Relational Modeling

Somesh Jha
Computer Science Department
University of Wisconsin
Madison, WI 53703

1 Introduction

Model checking is appropriate for verifying concurrency aspects of a system , e.g., a distributed system never admits a deadlock. However, model checking is not appropriate for expressing invariants about data structures, e.g., a list is always sorted. For talking about data structures relational calculus is much more suitable formalism. First, we will discuss a language (called *Nitpick*) based on relational calculus. *Ladybug* is a tool that checks specifications written in Nitpick.

2 Operators and Semantics

There are certain types in our relational calculus language. These types signify a domain of a certain kind. For example, `People` is the type that denotes the set of people. Each type has a domain associated with it. We will use overloading and use the name of the type to also denote the domain associated with it. For example, `People` also denotes the set of people. It will be clear from the context whether we are talking about types or the associated domain. There are four basic types of entities in our relational language:

- **Scalars** of type S simply takes value in the domain associated with type S.
- **Sets** of types S take values in the power set of the domain associated with S.
- A **relation** allows us to talk about relationships between different types. A relation between type S to type T will have the type $S \leftrightarrow T$.
- A **function** from type S to type T will be written as $S \rightarrow T$. Function is a special case of relation. A relation R of type $S \leftrightarrow T$ is a function iff every element in S has *at most* one element related to it.

We will use a running example to explain various operations in *Nitpick*. In our example there are four basic types `People`, `Males`, `Females`, and $\aleph$. $\aleph$ is the set of natural numbers. Notice that certain relationships hold between domains associated with types. For example, every male is a person, so `Males` is a subset of `People` or `Males` $\subseteq$ `People` (notice that we are talking about domain associated with types rather than types). The relations are shown in Figure 2. Frequently, we

Name	Type
`Father`	`People` $\rightarrow$ `Males`
`Mother`	`People` $\rightarrow$ `Females`
`Wife`	`Males` $\leftrightarrow$ `Females`
`Husband`	`Females` $\leftrightarrow$ `Males`
`Age`	`People` $\rightarrow \aleph$
`Brother`	`People` $\leftrightarrow$ `Males`
`Sister`	`People` $\leftrightarrow$ `Females`
`Friend`	`People` $\leftrightarrow$ `People`

Figure 1: Primitive relations and functions.

will depict a relation of type $S \leftrightarrow T$ by drawing two columns, where the left column corresponds to S and the right column corresponds to T, and an edge between element a and b represents that (a, b) is in the relation. For example, Figure 2 shows a fragment of the relation `Brother`. If (a, b) is an edge in a relation P, we say that a and b are P-related.

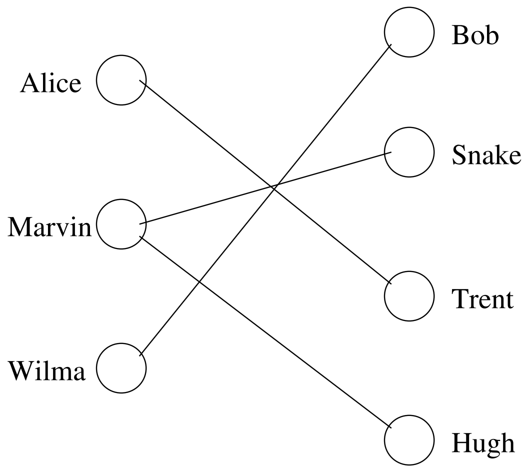

Figure 2: A sample relation

`Note:` We describe various operators in our relational calculus. The name of the symbol in *Ladybug* (the tool we are going to use) is also provided.

2.1 Set Operators

Each relation in the operators given below have the same type, i.e., each relation given below has type $S \leftrightarrow T$. All the operators given below also apply to sets.

- **Union** ($P \cup Q$)
  `Ladybug: U`
  $\{(a, b) | (a, b) \in P \vee (a, b) \in Q\}$

Two elements a and b are $(P \cup Q)$-related iff a and b are P-related *or* Q-related. We might define a relation `Parent` as

$$\texttt{Parent} = \texttt{Father} \cup \texttt{Mother}$$

Recall that the type of `Father` is `People` $\rightarrow$ `Males` and the type of `Mother` is `People` $\rightarrow$ `Females`. In order to perform the union both relations have to be interpreted as of type `People` $\leftrightarrow$ `People`. Fortunately, this can be done because *every function is a relation*, `Males` $\subseteq$ `People`, and `Females` $\subseteq$ `People`.

- **Intersection** $(P \cap Q)$
  `Ladybug: &`
  $\{(a,b)|(a,b) \in P \wedge (a,b) \in Q\}$
  Two elements a and b are $(P \cap Q)$-related iff a and b are P-related *and* Q-related. We might define a relation `FriendlyBrother` as

$$\texttt{FriendlyBrother} = \texttt{Brother} \cap \texttt{Friend}$$

- **Difference** $(P \setminus Q)$
  $\{(a,b)|(a,b) \in P \wedge (a,b) \notin Q\}$
  Two elements a and b are $(P \setminus Q)$-related iff a and b are P-related *and not* Q-related. We might define a relation `unFriendlyBrother` as

$$\texttt{unFriendlyBrother} = \texttt{Brother} \setminus \texttt{Friends}$$

- **Subset** $(P \subseteq Q)$
  `Ladybug: <=`
  $\forall(a,b)((a,b) \in P \Rightarrow (a,b) \in Q)$
  Notice that this is a logical formula, i.e, evaluates to **true** or **false**. The formula states that for all elements a and b (of appropriate type of course), if a and b are P-related, then they are Q-related. For example, we have the following logical-formula

$$\texttt{FriendlyBrother} \subseteq \texttt{Brother}$$

- **Proper Subset** $(P \subset Q)$
  `Ladubug: <`
  $(P \subseteq Q) \wedge (P \neq Q)$
  The logical formula states that if a and b are P-related, then they are Q-related, but there exists two elements a and b such that a and b are Q-related but not P-related. If there exists a brother who is unfriendly with one of his siblings, we have

$$\texttt{FriendlyBrother} \subset \texttt{Brother}$$

2.2 Exclusive set operators

- $x \in s$
  x : *scalar* S and s : *Set* S
  `Ladybug: in` or `:`
  The formula is true if x is in the set s.

- $x \notin s$
  x : *scalar* S and s : *Set* S
  `Ladybug: not in` or `!:`
  The formula is true if x is not in the set s

2.3 Relational Operators

- **Universal Relation** (Un)
  `Ladybug: Un`
  $Un : S \leftrightarrow T$
  $Un \;=\; S \times T$
  $S \times T$ is the Cartesian product and has all tuples (a, b) where $a \in S$ and $b \in T$.

- **Identity Relation** (Id)
  `Ladybug: Id`
  $Id : S \leftrightarrow S$
  $Id \;=\; \{(a, a) | a \in S\}$
  Identity relation only relates same elements.

- **Domain** (*DomP*)
  `Ladybug: dom`
  $P : S \leftrightarrow T$
  $\{a | (a, b) \in P\}$
  The set of people that have parents (denoted as `not-orphans`) is given by the following equation:

  $$\texttt{not-orphans} \;=\; \textit{Dom}(\texttt{Parent})$$

- **Range** (*range* P)
  `Ladybug: ran`
  $P : S \leftrightarrow T$
  $\{b | (a, b) \in P\}$
  Let `Parents` be the set of all people that are parents. `Parents` is given by the following set:

  $$\texttt{Parents} \;=\; \textit{range}\,(\texttt{Parent})$$

- **Domain Restriction** ($s \lhd P$)
  `Ladybug: <:`

$s : \textit{set}\ \ S, P : S \to T$
$\{(a, b) \in P | a \in S\}$
The domain of P is restricted to be in the set s. Example will be given in the next item.

- **Range Restriction** ($s \rhd P$)
  `Ladybug: >:`
  $s : \textit{set}\ \ T, P : S \to T$
  $\{(a, b) \in P | b \in T\}$
  Range of P is restricted to be in the set s.
  Let $[20]$ denote the set of numbers $\{1, 2, \cdots, 20\}$. We have the following equation:

$$\texttt{young} \quad = \quad \textit{Dom}(\texttt{Age} \rhd [20])$$

  Based on the set `young` we have the following derived sets:

$$\begin{aligned} \texttt{youngWives} &= \textit{Dom}(\texttt{young} \lhd \texttt{Husband}) \\ \texttt{youngHusbands} &= \textit{range}\ (\texttt{young} \rhd \texttt{Husband}) \end{aligned}$$

- **Negative Domain Restriction** ($s \unlhd P$)
  `Ladybug: <;`
  $s : \textit{set}\ \ S, P : S \to T$
  $\{(a, b) \in P \mid a \notin S\}$
  Force the domain to be not in the set s
  We have the following equation:

$$\texttt{oldWives} \quad = \quad \textit{Dom}(\texttt{young} \unlhd \texttt{Husband})$$

- **Negative Range Restriction** ($s \unrhd P$)
  `Ladybug: >;`
  $s : \textit{set}\ \ T, P : S \to T$
  $\{(a, b) \in P | b \notin S\}$
  Force the range to be not in the set s.
  The set of people whose age is above 20 is given by the following equation:

$$\texttt{old} \quad = \quad \textit{Dom}(\texttt{Age} \unrhd [20])$$

- **Relational Override** ($P \oplus Q$)
  `Ladybug: (+)`
  $P, Q : S \leftrightarrow T$
  $\{(\textit{Dom}(Q) \unrhd P) \cup Q\}$
  Here is how relational override works: if $a \in S$ is in the domain of Q, then it is related to all the elements that it is Q-related to. Elements that are not in the domain of Q are related to all the elements that it is P-related to. Succinctly speaking, first look in Q and then in P.

- **Composition** $(P; Q)$
  `Ladybug: ;`
  $P : S \leftrightarrow T, Q : T \leftrightarrow U$
  $\{(a, c) | \exists b((a, b) \in P \wedge (b, c) \in Q)\}$
  We have the following equation:

$$\mathit{Dom}(\mathtt{Wife}; \mathtt{Husband}) \;=\; \mathtt{MarriedMales}$$

- **Transpose** $(P^\top)$
  `Ladybug:` Tilde character.
  $P : S \leftrightarrow T$
  $\{(b, a) | (a, b) \in P\}$

$$\mathtt{Husband}^\top \;=\; \mathtt{Wife}$$

- **Transitive Closure** (P^+)
  $P : S \leftrightarrow S$
  `Ladybug:` $+$
  $P^+ \;=\; \bigcup_{i=1}^{\infty} P^i$
  P^i is relation P composed with itself i times.

$$\mathtt{FriendOfFriend} \;=\; \mathtt{Friend}^+$$

- **Reflexive/Transitive Closure** $(P^\star)$
  $P : S \leftrightarrow S$
  `Ladybug:` $\star$
  $P^\star \;=\; Id \cup P^+$
  Everybody is their own friend.

$$\mathtt{FriendofFriend}_1 \;=\; \mathtt{Friend}^\star$$

- **Application** $(P \cdot x)$
  $P : S \leftrightarrow T, x : S$
  single y *such that* $(x, y) \in P$

$$\mathtt{Father.Bob} \quad \text{(Father of Bob)}$$

- **Image** $(P \cdot S)$
  $P : S \leftrightarrow T, s : \mathit{set}\ S$
  range $(s \lhd P)$

$$\mathtt{Father.}\{\mathtt{Bob}, \mathtt{Alice}\} \quad \text{(Fathers of Bob and Alice)}$$

- **Functional Domain** (*fdom* (P))
  $P : S \leftrightarrow T$
  `Ladybug: fdom`
  $\{a||P.a| = 1\}$
  Find elements in domain of P that are related to exactly one element.

$$\begin{aligned} \texttt{siblings} &= \texttt{Brother} \cup \texttt{Sister} \\ \texttt{singleSibling} &= \textit{fdom}\,(\texttt{siblings}) \end{aligned}$$

- **Function Predicate** (*fun* (P))
  $P : S \leftrightarrow T$
  `Ladybug: fun`
  $\forall a \in S(|P.a| \leq 1)$
  The predicate *fun* returns true iff relation P is a function, i.e., every element of S is P-related to at most one element of T.

- **Injection Predicate** (*inj* (P))
  $P : S \leftrightarrow T$
  The predicate given above is true iff the transpose of P is a function.

- **Surjection Predicate** (*sur* (P))
  $P : S \leftrightarrow T$
  range $(P) = T$
  For every element $b \in T$ there exists at least one element $a \in S$ such that $(a, b) \in P$ or a and b are P-related.

- **Totality Predicate** (*tot* (P))
  $P : S \leftrightarrow T$
  DomP $= S$
  The predicate given above is true iff for every element $a \in S$ there is at least one element $b \in T$ such that a and b are P-related.

- **Singleton Predicate** (*one* (s))
  s : *set* S
  $|s| \leq 1$
  This predicate is true iff set s has at most *one* element.

3 Nitpick or Ladybug

We will be using Ladybug-an improved version of Nitpick written in JAVA. Ladybug has been developed by Craig Damon. First, we will describe a small example. Suppose we have two types `Phones` and `Numbers`. Imagine that we have a relation

$$\texttt{Called} : \texttt{Phones} \leftrightarrow \texttt{Numbers}$$

and a function

$$\texttt{Net} : \texttt{Numbers} \rightarrow \texttt{Phones}$$

If a has called a number n, we have $(a, n) \in \texttt{Called}$. $\texttt{Net}(n)$ is the phone associated with the number n. The relation Connection (with type $\texttt{Phones} \leftrightarrow \texttt{Numbers}$) has the semantics that if $(a, b) \in \texttt{Connection}$, then a has called b. We have the following relationship:

$$\texttt{Connections} \quad = \quad \texttt{Called}; \texttt{Net}$$

Suppose a phone p wants to call a number n. We allow this new connection to be made if n is not already being called. The operation $\texttt{Join}(p, n)$ can be written as:

$$n \notin \textit{range}\,(\texttt{Called}) \Rightarrow (\texttt{Called}' = \texttt{Called} \cup \{(p, n)\}$$

Think of $\texttt{Called}'$ as the relation Called in the next state[1]. For example, assuming that Net is a constant function (the value of Net never changes) the value of Connection in the next state is given by the following equation:

$$\texttt{Connection}' \quad = \quad \texttt{Called}'; \texttt{Net}$$

We want to make sure that no phone is calling itself and that the transpose of the relation Connection is a function (or a phone can receive only one call). These properties are listed below:

$$\begin{aligned} \texttt{invB} \quad &= \quad (\textit{Dom}(\texttt{Connection}) \cap \textit{range}\,(\texttt{Connection}) = \emptyset) \\ \texttt{invC} \quad &= \quad \textit{fun}\,(\texttt{Connection}^{\top}) \end{aligned}$$

We want to make sure that if the invariants given above are true before a Join operation they are true after the operation. This can be written as:

$$\begin{aligned} \texttt{Join}(p, n) \wedge \texttt{invB} \Rightarrow \texttt{invB}' \\ \texttt{Join}(p, n) \wedge \texttt{invC} \Rightarrow \texttt{invC}' \end{aligned}$$

Exercise 1 The expression `invB'` is a short hand for the expression corresponding to `invB` where we use `Connection'` instead of `Connection`. Expand the expression for `invB` in terms of `Called`, `Net`, and the definition of `Join(p,n)`.

3.1 Ladybug

In *ladybug* the example given before looks as follows.

```
[Ph, Num]

Switch = [
   Called: Ph <-> Num
```

[1]Denoting the value of the entity in the next state by *prime* is pretty common in formal methods.

```
   const Net: Num -> Ph
   Conns: Ph <-> Ph
|
   Conns = Called ; Net
]

Join  (p: Ph; n: Num) =  [
   Switch
|
   not (n in ran Called)
   Called' = Called U {p -> n}
]

invB =  [Switch | dom Conns & ran Conns  = {}]
invC = [Switch | fun (Conns~)]

InvB_preserved (p: Ph; n: Num)  :: (Join(p,n) and invB) =>invB'
InvC_preserved (p: Ph; n: Num)  :: (Join(p,n) and invC) =>invC'
```

`Note:` Both the claims (`invB` and `invC`) are not true. *Ladybug* produces a counter-example. The log file produced by *ladybug* is shown below.

```
Welcome to Ladybug 0.8 (beta release), Copyright 1998


Loaded phone-modified.np

Select a claim or schema and choose 'Check' from the menu
    or double click on a claim or schema

Completed translation of InvB_preserved
Required 0:00:00.6 starting at 4:38:42 PM

Found Counterexample to Claim InvB_preserved:
Called : Ph<->Num =
{  }
Called' : Ph<->Num =
{ p0 -> {n0 } }
Conns : Ph<->Ph =
{  }
Conns' : Ph<->Ph =
{ p0 -> {p0 } }
n : Num =
```

```
n0
Net : Num->Ph =
{ n0 -> p0 }
p : Ph =
p0

Found 1 Counterexamples
Checked 2 cases and 5 values
Covered 0% of the total assignment space
Required 0:00:00.0 starting at 4:38:42 PM
Completed translation of InvC_preserved
Required 0:00:00.3 starting at 4:39:01 PM

Found Counterexample to Claim InvC_preserved:
Called : Ph<->Num =
{ p0 -> {n1 } }
Called' : Ph<->Num =
{ p0 -> {n1 },
 p1 -> {n0 } }
Conns : Ph<->Ph =
{ p0 -> {p0 } }
Conns' : Ph<->Ph =
{ p0 -> {p0 },
 p1 -> {p0 } }
n : Num =
n0
Net : Num->Ph =
{ n0 -> p0,
 n1 -> p0 }
p : Ph =
p1

Found 1 Counterexamples
Checked 68 cases and 75 values
Covered 1% of the total assignment space
Required 0:00:00.0 starting at 4:39:01 PM
```

Exercise 2 Explain the counter-examples produced by *ladybug*. Fix the specification and explain your fix.

4 Further Properties

This section discusses further properties of relational operators. These properties will be useful while writing or simplifying specifications.

- *Reflexive*
  A relation $\mathtt{R} : S \leftrightarrow S$ is called reflexive if for every element a ($(a, a) \in \mathtt{R}$), i.e., every element is related to itself.

- *Transitive*
  A relation $\mathtt{R} : S \leftrightarrow S$ is called transitive if $(a, b) \in \mathtt{R}$ and $(b, c) \in \mathtt{R}$ implies that $(a, c) \in \mathtt{R}$.

- *Symmetric*
  A relation $\mathtt{R} : S \leftrightarrow S$ is called symmetric if $(a, b) \in \mathtt{R}$ implies that $(b, a) \in \mathtt{R}$.

Exercise 3 Prove the following properties of relational operators:

- *Associativity*

$$P; (Q; R) \;=\; (P; Q) : R$$

- *Monotonicity*

$$(P \subseteq Q \wedge R \subseteq S) \Rightarrow (P; R) \subseteq (Q; S)$$

- *Distributive*

$$P; (Q \cup R) = (P; Q) \cup (P; R)$$

- *Commutativity of transpose and closure*

$$P^{\star\top} \;=\; P^{\top\star}$$

Next we describe a small example in *ladybug* which illustrates various features that are idiosyncratic to the modeling language used by *ladybug*.

5 Types and Relations

We have two basic types `NAME` and `DATE`. There is one basic function `book` : `NAME` $\rightarrow$ `DATE`. There is a derived set `known` : *Set* `NAME` that satisfies the following equation:

$$\mathtt{known} \;=\; \mathit{Dom}(\mathtt{book})$$

Writing all the state variables and specifications explicitly is cumbersome. *Ladybug* has a macro feature (called *schemas*) which allows us to bundle declarations and assertions at one place. Schemas can be referred to in other schemas and assertions. No recursion is allowed. For example, the *ladybug* fragment for the types and relation defined above are:

```
[NAME, DATE]

Book = [
  book: NAME -> DATE
  known: set NAME
|
  known = dom book
]
```

Notice that `Book` is a schema that defines `book`, `known`, and relationship between `book` and `known`

It is very useful to interpret types as *entities* and relations, functions, and sets as *relationships* between entities. For example, our example corresponds to the entity-relationship diagram shown in Figure 3. These diagrams are very useful in depicting the basic types and relationships between them.

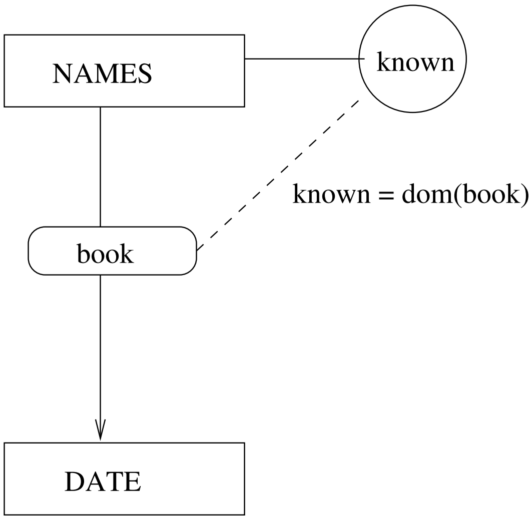

Figure 3: Entity Relationship Diagram

6 Operations

- `Insert`

  This operation takes a name and a date and inserts it in the book. The operation can be defined mathematically as:

  $$\mathtt{Insert}(n : \mathtt{NAME}, d : \mathtt{DATE}) \equiv (\mathtt{book}' = \mathtt{book} \cup \{n \rightarrow d\})$$

  The schema in *ladybug* corresponding to this is:

  ```
  Insert (n: NAME; d: DATE) = [
  ```

```
  Book
|
  book' = book U {n -> d}
]
```

Notice that the schema for `Insert` includes the schema `Book` which we defined before. Suppose `book` already has a pair (n, d) in it and one tries to perform the operation $\texttt{Insert}(n, d1)$. This operation is only valid if $d1 = d$ (*Why?*).

- `Delete`
  This operation allows us to delete a set of names from our birthday book. The operation is define as:

  $$\texttt{Delete}(ns : \textit{Set}\ \texttt{NAME}) \equiv (\texttt{book}' = (\texttt{ns} \trianglelefteq \texttt{book}))$$

  The schema for this operation in *ladybug* looks as follows:

  ```
  Delete (ns: set NAME) = [
    Book
  |
    book' = ns <; book
  ]
  ```

- `DeleteImplicit`
  This is another way of expressing the `Delete` operation.

  $$\begin{aligned}\texttt{DeleteImplicit}(ns : \textit{Set}\ \texttt{NAME}) \equiv\ & (ns \lhd \texttt{book}' = \emptyset) \wedge \\ & (ns \trianglelefteq \texttt{book} = ns \trianglelefteq \texttt{book}')\end{aligned}$$

  In *ladybug* the schema corresponding to this operation is:

  ```
  DeleteImplicit (ns: set NAME) = [
    Book
  |
    ns <: book' = {}
    ns <; book' = ns <; book
  ]
  ```

  Notice that in *ladybug* there is an implicit conjunction between various assertions.

- `Find`
  `Find` operation finds the birthday of a person with a certain name.

  $$\texttt{Find}(n : \texttt{NAME}, d : \texttt{DATE}) \equiv ((d = \texttt{book}.n) \wedge (\texttt{book}.\{n\} \subseteq \texttt{d}))$$

  The schema corresponding to this in *ladybug* is:

```
Find (n: NAME; d: DATE) = [
const Book
|
d = book.n
  book.{n} <= {d}
]
```

Notice the statement `const Book`. This means that all entities inside the schema `Book` are held constant by the operation `Find`.

7 Claims

- *Delete undoes Insert*
  This claim asserts that an `Insert` operation followed by a `Delete` operation results in the same state as we started with.

  ```
  DeleteUndoesInsert (n: NAME; d: DATE) :: [Book |
    Insert (n, d) ; Delete ({n})
      => book' = book
  ]
  ```

  Notice the `::` after the name of the claim. This is an indication to *ladybug* that `DeleteUndoesInsert` is a claim.

- *DeleteImplicit implies Delete*
  This says that operation `DeleteImplicit` is stronger than `Delete` or the post-condition of `DeleteImplicit` is stronger than that of `Delete`.

  ```
  Same (ns: set NAME) :: [Book |
    DeleteImplicit(ns) => Delete(ns)
  ]
  ```

- *Inserting a name makes it known*
  This assertion states that an `Insert` operation makes the name it is inserting known.

  ```
  InsertMakesKnown (n: NAME; d: DATE) :: [
    Book
  |
    Insert (n, d) => n in known'
  ]
  ```

- *Inserting something means that we can find it*
  This claim states that inserting (n, d) into the birthday book means that when we do a `Find` on n we get d.

```
InsertWorks (n: NAME; d, d2: DATE) :: [
  Book
|
  Insert (n, d) ; Find (n, d2) => d = d2
]
```

8 Counter Example

Consider a state where book $= \{(n0, d0)\}$. Let us say we perform an operation Insert$(n0, d0)$. In this case the value of book in the next state is also $\{(n0, d0)\}$. Performing Delete$(n0, d0)$ results in book$' = \emptyset$, and hence book is not equal to book$'$.

```
Welcome to Ladybug 0.8 (beta release), Copyright 1998


Loaded birthday-book.np

Select a claim or schema and choose 'Check' from the menu
    or double click on a claim or schema

Completed translation of DeleteUndoesInsert
Required 0:00:00.8 starting at 6:38:22 PM

Found Counterexample to Claim DeleteUndoesInsert:
book : NAME->DATE =
{ n0 -> d0 }
book' : NAME->DATE =
{  }
d : DATE =
d0
known : set NAME =
{ n0 }
known' : set NAME =
{  }
n : NAME =
n0

Found 1 Counterexamples
Checked 2 cases and 6 values
Covered 2% of the total assignment space
Required 0:00:00.0 starting at 6:38:22 PM
```

9 Entire Program

The entire program is shown below.

```
/*
a version of spivey's birthday book
*/

[NAME, DATE]

Book = [
  book: NAME -> DATE
  known: set NAME
|
  known = dom book
]

Insert (n: NAME; d: DATE) = [
  Book
|

  book' = book U {n -> d}
]

Delete (ns: set NAME) = [
  Book
|
  book' = ns <; book
]

DeleteImplicit (ns: set NAME) = [
  Book
|
  ns <: book' = {}
  ns <; book' = ns <; book
]

Find (n: NAME; d: DATE) = [
const Book
|
d = book.n
  book.{n} <= {d}
]

DeleteUndoesInsert (n: NAME; d: DATE) :: [Book |
```

```
  Insert (n, d) ; Delete ({n})
    => book' = book
]

Same (ns: set NAME) :: [Book |
  DeleteImplicit(ns) => Delete(ns)
]

InsertMakesKnown (n: NAME; d: DATE) :: [
  Book
|
  Insert (n, d) => n in known'
]

InsertWorks (n: NAME; d, d2: DATE) :: [
  Book
|
  Insert (n, d) ; Find (n, d2) => d = d2
]
```