

(We use procedure syntax since we are not interested in the output.) We can repeat this transaction several times in different threads. Because transactions are atomic, we are sure that  $@C1 + @C2 = 0$  will always be true. It is an invariant of our system. This would not be the case if the increment and decrement were executed outside a transaction. To read the contents of  $C1$  and  $C2$ , we have to use another transaction:

```
{Browse {Trans fun {$ T} {T.access C1}#{T.access C2} end _}}
```

### Another example

The previous example does not show the real advantages of transactions. The same result could have been achieved with locks. Our transaction ADT has two advantages with respect to locks: aborting causes the original cell states to be restored and the locks can be requested in any order without leading to deadlock. Let us give a more sophisticated example that exploits these two advantages. We will create a tuple with 100 cells and do transactional calculations with it. We start by creating and initializing the tuple:

```
D={MakeTuple db 100}
for I in 1..100 do D.I={NewCellT I} end
```

(We use a tuple of cells instead of an array because our transaction ADT only handles cells.) We now define two transactions, *Mix* and *Sum*. *Sum* calculates the sum of all cell contents. *Mix* “mixes up” the cell contents in random fashion but keeps the total sum unchanged. Here is the definition of *Mix*:

```
fun {Rand} {OS.rand} mod 100 + 1 end
proc {Mix} {Trans
  proc {$ T _}
    I={Rand} J={Rand} K={Rand}
    A={T.access D.I} B={T.access D.J} C={T.access D.K}
  in
    {T.assign D.I A+B-C}
    {T.assign D.J A-B+C}
    if I==J orelse I==K orelse J==K then {T.abort} end
    {T.assign D.K ~A+B+C}
  end _ _}
end
```

The random number generator *Rand* is implemented with the *OS* module. The mix-up function replaces the contents  $a$ ,  $b$ ,  $c$  of three randomly-picked cells by  $a + b - c$ ,  $a - b + c$ , and  $-a + b + c$ . To guarantee that three different cells are picked, *Mix* aborts if any two are the same. The abort can be done at any point inside the transaction. Here is the definition of *Sum*:

```
S={NewCellT 0}
fun {Sum}
  {Trans
    fun {$ T} {T.assign S 0}
```

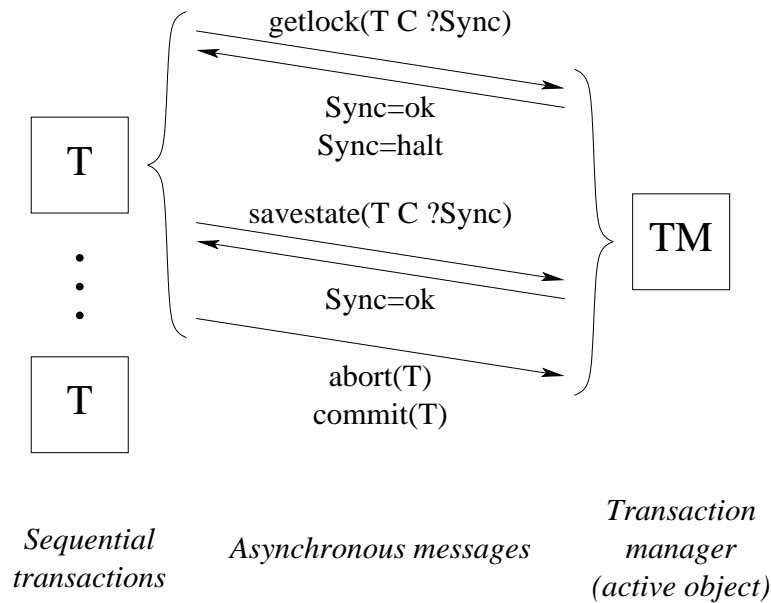


Figure 8.21: Architecture of the transaction system

```

for I in 1..100 do
    {T.assign S {T.access S}+{T.access D.I}} end
    {T.access S}
end _}
end

```

Sum uses the cell *S* to hold the sum. Note that Sum is a big transaction since it simultaneously locks *all* cells in the tuple. Now we can do some calculations:

```

{Browse {Sum}} % Displays 5050
for I in 1..1000 do thread {Mix} end end
{Browse {Sum}} % Still displays 5050

```

5050 is the sum of the integers from 1 to 100. You can check that the values of individual cells are well and truly mixed:

```

{Browse {Trans fun {$ T} {T.access D.1}#{T.access D.2} end _}}

```

This initially displays 1#2, but will subsequently display very different values.

### 8.5.4 Implementing transactions on cells

Let us show how to build a transaction system that implements our optimistic two-phase locking algorithm. The implementation consists of a transaction manager and a set of running transactions. (Transaction managers come in many varieties and are sometimes called *transaction processing monitors* [64].) The transaction manager and the running transactions each execute in its own thread. This allows terminating a running transaction without affecting the transaction

manager. A running thread sends four kinds of messages to the transaction manager: to get a lock (`getlock`), to save a cell's state (`savestate`), to commit (`commit`), and to abort (`abort`). Figure 8.21 shows the architecture.

The transaction manager is always active and accepts commands from the running transactions' threads. When a transaction is restarted, it restarts in a new thread. It keeps the same timestamp, though. We implement the transaction manager as an active object using the `NewActive` function of Section 7.8. The active object has two internal methods, `Unlockall` and `Trans`, and five external methods, `newtrans`, `getlock`, `savestate`, `commit`, and `abort`. Figures 8.22 and 8.23 show the implementation of the transaction system. Together with `NewActive` and the priority queue, this is a complete working implementation. Each active transaction is represented by a record with five fields:

- **stamp**: This is the transaction's *timestamp*, a unique integer that identifies the transaction and its priority. This number is incremented for successive transactions. High priority therefore means a small timestamp.
- **save**: This is a dictionary indexed by cell name (see below) that contains entries of the form `save(cell:C state:S)`, where `C` is a cell record (as represented below) and `S` is the cell's original state.
- **body**: This is the function `fun { $ T } <expr> end` that represents the transaction body.
- **state**: This is a cell containing `running`, `probation`, or `waiting_on(C)`. If `probation`, it means that the transaction will be restarted the next time it tries to obtain a lock. If `waiting_on(C)`, it means that the transaction will be restarted immediately if a higher priority transaction needs `C`.
- **result**: This is a dataflow variable that will be bound to `commit(Res)`, `abort(Exc)`, or `abort` when the transaction completes.

Each cell is represented by a record with four fields:

- **name**: This is a name value that is the cell's unique identifier.
- **owner**: This is either `unit`, if no transaction is currently locking the cell, or the transaction record if a transaction is locking the cell.
- **queue**: This is a priority queue containing pairs of the form `Sync#T`, where `T` is a transaction record and `Sync` is the synchronization variable on which the transaction is currently blocked. The priority is the transaction's timestamp. `Sync` will always eventually be bound by the transaction manager to `ok` or `halt`.
- **state**: This is a cell that contains the transactional cell's content.

```

class TMClass
  attr timestamp tm
  meth init(TM) timestamp:=0 tm:=TM end

  meth Unlockall(T RestoreFlag)
    for save(cell:C state:S) in {Dictionary.items T.save} do
      (C.owner):=unit
      if RestoreFlag then (C.state):=S end
      if {Not {C.queue.isEmpty}} then
        Sync2#T2={C.queue.dequeue} in
          (T2.state):=running
          (C.owner):=T2 Sync2=ok
        end
      end
    end
  end

  meth Trans(P ?R TS) /* See next figure */ end
  meth getlock(T C ?Sync) /* See next figure */ end

  meth newtrans(P ?R)
    timestamp:=@timestamp+1 {self Trans(P R @timestamp)}
  end
  meth savestate(T C ?Sync)
    if {Not {Dictionary.member T.save C.name}} then
      (T.save).(C.name):=save(cell:C state:@(C.state))
    end Sync=ok
  end
  meth commit(T) {self Unlockall(T false)} end
  meth abort(T) {self Unlockall(T true)} end
end

proc {NewTrans ?Trans ?NewCellT}
TM={NewActive TMClass init(TM)} in
  fun {Trans P ?B} R in
    {TM newtrans(P R)}
    case R of abort then B=abort unit
    [] abort(Exc) then B=abort raise Exc end
    [] commit(Res) then B=commit Res end
  end
  fun {NewCellT X}
    cell(name:{NewName} owner:{NewCell unit}
      queue:{NewPrioQueue} state:{NewCell X})
  end
end
end

```

Figure 8.22: Implementation of the transaction system (part 1)

```

meth Trans(P ?R TS)
  Halt={NewName}
  T=trans(stamp:TS save:{NewDictionary} body:P
          state:{NewCell running} result:R)
  proc {ExcT C X Y} S1 S2 in
    {@tm getlock(T C S1)}
    if S1==halt then raise Halt end end
    {@tm savestate(T C S2)} {Wait S2}
    {Exchange C.state X Y}
  end
  proc {AccT C ?X} {ExcT C X X} end
  proc {AssT C X} {ExcT C _ X} end
  proc {AboT} {@tm abort(T)} R=abort raise Halt end end
in
  thread try Res={T.body t(access:AccT assign:AssT
                           exchange:ExcT abort:AboT)}
    in {@tm commit(T)} R=commit(Res)
    catch E then
      if E\=Halt then {@tm abort(T)} R=abort(E) end
    end end
end

meth getlock(T C ?Sync)
  if @(T.state)==probation then
    {self Unlockall(T true)}
    {self Trans(T.body T.result T.stamp)} Sync=halt
  elseif @(C.owner)==unit then
    (C.owner):=T Sync=ok
  elseif T.stamp==@(C.owner).stamp then
    Sync=ok
  else /* T.stamp\=@(C.owner).stamp */ T2=@(C.owner) in
    {C.queue.enqueue Sync#T T.stamp}
    (T.state):=waiting_on(C)
    if T.stamp<T2.stamp then
      case @(T2.state) of waiting_on(C2) then
        Sync2#_={C2.queue.delete T2.stamp} in
          {self Unlockall(T2 true)}
          {self Trans(T2.body T2.result T2.stamp)}
          Sync2=halt
        [] running then
          (T2.state):=probation
        [] probation then skip end
      end
    end
  end

```

Figure 8.23: Implementation of the transaction system (part 2)

When a transaction  $T$  does an exchange operation on cell  $C$ , it executes the `ExcT` procedure defined in `Trans`. This first sends `getlock(T C Sync1)` to the transaction manager to request a lock on the cell. The transaction manager replies with `Sync1=ok` if the transaction successfully gets the lock and `Sync1=halt` if the current thread should be terminated. In the latter case, `getlock` ensures that the transaction is restarted. If the transaction gets the lock, then it calls `savestate(T C Sync2)` to save the original cell state.

### Priority queue

The transaction manager uses priority queues to make sure that high priority transactions get the first chance to lock cells. A *priority queue* is a queue whose entries are always ordered according to some priority. In our queue, the priorities are integers and the lowest value has the highest priority. We define the ADT as follows:

- $Q = \{\text{NewPrioQueue}\}$  creates an empty priority queue.
- $\{Q.\text{enqueue } X \ P\}$  inserts  $X$  with priority  $P$ , where  $P$  is an integer.
- $X = \{Q.\text{dequeue}\}$  returns the entry with the smallest integer value and removes it from the queue.
- $X = \{Q.\text{delete } P\}$  returns the entry with priority  $P$  and removes it from the queue.
- $B = \{Q.\text{isEmpty}\}$  returns true or false depending on whether  $Q$  is empty or not.

Figure 8.24 shows a simple implementation of the priority queue. The priority queue is represented internally as a cell containing a list of pairs `pair(X P)`, which are ordered according to increasing  $P$ . The `dequeue` operation executes in  $O(1)$  time. The `enqueue` and `delete` operations execute in  $O(s)$  time where  $s$  is the size of the queue. More sophisticated implementations are possible with better time complexities.

#### 8.5.5 More on transactions

We have just scratched the surface of transaction processing. Let us finish by mentioning some of the most useful extensions [64]:

- *Durability.* We have not shown how to make a state change persistent. This is done by putting state changes on stable storage, such as a disk. Techniques for doing this are carefully designed to maintain atomicity, no matter at what instant in time a system crash happens.

```

fun {NewPrioQueue}
  Q={NewCell nil}
  proc {Enqueue X Prio}
    fun {InsertLoop L}
      case L of pair(Y P)|L2 then
        if Prio<P then pair(X Prio)|L
        else pair(Y P)|{InsertLoop L2} end
      [] nil then [pair(X Prio)] end
    end
  in Q:={InsertLoop @Q} end

  fun {Dequeue}
    pair(Y _)|L2=@Q
  in
    Q:=L2 Y
  end

  fun {Delete Prio}
    fun {DeleteLoop L}
      case L of pair(Y P)|L2 then
        if P==Prio then X=Y L2
        else pair(Y P)|{DeleteLoop L2} end
      [] nil then nil end
    end X
  in
    Q:={DeleteLoop @Q}
    X
  end

  fun {IsEmpty} @Q==nil end
in
  queue(enqueue:Enqueue dequeue:Dequeue
        delete>Delete isEmpty:IsEmpty)
end

```

Figure 8.24: Priority queue

- *Nested transactions.* It often happens that we have a long-lived transaction that contains a series of smaller transactions. For example, a complex bank transaction might consist of a large series of updates to many accounts. Each of these updates is a transaction. The series itself should also be a transaction: if something goes wrong in the middle, it is canceled. There is a strong relationship between nested transactions, encapsulation, and modularity.
- *Distributed transactions.* It often happens that a database is spread over several physical sites, either for performance or organizational reasons. We would still like to perform transactions on the database.

## 8.6 The Java language (concurrent part)

The introduction of Section 7.7 only talked about the sequential part of Java. We now extend this to the concurrent part. Concurrent programming in Java is supported by two concepts: threads and monitors. Java is designed for shared-state concurrency. Threads are too heavyweight to support an active object approach efficiently. Monitors have the semantics of Section 8.4. Monitors are lightweight constructs that are associated to individual objects.

Each program starts with one thread, the one that executes `main`. New threads can be created in two ways, by instantiating a subclass of the `Thread` class or by implementing the `Runnable` interface. By default, the program terminates when all its threads terminate. Since threads tend to be heavyweight in current Java implementations, the programmer is encouraged not to create many of them. Using the `Thread` class gives more control, but might be overkill for some applications. Using the `Runnable` interface is lighter. Both techniques assume that there is a method `run`:

```
public void run();
```

that defines the thread's body. The `Runnable` interface consists of just this single method.

Threads interact by means of shared objects. To control the interaction, any Java object can be a monitor, as defined in Section 8.4. Methods can execute inside the monitor lock with the keyword `synchronized`. Methods without this keyword are called non-synchronized. They execute outside the monitor lock but can still see the object attributes. This ability has been strongly criticized because the compiler can no longer guarantee that the object attributes are accessed sequentially [24]. Non-synchronized methods can be more efficient, but they should be used extremely rarely.

We give two examples. The first example uses synchronized methods just for locking. The second example uses the full monitor operations. For further reading, we recommend [111].



### 8.6.1 Locks

The simplest way to do concurrent programming in Java is with multiple threads that access shared objects. Let us extend the class `Point` as an example:

```
class Point {
    double x, y;
    Point(double x1, y1) { x=x1; y=y1; }
    public double getX() { return x; }
    public double getY() { return y; }
    public synchronized void origin() { x=0.0; y=0.0; }
    public synchronized void add(Point p)
        { x+=p.getX(); y+=p.getY(); }
    public synchronized void scale(double s) { x*=s; y*=s; }
    public void draw(Graphics g) {
        double lx, ly;
        synchronized (this) { lx=x; ly=y; }
        g.drawPoint(lx, ly);
    }
}
```

Each instance of `Point` has its own lock. Because of the keyword `synchronized`, the methods `origin`, `add`, and `scale` all execute within the lock. The method `draw` is only partly synchronized. This is because it calls an external method, `g.drawPoint` (not defined in the example). Putting the external method inside the object lock would increase the likelihood of deadlocking the program. Instead, `g` should have its own lock.

### 8.6.2 Monitors

Monitors are an extension of locks that give more control over how threads enter and exit. Monitors can be used to do more sophisticated kinds of cooperation between threads accessing a shared object. Section 8.4.1 shows how to write a bounded buffer using monitors. The solution given there can easily be translated to Java, giving Figure 8.25. This defines a bounded buffer of integers. It uses an array of integers, `buf`, which is allocated when the buffer is initialized. The percent sign `%` denotes the modulo operation, i.e., the remainder after integer division.

## 8.7 Exercises

1. **Number of interleavings.** Generalize the argument used in the chapter introduction to calculate the number of possible interleavings of  $n$  threads, each doing  $k$  operations. Using Stirling's formula for the factorial function,  $n! \approx \sqrt{2\pi n} n^{n+1/2} e^{-n}$ , calculate a closed form approximation to this function.

```
class Buffer
{
    int[] buf;
    int first, last, n, i;

    public void init(int size) {
        buf=new int[size];
        n=size; i=0; first=0; last=0;
    }

    public synchronized void put(int x) {
        while (i<n) wait();
        buf[last]=x;
        last=(last+1)%n;
        i=i+1;
        notifyAll();
    }

    public synchronized int get() {
        int x;
        while (i==0) wait();
        x=buf[first];
        first=(first+1)%n;
        i=i-1;
        notifyAll();
        return x;
    }
}
```

Figure 8.25: Bounded buffer (Java version)

2. **Concurrent counter.** Let us implement a concurrent counter in the simplest possible way. The counter has an increment operation. We would like this operation to be callable from any number of threads. Consider the following possible implementation that uses one cell and an `Exchange`:

```
local X in {Exchange C X X+1} end
```

This attempted solution does not work.

- Explain why the above program does not work and propose a simple fix.
  - Would your fix still be possible in a language that did not have dataflow variables? Explain why or why not.
  - Give a solution (perhaps the same one as in the previous point) that works in a language without dataflow variables.
3. **Maximal concurrency and efficiency.** In between the shared-state concurrent model and the maximally concurrent model, there is an interesting model called the *job-based concurrent model*. The job-based model is identical to the shared-state concurrent model, except that whenever an operation would block, a new thread is created with only that operation (this is called a *job*) and the original thread continues execution.<sup>4</sup> Practically speaking, the job-based model has all the concurrency of the maximally concurrent model, and in addition it can easily be implemented efficiently. For this exercise, investigate the job-based model. Is it a good choice for a concurrent programming language? Why or why not?
4. **Simulating slow networks.** Section 8.2.2 defines a function `SlowNet2` that creates a “slow” version of an object. But this definition imposes a strong order constraint. Each slow object defines a global order of its calls and guarantees that the original objects are called in this order. This constraint is often too strong. A more refined version would only impose order among object calls within the same thread. Between different threads, there is no reason to impose an order. Define a function `SlowNet3` that creates slow objects with this property.
5. **The `mVar` abstraction.** An `mVar` is a box that can be full or empty. It comes with two procedures, `Put` and `Get`. Doing `{Put x}` puts `x` in the box if it is empty, thus making it full. If the box is full, `Put` waits until it is empty. Doing `{Get x}` binds `x` to the boxes’ content and empties the box. If the box is empty, `Get` waits until it is full. For this exercise, implement the `mVar` abstraction. Use whatever concurrency approach is most natural.

---

<sup>4</sup>The initial Oz language used the job-based model [180].

6. **Communicating Sequential Processes (CSP).** The CSP language consists of independent threads (called “processes” in CSP terminology) communicating through synchronous channels [83, 165]. The channels have two operations, send and receive, with rendezvous semantics. That is, a send blocks until a receive is present and vice versa. When send and receive are simultaneously present, then they both complete atomically, transferring information from send to receive. The Ada language also uses rendezvous semantics. In addition, there is a nondeterministic receive operation which listens to several channels simultaneously. As soon as a message is received on one of the channels, then the nondeterministic receive completes. For this exercise, implement these CSP operations as the following control abstraction:

- `C={Channel.new}` creates a new channel `C`.
- `{Channel.send C M}` sends message `M` on channel `C`.
- `{Channel.mreceive [C1#S1 C2#S2 ... Cn#Sn]}` listens nondeterministically on channels `C1`, `C2`, ..., and `Cn`. `Si` is a one-argument procedure **proc** `{ $ M } <stmt> end` that is executed when message `M` is received on channel `Ci`.

Now extend the `Channel.mreceive` operation with guards:

- `{Channel.mreceive [C1#B1#S1 C2#B2#S2 ... Cn#Bn#Sn]}`, where `Bi` is a one-argument boolean function **fun** `{ $ M } <expr> end` that must return true for a message to be received on channel `Ci`.

7. **Comparing Linda with Erlang.** Linda has a read operation that can selectively retrieve tuples according to a pattern (see Section 8.3.2). Erlang has a **receive** operation that can selectively receive messages according to a pattern (see Section 5.6.3). For this exercise, compare and contrast these two operations and the abstractions that they are part of. What do they have in common and how do they differ? For what kinds of application is each best suited?
8. **Termination detection with monitors.** This exercise is about detecting when a group of threads are all terminated. Section 4.4.3 gives an algorithm that works for a flat thread space, where threads are not allowed to create new threads. Section 5.5.3 gives an algorithm that works for a hierarchical thread space, where threads can create new threads to any nesting level. The second algorithm uses a port to collect the termination information. For this exercise, write an algorithm that works for a hierarchical thread space, like the second algorithm, but that uses a monitor instead of a port.
9. **Monitors and conditions.** Section 8.4.4 gives an alternative semantics for monitors in which there can be several wait sets, which are called *conditions*.

The purpose of this exercise is to study this alternative and compare it with main approach given in the text.

- Reimplement the bounded buffer example of Figure 8.16 using monitors with conditions.
  - Modify the monitor implementation of Figure 8.19 to implement monitors with conditions. Allow the possibility to create more than one condition for a monitor.
10. **Breaking up big transactions.** The second example in Section 8.5.3 defines the transaction `Sum` that locks all the cells in the tuple while it is calculating their sum. While `Sum` is active, no other transaction can continue. For this exercise, rewrite `Sum` as a series of small transactions. Each small transaction should only lock a few cells. Define a representation for a partial sum, so that a small transaction can see what has already been done and determine how to continue. Verify your work by showing that you can perform transactions while a sum calculation is in progress.
  11. **Lock caching.** In the interest of simplicity, the transaction manager of Section 8.5.4 has some minor inefficiencies. For example, `getlock` and `savestate` messages are sent on *each use* of a cell by a transaction. It is clear that they are only really needed the first time. For this exercise, optimize the `getlock` and `savestate` protocols so they use the least possible number of messages.
  12. **Read and write locks.** The transaction manager of Section 8.5 locks a cell upon its first use. If transactions `T1` and `T2` both want to read the same cell's content, then they cannot both lock the cell simultaneously. We can relax this behavior by introducing two kinds of locks, read locks and write locks. A transaction that holds a read lock is only allowed to read the cell's content, not change it. A transaction that holds a write lock can do all cell operations. A cell can either be locked with exactly one write lock or with any number of read locks. For this exercise, extend the transaction manager to use read and write locks.
  13. **Concurrent transactions.** The transaction manager of Section 8.5 correctly handles any number of transactions that execute concurrently, but each individual transaction must be sequential. For this exercise, extend the transaction manager so that the individual transactions can themselves be concurrent. Hint: add the termination detection algorithm of Section 5.5.3.
  14. **Combining monitors and transactions.** Design and implement a concurrency abstraction that combines the abilities of monitors and transactions. That is, it has the ability to wait and notify, and also the ability to abort without changing any state. Is this a useful abstraction?

15. (*research project*) **Transactional computation model.** Extend the shared-state concurrent model of this chapter to allow transactions, as suggested in Section 8.5. Your extension should satisfy the following properties:

- It should have a simple formal semantics.
- It should be efficient, i.e., only cause overhead when transactions are actually used.
- It should preserve good properties of the model, e.g., compositionality.

This will allow programs to use transactions without needing costly and cumbersome encodings. Implement a programming language that uses your extension and evaluate it for realistic concurrent programs.



# Chapter 9

## Relational Programming

“Toward the end of the thirteenth century, Ramón Llull (Raimundo Lulio or Raymond Lully) invented the thinking machine. [...] The circumstances and objectives of this machine no longer interest us, but its guiding principle—the methodical application of chance to the resolution of a problem—still does.”

– Ramón Llull’s Thinking Machine, *Jorge Luis Borges* (1899–1986)

“In retrospect it can now be said that the *ars magna Lulli* was the first seed of what is now called “symbolic logic,” but it took a long time until the seed brought fruit, this particular fruit.”

– Postscript to the “Universal Library”, *Willy Ley* (1957)

A procedure in the declarative model uses its input arguments to calculate the values of its output arguments. This is a *functional* calculation, in the mathematical sense: the outputs are functions of the inputs. For a given set of input argument values, there is only one set of output argument values. We can generalize this to become *relational*. A relational procedure is more flexible in two ways than a functional procedure. First, there can be any number of results to a call, either zero (no results), one, or more. Second, which arguments are inputs and which are outputs can be different for each call.

This flexibility makes relational programming well-suited for databases and parsers, in particular for difficult cases such as deductive databases and parsing ambiguous grammars. It can also be used to enumerate solutions to complex combinatoric problems. We have used it to automatically generate diagnostics for a RISC microprocessor, the VLSI-BAM [84, 193]. The diagnostics enumerate all possible instruction sequences that use register forwarding. Relational programming has also been used in artificial intelligence applications such as David Warren’s venerable WARPLAN planner [39].

From the programmer’s point of view, relational programming extends declarative programming with a new kind of statement called “choice”. Conceptually, the choice statement nondeterministically picks one among a set of alternatives.



During execution, the choice is implemented with search, which enumerates the possible answers. We call this *don't know nondeterminism*, although the search algorithm is almost always deterministic.

Introducing a choice statement is an old idea. E. W. Elcock [52] used it in 1967 in the Absys language and Floyd [53] used it in the same year. The Prolog language uses a choice operation as the heart of its execution model, which was defined in 1972 [40]. Floyd gives a lucid account of the choice operation. He first extends a simple Algol-like language with a function called *choice*(*n*), which returns an integer from 1 to *n*. He then shows how to implement a depth-first search strategy using flow charts to give the operational semantics of the extended language.

### Watch out for efficiency

The flexibility of relational programming has a reverse side. It can easily lead to highly inefficient programs, if not used properly. This cannot be avoided in general since each new choice operation multiplies the size of the search space by the number of alternatives. The *search space* is the set of candidate solutions to a problem. This means the size is exponential in the number of choice operations. However, relational programming is sometimes practical:

- **When the search space is small.** This is typically the case for database applications. Another example is the above-mentioned VLSI-BAM diagnostics generator, which generated all combinations of instructions for register forwarding, condition bit forwarding, and branches in branch delay slots. This gave a total of about 70,000 lines of VLSI-BAM assembly language code. This was small enough to be used as input to the gate-level simulator.
- **As an exploratory tool.** If used on small examples, relational programming can give results even if it is impractical for bigger examples. The advantage is that the programs can be *much* shorter and easier to write: no algorithm has to be devised since search is a brute force technique that avoids the need for algorithms. This is an example of *nonalgorithmic* programming. This kind of exploration gives insight into the problem structure. This insight is often sufficient to design an efficient algorithm.

To use search in other cases, more sophisticated techniques are needed, e.g., powerful constraint-solving algorithms, optimizations based on the problem structure, and search heuristics. We leave these until Chapter 12. The present chapter studies the use of nondeterministic programming as a tool for the two classes of problems for which it works well. For more information and techniques, we recommend any good book on Prolog, which has good support for nondeterministic programming [182, 39].

$\langle s \rangle ::=$	
<b>skip</b>	Empty statement
$\langle s \rangle_1 \langle s \rangle_2$	Statement sequence
<b>local</b> $\langle x \rangle$ <b>in</b> $\langle s \rangle$ <b>end</b>	Variable creation
$\langle x \rangle_1 = \langle x \rangle_2$	Variable-variable binding
$\langle x \rangle = \langle v \rangle$	Value creation
<b>if</b> $\langle x \rangle$ <b>then</b> $\langle s \rangle_1$ <b>else</b> $\langle s \rangle_2$ <b>end</b>	Conditional
<b>case</b> $\langle x \rangle$ <b>of</b> $\langle \text{pattern} \rangle$ <b>then</b> $\langle s \rangle_1$ <b>else</b> $\langle s \rangle_2$ <b>end</b>	Pattern matching
$\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$	Procedure application
<b>choice</b> $\langle s \rangle_1$ [ ] ... [ ] $\langle s \rangle_n$ <b>end</b>	<b>Choice</b>
<b>fail</b>	<b>Failure</b>

Table 9.1: The relational kernel language

## Structure of the chapter

The chapter consists of four parts:

- Section 9.1 explains the basic concepts of the relational computation model, namely *choice* and *encapsulated search*. Section 9.2 continues with some more examples to introduce programming in the model.
- Section 9.3 introduces logic and logic programming. It introduces a new kind of semantics for programs, the *logical semantics*. It then explains how both the declarative and relational computation models are doing logic programming.
- Sections 9.4–9.6 give large examples in three areas that are particularly well-suited to relational programming, namely natural language parsing, interpreters, and deductive databases.
- Section 9.7 gives an introduction to Prolog, a programming language based on relational programming. Prolog was originally designed for natural language processing, but has become one of the main programming languages in all areas that require symbolic programming.

## 9.1 The relational computation model

### 9.1.1 The **choice** and **fail** statements

The relational computation model extends the declarative model with two new statements, **choice** and **fail**:

- The **choice** statement groups together a set of alternative statements. Executing a **choice** statement provisionally picks one of these alternatives. If the alternative is found to be wrong later on, then another one is picked.
- The **fail** statement indicates that the current alternative is wrong. A **fail** is executed implicitly when trying to bind two incompatible values, for example  $3=4$ . This is a modification of the declarative model, which raises an exception in that case. Section 2.7.2 explains the binding algorithm in detail for all partial values.

Table 9.1 shows the relational kernel language.

### An example for clothing design

Here is a simple example of a relational program that might interest a clothing designer:

```

fun {Soft} choice beige [] coral end end
fun {Hard} choice mauve [] ochre end end

proc {Contrast C1 C2}
    choice C1={Soft} C2={Hard} [] C1={Hard} C2={Soft} end
end

fun {Suit}
    Shirt Pants Socks
in
    {Contrast Shirt Pants}
    {Contrast Pants Socks}
    if Shirt==Socks then fail end
    suit(Shirt Pants Socks)
end

```

This program is intended to help a clothing designer pick colors for a man's casual suit. **Soft** picks a soft color and **Hard** picks a hard color. **Contrast** picks a pair of contrasting colors (one soft and one hard). **Suit** returns a complete set including shirt, pants, and socks such that adjacent garments are in contrasting colors and such that shirt and socks are of different colors.

#### 9.1.2 Search tree

A relational program is executed sequentially. The **choice** statements are executed in the order that they are encountered during execution. When a **choice** is first executed, its first alternative is picked. When a **fail** is executed, execution “backs up” to the most recent **choice** statement, which picks its next alternative. If there are none, then the next most recent **choice** picks another alternative, and so forth. Each **choice** statement picks alternatives in order from left to right.

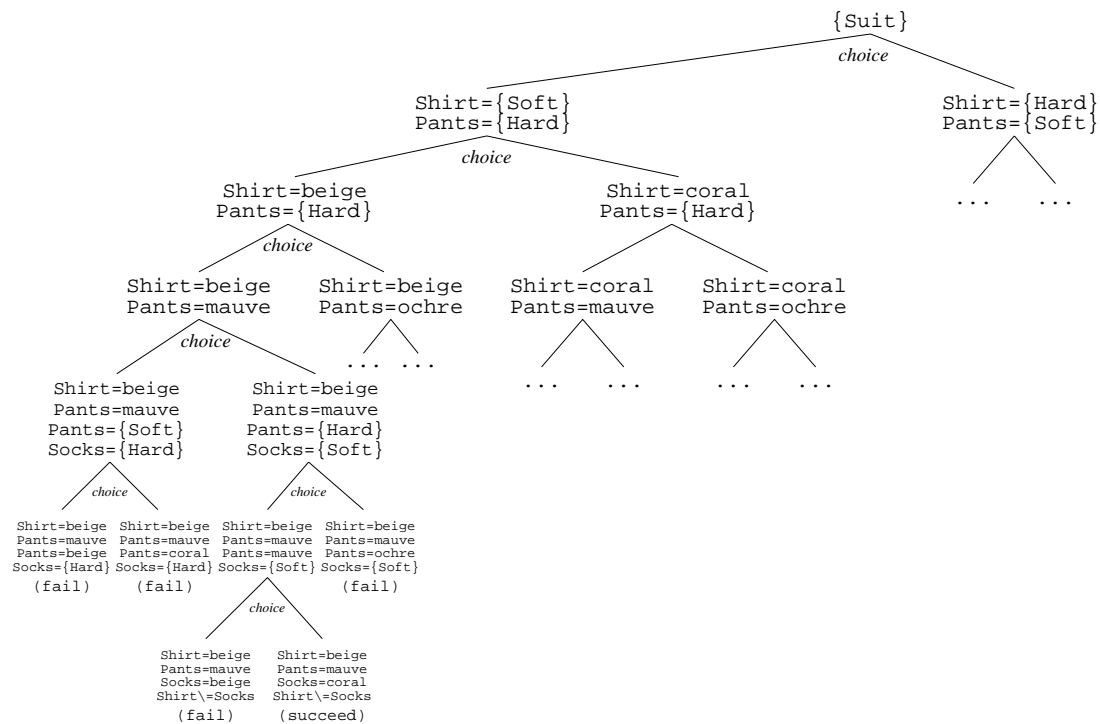


Figure 9.1: Search tree for the clothing design example

This execution strategy can be illustrated with a tree called the *search tree*. Each node in the search tree corresponds to a **choice** statement and each subtree corresponds to one of the alternatives. Figure 9.1 shows part of the search tree for the clothing design example. Each path in the tree corresponds to one possible execution of the program. The path can lead either to no solution (marked “fail”) or to a solution (marked “succeed”). The search tree shows all paths at a glance, including both the failed and successful ones.

### 9.1.3 Encapsulated search

A relational program is interesting because it can potentially execute in many different ways, depending on the choices it makes. We would like to control which choices are made and when they are made. For example, we would like to specify the search strategy: depth-first search, breadth-first search, or some other strategy. We would like to specify how many solutions are calculated: just one solution, all solutions right away, or new solutions on demand. Briefly, we would like the same relational program to be executed in many different ways.

One way to exercise this control is to execute the relational program with *encapsulated search*. Encapsulation means that the relational program runs inside a kind of “environment”. The environment controls which choices are made by the relational program and when they are made. The environment also protects the rest of the application from the effects of the choices. This is important

because the relational program can do multiple bindings of the same variable when different choices are made. These multiple bindings should not be visible to the rest of the application. Encapsulated search is important also for modularity and compositionality:

- For modularity: with encapsulated search there can be more than one relational program running concurrently. Since each is encapsulated, they do not interfere with each other (except that they can influence each other's performance because they share the same computational resources). They can be used in a program that communicates with the external world, without interfering with that communication.
- For compositionality: an encapsulated search can run inside another encapsulated search. Because of encapsulation, this is perfectly well-defined.

Early logic languages with search such as Prolog have global backtracking, in which multiple bindings are visible everywhere. This is bad for program modularity and compositionality. To be fair to Prolog, it has a limited form of encapsulated search, the `bagof/3` and `setof/3` operations. This is explained in Section 9.7.

#### 9.1.4 The `Solve` function

We provide encapsulated search by adding one function, `Solve`, to the computation model. The call `{Solve F}` is given a zero-argument function `F` (or equivalently, a one-argument procedure) that returns a solution to a relational program. The call returns a lazy list of all solutions, ordered according to a depth-first search strategy. For example, the call:

```
L={Solve fun {$} choice 1 [] 2 [] 3 end end}
```

returns the lazy list `[1 2 3]`. Because `Solve` is lazy, it only calculates the solutions that are needed. `Solve` is compositional, i.e., it can be nested: the function `F` can contain calls to `Solve`. Using `Solve` as a basic operation, we can define both one-solution and all-solutions search. To get one-solution search, we look at just the first element of the list and never look at the rest:

```
fun {SolveOne F}
  L={Solve F}
in
  if L==nil then nil else [L.1] end
end
```

This returns either a list `[X]` containing the first solution `X` or `nil` if there are no solutions. To get all-solutions search, we look at the whole list:

```
fun {SolveAll F}
  L={Solve F}
  proc {TouchAll L}
```

```

        if L==nil then skip else {TouchAll L.2} end
    end
in
    {TouchAll L}
    L
end

```

This returns a list of all solutions.

### Computation spaces

We have introduced **choice** and **fail** statements and the `Solve` function. These new operations can be programmed by extending the declarative model with just one new concept, the *computation space*. Computation spaces are part of the constraint-based computation model, which is explained in Chapter 12. They were originally designed for constraint programming, a powerful generalization of relational programming. Chapter 12 explains how to implement **choice**, **fail**, and `Solve` in terms of computation spaces. The definition of `Solve` is also given in the supplements file on the book's Web site.

### Solving the clothing design example

Let us use `Solve` to find answers to the clothing design example. To find all solutions, we do the following query:

```
{Browse {SolveAll Suit}}
```

This displays a list of the eight solutions:

```

[suit(beige mauve coral) suit(beige ochre coral)
 suit(coral mauve beige) suit(coral ochre beige)
 suit(mauve beige ochre) suit(mauve coral ochre)
 suit(ochre beige mauve) suit(ochre coral mauve)]

```

Figure 9.1 gives enough of the search tree to show how the first solution `suit(beige mauve coral)` is obtained.

## 9.2 Further examples

We give some simple examples to show how to program in the relational computation model.

### 9.2.1 Numeric examples

Let us show some simple examples using numbers, to show how to program with the relational computation model. Here is a program that uses **choice** to count from 0 to 9:

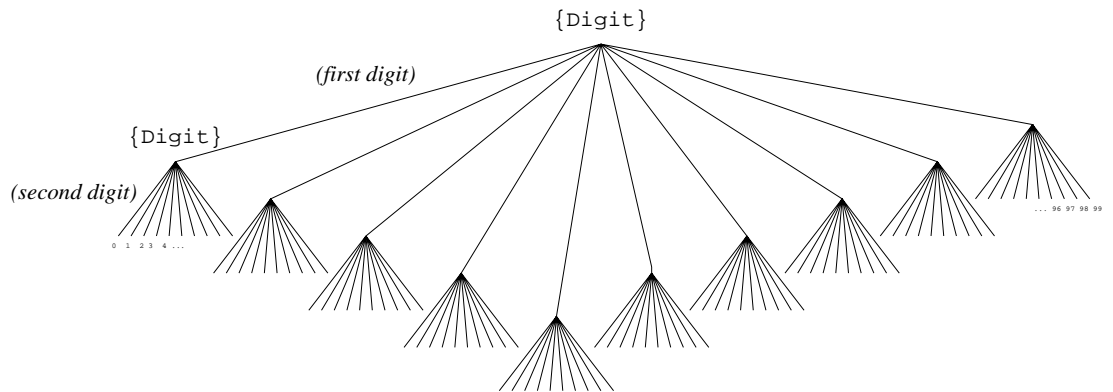


Figure 9.2: Two digit counting with depth-first search

```

fun {Digit}
  choice 0 [] 1 [] 2 [] 3 [] 4 [] 5 [] 6 [] 7 [] 8 [] 9 end
end
{Browse {SolveAll Digit}}

```

This displays:

```
[0 1 2 3 4 5 6 7 8 9]
```

(Note that the zero-argument function `Digit` is the same as a one-argument procedure.) We can combine calls to `Digit` to count with more than one digit:

```

fun {TwoDigit}
  10*{Digit}+{Digit}
end
{Browse {SolveAll TwoDigit}}

```

This displays:

```
[0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 ... 98 99]
```

This shows what it means to do a depth-first search: when *two* choices are done, the program first makes the first choice and then makes the second. Here the function chooses first the tens digit and then the ones digit. Changing the definition of `TwoDigit` to choose digits in the opposite order will give unusual results:

```

fun {StrangeTwoDigit}
  {Digit}+10*{Digit}
end
{Browse {SolveAll StrangeTwoDigit}}

```

This displays:

```
[0 10 20 30 40 50 60 70 80 90 1 11 21 31 41 ... 89 99]
```

In this case, the tens digit is chosen second and therefore changes quicker than the ones digit.

### Palindrome product problem

Using `Digit`, we can already solve some interesting puzzles, like the “palindrome product” problem. We would like to find all four-digit palindromes that are products of two-digit numbers. A *palindrome* is a number that reads the same forwards and backwards, when written in decimal notation. The following program solves the puzzle:

```

proc {Palindrome ?X}
  X=(10*{Digit}+{Digit})*(10*{Digit}+{Digit})  % Generate
  (X>0)=true                                     % Test 1
  (X>=1000)=true                                 % Test 2
  (X div 1000) mod 10 = (X div 1) mod 10        % Test 3
  (X div 100) mod 10 = (X div 10) mod 10       % Test 4
end

{Browse {SolveAll Palindrome}}
```

This displays all 118 palindrome products. Why do we have to write the condition `X>0` as `(X>0)=true`? If the condition returns **false**, then the attempted binding **false=true** will fail. This ensures the relational program will fail when the condition is false.

Palindrome product is an example of a *generate-and-test* program: it generates a set of possibilities and then it uses tests to filter out the bad ones. The tests use unification failure to reject bad alternatives. Generate-and-test is a very naive way to explore a search space. It generates *all* the possibilities first and only filters out the bad ones afterwards. In palindrome product, 10000 possibilities are generated.

Chapter 12 introduces a much better way to explore a search space, called *propagate-and-search*. This approach does the filtering during the generation, so that many fewer possibilities are generated. If we extend palindrome product to 6-digit numbers then the naive solution takes 45 seconds.<sup>1</sup> The propagate-and-search solution of Chapter 12 takes less than 0.4 second to solve the same problem.

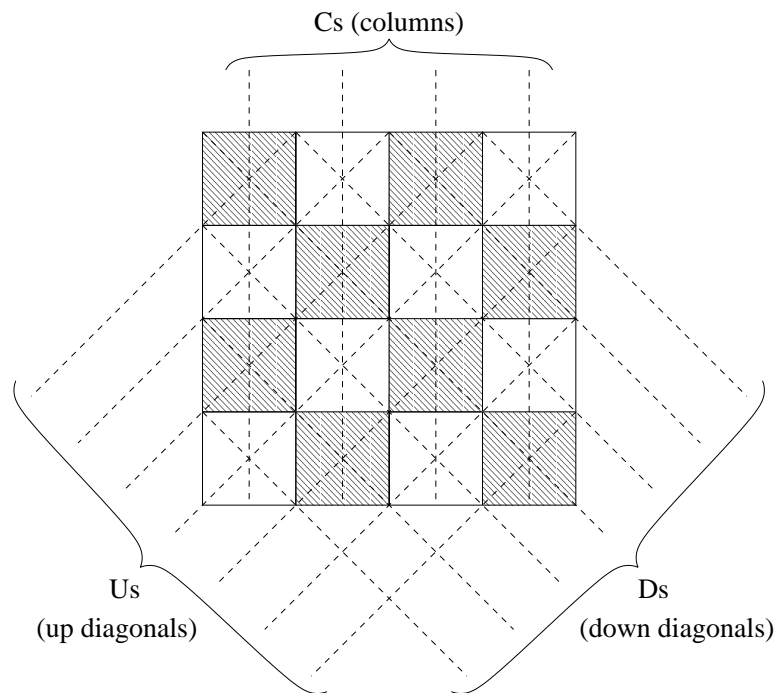
#### 9.2.2 Puzzles and the *n*-queens problem

The *n*-queens problem is an example of a combinatoric puzzle. This kind of puzzle can be easily specified in relational programming. The resulting solution is not very efficient; for more efficiency we recommend using constraint programming instead, as explained in Chapter 12. Using relational programming is a precursor to constraint programming.

The problem is to place *n* queens on an  $n \times n$  chessboard so that no queen attacks another. There are many ways to solve this problem. The solution given in Figure 9.4 is noteworthy because it uses dataflow variables. We can get the

<sup>1</sup>On a 500 MHz Pentium III processor running Mozart 1.1.0.



Figure 9.3: The  $n$ -queens problem (when  $n = 4$ )

first solution of an 8-queens problem as follows:

```
{Browse {SolveOne fun {$} {Queens 8} end}}
```

This uses higher-order programming to define a zero-argument function from the one-argument function `Queens`. The answer displayed is:

```
[[1 7 5 8 2 4 6 3]]
```

This list gives the placement of the queens on the chessboard. It assumes there is one queen per column. The solution lists the eight columns and gives for each column the queen's position (first square of first column, seventh square of second column, etc.). How many solutions are there to the 8-queens problem (counting reflections and rotations as separate)? This is easy to calculate:

```
{Browse {Length {SolveAll fun {$} {Queens 8} end}}}
```

This displays the number 92, which is the answer. `Queens` is not the best possible program for solving the  $n$ -queens problem. It is not practical for large  $n$ . Much better programs can be gotten by using constraint programming or by designing specialized algorithms (which amounts almost to the same thing). But this program is simple and elegant.

How does this magical program work? We explain it by means of Figure 9.3. Each column, up diagonal, and down diagonal has one dataflow variable. The lists `Cs`, `Us`, and `Ds` contain all the column variables, up variables, and down variables, respectively. Each column variable “guards” a column, and similarly

```

fun {Queens N}
  fun {MakeList N}
    if N==0 then nil else _|{MakeList N-1} end
  end

  proc {PlaceQueens N ?Cs ?Us ?Ds}
    if N>0 then Ds2
      Us2=_|Us
    in
      Ds=_|Ds2
      {PlaceQueens N-1 Cs Us2 Ds2}
      {PlaceQueen N Cs Us Ds}
    else skip end
  end

  proc {PlaceQueen N ?Cs ?Us ?Ds}
    choice
      Cs=N|_ Us=N|_ Ds=N|_
    [ ] _|Cs2=Cs _|Us2=Us _|Ds2=Ds in
      {PlaceQueen N Cs2 Us2 Ds2}
    end
  end
  Qs={MakeList N}
in
  {PlaceQueens N Qs _ _}
  Qs
end

```

Figure 9.4: Solving the  $n$ -queens problem with relational programming

for the variables of the up and down diagonals. Placing a queen on a square binds the three variables to the queen's number. Once the variables are bound, no other queen can bind the variable of the same column, up diagonal, or down diagonal. This is because a dataflow variable can only have one value. Trying to bind to another value gives a unification failure, which causes that alternative to be rejected.

The procedure `PlaceQueens` traverses a column from top to bottom. It keeps the same `Cs`, but “shifts” the `Us` one place to the right and the `Ds` one place to the left. At each iteration, `PlaceQueens` is at one row. It calls `PlaceQueen`, which tries to place a queen in one of the columns of that row, by binding one entry in `Cs`, `Us`, and `Ds`.

## 9.3 Relation to logic programming

Both the declarative computation model of Chapter 2 and the relational computation model of this chapter are closely related to logic programming. This section examines this relationship. Section 9.3.1 first gives a brief introduction to logic and logic programming. Sections 9.3.2 and 9.3.3 then show how these ideas apply to the declarative and relational computation models. Finally, Section 9.3.4 briefly mentions pure Prolog, which is another implementation of logic programming.

The advantage of logic programming is that programs have *two* semantics, a logical and an operational semantics, which can be studied separately. If the underlying logic is chosen well, then the logical semantics is much simpler than the operational. However, logic programming cannot be used for all computation models. For example, there is no good way to design a logic for the stateful model. For it we can use the axiomatic semantics of Section 6.6.

### 9.3.1 Logic and logic programming

A *logic program* is a statement of logic that is given an operational semantics, i.e., it can be executed on a computer. If the operational semantics is well-designed, then the execution has two properties: it is *correct*, i.e., it respects the logical semantics (all consequences of the execution are valid logical consequences of the program considered as a set of logical axioms) and it is *efficient*, i.e., it allows to write programs that execute with the expected time and space complexity. Let us examine more closely the topics of logic and logic programming. Be warned that this section gives only a brief introduction to logic and logic programming. For more information we refer interested readers to other books [114, 182].

#### Propositional logic

What is an appropriate logic in which to write logic programs? There are many different logics. For example, there is propositional logic. Propositional formulas consist of expressions combining symbols such as  $p$ ,  $q$ ,  $r$ , and so forth together with the connectors  $\wedge$  (“and”),  $\vee$  (“or”),  $\leftrightarrow$  (“if and only if”),  $\rightarrow$  (“implies”), and  $\neg$  (“not”). The symbols  $p$ ,  $q$ ,  $r$ , and so forth are called *atoms* in logic. An atom in logic is the smallest indivisible part of a logical formula. This should not be confused with an atom in a programming language, which is a constant uniquely determined by its print representation.

Propositional logic allows to express many simple laws. The contrapositive law  $(p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)$  is a formula of propositional logic, as is De Morgan’s law  $\neg(p \wedge q) \leftrightarrow (\neg p \vee \neg q)$ . To assign a truth value to a propositional formula, we have to assign a truth value to each of its atoms. We then evaluate the formula using the usual rules for  $\wedge$ ,  $\vee$ ,  $\leftrightarrow$ ,  $\rightarrow$ , and  $\neg$ :

$a$	$b$	$a \wedge b$	$a \vee b$	$a \leftrightarrow b$	$a \rightarrow b$	$\neg a$
false	false	false	false	true	true	true
false	true	false	true	false	true	true
true	false	false	true	false	false	false
true	true	true	true	true	true	false

If the formula is true for all possible assignments of its atoms, then it is called a *tautology*. Both the contrapositive law and De Morgan's law are examples of tautologies. They are true for all four possible truth assignments of  $p$  and  $q$ .

### First-order predicate calculus

Propositional logic is rather weak as a base for logic programming, principally because it does not allow expressing data structures. First-order predicate calculus is much better-suited for this. The predicate calculus generalizes propositional logic with variables, terms, and quantifiers. A logical formula in the predicate calculus has the following grammar, where  $\langle a \rangle$  is an atom and  $\langle f \rangle$  is a formula:

$$\begin{aligned}
 \langle a \rangle &::= p(\langle x \rangle_1, \dots, \langle x \rangle_n) \\
 \langle f \rangle &::= \langle a \rangle \\
 &\quad | \langle x \rangle = f(l_1 : \langle x \rangle_1, \dots, l_n : \langle x \rangle_n) \\
 &\quad | \langle x \rangle_1 = \langle x \rangle_2 \\
 &\quad | \langle f \rangle_1 \wedge \langle f \rangle_2 \mid \langle f \rangle_1 \vee \langle f \rangle_2 \mid \langle f \rangle_1 \leftrightarrow \langle f \rangle_2 \mid \langle f \rangle_1 \rightarrow \langle f \rangle_2 \mid \neg \langle f \rangle \\
 &\quad | \forall \langle x \rangle. \langle f \rangle \mid \exists \langle x \rangle. \langle f \rangle
 \end{aligned}$$

Atoms in predicate calculus are more general than propositional atoms since they can have arguments. Here  $\langle x \rangle$  is a variable symbol,  $p$  is a predicate symbol,  $f$  is a term label, and the  $l_i$  are term features. The symbols  $\forall$  (“for all”) and  $\exists$  (“there exists”) are called *quantifiers*. In like manner as for program statements, we can define the *free identifier occurrences* of a logical formula. Sometimes these are called free variables, although strictly speaking they are not variables. A logical formula with no free identifier occurrences is called a *logical sentence*. For example,  $p(x, y) \wedge q(y)$  is not a logical sentence because it has two free variables  $x$  and  $y$ . We can make it a sentence by using quantifiers, giving for instance  $\forall x. \exists y. p(x, y) \wedge q(y)$ . The free variables  $x$  and  $y$  are captured by the quantifiers.

### Logical semantics of predicate calculus

To assign a truth value to a sentence of the predicate calculus, we have to do a bit more work than for the propositional calculus. We have to define a *model*. The word “model” here means a logical model, which is a very different beast than a computation model! A logical model consists of two parts: a domain of discourse (all possible values of the variables) and a set of relations (where a relation is a set of tuples). Each predicate has a relation, which gives the tuples for which the predicate is true. Among all predicates, equality ( $=$ ) is particularly important. The equality relation will almost always be part of the model. The quantifiers

$\forall x$  (“for all  $x$ ”) and  $\exists x$  (“there exists  $x$ ”) range over the domain of discourse. Usually the logical model is chosen so that a special set of sentences, called the *axioms*, are all true. Such a model is called a *logical semantics* of the axioms. There can be many models for which the axioms are true.

Let us see how this works with an example. Consider the following two axioms:

$$\begin{aligned}\forall x, y. \text{grandfather}(x, y) &\leftrightarrow \exists z. \text{father}(x, z) \wedge \text{father}(z, y) \\ \forall x, y, z. \text{father}(x, z) \wedge \text{father}(y, z) &\rightarrow x = y\end{aligned}$$

There are many possible models of these axioms. Here is one possible model:

Domain of discourse: {george, tom, bill}  
 Father relation: {father(george, tom), father(tom, bill)}  
 Grandfather relation: {grandfather(george, bill)}  
 Equality relation: {george = george, tom = tom, bill = bill}

The relations contain only the true tuples; all other tuples are assumed to be false. With this model, we can give truth values to sentences of predicate calculus. For example, the sentence  $\exists x, y. \text{father}(x, y) \rightarrow \text{father}(y, x)$  can be evaluated as being false. Note that the equality relation is part of this model, even though the axioms might not mention it explicitly.

## Logic programming

Now we can state more precisely what a logic program is. For our purposes, a *logic program* consists of a set of axioms in the first-order predicate calculus, a sentence called the query, and a *theorem prover*, i.e., a system that can perform deduction using the axioms in an attempt to prove or disprove the query. Performing deductions is called *executing* the logic program. Can we build a practical programming system based on the idea of executing logic programs? We still need to address three issues:

- Theoretically, a theorem prover is limited in what it can do. It is only guaranteed to find a proof or disproof for queries that are true in *all* models. If we are only interested in some particular models, then there might not exist a proof or disproof, even though the query is true. We say that the theorem prover is *incomplete*. For example, we might be interested in number theory, so we use the model of integers with integer arithmetic. There is a famous result in mathematics called Gödel’s Incompleteness Theorem, from which it follows that there exist statements of number theory that cannot be proved or disproved within any finite set of axioms.
- Even in those cases where the theorem prover could theoretically find a result, it might be too inefficient. The search for a proof might take exponential time. A theorem prover intended for practical programming should have a simple and predictable operational semantics, so that the programmer can define algorithms and reason about their complexity.

- A final point is that the deduction done by the theorem prover should be *constructive*. That is, if the query states that there exists an  $x$  that satisfies some property, then the system should construct a witness to the existence. In other words, it should build a data structure as an output of the logic program.

Two approaches are taken to overcome these problems:

- We place restrictions on the form of the axioms so that an efficient constructive theorem prover is possible. The Prolog language, for example, is based on *Horn clauses*, which are axioms of the form:

$$\forall x_1, \dots, x_k . \langle a \rangle_1 \wedge \dots \wedge \langle a \rangle_n \rightarrow \langle a \rangle,$$

where  $\{x_1, \dots, x_k\}$  are chosen so that the axiom has no free variables. Horn clauses are interesting because there is an efficient constructive theorem prover for them using an inference rule called *resolution* [114]. The relational computation model of this chapter also does logic programming, but without using resolution. It uses a different set of axioms and theorem prover, which are discussed in the next section.

- We give the programmer the possibility of helping the theorem prover with *operational knowledge*. This operational knowledge is essential for writing efficient logic programs. For example, consider a logic program to sort a list of integers. A naive program might consist of axioms defining a permutation of a list and a query that states that there exists a permutation whose elements are in ascending order. Such a program would be short but inefficient. Much more efficient would be to write axioms that express the properties of an efficient sorting algorithm, such as mergesort.

A major achievement of computer science is that practical logic programming systems have been built by combining these two approaches. The first popular language to achieve this goal was Prolog; it was subsequently followed by many other languages. High-performance Prolog implementations are amazingly fast; they can even rival the speed of imperative language implementations [195].

### 9.3.2 Operational and logical semantics

There are two ways to look at a logic program: the logical view and the operational view. In the logical view, it is simply a statement of logic. In the operational view, it defines an execution on a computer. Before looking at the relational model, let us look first at the declarative model of Chapter 2. We will see that programs in the declarative model have a logical semantics as well as an operational semantics. It is straightforward to translate a declarative program into a logical sentence. If the program terminates correctly, i.e., it does not block, go into an infinite loop, or raise an exception, then all the bindings it does are