Automatic Verification of Database Transaction Safety

Tim Sheard
David Stemple

COINS Technical Report 88-29
April 1988

Revised version of COINS Technical Report 86-30.
This paper is based on work supported by the National Science Foundation under grants DCR-8503613 and IRI-8606424, and by the Office of Naval Research University Research Initiative contract number N00014-86-K-0764.

Automatic Verification of Database Transaction Safety

Tim Sheard
David Stemple
Department of Computer and Information Science
University of Massachusetts, Amherst, MA. 01003

April 11, 1988

Abstract

Maintaining the integrity of databases is one of the promises of database management systems. This includes assuring that integrity constraints are invariants of database transactions. This is very difficult to accomplish efficiently in the presence of complex constraints and large amounts of data. One way to minimize the amount of processing required to maintain database integrity over transaction processing is to prove at compile-time that transactions cannot, if run atomically, disobey integrity constraints. We report on a system which performs such verification for a robust set of constraint and transaction classes. The system accepts database schemas written in a more or less traditional style and accepts programs in a high level programming language. Automatic verification fast enough to be effective on current workstation hardware is performed.

1 Introduction

Maintaining database integrity is a difficult task. It can be accomplished by ensuring that a database is only updated by transactions and that it obeys a set of predicates called integrity constraints after any transaction updates it, assuming that the database starts in a consistent state. This can be done by testing at run-time all integrity constraints that could be affected by each transaction before allowing the transaction to commit. Not only could such testing be very expensive, but the determination of the minimal set of integrity constraints violatable by a given transaction is itself difficult. Many systems require that users specify the points at which individual integrity constraints are to be checked, for example, on

specific updates or at commit time of any transaction. This, too, is problematical since users will, in even moderately complex cases, make many errors both by specifying points at which constraints could not be violated and missing points at which checks should be made.

This paper describes a tool for use in developing systems which maintain the integrity of highly constrained databases, without the overhead of unnecessary runtime tests. Integrity is maintained by allowing only *safe* transactions to update the database. A safe transaction is one that is guaranteed not to violate the integrity constraints. The tool is a theorem prover that can be used to prove that database transactions are safe. Whenever a transaction cannot be proved to be safe, it is helpful to a designer if the parts of the transaction and schema that caused the failure are identified. The system could generate feedback to the transaction designer by suggesting either a set of runtime tests on the database or additional updates that would make the transaction safe. Such feedback is included in the design tools that we have built, but are described elsewhere [20].

We have built a system which is effective enough on a robust set of transactions and constraints to be useful in a database development system running in a workstation environment. The system accepts database schemas containing constraints in a relatively standard schema language and transactions in a high level programming language and attempts to prove that the transactions obey the constraints stated in the schema.

The major system components are:

- A theorem prover used to build a formal theory about database systems. This theory is based on a functional model of tuples, finite sets, and natural numbers. The theory is in the style of Boyer and Moore [2], but uses a novel axiomatization of finite sets and includes higher order functions and theorems [17], [15], [18].

- A schema translator used to extend the theory to a particular database by generating specific knowledge about an application from the structures and constraints contained in the database schema. The classes of constraints currently expressible in our schema language include functional dependencies, inclusion dependencies, aggregate constraints, intersection dependencies, and inter-relational redundancies, all specifiable on arbitrary subsets and derivations from a database.

- A transaction safety verifier which uses the general knowledge about database systems embodied in our database theory along with the specific knowledge generated from a schema to verify the safety of each transaction in a system. Our transactions are constructed from arbitrary conditional combinations of simple and complex updates of multiple relations. Complex updates are mod-

elled as generic update functions which take functions as parameters to make them specific. Knowledge about the generic updates (and generic constraints which are handled in the same manner) is stored in special second order theorems, called *meta-lemmas*. Meta-lemmas are used to reason about the generic constraints and updates during the safety proof. Much of the power and efficiency of our system derives from the use of meta-lemmas.

- A test generator which provides feedback for unsafe transactions. Feedback can consist of simply identifying a constraint that cannot be proven or in some cases a sufficient set of run-time tests that could be added to the transaction to make it safe [20].

Each of these components except the last is discussed at length in succeeding sections of this paper. Both the class of constraints and the class of transactions are expandable by adding to our theory new function definitions and proving the proper theorems. But this expansion of the constraint and update set is currently a job for an expert, though it is possible that it could be accomplished by database system designers when additional tools are built.

Our work is related to other work on mechanical proofs of transaction safety [6], [8], [3] and [4]. It is also related to work on simplifying constraint tests [1], [21], [11], [12], and [16]. A major difference between our work and that cited is that our work makes use of a general purpose, computational logic theorem prover, and thus is based on (higher order) recursive functions rather than on first order predicate logic. Our work also deals with larger classes of integrity constraints and transactions. Section 2.5 discusses these differences. The system represents, to the best of our knowledge, the first implementation of a transaction safety verifier which handles a realistically robust set of constraints and transactions entered in traditional schema and transaction program form. Timings on a workstation indicate that the system performs well enough to make it an effective tool for database system designers.

In the rest of this paper we first define the range of constraints and updates currently managed by our system. We then present a sample schema and three transactions to illustrate how a designer would use the system's specification capabilities. We follow this with a comparison of our work with previous work in the area. We then discuss the theory building and proof techniques behind the tools we have developed. We finish by stating our conclusions and discussing future work to be done.

2 Constraint and Transaction Classes

We call our specification language the Abstract Database Type Programming Language (ADABTPL, which we pronounce adaptable). It is used to write schemas

and transactions. A database system specification in ADABTPL contains three sections. The first is a type definition section wherein abstract data types are constructed using tuples, lists, and finite sets along with derivation by arbitrary predicates in *where* clauses attached to each type definition. These types are used to build a database type (the second section) and to constrain transaction input. The second section describes the database object itself. The database object can also be restricted by a predicate stating the integrity constraints on the database. The first and second sections constitute the database schema. The third section describes the set of transactions to be run on the database.

The database object can become constrained in two ways: directly by placing constraints in the *where* clause of the database type definition, or indirectly by inheriting constraints from the types of its components. A database can be specified as a relational database [1] in which case it is defined as a tuple whose components are relations. In this case, inter-relational constraints are specifiable only in the direct method as nowhere else in the schema do two entities of type relation (i.e., a set of tuples) appear. Relational and domain constraints are more flexible as they can originate either directly in the database type where clause or by inheritance via a component type.

ADABTPL provides special constructs for expressing some of the relational algebra operators. We use the dot notation to express both tuple and relational projection. If **X** is a tuple object with a component called **name** then **X.name** is the **name** component of **X**. If **X** is a relation then **X.name** is the set of all the **name** components in **X**. We indicate the projection of more than one component by listing them in square brackets. For example, **X.[Name, Age]**. We indicate relational selection with the **(All x in R where P(x))** construct. This selects only those tuples in **R** where the predicate **P** holds.

2.1 Example Database

We now introduce a particular database to illustrate the use of our system. The database describes the workings of a job matching agency. Entities in the database include people, who apply for jobs and are placed with companies. Companies offer positions and hire people. Skills are required for certain jobs, and people have abilities that satisfy skill requirements. An entity relationship (ER) diagram for the database is shown in Figure 1. The rectangles of the ER diagram represent entity sets and these will be specified in ADABTPL as keyed relations, represented by tuple and set types and a relation component (typed as a set of tuples) of the database. The diamonds stand for relationship sets. We will specify these in our ADABTPL

[1] The example we use in this paper is of a first normal form relational database, but neither the ADABTPL language nor our database theory is limited to first normal form relations or even to relations.

schema as relations containing foreign keys. The identifiers in the rectangles and diamonds are attributes and, if underlined, are keys. The ER diagram imposes integrity constraints on the relational specification, namely key constraints and the referential integrity of the foreign keys in the relationship relations. For example, there should never appear a [**Pid, Jid**] pair in the **Applications** relation in which **Pid** does not appear in the **Pid** column of the **Persons** relation, or **Jid** does not appear in the **Jid** column of the **Jobs** relation. The relational schema given in Figure 2 for this database system explicitly states these constraints.

We also constrain this system with constraints that are not expressible in an ER diagram. The "clouds" in Figure 1 indicate relational and inter-relational constraints which we also want the system to enforce. These constraints are as follows.

1. A person should never simultaneously be placed in a job and have an application for a job, i.e. only unemployed persons can apply for jobs.

2. Each company shall offer jobs for which they have one or more openings. A company should never offer a job with zero openings.

3. All persons who are applying for a job should have the skills required for the job they are applying for.

4. The **Placed** field of the **Persons** relation is a redundant field. That is, it could be computed by testing if the person is in the **Placement** relation. We will represent this value redundantly for convenience, (and quicker access if the physical database uses the tuple structure specified), and we want the system to ensure that the two representations of this fact always agree.

5. In a similiar manner the **TotSal** component of each **Company** tuple in the **Companies** relation should agree with the sum of all the company's employees' salaries in the **Placement** relation.

These constraints are also expressed in the ADABTPL schema in Figure 2. We now present the set of constraint primitives that can be used in an ADABTPL schema.

2.2 Individual Constraint Mechanisms

In ADABTPL, integrity constraints appear in **where** clauses in type definitions and act to restrict the values that are members of the type. They may appear in any type including all those that are used to specify the database type. (Other type definitions specify the types of inputs to transactions.) In specifying first normal form databases, integrity constraints fall into four broad categories.

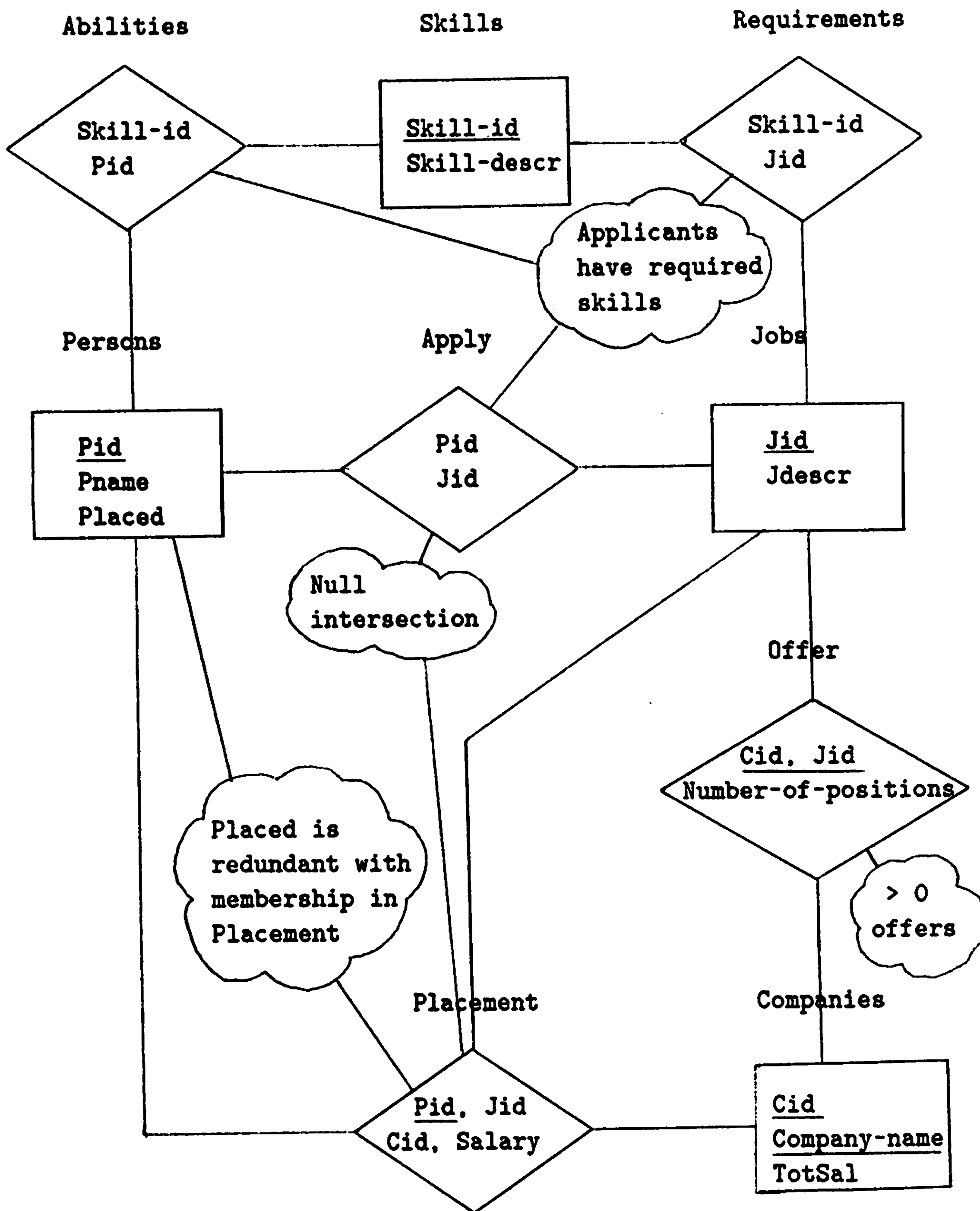

Figure 1: ER Diagram for Job Agency Database

```
Person = [ Pid: number; Pname: string; Placed: boolean ];
Person-rel = Set( Person) where  Key(Person-rel, Pid);
Job = [ Jid: number; Jdescr: string ];
Job-rel = Set( Job) where Key(Job-rel, Jid);
Skill = [ Skill-id: number; Skill-descr: string ];
Skill-rel = Set( Skill) where Key(Skill-rel, Skill-id);
Ability-rel = Set( [ Skill-id: number; Pid: number ]);
Requirement-rel = Set( [ Skill-id: number; Jid: number ]);
Company = [ Cid: number; Company-name: string; TotSal: number];
Company-rel = Set( Company)
  where  Key(Company-rel, Cid) and  Key(Company-rel, Company-name)
Application-rel = Set( [Pid: number; Jid: number ]);
Offer = [ Cid: number; Jid: number; Number-of-positions: number]
  where Number-of-positions <> 0 ;
Offer-rel = Set( Offer) where  Key(Offer-rel, Cid, Jid);
Placement = [Pid: number; Jid:number; Cid:number; Salary:number];
Placement-rel = Set( Placement) where Key(Placement-rel, Pid);
Database Job-agency :
    [ Persons: Person-rel; Jobs:  Job-rel; Skills: Skill-rel;
      Abilities: Ability-rel; Requirements: Requirement-rel;
      Applications: Application-rel; Offerings: Offer-rel;
      Companies: Company-rel; Placements: Placement-rel ]
where Contains(Persons.Pid, Placements.Pid) ;
      Contains(Companies.Cid, Placements.Cid) ;
      Contains(Jobs.Jid, Placements.Jid) ;
      Contains(Jobs.Jid, Offerings.Jid) ;
      Contains(Companies.Cid, Offerings.Cid) ;
      Contains(Persons.Pid, Abilities.Pid) ;
      Contains(Persons.Pid, Applications.Pid) ;
      Contains(Jobs.Jid, Applications.Jid) ;
      Contains(Jobs.Jid, Requirements.Jid) ;
      For all x in Persons
          x.Placed is-redundant-with x.Pid in Placements.Pid ;
      For all a in Applications:
       Contains((All x in Abilities where a.Pid = x.Pid).Skill-id,
            (All y in Requirements where a.Jid = y.Jid).Skill-id);
      Null-intersection (Applications.Pid, Placements.Pid);
      For all c in Companies c.TotSal is-redundant-with
          Sum( (All p in Placements where p.Cid = c.Cid), Salary);
```

Figure 2: The Job Agency Schema

1. Domain constraints which limit the values a particular component of a tuple may take on, independently of the values of other components of the tuple.

2. Tuple constraints which limit the combinations of components constituting valid tuples of a type.

3. Relational constraints which limit the value a set of tuples can take on. The key constraint and universal and existential quantification over the elements of a relation are examples of this type.

4. Inter-relational constraints which limit the value a relation can take on by relating it to some other relation or relations in the database. Inclusion dependencies, referential integrity, intersection dependencies, and redundancies are examples of this class of constraint.

All these kinds of constraints are expressible in our schema language using the constraint primitives discussed in this section. [2]

2.2.1 Arithmetic Constraints

Currently the system reasons about the natural numbers. It has definitions and knowledge about the following numeric relations: equality, less-than, greater-than, less-than-or-equal, greater-than-or-equal, and non-equality. In addition it knows about the arithmetic operators, addition, subtraction, and multiplication. In the schema language these operations are indicated by the use of the symbols, =, <, >, <=, >=, <>, +, -, and *, respectively.

2.2.2 Set Membership and Containment

We allow constraints and tests which test for set membership (or non membership). In our schema and transaction languages we use the infix operator **in** to denote set membership. Set containment is useful in stating referential integrity and other inclusion dependencies. We denote set containment using the prefix operator **Contains**. By combining projection on sets and containment we can define referential integrity. For example, to declare that all those applying for jobs must be subscribing persons we state

```
Contains ( Person.Pid, Applications.Pid )
```

[2] We do not currently handle transition constraints (predicates on pairs of states used to require or preclude certain properties when changes are made), though our proof techniques should work as effectively on verifying that transactions obey these constraints as they do in verifying database invariants.

More complex inclusion dependencies can also be stated with **Contains**. The constraint that all persons applying for a job must have abilities which include all skills required for that job can be stated as

```
For all a in Applications:
    Contains((All x in Abilities where a.Pid = x.Pid).Skill-id,
             (All y in Requirements where a.Jid = y.Jid).Skill-id)
```

Note the use of the **All x in R where P(x)** construct which denotes set selection. We use the dot notation to denote set projection even when the set being projected is an expression rather than a variable.

2.2.3 Universal and Existential Quantification

The above constraint is also an example of universal quantification. We specify quantification over sets as

```
For all  x in R: P(x)
For some x in R: Q(x)
```

where **P** and **Q** can be any predicate (even involving another quantifier). In addition to the bound variable **x**, the predicates **P** and **Q** can contain unbound variables denoting the component relations defined in the database section of the schema. For example, the only unbound variables in the **For all** statement in 2.2.2 are **Abilities** and **Requirements** which are database components.

2.2.4 Null-intersection

Currently our intersection dependencies are limited to specifying that two sets have a null intersection. We use this to state that two sets do not overlap. In the **Job-agency** example the constraint that the same person cannot simultaneously be placed and be applying for a job is stated as

```
Null-intersection ( Applications.Pid, Placements.Pid )
```

2.2.5 Redundancies

A database contains redundant information when some components of the database are computable from others. This is often done to provide convenient reference (and fast access if the redundant information is actually stored in the physical database) to the redundant information. It is important that redundant parts of the database remain consistent. Our schema and transaction language provides a special construct for stating a particular kind of redundancy between two relations. Let **R1** and **R2** be two relations. Then

```
For all x in R1 F(x) is-redundant-with G(x,R2)
```

states the redundancy that the two functions **F** and **G** compute the same values when applied to any x in **R1** and to x and **R2**, respectively. A convenient **F** is a component function which identifies a component of the tuples that make up **R1**. For example,

```
For all x in Persons x.Placed is-redundant-with x.Pid in Placements.Pid
```

states that the **Placed** component of all tuples in **Persons**, a **boolean** value, agrees with the membership of the person identifier in the **Pid** projection of **Placements**. The **G** function in this example is set membership, denoted by the infix **in** operator, and **R2** is **Placements.Pid**.

2.2.6 Aggregate Constraints

Our schema and transaction language currently allows two aggregate functions, **Sum** and **count**. These aggregate functions can exist within the normal constraint mechanisms to provide aggregate constraints. The **Sum** function takes two inputs, a relation and the name of a column over which the sum is to be computed. The column name must be numeric. Consider the constraint:

```
For all c in Companies c.TotSal is-redundant-with
     Sum( (All p in Placements where p.Cid = c.Cid), Salary)
```

which states that the **TotSal** component of each tuple in the **Companies** relation agrees with the sum of the **Salary** component of all those tuples in the **Placements** relation that match the **Companies** tuple's **Cid** component. Note that this constraint is a mixture of an aggregate function, a set selection and a redundancy, and is handled quite naturally by our system with no special mechanisms for dealing with aggregates.

Other aggregate functions such as **Max** and **Min** could be handled easily in a similar manner by extending the knowledge base of the sytem with function definitions and lemmas about them. The **Average** function will have to wait until the system is extended to handle rational numbers.

2.3 Updating Mechanisms

A transaction verification system should handle a diverse range of updating constructs, powerful enough to easily and succinctly express real database changes. It is easy to see that simple inserts and deletes which most previous work has dealt with are insufficient or inadequate for this task. At present our system provides, in addition to simple inserts and deletes, four complex updating constructs. We have

developed our theory of constraint interaction with these six types of updates to such a degree that we can prove transactions safety theorems quickly enough that a transaction designer should not be bored by long, tedious proofs. The system can accommodate additional updating constructs provided suitable knowledge about their interaction with our constraints can be discovered, proved, and entered into the system. Several new constructs are presently being evaluated for inclusion in the system. Currently this is a job for system experts. Building tools to enable database system designers to successfully add new or unique updating constructs is currently an area of research.

2.3.1 Simple Inserts and Deletes

Insertion into (or deletion from) relations is handled by building a tuple using the tuple constructor (square brackets [ ]) and inserting (or deleting) the tuple from the required relation. For example

```
Insert [ id, name, false ] into Persons
```

inserts a **Person** tuple with person identifier component, **id**, **Pname** component, **name**, and **Placed** component, **false**.

```
Delete [ id, name, false ] from Persons
```

would delete that tuple from the persons relation. Since these are true set operations the **Insert** will add a tuple to a set only if it is not already in the set (i. e., no duplicates are allowed). If a tuple is already in a set and it is inserted the insert does not change the set. In a similiar manner, **Delete x from R** has no effect whenever **x** is not in **R**.

2.3.2 Remove

Often the exact tuple (or tuples) to be removed from a relation are not fully known. That is, we do not know all the components of every tuple which we want to delete. In this case the **Remove** construct is useful. The **Remove** construct deletes all tuples meeting a certain predicate from a relation. For example,

```
Remove p from Applications where p.Pid = hiree
```

deletes all tuples from **Applications** where the **Pid** component has the same value as the variable **hiree**.

2.3.3 Multiple Inserts

Another common type of update is the insertion of several tuples of one relation into another relation. Often all the tuples to be inserted share some common components but differ on the key component(s). We model this type of update with our **For each** construct.

```
For each x in R1 Insert F(x) into R2
```

Here **R1** is some set (or relation). We take each tuple in the set, transform it by the function **F** and insert the transformed tuple into **R2**. Since **F** is an arbitrary function, it could be the identity function. In this case the **For each** construct models set union. Normally **F** constructs tuples which match the type of the target relation **R2** from tuples of the type of the source relation **R1**. In the **Job-agency** database when a new person subscribes to the job agency service we would like to add to the **Abilities** relation a set of tuples matching each of the the new subscriber's skills with his person identifier. Let **pskills** be a set of **skill-ids** for the person with person identifier id. We do this as follows

```
For each s in pskills Insert [s, id] into Abilities
```

2.3.4 Update

When one wants to change some components of selected tuples in a relation the **Update** construct is the appropriate choice. The format of the **Update** construct is

```
Update x in R
        where P(x) by [ comp1 := F(x), comp2 := G(x),   ... ]
```

P is a predicate function on tuples which selects those tuples to be updated. The construction inside the square brackets tells how to construct the new value that the tuple x is to be replaced by. Components not mentioned in the square brackets are left unchanged. For example,

```
Update p in Persons where p.Pid = hiree by  [ Placed := true ]
```

changes only those tuples in **Persons** whose **Pid** component is the same as the value in the variable **hiree**. The changed tuples are identical to their old values except that their **Placed** components are now **true**.

2.3.5 Selective Updating or Deleting

Our last updating construct allows one of three possible outcomes to be applied to each tuple. Each tuple can be 1) left unchanged, 2) deleted, or 3) have some of its components changed as is done in the update just described. We specify the use of this construct with the following format.

```
Update x in R where P(x) by
       if Q(x) then delete x
               else [ comp1 := F(x), comp2 := G(x), ...]
```

Each tuple for which **P** is not true is left unchanged. Tuples for which both **P** and **Q** are true are deleted. Tuples for which **P** but not **Q** is true are updated. For example, when a person is hired, it is necessary to decrement the **Number-of-positions** available in the **Offerings** relation by one, since one fewer positions now exist. If there was only one position available the **Offering** tuple should be deleted since the integrity constraint on the **Offer** type states that no **Offer** tuple should have a **Number-of-positions** component of 0.

```
Update o in Offerings where o.Cid = comp and o.Jid = jb by
       if o.Number-of-positions = 1
          then Delete o
          else [ Number-of-positions := Number-of-positions - 1 ]
```

Transactions are constructed from any number of the updates described above in some order and nested in arbitrary if-then-else structures. Changes made by one statement's actions are seen by subsequent statements. Whenever a loop is controlled by a set, as in the **For each** construct, the set maintains its original members unchanged for the purpose of controlling the loop, though an intermediate state of the set can be passed to a function called inside the loop. A transaction may optionally contain an arbitrary predicate on the database and input as a precondition. A transaction is not run if its precondition is not true at the time of the request to run the transaction.

2.4 Example Transactions

In this section, we define a few transactions on the **Job-agency** schema and use them to illustrate how a designer would use our verification system. The **Job-agency** database consists of nine relations (one for each entity and relationship) and is constrained by twenty-one integrity constraints, comprising nine referential integrity constraints implied by the ER diagram, seven key constraints, and the five constraints deriving from the clouds in the ER diagram.

The transactions are non-trivial, one updating five relations. We believe that the transactions are of average complexity, comparable to those found in real,

non-toy systems and would be typical of complex transactions even on databases with much larger numbers of relations and constraints. Because of the manner in which *inertia* (the property of unupdated parts of the database continuing to obey local integrity constraints) is handled, the size of a database's structure does not unduly increase the time of a verification. Since the verification is a transaction compile-time operation, the size (extent) of the database does not affect the speed of verification at all.

We now discuss each transaction and give the time needed to to prove it safe.

```
Transaction Subscribe(id:number; name: string; pskills: set of number);
Preconditions
   id not-in Persons.Pid;
Begin
Insert [id, name, false] into Persons;
For each s in pskills Insert [s, id] into Abilities
End;
```

The simplest transaction is the **Subscribe** transaction. It models the event of a new person subscribing to the job agencies services. The transaction takes as input a person identifier, a subscriber name, and a set of skills which the subscriber has mastered. It adds a new person tuple to the **Person** and a set of [**skill, person-identifier**] pairs to the **Abilities** relation. It is necessary to test the precondition that the new person does not already exist in the database, or the key constraint may be violated. If the transaction were defined with the precondition omitted the result of the safety proof would be exactly this test as an unresolved subgoal. This illustrates one segment of the current test generation ability of the system. The transaction safety theorem for **Subscribe** states that if the database state obeys the types and integrity constraints in the schema, the inputs are of their declared types, and the preconditions are met, then the changes made by **Subscribe** will leave the database obeying the schema constraints and types. (We tacitly assume atomicity of transactions, i. e., serializable schedules of interleaving.) This theorem is proven by our verifier in 18 CPU seconds on a Texas Instruments Explorer II. (Our system is written in Common LISP.)

```
Transaction Fire(emp, comp, jb, sal:number);
Preconditions
   [emp, jb, comp, sal] in placements;

Begin
 Remove x from Placements where x.Pid = emp;
 Update p in Persons where p.Pid = emp by [Placed := false];
 Update c in Companies where c.Cid = comp by [TotSal := TotSal - sal]
End;
```

The **Fire** transaction removes an employed person from the **Placements** relation. It must set the person's **Placed** field to **false** in the **Persons** relation, and adjust the company's **TotSal** field in the **Companies** relation. Note that the precondition tests only that the **Placements** tuple, [**emp, jb, comp, sal**] exists. The existence of similar tuples in **Persons** and **Companies** is proven from the system using this condition and the referential integrity constraints of the schema. The **Fire** transaction is proven safe in 25 seconds. A partial transcript of **Fire**'s safety proof is given in Appendix 2.

```
Transaction Hire (comp, hiree, jb, sal: number ) ;
Preconditions
   hiree in Persons.id ;
   [comp, jb] in Offerings.[Cid, Jid] ;
   hiree not-in Placements.Pid ;
Begin
 Update p in Persons where p.Pid = hiree by [ Placed := true];
 Update o in Offerings where o.Cid = comp and o.Jid = jb by
        if o.Number-of-positions = 1
           then Delete o
           else [Number-of-positions := Number-of-positions - 1];
 Insert [hiree, jb, comp, sal] into Placements;
 Remove p from Applications where p.Pid = hiree;
 Update c in Companies where c.Cid = comp
        by [ TotSal := TotSal + sal ]
End;
```

The **Hire** transaction involves placing an applicant in a job which is currently offered. For this transaction to be safe it must update five separate relations in the correct manner to ensure database integrity. In addition to adding the person to the **Placements** relation, his/her **Placed** field must be set to **true** in the **Persons** relation. In the **Offerings** relation the **Number-of-positions** open must be decremented, or the tuple removed if the number falls to zero. The application for the hiree must be removed, and the **TotSal** of the hiring company must be adjusted. This transaction generates a theorem containing around six hundred terms that is proven in 38 CPU seconds.

2.5 Relation of Our Constraints and Update Mechanisms to Previous Work

All previous formal work on constraint maintenance has been done in the context of the relational model. Our work has also concentrated on the relational model,

but is not restricted to it. (See, for example, [19].) The work of Bernstein and Blaustein deals with simple updates and integrity constraints expressed in a limited form of predicate calculus. Hsu and Imielinski continue this work by expanding the set of constraints and dealing with sets of simple updates. Gardarin and Melkanoff take a more encompassing view, reasoning about manipulations of first normal form relations using proof techniques based on the Hoare axiomatic method [9] applied to programs in ALGOL 60 extended with relational operators and first order predicates. Casanova and Bernstein's work is similiar except that they use an extended form of dynamic logic.

Nicolas uses range-restricted well formed formulas in prenex normal form. He considers single insertions and deletions (an update is a deletion followed conditionally by an insertion) of tuples. He does not allow deletes of the form "delete all tuples of employee where salary > 40000" - the user needs to find all the tuples and delete them one by one. He derives conditions for simplification of the integrity constraints under single inserts and deletes.

Henschen, McCune and Naqvi allow domain independent well formed formulas - a broader class of well formed formulas than Nicolas - in prenex form, converting the well formed formulas into clauses through skolemization. They too treat single-tuple inserts and deletes as well as a relation based update which updates all tuples which meet an equality predicate. They use a resolution theorem prover equipped with special-purpose heuristics to prove the consistency of the resulting database state, failing which they suggest that since the theorem is a disjunction any subset of the remaining clauses can be taken as a sufficient test to be performed at run-time.

Bernstein and Blaustein allow range-restricted well formed formulas in the prenex form such that a quantified variable may range over only a single relation and further, only one variable may range over a given relation. They analyse only single-tuple inserts and deletes, and allow quantifiers. They analyse cases where the constraint is automatically satisfied, others where the tests need only involve the input data, and still others where either a simplified form of the constraint can be tested, or a simple test can be performed on a reduced substate of the database stored for the purpose.

Hsu and Imielinski allow range-restricted well formed formulas in prenex form where a quantified variable may range over a single relation. Thus they allow co-range constraints where two or more variables may range over a single relation. They consider multiple inserts and deletes by analysing the pattern of quantifiers in the prefix of the constraints. Their algorithm attempts to find patterns that allow a decomposition of the universal and existential quantifier into a conjunction and disjunction respectively of simpler constraints. They too, like Bernstein and Blaustein, imply that some reduced substates of relations may be stored to remember the information guaranteeing existence predicates for quick computation.

Gardarin and Melkanoff express the constraints in first-order predicate calculus. Their transactions are non-trivial ALGOL-like programs with assignment statements that take care of inserts, deletes, and updates, as well as conditional statements and predicates to select specific tuples of the database. They show how Hoare's axiomatic methods can be used to verify that consistency is maintained, but their procedures are not automated.

Casanova and Bernstein also consider first order predicate calculus as the language for integrity constraints. They model transition constraints as a pair of formulas. They extend dynamic logic by adding axioms to model a data manipulation language enriched with relational assignment, random tuple selection, and aggregate functions and use it to prove the invariance of the constraints under the transaction programs. No steps towards mechanisation of such proofs are evident.

The constraint and update mechanisms in our work subsume those dealt with in all the research on database transaction safety verification with which we are familiar. Two more important differences between our work and others is our use of a heuristic, lemma-driven theorem prover that uses higher order theory, and the fact that the types of constraints and updates which can be handled effectively by our system is extendable by adding new function definitions and theorems to the knowledge base of our tool. We now turn to the details of proving safety theorems.

3 Proving Safety Theorems

The proof of safety theorems is a complex undertaking that is accomplished in a number of phases, starting with the axiomatization of the primitive constructs from which database systems are built. The following steps represent our attack on this problem:

- Devise axioms for the primitive abstract data types from which database systems can be constructed: tuples, finite sets, lists and natural numbers.

- Prove properties of basic types, e. g.,

  ```
  insert(a, insert(b ,s)) = insert(b, (insert(a ,s))
  ```

- Add concepts to the basic theories by adding recursive function definitions for new predicates and operations.

- Define functions that cross types, e. g., relational projection in the combined theory of tuples and finite sets, and column sum in tuples, finite sets and natural numbers.

- Prove useful theorems in combined theories, e. g.,

```
sum(insert(a,s),c) = if member(a,s) then sum(s,c) else a.c+sum(s,c)
```

- Design a usable database specification language that has a clean mapping into the combined theory of the abstract data types.
- Write a compiler that builds a tailored theory from the specifications (schema and transactions) of particular database applications.
- Implement a verifier that works from a tailored theory to prove that the transactions in an application obey the database schema.

3.1 Building the Basic Theory

In this section, we discuss the first five phases of the process outlined above, those that constitute building the basic theory. We include an introduction to Boyer-Moore computational logic which forms the core of our inference techniques.

We first decided that tuples, finite sets and natural numbers would constitute the fundamental data types out of which we would structure databases. (We have since added simple lists, but we will not deal with them in this paper.) We also decided that Boyer-Moore computational logic would be the foundation of our mechanical reasoning about database systems. The primitives of this logic are essentially the boolean type with an if-then-else function. In this logic, abstract data types can be axiomatized in terms of rewrite rules on functional expressions. Boyer-Moore proofs are mainly a matter of reducing the functional expression of a theorem to true by using the axioms, function definitions and previously proven theorems as rewrite rules. Other techniques employed include generalization and induction. The order in which rewrite rules and other techniques are applied is based on heuristics described in [2]. It is not necessary to understand the details of these heuristics to understand our method.

It is well known that all sets of axioms are not created equal. Axioms can be incomplete, inconsistent, or just plain useless for the purposes of mechanical reasoning. Many have attempted to codify the properties necessary for being a good axiom set, such as possessing the Church-Rosser property, having *reciprocity* [10], containing *acyclicity* axioms [13], being *sufficiently complete* [7], or being *safe*, [14]. Our exposure to Boyer-Moore logic led us to adopt what we will call the "constructor-destructor" approach to the syntax and axioms of our base abstract data types. In this approach, a type has constructor and destructor (selector) operations, at least one of the latter for each input to the major constructor. An example is a stack type with two constructors, **EmptyStack** and **Push** and two destructors, **Top** and **Pop**, since **Push** takes two inputs (the semantics of **Pop** is the stack with the **Top** removed). The axioms in this approach should be equations in which expressions having a selector followed by the multi-operand constructor

are on the left-hand side of the equals, and the right-hand side is an expression containing only "simpler" terms. In the stack example, the main axioms of this form are:

```
Top(Push(e,s))  = e
Pop(Push(e,s))  = s
```

The simplicity of these axioms reflect the fundamental nature of the stack as a data structure. Note that natural numbers in this form have two constructors, **PlusOne** and **Zero**, but only one selector, **MinusOne**, since **PlusOne** has arity one, and only one major axiom:

```
MinusOne(PlusOne(n)) = n
```

Tuple types can be axiomatized quite conveniently with axioms of this style. Let **TupleCons** be a constructor of n-ary tuples with components from the n sorts or types **C1**,...,**Cn**, and let **S1**,...,**Sn** denote the selector functions that select from a tuple the ith component, for i=1,...,n. Then the following n axioms capture nearly all of the behavior of such a type:

```
Si(TupleCons(e1,...,en)) = ei,  i=1,...,n
```

Note that we have only introduced one constructor function. A constructor such as **EmptyStack** or **Zero** is required only if the constructor that takes more than zero inputs needs at least one object of the type being constructed (in which case, we call the operation inductive and the type recursive). **Push** is such a constructor. In these cases, the type needs a constructor that can produce an object from nothing, or from an element of another independent type. This constructor is often identified as an object, namely, the *bottom* object of the type. We chose to define tuples as a simple (non-recursive) type in which tuple components are not of the same type as the type being constructed, nor of a type that depended on it. (The stack type above can be considered to be a tuple type without this restriction.) The only other axiom needed to fully capture the tuple structure is:

```
for T1 and T2, n-ary tuples of the same type,
T1 = T2 -> Si(T1) = Si(T2),  i=1,...,n
```

It would be convenient if all interesting primitive types had such simple axioms as suffice for natural numbers, stacks (which, in our axiomatization above are the same as simple lists), and tuples. Unfortunately, finite sets cannot be axiomatized using such a scheme unless the right-hand sides of the major axioms are more complex. If we decide to name the two constructor functions **emptyset** and **insert**, and the two selector functions **choose** and **rest**, then the required axioms are given in Figure 3. Note that the right hand sides of the two basic

axioms, though complex, only have terms at the different places in the right-hand side if-then-elses in which either the **insert** has been eliminated or has had its second (inductive) argument changed from **S** to **rest(S)**. The purpose of this is to set the stage for case-based reasoning about complex set expressions, in which each case has been simplified and in some cases prepared for an induction proof.

There are two restrictions on the assignment of sorts to the sort variable **elements**. The first is that, though **elements** may be a set type, it may not be the set type being defined or be based indirectly on it. The sort substituted for **elements** must have an equality relation and allow a **before** function which obeys the axioms involving **before**. The requirement that **smaller** be well-founded is a stronger version of the acyclicity axioms of Oppen.

The axioms specify the expected behavior of finite sets. How do we know this? Because we have mechanically proven the theorems that state these behaviors. For example, the following theorems have been proven:

```
insert(a,insert(b,s)) = insert(b,(a,s))
insert(a,insert(a,s)) = insert(a,s)
```

Are there unexpected properties that are specified by these axioms? We don't know. (Remember that it is not known whether classical set theory is even consistent.) We haven't found any, but maybe we haven't looked in the right places. The most problematic aspect of these axioms is the presence of the **before** relation on the domain set. Neither the **before** nor the **smaller** relation is meant to be visible or used in functions defined on finite sets in our system. How restrictive is this feature of the axioms? Again, we cannot be sure. The actual culprit may be simply the stipulation that **choose** is a function. This induces an order on the elements in any finite set. All that the **before** relation does is name it and posit its definition on the element domain rather than allow it to change from set to set. We can have a different **before** for each set type we define (these axioms are generic) even to the point of having two different set types with the same element domains. More importantly, the only requirement on **before** in the theory is that it be a total order (anti-symmetric and anti-reflexive). Any particular order is valid and is a feature of an interpretation (model) of the theory. Reordering the elements of a set is simply to change the model, not the theory.

The set theory determined by the axioms so far is a meager theory. In it we cannot express anything about operations such as **delete**, **union** and **intersection**, nor anything about membership, the quintessential property of classical sets. Thus, we next extend the theory by giving recursive definitions for these concepts:

```
member(e,s) = if s = emptyset
                then false
                else if e = choose(s)
```

```
Syntax (signature)

emptyset: --> fsets
rest: fsets --> fsets
choose: fsets --> elements
insert: elements X fsets --> fsets
before: elements X elements --> boolean
smaller: fsets X fsets --> boolean

Axioms

rest(insert(e, s)) = if s = emptyset
                       then emptyset
                       else if not (e = choose(s))
                             then if before(e, choose(s))
                                   then s
                                   else insert(e, rest(s))
                             else rest(s)
choose(insert(e, s)) = if s = emptyset
                         then e
                         else if before(e, choose(s))
                              then e
                              else choose(s)
(not (s = emptyset)) --> insert(choose(s), rest(s)) = s
not (insert(e, s) = emptyset)

a = b --> not before(a, b)
before(a, b) --> not before(b, a)
before(a, b) and before(b, c) --> before(a, c)
(not (a = b)) --> before(a, b) or before(b, a)

smaller is a well-founded relation and
(not (s = emptyset)) --> smaller(rest(s), s)
```

Figure 3: Finite Set Axioms

```
                            then true
                            else member(e, rest(s))
delete(e,s) = if s = emptyset
                then emptyset
                else if e = choose(s)
                        then rest(s)
                        else insert(choose(s), delete(e, rest(s)))
union(r,s) = if r = emptyset
               then s
               else insert(choose(r), union(rest(r), s))
```

After defining new concepts, both predicates, e. g., **member**, and actions, e. g., **delete** and **union**, we prove new theorems involving the new concepts, such as:

```
member(e, insert(e, s))
not(member(e, emptyset))
not(member(e, delete(e,s)))
member(e, r) -> member(e, union(r, s))
union(s, emptyset) = s
union(r, s) = union(s, r)
```

We also add concepts that normally come with the underlying logic, such as universal and existential quantification over sets:

```
for-some(s, p) = if s = emptyset
                   then false
                   else if p(choose(s))
                           then true
                           else for-some(rest(s))
for-all(s, p) = if s = emptyset
                  then true
                   else if p(choose(s))
                           then for-all(rest(s), p)
                           else false
```

Note that these definitions are not strictly first order in that the **p** parameter is a (predicate) function from the element type to boolean. We will see many such functions when we define functions over tuples and finite sets, such as projection, in which sets of column names (functions) are given to the projection function. This has not proven to be a problem. In fact, the escalation of this feature to theorems with true function variables is the feature of our system that is most responsible for its speed and effectiveness. We will address this issue in the following sections. For now, let us consider the proof of properties involving the concepts defined above.

3.1.1 A Sample Proof of a Basic Theorem

In this section we present, in some detail, a proof of a basic theorem about existential quantification over a set. We will prove that adding a tuple to a relation does not affect the existence of other tuples that have some property **p** defined as a predicate on single tuples. This theorem is stated by:

```
for-some(s, p) --> for-some(insert(e,s), p)
```

In all our theorems, variables are universally quantified over the proper type domains. In the running system, all types are checked prior to proof time by a type checking algorithm included in the compiler from ADABTPL into the internal form.

We will now go through the proof as produced by our basic theorem prover (our implementation of the prover described in [2] with some modifications and extensions). The prover decides that an induction proof is indicated and transforms the theorem into the conjunction

```
   s = emptyset --> (for-some(s, p) --> for-some(insert(e, s), p))
and
                 ((not (s = emptyset)) and
   (for-some(rest(s), p) --> for-some(insert(e, rest(s),p)))
                                 -->
   (for-some(s, p) --> for-some(insert(e,s), p))
```

This shows the form of induction proofs over finite sets, namely, to prove **P**(s) prove **P**(emptyset) and (s $\neq$ **emptyset and P(rest(s)))** -> **P**(s).

After forming the induction formula above, the proof continues in three steps. This is because the internal form of the formula is in conjunctive normal form and consists of three conjuncts or *clauses*, each of which must be proven for the proof to be accomplished. The first clause is the base case, shown on the left in disjunctive form, as held internally, and on the right as an implication. The numbers label the terms for identification purposes in the discussion that follows.

```
1    not(s = emptyset) or              s = emptyset and
2    not(for-some(s, p) or             for-some(s, p)
                                                    -->
3    for-some(insert(e, s), p)         for-some(insert(e, s), p)
```

The basic strategy in dealing with a clause is to consider each disjunct (term) in turn and try to prove it from the negation of all the other disjuncts. One success in this ploy proves the clause (remember that **p or q or r** is equivalent to **not p and not q implies r**, as well as to **not p and not r implies q**, etc.). We show a form on the right that we have found easier to think about. The clause above is proven when the prover tries to prove the second term from the negation of the first and

third. Only the negation of the first is needed. The prover "opens" the **for-some** function (i. e., substitutes the function body for the call) in the second term and uses the negation of the first term, (s = **emptyset**), to reduce the call to **false**.

```
1  not(s = emptyset) or
2  not( if s = emptyset
            then false
            else if p(choose(s))
                 then true
                 else for-some(rest(s),p)
   or
3  for-some(insert(e, s), p)
```

```
s = emptyset and
if s = emptyset
   then false
   else if p(choose(s))
        then true
        else for-some(rest(s),p)
             -->
   for-some(insert(e, s), p)
```

This **false** is negated to **true** (in the disjunctive form) in the term, and thus the clause is proven.

The second clause follows. From now on we will show only the implication form.

```
1  not(s = emptyset) and
2  not(for-some(rest(s, p)) and
3  for-some(s, p)
       -->
4  for-some(insert(e, s), p)
```

This is simplified by opening term 3, the simplest call to **for-some**, producing

```
1  not(s = emptyset) and
2  not(for-some(rest(s, p)) and
3  if s = emptyset
      then false
      else if P(Choose(s)) then true else for-some(Rest(s),p)
      -->
4  for-some(insert(e, s), p)
```

Using **not(s = emptyset)** (term 1) and **not(for-some(rest(s, p)))** (term 2), the newly opened term 3 is simplified to if p(choose(s)) **then true else false**, which, by using a simple identity about if terms, leaves the whole clause as follows,

```
1  not(s = emptyset) and
2  not(for-some(rest(s, p)) and
3  p(choose(s))
       -->
4  for-some(insert(e, s), p)
```

The prover now starts simplifying the fourth term by opening the **for-some** function term to

```
4  if insert(e, s) = emptyset
      then false
      else if p(choose(insert(e, s)))
              then true
              else for-some(rest(insert(e, s)), p)
```

This is further simplified by using the finite set axioms. Using the axiom that states that an **insert** cannot be the **emptyset** eliminates the false **then** clause. The axioms that equate the **choose** and **rest** of an **insert** can be used to rewrite the **insert** expressions. Thus, p(**choose**(**insert**(e, s)) is replaced by the following term, which we show two ways. The form on the right shows the **if** expressions distributed out of the call to **p**.

```
p(if s = emptyset
       then e
       else if before(e, choose(s))
              then e
              else choose(s))
```

```
if s = emptyset
   then p(e)
   else if before(e, choose(s))
           then p(e)
           else p(choose(s))
```

Since we are using the assumption that s is not the **emptyset** (term 1) and p(choose(s)) is **true** (term 3), this simplifies to

```
if before(e, choose(s))
   then p(e)
   else true
```

The fourth term (**for-some**(**insert**(e, s), **p**)) has now been changed to

```
4   if (if before(e, choose(s))
             then p(e)
             else true)
       then true
       else for-some(rest(insert(e, s)), p)
```

which by the tautology

```
(if (if p then q else r) then left else right) =
(if p then (if q then left else right) else (if r then left else right))
```

is

```
4    if before(e, choose(s))
        then if p(e) then true else for-some(rest(insert(e, s)), p)
        else if true then true else for-some(rest(insert(e, s)), p)
```

By trivial rearrangement this becomes

```
4    if before(e,choose(s)) and not(p(e))
        then for-some(rest(insert(e, s)),p)
        else true
```

The **rest(insert(e, s))** in the **then** clause of term 4 is now rewritten using the **rest-insert** axiom to

```
if s = emptyset
       then emptyset
       else if (not (e = choose(s)))
               then if before(e, choose(s))
                       then s
                       else insert(e, rest(s))
               else rest(s)
```

which hardly seems like progress until we note that this reduces to s when the assumed facts are taken into account (terms 1, 2, and 3 and those produced by simplifying the **choose(insert(e,s))** term just dealt with). Thus term 4 becomes

```
4    if before(e,choose(s)) and not(p(e))
        then for-some(s, p)
        else true
```

This allows the prover to rewrite the previous clause as follows. The new terms 4a and 4b come from the test of the **if** term above, and 4c comes from the **then** clause.

```
1    not(s = emptyset) and
2    not(for-some(rest(s, p)) and
3    p(choose(s)) and
4a   before(e, choose(s) and
4b   not p(e)
      -->
4c   for-some(s, p)
```

The prover now needs only to open the last **for-some** (term 4c) yielding

```
if s = emptyset
   then false
   else if p(choose(s))
           then true
           else for-some(Rest(s),p)
```

and use the assumed

```
not( s = emptyset) and p(choose(s))
```

to reduce this clause to **true**.

The third and remaining clause is

```
not(s = emptyset) and
for-some(rest(s), p) and
for-some(insert(e, rest(s)), p) and
for-some(s, p) -->
for-some(insert(e, s), p)
```

The same process as was applied in the next to last step on the previous clause, i. e., using the three axioms to rewrite and simplify the last two **for-some** calls, reduces this clause to **true**, which completes the proof. (The proof takes 7 seconds of CPU time on a Texas Instruments Explorer II.)

3.1.2 Building the Knowledge Base

Once the prover has proven a theorem, the theorem can be put in a database or knowledge base for use in proving new theorems. We call these saved theorems lemmas for obvious reasons. New function definitions are also made available to the prover when appropriate. As the knowledge, in terms of lemmas and function definitions, grows, the job of deciding what to do at a given time in the proof process becomes more complex. In order to keep this from becoming a problem, we choose to delete certain theorems and function definitions from the prover's knowledge base. For example, after we have proven enough theorems about **for-some**, we "close" the definition, which tells the theorem prover that it is never to open the definition in the process of proving a theorem about **for-some**. This amounts to a kind of abstraction, not unlike data abstraction and certainly a normal and necessary part of doing any mathematics. We can also choose to discard certain theorems that were necessary in establishing other theorems, but which may be too detailed or specific to be of any use once other, more powerful theorems have been established. The choice of which theorems to leave in the knowledge base, and which functions to leave "open" is a knowledge engineering problem similar to those faced in the process of building expert systems. The major difference in our knowledge building process is that we are enforcing the discipline that our rules are proven to be sound within our axiomatic theory at every stage.

We can prove properties of the combined theories, e. g., for sets and tuples, by defining functions and proving theorems. For example, the sum of a column **c** (of integers) in a set of tuples having **c** as a selector is defined by

```
sum(s, c) = if s = emptyset
              then 0
              else c(choose(s)) + sum(rest(s), c)
```

From this we can prove the following theorem:

```
sum(insert(e, s)) = if member(e, s)
                      then sum(s, c)
                      else c(e) + sum(s, c)
```

This is a property of natural numbers, sets and tuples. Note that this has the same property as the major axioms for sets: the **insert** on the left-hand side of the equals either does not appear on the right (as in this case) or appears as an **insert** into **rest(s)** where **insert(s)** appeared on the right. This property allows such theorems to be kept around and used by the prover to simplify expressions that occur in new theorems to be proven.

To underscore the importance of the form of theorems that are left in the lemma base, consider the theorem proven in the last section,

```
for-some(s, p) --> for-some(insert(e,s), p)
```

The form of this theorem, (actually of a slightly more general theorem), that we store is

```
for-some(insert(e,s), p) = if p(e) then true
                                 else for-some(s, p)
```

The left-hand side of this form is a term that is likely to occur in the consequent of a transaction safety theorem since it is a predicate on an updated set. Furthermore, the **for-some(s, p)** is likely to appear in the antecedent of a safety theorem with the left-hand side in the consequent (as a result of the assumption that the database is consistent before the transaction is executed). Such patterns are quickly and simply proven. This indicates the importance, in computational logic, of leaving strategically powerful forms of theorems around, where the purpose of the theory is known.

By repeating the process of defining new functions, proving new theorems, hiding detailed theorems, and closing function definitions we have built a general theory of databases. It is in no sense complete, but it is extendable and has no ad hoc limitations beyond the fact that it does not include non-terminating functions. [3] However, the mechanisms we have presented so far are not suitable to the task of proving safety transactions. The general purpose nature of Boyer-Moore theorem

[3] Whenever a new function, such as those that define **delete** and **member**, is defined, the system tries to prove that it terminates. If termination cannot be established, the definition is not accepted.

proving seems to demand a lemma base too specific for use in proving the safety of a robust class of transactions over arbitrary schemas. When the lemma base is not just right, proofs can wander off into never-never land trying an unbounded number of possibilities. This phenomenon can be coped with by a knowledge base designer in the process of building a lemma base for some domain, but should not be visited on an innocent database designer, (no more than writing axioms should be required of programmers). The lemma base builder simply aborts these errant proof attempts, studies the offending clause, and posits a lemma that might help prove the theorem. In order to avoid the necessity of database designers having to get into this business, and to make the theorem prover execute efficiently, we had to raise the level of generality of theorems and produce an effective way of making access to this general knowledge. To this end, we extended Boyer-Moore theorem proving to deal with higher order (generic) functions and a special form of theorems (called *meta-lemmas*), both of which contain variables that range over typed function domains. The rest of this paper deals mainly with the definition and use of higher order functions and meta-lemmas.

3.2 Higher Order Functions and Meta-lemmas

Higher order (generic) functions and their associated meta-lemmas have proven to be an exceedingly parsimonious and efficient way of capturing useful generic knowledge about the domain of updating constrained databases. Higher order functions are used to encapsulate structured updates and integrity constraint primitives. We have also used higher order functions and meta-lemmas to capture knowledge common to groups of predicates sharing fundamental structure such as relational selection, set containment and universal quantification. In this section, we discuss the definition of higher order functions, the form of meta-lemmas, and the use of meta-lemmas at various times. One of the most important heuristics of our system is that which determines when to generate specific lemmas in anticipation of their need versus when to generate them on demand. Unlimited anticipation, which we tried, swamps the system with lemmas, few of which are used, and consumes inordinate amounts of time in the generation process. On the other hand, relying completely on demand for generation causes certain knowledge to be missed, and incurs high search costs.

We have defined **key**, **update**, **null-intersection**, **remove**, and a number of other important functions, both updating and predicate functions, as recursive functions with sets and functions as parameters. For example, the **update** function takes as input a set r, a predicate !p which tells which tuples to change, and a tuple transforming function !f (as well as some optional extra inputs &x). We use the convention that a variable that starts with an exclamation mark (!) is a function variable, and that a variable that starts with an ampersand (&) is optional. The

following is the form of our declaration to the function validation front-end.

```
(fun (name update)
(generic-types alpha &extra)
(params (r (set of alpha))
        (!f (function ((cross alpha &extra) where (x &y) (!p x &y))
                      alpha))
        (!p (function (cross alpha &extra) boolean))
        (&x &extra))
(body (if (equal r emptyset)
          emptyset
          (if (!p (choose r) &x)
              (insert (!f (choose r) &x)
                      (update (rest r) !f !p &x))
              (insert (choose r)
                      (update (rest r) !f !p &x)))))
(result (set of alpha)))
```

The definition can be read as follows. The function name is **update** and it is defined in terms of two arbitrary generic types, **alpha** and **&extra**. The function has four parameters. The first is a set of **alpha**, where **alpha** can be any type since it is generic. The second and third parameters are functions. **!p** is a predicate from **(alpha, &extra)** to **boolean**, and **!f** is a function from **(alpha, &extra)** to **alpha**. The **where** part of the function type declaration for **!f** states that **(!f x &y)** may only be called where **(!p x &y)** is **true**. This is an example of the strong typing which the system enforces. The body of the function is the recursive expression in the body clause. The form of the body is prefix pure LISP with all expressions starting with a function name followed by the function arguments. For example, **f(x, y)** is written **(f x y)**. This includes the basic computational logic operator **if**, which is always written **(if condition then-expression else-expression)**. We will use this form for all function bodies and theorems in the rest of the paper, since we believe it makes the structure of the reasoning a bit clearer. The **result** line declares that the type of the result that **update** returns is a set of **alpha**.

A few comments on optional parameters are in order. In our syntax, a parameter whose first character is "&" is an extra parameter. An extra parameter stands for zero or more optional parameters. This is particularly useful in allowing arguments substituted for the function parameters **!p** and **!f** to be functions of more than one parameter. Extra parameters are necessary because the functions in our system must be pure functions. The body of a function cannot refer to any variable not in the function's parameter list. The reason for this is that the theorem prover relies on the fact that two textually equal function calls are always equal.

If we allowed "free" variables in the body of a function, then this property would not hold. For example, consider the simple function defined below.

```
(fun (name test)
     (params (x number))
     (body (plus x y))
     (result number))
```

Here the equality of the two calls (**test z**) and (**test z**) depends upon the value of the "global" variable y, the value of which may be different at the times of the two calls. Thus any data which is used in the body of a function must be an explicit parameter.

Consider the **update** function. Suppose that the function **!p** that tests if an item in the set r is to be updated depends upon some data not in the item alone (for example, the item is tested to see if it is a member of some totally independent set S). An **update** function without the extra parameter feature would be less than general, and this particular case couldn't be modelled by it, since the value of the set S could not be supplied to the **!p** function. So, rather than write n different **update** functions with 0, 1, 2 ...n-1 extra parameters we have chosen to use the **¶meter** notation to represent all these cases in one general function.

We need also to handle the case where **!f** and **!p** depend on completely separate extra data. For example, suppose **!f** needs set S and **!p** needs set **T**. In this case we call **update** with two extra parameters, S and **T**, and new forms of !f and !p, !f' and !p', e. g., (**update r !p' !f' S T**). The transformed functions, !p' and !f' are the following lambda expressions, which ignore the unwanted parameter.

```
!p' = (lambda (x S T) (!p x T))
!f' = (lambda (x S T) (!f x S))
```

Meta-lemmas are theorems about generic functions and have preconditions that involve the function parameters of the generic functions. Meta-lemmas have the following internal form, though they may be written in somewhat simpler form when entered into the system:

```
(implies MH (implies H (equal L R)))
```

where **MH** stands for meta-hypothesis and is a condition on the functional parameters in the generic functions contained in the meta-body, (**implies H (equal L R)**). A meta-body is to be used eventually as a rewrite lemma. A rewrite lemma has a hypothesis and a rewrite rule stored as an equation and used in a left to right manner (the left-hand side is to be replaced by the right-hand side). [4]

[4] Rules not in this form can always be recast this way. For example, (p x) becomes (**equal** (p x) **true**) and (**not** (q y)) becomes (**equal** (q y) **false**).

Meta-lemmas are proven by temporarily assuming the meta-hypothesis as an axiom, and then attempting to prove the meta-body as an ordinary theorem. A meta-lemma is only applicable to cases where a generic function has been given concrete function arguments that satisfy the meta-hypothesis. Whenever such cases arise, specific lemmas are generated from a meta-lemma and consist of the meta-body with its functional variables replaced by the concrete function argument names or lambda expressions. For example, a meta-lemma about **update**, **update-preserves-key**, is given below. Its meta-hypothesis describes properties of !p and !f, and the meta-body describes a property of **update** if the meta-hypothesis holds. **L** and **R** of the rewrite rule stored internally will be (**key** (**update** s !f !p &m) !c)) and **true**, respectively. The meta-lemma states that if !f does not change the column !c [5] in s on which a key constraint is declared, then **update** does not change the keyed property of s.

```
update-preserves-key: meta
(implies (equal (!c (!f x &m)) (!c x))
      (implies (key s !c) (key (update s !f !p &m) !c)))
```

3.2.1 Proof-time Use of Meta-lemmas

In the process of a safety proof, whenever a generic function term is encountered, its functional parameters are concrete functions. If a meta-lemma exists whose left-hand side unifies with the term, then a specific lemma could possibly be generated. In order to generate the specific lemma we must prove the meta-hypothesis with the concrete functions. For example, if the term, (**key** (**update** s f p &m) c)) where f, p and c are concrete functions, is encountered, it is now possible to generate a valid lemma applicable to this term if we can prove some properties of f and c. The properties that must be proven in order to use **update-preserves-key** are that (**equal** (c (f x &m)) (c x)). This is a trivial example of how we use meta-lemmas during the proofs of safety theorems. It represents an example of demand driven generation of specific knowledge. The next two sections discuss examples of anticipatory generation of specific lemmas.

3.2.2 Generic Function Definition Time Use of Meta-lemmas

Generic functions can be used to gather very general knowledge about classes of functions. For example, the set operations of membership, containment, existential and universal quantification, union, intersection, projection, sum, and selection all have common properties based on the fact that all look at each item in a set and accumulate a result. This commonality is captured by defining them all as specific

[5]Remember that column names in our theory stand for selector functions and thus a variable standing for a column is a function variable and its use here is as a generic function (**key**) argument.

instantiations of a generic mapping function. We define **s-map** (set map) recursively in terms of the set primitives **choose**, **rest**, a base object, and two generic functions, !f (the application function) and !acc (the accumulator function).

```
(fun (name s-map)
(generic-types alpha beta gamma &rest)
(params (s (set of alpha))
        (!f (function (cross alpha &rest) beta))
        (!acc (function (cross beta gamma) gamma))
        (base gamma)
        (&extra &rest))
(body (if (equal s emptyset)
          base
          (!acc (!f (choose s) &extra)
                (s-map (rest s) !f !acc base &extra))))
(result gamma))
```

Using this definition as a base, we define all the other functions we mentioned above. See Appendix 1 for the complete definitions and some inherited lemmas. We then prove meta-lemmas about **s-map** and selectively inherit them, instantiated and simplified, as theorems about the derivative functions. For example, consider the definition of **intersect**:

```
(fun (name intersect)
(generic-types alpha)
(params (r (set of alpha)) (s (set of alpha)))
(body (s-map r
             (lambda (x y) (if (mem x y) (insert x emptyset) emptyset))
             union
             emptyset
             s))
(result (set of alpha)))
```

Here the accumulating function is **union** and the base object is the **emptyset**. We add each element of **r** to the result if and only if the element is a member of s. Note the use of s in the position of the optional parameter **&extra** of **s-map**. The lambda expression in the place of the function parameter **!f** is a function of two variables, **x** which successively takes on the all the values in **r**, and **y** which is bound to s in the optional position.

Intersect inherits properties known about its generic parent, **s-map**. Consider the meta-lemma about **s-map**, **s-map-insert-expands**:

```
(implies (and (equal (!acc m (!acc n o)) (!acc n (!acc m o)))
```

```
        (equal (!acc m (!acc m n)) (!acc m n))))
    (equal (s-map (insert a s) !f !acc base &x)
           (!acc (!f a &x) (s-map s !f !acc base &x))))
```

When we define **intersect** in terms of **s-map** the meta-hypothesis is instantiated as:

```
(and (equal (union m (union n o)) (union n (union m o)))
     (equal (union m (union m n)) (union m n)))
```

Applying the lemmas which allow the arguments of nested unions to be rearranged in the required manner and state that **union** is idempotent (part of our basic theory) the meta-lemma hypothesis is easily proven. Thus the meta-body is applicable to **intersect**. In the meta-body if we open the functions **union** and apply the lemmas **union-insert-expands** (see Appendix 1) the meta-lemma body is specialized to:

```
(equal (intersect (insert a r) s)
       (if (member a s)
           (insert a (intersect r s))
           (intersect r s)))
```

which is a useful lemma about **intersect**. Note the introduction of the **member** predicate and the absence of **union** in this lemma. This form has been generated by use of the term rewriting system (applying the lemma **union-insert-expands**) during the inheritance process when the instantiated version of the theorem is simplified by using rewrite lemmas.

Each of the functions defined in terms of **s-map** (or in terms of a function derived from **s-map**, such as **for-all**) inherits the **s-map-insert-expands** lemma if its instantiating functions meet the meta-hypothesis, though the final forms of the instantiated theorem can be radically different due to the rewriting made possible by knowledge about the instantiating functional arguments. Thus, useful knowledge about the generic function s-map is inherited by the instantiated functions defined in terms of s-map. Appendix 1 contains a list of lemmas inherited from s-map.

3.2.3 Use of Meta-lemmas at Schema Translation Time

Whenever the term rewriting kernel rewrites a term, it chooses a rewrite lemma (if one exists) whose left-hand side unifies with the term to be rewritten. In order to avoid attempting to unify every term against the left-hand side of every rewrite rule, we index our rewrite rules by the outermost function in the left-hand term.

Because meta-lemmas allow function variables, it is possible to state a meta-lemma which cannot be indexed by a function name, since a function variable can be in the outermost function position. The only way to use such a lemma would be

to attempt to apply it to every term, which is computationaly infeasible. Consider the meta-lemma with no functional preconditions:

```
(implies true
      (implies (and (for-all r !p &a) (member x r))
            (!p x &a)))
```

This theorem could never be applied at proof time, since it is a theorem about **!p**, but **!p** is a function variable not a function name. We have no effective means of indexing this lemma. Instead, when a meta-lemma is added to the system, the user informs the system that the meta-lemma is "about" the higher order function (or functions) it contains, and at schema translation time, any expressions using this function (or functions) cause the instantiation of concrete lemmas. These lemmas are indexed by the concrete functions replacing the function variable. For example, the schema fragment,

```
For all p in Persons:
    p.Placed = (p.Pid in Placements.Pid)
```

translates to:

```
(for-all persons
      (lambda (x y)
            (equal (placed x)
                  (member (pid x) (project y pid))))
      placements)
```

Note that the predicate in the ADABTPL For all statement translates into the lambda expression. This corresponds to the function variable **!p** in the meta-lemma. The tuple variable, **p**, of the **For all** statement corresponds to the lambda bound variable **x**, and the **Placements** relation instantiates the optional variable **&a** of the meta-lemma and corresponds to the lambda bound variable **y**. At schema translation time, we anticipate the need for a particular lemma and use the concrete values supplied for **!p** and **&a** to generate the following lemma from the **for-all** meta-lemma above:

```
(implies (and (for-all persons
                  (lambda (x y)
                        (equal (placed x)
                        (member (pid x) (project y pid))))
                  placements)
            (member x persons))
      (equal (member (pid x) (project placements pid))
            (placed x)))
```

This lemma is indexed under member and will be available for use by the lemma choosing heuristics. This is an example of anticipatory generation of specific knowledge and works very well in keeping both the amount of knowledge stored and search time to acceptable levels.

4 Heuristics for Safety Proofs

We will now discuss the six heuristics we use to prove safety theorems. They are:

1. Transaction and database predicate evaluation.
2. Generalization and type inheritance.
3. Elimination of inertia.
4. Subgoal generation.
5. Meta-lemma instantiation.
6. Standard lemma application and function evaluation.

The first four heuristics are applied in the order listed. The last two heuristics are then applied to each subgoal alternately until the subgoal is proved or further application of these two heuristics makes no change in the subgoal. Subgoals that are left unproved are reported as unresolved and passed to the test generation front-end if the user desires. We now explain in detail each of our heuristics.

4.1 Transaction and Database Predicate Evaluation

A safety theorem has the form **(implies (P db) (P (T input db)))** where **T** is the transaction and **P** the database integrity predicate. A database is built up from its component relations using a database constructor function. The component relations of a database are accessed using database selector functions. The database predicate and the transaction are defined in terms of the database constructor and selector functions. In this heuristic we open up their definitions. Since both are usually large complex functions the resulting term is very large and complex. Fortunately, many of its subterms are of the form

```
(selector-i (constructor a1 a2 ... ai ... an))
```

which reduces to ai. To illustrate this, consider the partial schema and a transaction for a database named test.

```
Database test : [ R1 : type1; R2 : type2; R3 : type3 ]
        where Key(R1, column);
              Contains(R2.column2, R3.column3);
              For all r in R3 r.column3 = "tom";

Transaction T (i1 : tuple-type1; i2 : string);
Begin
Insert i1 into R1;
Remove x from R3 where x.column3 = i2
End;
```

These ADABTPL definitions generate the functional definitions below. Notice the use of the constructor function **test** and the selector functions **R1**, **R2**, **R3**. The result **test-type** is the database object's type. The predicate function, **thedbpred**, expresses the integrity constraints on a **test** database.

```
(fun (name T)
(params (i1 tuple1-type) (i2 string) (db test-type))
(body (test (insert i1 (R1 db))
            (R2 db)
            (rem (R3 db) (lambda (x i2) (equal (column3 x) i2)) i2)))
(result test-type))


(fun (name thedbpred)
(params (db test-type))
(body (and (for-all (R1 db) tuple1-type)
           (for-all (R2 db) tuple2-type)
           (for-all (R3 db) tuple3-type)
           (key (R1 db) column1)
           (Contains (project (R2 db) column2) (project (R3 db) column3))
           (for-all (R3 db) (lambda (r) (equal (column3 r) "tom")))))
(result boolean))
```

When we open up the definition of the database integrity predicate, **thedbpred**, and **T** in the safety theorem,

```
(implies (thedbpred db) (thedbpred (T input db)))
```

we get a very large term. One of its subterms is:

```
(for-all (R3 (test (insert i1 (R1 db))
                   (R2 db)
```

```
                    (rem (R3 db)
                         (lambda (x i2) (equal (column3 x) i2))
                         i2)))
          (lambda (r) (equal (column3 r) "tom")))
```

This subterm comes from applying the last predicate of the conjunction which makes up the body of **thedbpred** to the expanded body of **T**. Each predicate in the body of **thedbpred** creates a similiar subterm. Note though that the set parameters of the **for-all** predicate can be simplified to test only the **R3** portion of the database and that this subterm can be simplified to:

```
  (for-all (remove (R3 db)
                   (lambda (x i2) (equal (column3 x) i2))
                   i2)
         (lambda (r) (equal (column3 r) "tom")))
```

Recapping, our first heuristic is to open up the definitions of the database integrity predicate, and the transaction function. We then apply the constructor selector axioms for the database type. The application of other rewrite rules is deferred until later.

4.2 Generalization and Type Inheritance

After transaction and predicate evaluation we have a large term. This term has many subterms of the form **(selector-i db)**. In the previous example these terms are **(R1 db)**, **(R2 db)** and **(R3 db)**. At this point the usefulness of expressing the database as a single object, **db**, is over. We now generalize these subterms into variables. We know the type of these variables because we know the types of the database components. Since **db** satisfies the database integrity predicate, we know the **where** clauses of the database component types apply to the new variables. Generalizing the terms **(R1 db)**, **(R2 db)**, and **(R3 db)** into the variables **R1**, **R2**, **R3** we get:

```
(implies
  (and (for-all R1 tuple1-type)
       (for-all R2 tuple2-type)
       (for-all R3 tuple3-type)
       (key R1 column1)
       (contains (project R2 column2) (project R3 column3))
       (for-all R3 (lambda ($11) (equal (column3 $11) "tom"))))
  (and (for-all (insert i1 R1) tuple1-type)
       (for-all (remove R3 (lambda ($11 $12)
                             (equal (column3 $11) $12)) i2)
```

```
                tuple3-type)
        (key (insert i1 R1) column1)
        (contains (project R2 column2)
                  (project (remove R3
                     (lambda ($l1 $l2) (equal (column3 $l1) $l2)) i2)
                         column3))
        (for-all (remove R3 (lambda ($l1 $l2)
                                   (equal (column3 $l1) $l2)) i2)
                 (lambda ($l1) (equal (column3 $l1) "tom")))))
```

The generalization adds the following facts about the new variables generated from the where clauses of their type definitions, i. e., the new variables inherit their **where** clause type restrictions.

```
(for-all R1 tuple1-type)
(for-all R2 tuple2-type)
(for-all R3 tuple3-type)
```

where **tuple1-type** is a predicate defined by the schema translator to test the **where** clause of the **tuple1** type definition.

The purpose of this heuristic is to add new terms to the antecedents of the clauses we will prove. It is safe to do this since we know the old database met the integrity constraints, so its components must meet their type restrictions.

4.3 Elimination of Inertia

If the database integrity predicate contains a predicate in its body which involves only relations not changed by the transaction **T**, then this predicate survives unchanged in the consequent of the implication which makes up the safety proof. We call such terms *inertial terms*, or *inertia.* After generalization and inheritance, inertial terms are removed from the consequent since they also appear in the antecedent of the implication. We call this process the elimination of inertia. This heuristic removes goals that would be proven trivially by the remaining heuristics.

4.4 Subgoal Generation

Each remaining term in the consequent generates a subgoal to be proven. For each subgoal we assume the facts in the antecedent of the original formula along with all the facts generated in the generalization and type inheritance phase. Each of these goals is passed individually to the last two heuristics. If all of them can be proven the safety theorem is proven.

4.5 Transaction Proof Time Use of Meta-lemmas

Each subgoal represents the interaction of one part of the database integrity predicate with the parts of the database that have been changed by the transaction whose safety we are trying to prove. But these kinds of interactions are exactly the knowledge expressed by our meta-lemmas, in which the parameter functions were represented as function variables. In our subgoals we have actual functions (most often represented as lambda expressions). We search the subgoal for a subterm which matches the body of a meta-lemma. We then try to prove the meta-hypothesis with the actual functions replacing the function variables. Consider the subgoal that comes from the **Hire** transaction safety proof, which involves proving the keyed property of the **Offerings** relation after it is updated: [6]

```
(key (update3 offerings
              (lambda (l1 l2 l3)
                      (and (equal (cid l1) l2) (equal (jid l1) l3)))
              (lambda (l1 l2 l3)
                      (equal (number-of-positions l1) (add1 0)))
              (lambda (l1 l2 l3)
                      (offer (cid l1)
                             (jid l1)
                            (sub1 (number-of-positions l1))))
              comp jb)
     (lambda (l1)
              (cross (cid l1) (jid l1))))
```

The meta-lemma: **update3-preserves-keyness** matches this subgoal.

```
META-HYP:
(equal (!d (!g x &y)) (!d x))
META-BODY:
(implies (key s !d) (key (update3 s !q !f1 !g &y) !d))
```

In order to apply the meta-body of the meta-lemma we must prove the instantiated meta-hypothesis which is that the two terms listed below are equal:

```
(cross (cid ((lambda (l1 l2 l3) (offer (cid l1)
                                       (jid l1)
                                       (sub1 (number-of-positions l1)))
                  x comp jb))
```

[6] The higher order function **update3** is the update form described in section 2.3.5. Its effect is on the tuples in its first argument which meet the second argument (a predicate function). These tuples are deleted if they satisfy the third argument, or if they do not, they are modified by the fourth argument.

```
(jid ((lambda (l1 l2 l3) (offer (cid l1)
                                (jid l1)
                                (sub1 (number-of-positions l1))))
      x comp jb)))
```

```
(cross (cid x) (jid x))
```

This is easily done by applying the lambda expressions to their arguments and using the **jid-offer**, **cid-offer** constructor selector axioms. The body of the meta-lemma can now be applied to the subgoal provided the term,

```
(key offerings (lambda (l1) (cross (cid l1) (jid l1))))
```

can be proven. This term is one of the terms in the antecedent of the subgoal, since it represents the keyed property of the original offerings relation. Thus the subgoal is proven.

4.6 Standard Lemma Application and Function Evaluation

Most of the time, simply applying a meta-lemma to a subgoal is not sufficient to prove the subgoal. Many times there does not even exist a meta-lemma that matches the subgoal. If this is the case we apply our standard rewrite rules and function evaluation rules in the hope that these will either prove the subgoal or transform it to a form where a meta-lemma applies. In the job agency database we have an integrity constraint which states that no person is simultaneously in both the applications and applied relations. Since the **Hire** transaction changes both these relations the safety proof for **Hire** generates the following subgoal:

```
(null-intersection
  (project (remove applications
                   (lambda (l1 l1) (equal (pid l1) l2) hiree)
                     pid)
  (project (insert (placement hiree jb comp sal) placements)
                     pid))
```

No meta-lemmas apply to this subgoal. However, the following rewrite lemmas apply to subterms of the the subgoal.

```
pull-insert-out-of-project: rewrite
(equal (project (insert a x) !d) (insert (!d a) (project x !d)))

project-rem-to-delete-project: rewrite meta
(equal (project (remove a (lambda (l1 l2) (equal (!c1 l1) l2)) n) !c1)
       (delete n (project a !c1)))
```

Once these lemmas are applied, the subgoal becomes

```
(null-intersection (delete hiree (project applicatons pid))
              (insert (pid (placement hiree jb comp sal))
                    (project placements pid)))
```

and the following lemmas become applicable.

```
null-inter-insert-expands2: rewrite
(equal (null-intersection x (insert a y))
       (if (null-intersection x y) (not (member a x)) false))

null-inter-means-null-inter-delete: rewrite
(implies (null-intersection x y) (null-intersection x (delete a y)))
```

The second rewrite rule can be applied only if we can prove the premise:

```
(null-intersection (project applications pid) (project placements pid))
```

But this premise is in the antecedent of the subgoal since it represents the fact that the database met the integrity constraints before the transaction started. Thus the consequent of the subgoal rewrites to:

```
(not (member (pid (placement hiree jb comp sal))
            (delete hiree (project applications pid))))
```

Applying the lemmas **member-delete-rewrites**, and the constructor selector axiom **pid-placement** the subgoal is proven.

```
member-delete-rewrites: rewrite
(equal (member a (delete b x)) (if (equal a b) false (member a x)))
```

5 Amount of Theory Generation

The amount of tailored lemma generation in the safety verification process is indicated by the following numbers. The basic theory of sets and tuples that is made visible to the safety verifier is contained in fifty-five function definitions and two hundred and nine lemmas. Sixty-four of the lemmas are meta-lemmas. Compiling the job agency schema generates fifty-five new function definitions and one hundred and thirty-nine new lemmas. Thirty-two of the new lemmas are constructor-destructor lemmas for the different tuple types, and the bulk of the other lemmas are generated from meta-lemmas. Additional lemmas are generated from meta-lemmas during the safety proof as each clause is examined. These lemmas are not placed in the lemma base, and persist only during the processing of one clause.

6 Summary

We have demonstrated that mechanical theorem proving in higher order computational logic can be used effectively on the problem of proving that complex database transactions obey complex database constraints. To accomplish this, it was necessary to

- axiomatize a few abstract data types from which databases could be structured,
- build a basic theory combining the theories of these abstract data types,
- extend Boyer and Moore style theorem proving to higher order functions, treating functions (including predicates) as first class objects,
- build different control front-ends for proving basic theorems, transaction safety theorems, and theorems involving higher order functions and conditions on their functional inputs, and
- develop a general theory of database systems which has the right structure for guiding the safety verifier to effective safety proofs.

The system we have implemented allows database system designers to specify a database system using a more or less standard schema language along with a high level transaction programming language. A designer need know nothing about the internal logical form of the resulting system specification nor any details of proof procedures to use the system effectively. The schema and programming language is suitable as the source for translation into a lower level programming language and database management system.

The current system deals with a robust set of constraint constructs and transaction forms and can be extended with relative ease. Currently extensions can only be made by a system expert (a knowledge base engineer, if you will), not by database designers. Current research addresses the problem of database system designers extending the system's reasoning capabilities. A basic facility for generating run-time tests for unsafe transactions has been implemented, and an improved facility is under development. Other improvements to our system include allowing user-defined abstract data types in databases, semantic query optimization, and compilation of our high level language into target database management systems.

Our verification system is an expert system in which the expert knowledge is embedded in the heuristics of the provers, encoded procedurally in LISP code, and in the lemmas of the different levels of the system, encoded nonprocedurally (more or less) in computational logic lemmas. The interpreter of the knowledge is probably more sophisticated than is common in most expert system work, though

we do not claim to be familiar with the state of the art in expert systems. One of the differences between our non-procedural knowledge base, the lemmas, and that of most expert systems is that our base has been formally and mechanically verified in its entirety. The sophistication of our interpreter is only one of the similarities between our approach and that of Boyer and Moore, to whom we owe so much for inspiration as well as for concrete techniques, heuristics and guidance in knowledge base building. One of the differences between our work and theirs is our dependence on automatic generation of specific knowledge from generic knowledge. Perhaps the most important lesson that we have learned in our efforts is that building a theory for use in practical reasoning is a difficult and delicate operation, to be attempted only if sufficient patience for the long haul is possessed. This is most likely true even if the knowledge interpreter is less sophisticated than ours. In other words, writing programs to be interpreted by sophisticated knowledge oriented interpreters is even harder than writing programs for interpreters with simpler evaluation loops such as LISP interpreters, Prolog systems, Pascal machines, or 8086 processors. However, we believe the results which can be attained by combining heuristics with formally verified knowledge bases to be worth the effort and achievable, in many cases, in no other way.

Anyone claiming to have created an effective tool for program verification must face the scepticism created by the infamous "Social Processes and Proofs" paper of De Millo, Lipton and Perlis. [5] In their paper, the authors adeptly describe the process of establishing acceptance of a mathematical proof. They further use this description of proofs to disparage the production of verifiers that establish formal properties of programs, stating that "verifications cannot really be read", "evidence [indicates] that fully automatic verifying systems are out of the question," and that abstraction in the form of "creating more general structures" in programming is a "totally different notion of abstraction [from that] in mathematics." In this paper, we have argued and, we believe, demonstrated that the repeated process of introducing new functions, proving lemmas, closing the function definitions, and discarding some lemmas, is precisely the sort of abstraction that mathematicians perform in building mathematical theories and basically indistinguishable from abstraction in programming. In order to be useful and socially acceptable to system designers, a verification tool must work with abstractions that are useful cognitive chunks for the writer of the specifications. What we have shown, in one domain, is that the proper abstractions can both supply the leverage needed for efficient verification, in that they allow generally applicable theory to be prepared for verification tasks, and simultaneously provide the means of making proofs readable. We believe that De Millo et al. did not appreciate the form of abstraction that computational logic facilitates in building theory for use in verification. We hope that our approach to the verification of transaction safety shows that verification can be believable and useful in nontrivial domains.

7 Acknowledgements

This paper is based on work supported by the National Science Foundation under grants DCR-8503613 and IRI-8606424, and by the Office of Naval Research University Research Initiative contract number N00014-86-K-0764. We want to thank the anonymous reviewers for their constructive comments that helped us make this paper more accessible.

References

[1] Bernstein, P. A. and Blaustein, B. T. Fast Methods for Testing Quantified Relational Calculus Expressions. In *Proceedings ACM SIGMOD Conference* (1982), 39-50.

[2] Boyer, R. S. and Moore, J. S. *A Computational logic*, Academic Press, New York, 1979.

[3] Casanova, M. A. and Bernstein, P. A. Logic of a Relational Data Manipulation Language. In *Proceedings of the Sixth ACM Symposium on Principles of Programming Languages* (1979), 101-120.

[4] Casanova, M. A. and Bernstein, P. A. Formal System for Reasoning about Programs Accessing a Relational Database. *ACM Trans. Programming Lang. and Syst.* 2, 3 (Jul. 1980), 386-414.

[5] De Millo, R. A., Lipton, R. J. and Perlis, A. J. Social Processes and Proofs of Theorems and Programs. *Comm. of the ACM* 22, 5 (May 1979), 271-280.

[6] Gardarin, G. and Melkanoff, M. Proving the Consistency of Database Transactions. In *Proceedings of the 5th International Conference on Very Large Databases* (1979), 291-298.

[7] Guttag, J. Notes on Type Abstractions (Version 2). *IEEE Transactions on Software Engineering* 6, 1 (Jan. 1980), 13-23.

[8] Henschen, L. J., McCune, W. W., and Naqvi, S. A. Compiling Constraint-checking Programs from First-order Formulas. in *Advances in Database Theory*, Vol 2. Edited by Gallaire, Minker and Nicolas. Plenum Press, New York, 1984, 145-170.

[9] Hoare, C. A. An Axiomatic Basis for Computer Programming. *Comm. of the ACM* 12, 10 (Oct. 1969), 576-580.

[10] Hoare, C. A. Recursive Data Structures. *International Journal of Computer and Information Sciences* 4, 2 (June, 1975), 105-132.

[11] Hsu, T. and Imielinski, T. Integrity Checking for Multiple Updates. In *Proceedings ACM SIGMOD Conference* (1985), 152-168.

[12] Nicolas, J. M. Logic for Improving Integrity Checking in Relational Databases. *Acta Informatica* 18, 3 (Dec. 1982), 227-253.

[13] Oppen, D. C. Reasoning about Recursively Defined Data Structures. In *Proceedings of Fifth Symposium on Principles of Programming Languages* (1978), 151-157.

[14] Phillips, N. C. K. Safe Data Type Specifications. *IEEE Transactions on Software Engineering* 10, 3 (May 1984), 285-289.

[15] Sheard, T. and Stemple, D. Coping With Complexity In Automated Reasoning About Database Systems. In *Proccedings of the 11th International Conference on Very Large Databases* (1985), 426-435.

[16] Simon, E. and Valduriez, P. Design and Analysis of a Relational Integrity Subsystem. MCC Technical Report Number DB-015-87.

[17] Stemple, D. and Sheard, T. Specification and Verification of Abstract Database Types. In *Proceedings of the Third Symposium on Principles of Database Systems* (1984), 248-257.

[18] Stemple, D. and Sheard, T. Database Theory for Supporting Specification-based Database System Development. In *Proceedings of the Eighth International Software Engineering Conference* (1985), 43-49.

[19] Stemple, D., Sheard, T. and Bunker, R. Abstract Data Types in Databases: Specification, Manipulation and Access. In *Proceedings of the IEEE Second International Conference on Data Engineering* (1986), 590-597.

[20] Stemple, D., Mazumdar, S. and Sheard, T. On the Modes and Meaning of Feedback to Transaction Designers. In *Proceedings ACM SIGMOD Conference* (1987), 374-386.

[21] Walker, A. and Salveter, S.C. Automatic Modification of Transactions to Preserve Data Base Integrity Without Undoing Updates. State University of New York, Stony Brook, New York: Tech. Report 81/026 (June 1981).

8 Appendix 1

In this appendix we list the meta-lemma **s-map-insert-expands** and the specific lemmas generated from it by our definition time inheritance mechanism. In each case we give the definition of the child function in terms of **s-map**, and the lemma generated. After the inherited lemma's name we list how we use the lemma. Note some of the inherited lemmas are themselves meta-lemmas. Note also the different forms of the instantiations of the body of **s-map-insert-expands** for the different specializations of **s-map**. These forms are the result of using the term rewriting system and knowledge about the instantiating functional arguments during the inheritance process.

```
s-map-insert-expands: meta
(implies (and (equal (!acc m (!acc n o)) (!acc n (!acc m o)))
              (equal (!acc m (!acc m n)) (!acc m n)))
         (equal (s-map (insert a s) !f !acc base &x)
                (!acc (!f a &x) (s-map s !f !acc base &x))))

(fun (name for-some)
     (types alpha &beta)
     (params (x (set of alpha))
             (!p (function (cross alpha &beta) boolean))
             (&extra &beta))
     (body (s-map x !p or false &extra))
     (result boolean)
     (inherit meta))

for-some-insert-expands: rewrite meta
(equal (for-some (insert a x) !p &extra)
       (if (!p a &extra) true (for-some x !p &extra)))

(fun (name for-all)
     (types alpha &beta)
     (params (x (set of alpha))
             (!p (function (cross alpha &beta) boolean))
             (&extra &beta))
     (body (s-map x !p and true &extra))
     (result boolean)
     (inherit meta))

for-all-insert-expands: rewrite meta
```

```
(equal (for-all (insert a x) !p &extra)
       (if (!p a &extra) (for-all x !p &extra) false))

(fun (name mem)
     (types alpha)
     (params (a alpha) (x (set of alpha)))
     (body (for-some x equal a))
     (result boolean))

mem-insert-expands: rewrite
(equal (mem a (insert a1 x)) (if (equal a1 a) true (mem a x)))

(fun (name contains)
     (types set-elem)
     (params (r (set of set-elem)) (s (set of set-elem)))
     (body (for-all s mem r))
     (result boolean))

contains-insert-expands: rewrite
(equal (contains r (insert a s)) (if (mem a r) (contains r s) false))

(fun (name union)
     (types set-elem)
     (params (r (set of set-elem)) (s (set of set-elem)))
     (body (s-map r identity insert s))
     (result (set of set-elem)))

union-insert-expands: rewrite
(equal (union (insert a r) s) (insert a (union r s)))

(fun (name select)
     (types alpha)
     (params (r (set of alpha)) (!p (function alpha boolean)))
     (body (s-map r (lambda (x) (if (!p x) (insert x emptyset) emptyset))
                 union emptyset))
     (result (set of alpha)))

select-insert-expands: rewrite
(equal (select (insert a r) !p)
       (if (!p a) (insert a (select r !p)) (select r !p)))
```

```
(fun (name intersect)
     (types alpha)
     (params (r (set of alpha)) (s (set of alpha)))
     (body (s-map r (lambda (x y)
                      (if (mem x y) (insert x emptyset) emptyset))
                    union
                    emptyset
                    s))
     (result (set of alpha)))
```

```
intersect-insert-expands: rewrite
(equal (intersect (insert a r) s)
       (if (mem a s) (insert a (intersect r s)) (intersect r s)))
```

```
(fun (name project)
     (types set-elem new-set-elem)
     (params (r (set of set-elem))
             (!p (function set-elem new-set-elem)))
     (body (s-map r !p insert emptyset))
     (result (set of new-set-elem)))
```

```
project-insert-expands: rewrite
(equal (project (insert a r) !p) (insert (!p a) (project r !p)))
```

9 Appendix 2

In this appendix we list parts of the proof of the safety theorem for the **Fire** transaction as annotated by the transaction safety verifier. We use a verbose mode for the first subgoal to show the use of meta-lemmas and a terse mode for the other subgoals to save space.

```
We will attempt to prove the conjecture:

   (implies (thedbpred db) (thedbpred (fire emp comp jb sal db)))

Opening the functions fire, thedbpred and applying the constructor-
destructor lemmas placements-job-agency, companies-job-agency,
offerings-job-agency, applications-job-agency, requirements-job-agency,
abilities-job-agency, skills-job-agency, jobs-job-agency, persons-job-
agency then generalizing the terms
(persons db), (jobs db), (skills db), (abilities db), (requirements db),
(applications db), (offerings db), (companies db), (placements db)
into the variables
persons, jobs, skills, abilities, requirements, applications, offerings,
companies, placements, we get a formula whose antecedent is:

(and (key persons pid)
     (for-all persons person-type)
     (key jobs jid)
     (for-all jobs job-type)
     (key skills skill-id)
     (for-all skills skill-type)
     (for-all abilities ability-type)
     (for-all requirements requirement-type)
     (for-all applications application-type)
     (key offerings (lambda ($11) (cross (cid $11) (jid $11))))
     (for-all offerings offer-type)
     (key companies cid)
     (key companies company-name)
     (for-all companies company-type)
     (key placements pid)
     (for-all placements placement-type)
     (contains (project persons pid) (project placements pid))
     (contains (project companies cid) (project placements cid))
     (contains (project jobs jid) (project placements jid))
```

```
(contains (project jobs jid) (project offerings jid))
(contains (project companies cid) (project offerings cid))
(contains (project persons pid) (project abilities pid))
(contains (project persons pid) (project applications pid))
(contains (project jobs jid) (project applications jid))
(contains (project jobs jid) (project requirements jid))
(redun persons
       placed
       (lambda ($11 $12) (mem (pid $11) (project $12 pid)))
      placements)
(for-all applications
      (lambda ($11 $12 $13)
              (contains (project (select $12
                                        (lambda ($11 $12)
                                                (equal (pid $11) $12))
                                        (pid $11))
                                 skill-id)
                        (project (select $13
                                        (lambda ($11 $12)
                                                (equal (jid $11) $12))
                                        (jid $11))
                                 skill-id)))
      abilities
      requirements)
(null-intersection (project applications pid)
                   (project placements pid))
(redun companies
      totsal
      (lambda ($11 $12)
              (sum (select $12
                           (lambda ($11 $12)
                                   (equal (cid $11) $12))
                           (cid $11))
                     salary))
      placements)
      (mem (placement emp jb comp sal) placements))
```

The generalization adds the following facts about the new variables generated from the where clauses of their type definitions:

```
(for-all persons person-type)
```

```
(key persons pid)
(for-all jobs job-type)
(key jobs jid)
(for-all skills skill-type)
(key skills skill-id)
(for-all abilities ability-type)
(for-all requirements requirement-type)
(for-all applications application-type)
(for-all offerings offer-type)
(key offerings (lambda ($11) (cross (cid $11) (jid $11))))
(for-all companies company-type)
(key companies company-name)
(key companies cid)
(for-all placements placement-type)
(key placements pid)
```

The consequent of the formula has 16 subgoals. The functions delete, rest are disabled. No lemmas are disabled.

****

The consequent for the 1st subgoal is:

```
(key (update persons
             (lambda ($11 $12)
                     (person (pid $11) (pname $11) false))

             (lambda ($11 $12)
                     (equal (pid $11) $12))

             emp)
     pid)
```

We will attempt to use the meta-lemma: update-preserves-key

```
HYP:
(equal (*!c (*!f x &m)) (*!c x))
BODY:
(implies (key *s *!c) (key (update *s *!f *!p &m) *!c))
```

Applying the lemmas pid-person the (instantiated) meta hypothesis

```
(equal (pid (person (pid x) (pname x) false)) (pid x))
```

of the meta-lemma update-preserves-key is proven. Applying the lemmas update-preserves-key

```
update-preserves-key: meta
(implies (equal (!c (!f x &m)) (!c x))
         (implies (key s !c) (key (update s !f !p &m) !c)))
```

and referring to term 1 of the antecedent

```
1
(key persons pid)
```

The consequent is proven.

****
The consequent for the 2nd subgoal is:

```
(for-all (update persons
                 (lambda ($l1 $l2)
                         (person (pid $l1) (pname $l1) false))
                 (lambda ($l1 $l2)
                         (equal (pid $l1) $l2))
                 emp)
         person-type)
```

We will attempt to use the meta-lemma: for-all-update
Applying the lemmas person-type the (instantiated) meta hypothesis of the meta-lemma for-all-update is proven. Applying the lemmas for-all-update and referring to term 2 of the antecedent the consequent is proven.

****
The consequent for the 3rd subgoal is:

```
(key (update companies
             (lambda ($l1 $l2 $l3)
                     (company (cid $l1)
                              (company-name $l1)
                              (difference (totsal $l1) $l3)))
```

```
            (lambda ($l1 $l2 $l3)
                    (equal (cid $l1) $l2))
            comp
            sal)
    cid)
```

We will attempt to use the meta-lemma: update-preserves-key
Applying the lemmas cid-company the (instantiated) meta hypothesis of the meta-lemma update-preserves-key is proven. Applying the lemmas update-preserves-key and referring to term 12 of the antecedent the consequent is proven.

.

.

.

****

The consequent for the 6th subgoal is:

```
(key (rem placements (lambda ($l1 $l2)
                             (equal (pid $l1) $l2))
                     emp) pid)
```

Applying the lemmas key-means-key-delete, key-means-rem-rewrites, pid-placement and referring to the terms 15, 30 of the antecedent the consequent is proven.

****

The consequent for the 7th subgoal is:

```
(for-all (rem placements (lambda ($l1 $l2)
                                 (equal (pid $l1) $l2))
                         emp)
         placement-type)
```

We will attempt to use the meta-lemma: for-all-rem
Applying the lemmas for-all/placement-rel and referring to term 16 of the antecedent the (instantiated) meta hypothesis of the meta-lemma for-all-rem is proven. Applying the lemmas for-all-rem and referring to term 16 of the antecedent the consequent is proven.

****

```
The consequent for the 8th subgoal is:

(contains (project (update persons
                           (lambda ($11 $12)
                                   (person (pid $11) (pname $11) false))
                           (lambda ($11 $12)
                                   (equal (pid $11) $12))
                           emp)
                   pid)
          (project (rem placements
                        (lambda ($11 $12)
                                (equal (pid $11) $12))
                        emp)
                   pid))

We will attempt to use the meta-lemma: project-update-simplifies
Applying the lemmas pid-person the (instantiated) meta hypothesis of the
meta-lemma project-update-simplifies is proven. Applying the lemmas
contains-delete-small-set, pull-delete-from-project,
key-means-rem-rewrites, pid-placement, project-update-simplifies and
referring to the terms 15, 17, 30 of the antecedent the consequent is
proven.
.
.
.
*************************************************************
The consequent for the 14th subgoal is:

(redun (update persons
               (lambda ($11 $12)
                       (person (pid $11) (pname $11) false))
               (lambda ($11 $12)
                       (equal (pid $11) $12))
               emp)
       placed
       (lambda ($11 $12)
               (mem (pid $11) (project $12 pid)))
       (rem placements (lambda ($11 $12)
                               (equal (pid $11) $12))
                       emp))
```

We will attempt to use the meta-lemma: redun-update-rem
Applying the lemmas mem-project-placement-on-pid, pull-delete-from-project, key-means-rem-rewrites, pid-placement, placed-person, pid-person, mem-delete-rewrites, project-rem-to-delete-project and referring to the terms 15, 30 of the antecedent the (instantiated) meta hypothesis of the meta-lemma redun-update-rem is proven. Applying the lemmas redun-update-rem and referring to term 26 of the antecedent the consequent is proven.

****
The consequent for the 15th subgoal is:

```
(null-intersection (project applications pid)
                   (project (rem placements
                                 (lambda ($11 $12)
                                         (equal (pid $11) $12))
                                 emp)
                            pid))
```

Applying the lemmas null-inter-means-null-inter-delete, pull-delete-from-project, key-means-rem-rewrites, pid-placement and referring to the terms 15, 28, 30 of the antecedent the consequent is proven.

****
The consequent for the 16th subgoal is:

```
(redun (update companies
               (lambda ($11 $12 $13)
                       (company (cid $11)
                                (company-name $11)
                                (difference (totsal $11) $13)))
               (lambda ($11 $12 $13)
                       (equal (cid $11) $12))
               comp
               sal)
       totsal
       (lambda ($11 $12)
               (sum (select $12
                            (lambda ($11 $12)
                                    (equal (cid $11) (cid $12)))
```

```
                    $11)
              salary))
    (rem placements (lambda ($11 $12)
                            (equal (pid $11) $12))
                emp))
```

We will attempt to use the meta-lemma: redun-update-rem
Applying the lemmas salary-placement, sum-delete-expands,
mem-select-expands, select-delete-expands, cid-placement, cid-company,
redun-means/sum3, key-means-rem-rewrites, pid-placement,
totsal-company, select-lambda-special and referring to the terms 15,
29, 30 of the antecedent the (instantiated) meta hypothesis of the
meta-lemma redun-update-rem is proven. Applying the lemmas
redun-update-rem, select-lambda-special and referring to term 29 of the
antecedent the consequent is proven.

The Safety Theorem is Proven!

The following are the meta-lemmas referred to in the proof.

```
update-preserves-key: meta
(implies (equal (!c (!f x &m)) (!c x))
         (implies (key s !c) (key (update s !f !p &m) !c)))

for-all-update: meta
(implies (implies (!p x &rest1) (!q (!f x &rest1) &rest2))
         (implies (for-all r !q &rest2)
                  (for-all (update r !f !p &rest1) !q &rest2)))

for-all-rem: meta
(implies (if (mem x *r) (if (*!c x &n) true (*!q x &rest2)) true)
         (implies (for-all *r *!q &rest2)
                  (for-all (rem *r *!c &n) *!q &rest2)))

project-update-simplifies: meta
(implies (equal (!d (!f x &m)) (!d x))
         (equal (project (update r !f !p &m) !d) (project r !d)))

redun-update-rem: meta
(implies (and (implies (and (mem x r1) (not (!p x &b)))
                       (equal (!q x (rem r2 !s &a)) (!q x r2)))
              (implies (and (mem x r1) (!p x &b) (equal (!f x) (!q x r2))
```

```
                (equal (!q (!t x &b) (rem r2 !s &a))
                       (!f (!t x &b)))))
(implies (redun r1 !f !q r2)
         (redun (update r1 !t !p &b) !f !q (rem r2 !s &a))))
```