting the time, place and (non-empty) list of people. The one in which the first element was an input and the second was an output would iterate over the row (or column) of the array corresponding to an input time, and report for each (non-empty) element of that row (or column) the place and list of people.

5.1 Translation of iterations

Iteration over a relation is analogous to iteration over a building block. One simplification is that there are no output building blocks. However, an additional complication is that slots may be classified not only as inputs and outputs but also as “any” slots. If the relation is Schedule(person, time, place), where the person slot is an input, the time slot is an output and the place slot is an any, then we need an algorithm which accepts a person and iterates over all times that are related to that person (and any place). A nested representation that maps people to maps which map times to sets of places can do this very efficiently if there is an efficient algorithm for doing an input iteration (a test/fetch) on the first map, an efficient algorithm for doing an output iteration on the second map, and empty building blocks are eagerly reclaimed. If empty building blocks are stored (for instance hash tables with no entries) it will suffice to have an efficient way to determine whether they are empty. Otherwise it might be necessary to do more search in order to make sure that there really was a place related to a particular person and time.

Each iteration algorithm for a relation corresponds to a sequence of building block descriptions in the relation representation specification, where

- the first element of the sequence is a root.
- every other building block description is a child of the one before it, and
- every building block is assigned one of its iteration methods.

Intuitively the algorithm looks like a nesting of iterations (where a lookup is a special case of a loop that can only execute zero or one times). The outermost loop corresponds to the root. The description of the resulting composed algorithm assigns to each slot of the relation the description assigned to that slot by the iteration algorithm for the first building block description in the sequence containing that slot, or if there is none then “any”. For example, consider a description containing a map from people to maps from times to sets of places, where the first map is assigned an output iteration and the second is assigned an input iteration. The resulting algorithm iterates over people that are related to an input time and any place. It iterates over the root map to find all of the people in the relation. For each person it looks up the input time. If it’s there (and if either we eagerly reclaim empty sets or we verify that the associated set is non-empty), then it outputs the person. Our translator can automatically convert output slots of iteration algorithms to input slots or any slots. That is, an output iteration can be used to build an (inefficient) input iteration by simply filtering out the undesired outputs. This makes it unnecessary, when adding a building block, to supply trivial iteration algorithms.

The cost of the composed algorithm can be estimated from the costs of the building block iteration algorithms along with the sizes of the relations that the building blocks represent. These are estimated from the size characterizations of the original relation. Our translators actually support size characterizations with enough detail to allow the translator to accurately estimate the sizes of building blocks. This facility can be used to tell the translator just what it needs in order to optimize a particular query.

5.2 Translation of add/add-new

The Add/Add-new function for relations is analogous to the Add/Add-new function on building blocks. It must produce code (and estimate its cost) which, given a tuple of objects, adds the tuple to the relation if it is not already there. Not all representations support all updates. For example, the subassembly representation does not support the operation of adding a tuple, since the parent field can only represent singleton sets.\footnote{Typically the useful operations for updating relations with keys not only add a tuple but also delete any previous tuple with...}
An add involves several conceptually different tasks. One is determining whether the tuple is already in the relation. If so the add is a noop and no more work is needed. (This does not apply to the add-new case.) If not, it is necessary to determine which of the building blocks needed to represent the tuple already exist and which do not. Note that every tuple is represented by one building block of each description in the relation representation description. These building blocks can be classified into three groups. There are some (near the roots) which need not be changed at all. These are the ones whose children already exist. There are others, near the leaves, that cannot be found without first finding at least some of their ancestors.

These are the ones that we really want to find. To these we must add the children that are to be created. Of course, these boundary building blocks generally cannot be found without first finding at least some of their ancestors.

There are several opportunities for optimization in adding tuples. First, we have to look up tuples in some building blocks which may or may not already be there. If they are not there we will want to add them.

This motivates test-add algorithms. Second, the presence or absence of a tuple in one building block may imply the presence or absence of a tuple in another one. When we know that the tuple is present we can improve the test-add to a fetch, and when we know it is not we can improve it to an add-new. Finally, structure sharing presents choices of how to find building blocks.

The translator groups all of the building block descriptions in the relation representation description by their slots\(^4\). The import of the group is that at most one member need be tested. In particular, the test of every member will return true if the relation has a non-empty residue with respect to the partial tuple assigning the slots\(^4\) slots to the corresponding elements of the input tuple, i.e., if the slots\(^4\) slots of the input are already related. More generally, if a member of another group with a smaller (subset) slots\(^4\) turns out to not already exist, then this group need not be tested at all, e.g., if there is no tuple containing the given the same key, e.g., delete any previous parent of the subassembly. Such operations are likely to be presented in the syntax of the source language as "assignments", e.g., make the parent of assembly A1 be assembly A2.

6 Related work

In addition to our own relational languages, there are at least two other set oriented languages. These are Setl\([4]\) and Refine\([5]\). These do not support general relations but rather provide direct support for tuples, sets and maps.\(^5\) An extension to Refine\([6]\) allows programmers to choose representations for sets, maps and tuples, much as AP5 allows them to choose representations for relations. Setl has a somewhat more limited facility (and has no facility for user extension).

The setl researchers have also worked on automatically choosing efficient data structures\([7]\) for algorithms already written by programmers.

However, in these languages programs are written directly in terms of maps, sets and tuples rather than in terms of the more abstract relations which we have described here. That is, one would have to substantially rewrite the program in order to change the representation of the Schedule relation from a set of tuples to a map from places to maps from times to sets of people. This rewriting, as well as the corresponding part of the initial version, is what out compiler does automatically.

Another form of related work is query optimizers that have been written for relational languages. These are necessary components of a programming language that separates performance related implementation decisions from behavioral specifications. Consider, for example, a program to find which parts in an assembly are available in a warehouse. There may be several ways to do this, e.g., iterating over the parts in the assembly and for each checking whether it is in the warehouse, or iterating over the parts in the warehouse and for each checking whether it is in the assembly. Since the choice of algorithm depends on the representations of the relations (or at least what operations they support and at what cost), it certainly does not belong in the specification. Furthermore, since representation decisions should be easy to change, it makes no sense for the programmer to expend any effort choosing algorithms that depend on particular representation choices. Instead he should write the specification in a form that does not commit to a particular algorithm, e.g., using the language of

\(^4\)There are some differences between the tuples, sets and maps described here and those offered by Setl and Refine. However, this characterization is a close approximation to the net effect.

\(^5\)There are other opportunities for optimization in adding tuples. First, we have to look up tuples in some building blocks which may or may not already be there. If they are not there we will want to add them. This motivates test-add algorithms. Second, the presence or absence of a tuple in one building block may imply the presence or absence of a tuple in another one. When we know that the tuple is present we can improve the test-add to a fetch, and when we know it is not we can improve it to an add-new. Finally, structure sharing presents choices of how to find building blocks.

The translator groups all of the building block descriptions in the relation representation description by their slots\(^4\). The import of the group is that at most one member need be tested. In particular, the test of every member will return true if the relation has a non-empty residue with respect to the partial tuple assigning the slots\(^4\) slots to the corresponding elements of the input tuple, i.e., if the slots\(^4\) slots of the input are already related. More generally, if a member of another group with a smaller (subset) slots\(^4\) turns out to not already exist, then this group need not be tested at all, e.g., if there is no tuple containing the given the same key, e.g., delete any previous parent of the subassembly. Such operations are likely to be presented in the syntax of the source language as "assignments", e.g., make the parent of assembly A1 be assembly A2.
relational calculus, and rely on a query optimizer to find an algorithm suitable to whatever representations are eventually chosen. There are many query optimizers for relational languages supported by database systems[8]. These generally optimize on the basis of a cost model, and generally support a variety of representations. However the set of representations cannot be extended by programmers, and both the representations and optimization strategies are specialized to large disk resident sets of data. In AP5 we have provided optimizers for relational calculus[1], for rule triggering[3] and several other small languages for defining relations in terms of other relations (such as transitive closures, summations, maxima, etc.).

7 Current status and future directions

Our group is still in the early stages of building a new family of relational languages. We are currently building a new relational extension to commonlisp and our first relational extension to Ada. We also foresee a relational extension to C++. At the moment our compiler for relations composed of building blocks only produces lisp code, but we expect retargeting to be very easy. (The same is true of the compiler for relations defined by relational calculus.) The building block library is still quite small. It does, however, include all of the building blocks used in the examples, with the exception of small-hash-map and record-tuple. (In AP5 we used lists to represent tuples, and these are in the library.) Of course, the building blocks themselves must also be extended to produce code for different target languages. Finally we need to provide compilers for more operations on relations. So far we have concentrated on the minimal set used in AP5: iteration (including test, fetch), add and delete. The next priorities are assignment and an operation to remove all tuples (needed for destroying relations in languages without garbage collection). We are also very interested in linguistic support for residues. For instance, we would like to be able to iterate over the people in the Schedule relation and inside the iteration treat the residue for that person as another relation which could be tested, added to, etc. Operations analogous to test-add for relations would be a special case of these residue relations. (The add would be to a zero-ary residue defined with respect to a complete tuple.) We later hope to experiment with various kinds of rules in these languages similar to the kinds we already use in AP5.

8 Conclusion

We have described a language which allows a programmer to succinctly describe composite representations for relations, and a compiler that implements relational interfaces to the representations described in that language. The language allows nested data structures, redundant representation of data and shared data structures. The compiler is analogous to a query optimizer in that it searches a space of possible algorithms and evaluates each with respect to a cost model in order to choose an implementation for a relational interface.

References