```
fun {DFE S}
   case {Ask S}
   of failed then nil
   [] succeeded then [S]
   [] alternatives(2) then C={Clone S} in
      {Commit S 1}
      case {DFE S} of nil then {Commit C 2} {DFE C}
      [] [T] then [T]
      end
   end
end

% Given {Script Sol}, returns solution [Sol] or nil:
fun {DFS Script}
   case {DFE {NewSpace Script}} of nil then nil
   [] [S] then [{Merge S}]
   end
end
```

Figure 12.3: Depth-first single solution search

⟨statement⟩ ::=	`{NewSpace` ⟨x⟩ ⟨y⟩ `}`
	| `{Choose` ⟨x⟩ ⟨y⟩ `}`
	| `{Ask` ⟨x⟩ ⟨y⟩ `}`
	| `{Commit` ⟨x⟩ ⟨y⟩ `}`
	| `{Clone` ⟨x⟩ ⟨y⟩ `}`
	| `{Inject` ⟨x⟩ ⟨y⟩ `}`
	| `{Merge` ⟨x⟩ ⟨y⟩ `}`

Table 12.1: Primitive operations for computation spaces

A depth-first search engine

Figure 12.3 shows how to program depth-first single solution search, in the case of binary choice points. This explores the search tree in depth-first manner and returns the first solution it finds. The problem is defined as a unary procedure `{Script Sol}` that gives a reference to the solution `Sol`, just like the examples of Section 12.2. The solution is returned in a one-element list as `[Sol]`. If there is no solution, then `nil` is returned. In `Script`, choice points are defined with the primitive space operation `Choose`.

The search function uses the primitive operations on spaces `NewSpace`, `Ask`, `Commit`, `Clone`, and `Merge`. We will explain each operation in detail as it comes in the execution. Table 12.1 lists the complete set of primitive operations.

A script example

Let us run the search engine on the example given in Section 12.1.3. The problem was specified by the procedure `Rectangle`.

```
proc {Rectangle ?Sol}
   sol(X Y)=Sol
in
   X::1#9   Y::1#9
   X*Y=:24  X+Y=:10  X=<:Y
   {FD.distribute naive Sol}
end
```

We start the execution with the statement `Sol={DFS Rectangle}`, where `DFS` and `Rectangle` are defined as above, and `Sol` is a fresh variable. If we expand the body of the function, it should create two variables, say `S` and `L`, leading to a configuration like the following. The box represents the thread that executes the statements, and below it is a representation of the store.

```
S={NewSpace Rectangle}
L={DFE S}
Sol=case L of ... end

Rectangle=<proc>   Sol   L   S
```

Space creation

The first primitive space operation we use is `NewSpace`. In our example, it creates a new computation space `S`, with a *root variable* `Root`, and one thread that executes `{Rectangle Root}`. Both the new thread and the new store are shown inside a box, which delimits the "boundaries" of the space.

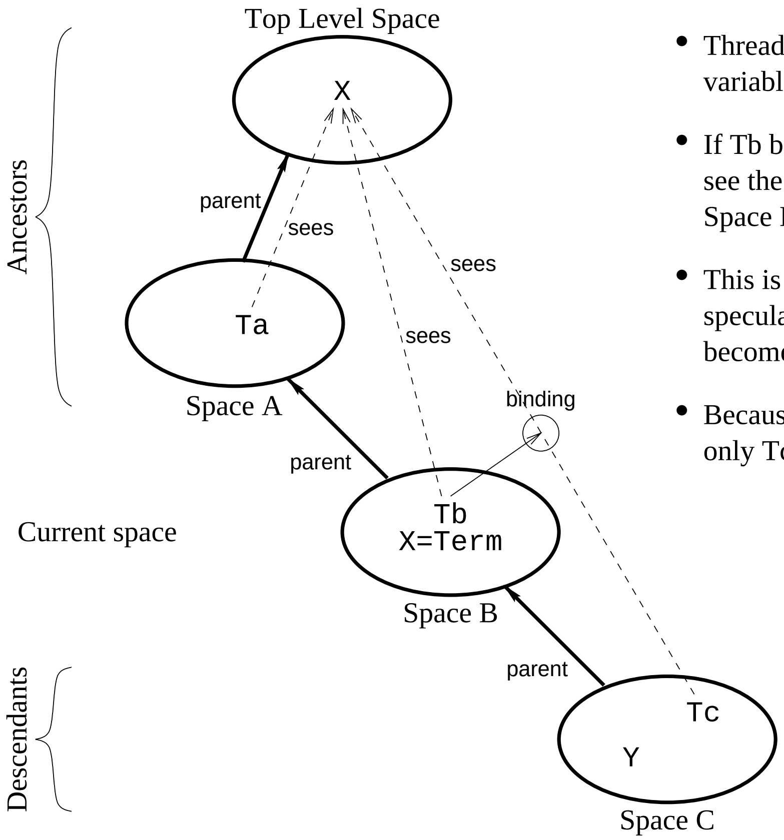
```
L={DFE S}
Sol=case L of ... end

Rectangle=<proc>   Sol   L   S= {Rectangle Root}
                                      Root
```

A precise definition of `NewSpace` is

- `S={NewSpace P}`, when given a unary procedure `P`, creates a new computation space and returns a reference to it. In this space, a fresh variable `R`, called the *root variable*, is created and a new thread, and `{P R}` is invoked in the thread.

Recall that a computation space encapsulates a computation. It is thus an instance of the stateful concurrent model, with its three parts: thread store, constraint store, and mutable store. As it can itself nest a computation space, the spaces naturally form a tree structure:



- Threads Ta, Tb, and Tc all see variable X.
- If Tb binds X then Tb & Tc will see the binding. Ta won't unless Space B is merged into Space A.
- This is because child spaces are speculative: they may or may not become part of their parent store.
- Because Space C is speculative, only Tc sees Y (Ta and Tb don't).

Figure 12.4: Visibility of variables and bindings in nested spaces

- *Tree structure.* There is always a *top level* computation space where threads may interact with the external world. A thread may create a new computation space. The new space is called a *child space.* The current space is the child's *parent space.* At any time, there is a tree of computation spaces in which the top level space is the root. With respect to a given space, a higher one in the tree (closer to the root) is called an *ancestor* and a lower one is called a *descendant.*

- *Threads and variables belong to spaces.* A thread always belongs to exactly one computation space. A variable always belongs to exactly one computation space.

Space execution

Now let us focus on the space `S`. The thread inside is runnable, so we will run it. The reduction of the procedure call `{Rectangle Root}` gives

```
S= local sol(X Y)=Root in
      X::1#9   Y::1#9
      X*Y=:24  X+Y=:10  X=<:Y
      {FD.distribute naive Root}
   end
                 Root
```

You might have noticed that the variable `Rectangle` is bound *outside* the space, which did not prevent the inner thread to read its value and use it. Computation spaces do respect precise visibility rules. Those rules provide a certain degree of isolation from the "external" computation.

- *Variable visibility.* A thread sees and may access variables belonging to its space as well as to all ancestor spaces. The thread cannot see the variables of descendant spaces. Figure 12.4 gives an example with bindings.

- *Basic constraint visibility.* A thread may add basic constraints to variables visible to it. This means that it may constrain variables belonging to its space or to its ancestor spaces. The basic constraint will only be visible in the current space and its descendants. That is, the parent space does not see the binding unless the current space is merged with it (see later).

Posting constraints

The thread inside the space continues its execution. It creates two new variables `X` and `Y` inside the space, and binds `Root` to `sol(X Y)`. This gives

```
   X::1#9   Y::1#9
S= X*Y=:24  X+Y=:10  X=<:Y
   {FD.distribute naive Root}
     Root=sol(X Y)   X   Y
```

It then tells the basic constraints `X::1#9` and `Y::1#9` to the constraint store of the space, and creates new propagators, each one in its own thread. We have

```
S= X*Y=:24  X+Y=:10  X=<:Y  {FD.distribute naive Root}
            Root=sol(X Y)   X::1#9   Y::1#9
```

Concurrent propagation

Now propagators enter the scene. As we have seen in Section 12.1.3, they propagate concurrently, reducing the domains to `4#6`. The space becomes

```
S= X*Y=:24  X+Y=:10  X=<:Y  {FD.distribute naive Root}
            Root=sol(X Y)   X::4#6   Y::4#6
```

Execution in a computation space does a variant of the maximally concurrent model. It avoids the difficulties usually associated with this model. Let us see why this is possible. Each constraint is implemented as a thread (called "propagator") that executes concurrently with the other propagators. Each propagator adds information to the store until no more information can be added. Constraint programming avoids the difficulties of the maximally concurrent model because propagator execution is *monotonic*: they only add information, they never change or remove information. (This is essentially the same reason why concurrent declarative programming is simpler than concurrent stateful programming.) Furthermore, propagators have a logical semantics. All the information they add is consistent with this semantics. If they are written correctly, then the exact order in which they execute does not matter. When they reach a fixpoint (space stability), i.e., when no propagator can add any more information, the result is always the same.

Distribution

The propagators in the space are no longer runnable. At this point, `FD.distribute` becomes runnable. This procedure implements the *distribution strategy*. It picks a variable and a value following a heuristic, in this case `X` and `4`, and proposes a "guess". For this it executes the statement `{Choose 2}`, which creates a choice point with two alternatives, and blocks until a call to `Commit` unblocks it. The interaction between `Choose` and `Commit` is explained in detail later. The whole computation (including the parent space) now looks like

```
              L={DFE S}
              Sol=case L of ... end
            Rectangle=<proc>   Sol   L

       X*Y=:24   case {Choose 2}
                 of 1 then X=4 {FD.distribute [Y]}
S=     X+Y=:10   [] 2 then X\=:4 {FD.distribute Root}
                 end
        X=<:Y
           Root=sol(X Y)   X::4#6   Y::4#6
```

The definition of `Choose` is

- `Y={Choose N}` is the only operation that is called from *inside* the space, while the other operations are called from *outside* the space. It creates a choice point with `N` alternatives. Then it blocks, waiting for an alternative to be chosen by a `Commit` operation on the space. The `Choose` call defines only the *number* of alternatives; it does not specify what to do for any given alternative. `Choose` returns with `Y=I` when alternative `1`≤`I`≤`N` is chosen. A maximum of one choice point may exist in a space at any time.

State of a space

The space was running concurrently with its parent space. The thread of the search engine now executes the statement `L={DFE S}`, which evaluates `{Ask S}`. This operation asks the space for its status. In this case, it returns `alternatives(2)`, meaning that a choice point with two alternatives has been created inside the space. After reduction of the **case** statement, the whole computation becomes

```
local C={Clone S} in
   {Commit S 1}
   L=case {DFE S} of ... end
end
Sol=case L of ... end
```

```
Rectangle=<proc>   Sol   L   S=<space>
```

Here we give a precise definition of the various states of a space. A space is *runnable* if it or a descendant contains a runnable thread, and *blocked* otherwise. Let us run all threads in the space and its descendants, until the space is blocked. Then the space can be in one of the following further states:

- The space is *stable*. This means that no additional basic constraints done in an ancestor can make the space runnable. A stable space can be in four further states:
  - The space is *succeeded*. This means that it contains no choice points. A succeeded space contains a solution to the logic program.
  - The space is *distributable*. This means that the space has one thread that is suspended on a choice point with two or more alternatives. A space can have at most one choice point; attempting to create another gives an error.
  - The space is *failed*. This means that the space attempted to tell inconsistent basic constraints, for instance binding the same variable to two different values. No further execution happens in the space.
  - The space is *merged*. This means that the space has been discarded and its constraint store has been added to a parent. Any further operation on the space is an error. This state is the end of a space's lifetime.
- The space is *suspended*. This means that additional basic constraints done in an ancestor can make the space runnable. Being suspended is usually a temporary condition due to concurrency. It means that some ancestor space has not yet transferred all required information to the space. A space that stays not stable indefinitely usually indicates a programmer error.

The operation `Ask` is then defined as

- `A={Ask S}` asks the space `S` for its status. As soon as the space becomes stable, `A` is bound. If `S` is failed, merged, or succeeded, then `Ask` returns `failed`, `merged`, or `succeeded`. If `S` is distributable, then it returns `alternatives(N)`, where `N` is the number of alternatives.

Cloning a space

The next statement of the search engine thread declares a variable `C`, and creates a *copy* of the space `S`. Note that variables and threads belonging to `S` are copied too, so that both spaces are independent of each other. For the sake of simplicity, we have kept the same identifiers for `S` and `C` in the picture below. But they actually denote different variables in the stores.

```
{Commit S 1}
L=case {DFE S} of ... end
Sol=case L of ... end

Rectangle=<proc>   Sol   L

S=  X*Y=:24    case {Choose 2}
    X+Y=:10    of 1 then X=4 {FD.distribute [Y]}
    X=<:Y      [] 2 then X\=:4 {FD.distribute Root}
               end
          Root=sol(X Y)   X::4#6   Y::4#6

C=  X*Y=:24    case {Choose 2}
    X+Y=:10    of 1 then X=4 {FD.distribute [Y]}
    X=<:Y      [] 2 then X\=:4 {FD.distribute Root}
               end
          Root=sol(X Y)   X::4#6   Y::4#6
```

The definition of `Clone` is

- `C={Clone S}`, if `S` is a stable space, creates an identical copy (a *clone*) of `S` and returns a reference to it. This allows both alternatives of a distributable space to be explored.

Committing to an alternative

The search engine then executes `{Commit S 1}`. This indicates to the space `S` to enter the first alternative. So the call to `Choose` inside the space unblocks and returns `1`. The distributor thread then binds `X` to `4`, which leads to the space

```
S=  X*Y=:24  X+Y=:10  X=<:Y  {FD.distribute [Y]}
          Root=sol(X Y)   X=4   Y::4#6
```

We define `Commit` as

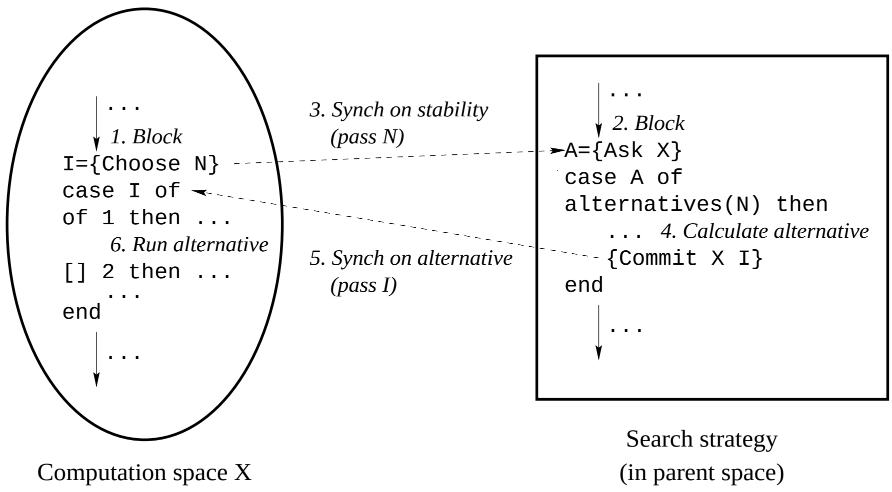


Figure 12.5: Communication between a space and its distribution strategy

- `{Commit S I}`, if `S` is a distributable space, causes the `Choose` call in the space to complete and return `I` as its result. This may cause the space to resume execution. The integer `I` must satisfy `1≤I≤N`, where `N` is the first argument of the `Choose` call.

Now we see precisely how to make the search strategy interact with the distribution strategy. The basic technique is to use `Choose`, `Ask`, and `Commit` to communicate between the inside of a space and the search strategy, which is programmed in the parent space. Figure 12.5 shows how the communication works. Within the space, calling `I={Choose N}` first informs the search strategy of the total number of alternatives (`N`). Then the search strategy picks one (`I`) and informs the space. The synchronization condition between the inside of the space and the search strategy is *stability*, i.e., that there are no more local deductions possible inside the space.

Merging a space

The propagators inside `S` now run until both variables become determined. All the propagators are *entailed* by the store, they simply disappear from `S`:

```
     {FD.distribute [Y]}
S=
   Root=sol(X Y)   X=4   Y=6
```

The distributor thread terminates too, because `Y` is determined, so the whole computation becomes

```
L=case {DFE S} of ... end
Sol=case L of ... end

Rectangle=<proc>   Sol   L   S= Root=sol(X Y)   X=4   Y=6
```

```
   X*Y=:24    case {Choose 2}
              of 1 then X=4 {FD.distribute [Y]}
C= X+Y=:10    [] 2 then X\=:4 {FD.distribute Root}
   X=<:Y      end
       Root=sol(X Y)   X::4#6   Y::4#6
```

The search engine calls again `{DFE S}`, which performs `{Ask S}`. The returned value is now `succeeded`, which means that the computation inside `S` has terminated with a consistent store. The search engine continues its execution. The call to `{DFE S}` then returns `[S]`. The latter matches the second clause in `DFS`, and the search ends with the statement `Sol=[{Merge S}]`. The call `{Merge S}` *merges* `S` with the current space, and returns the root variable of `S`. The computation becomes

```
Rectangle=<proc>   Sol=Root   L=[S]   S=<merged>
           Root=sol(X Y)   X=4   Y=6

   X*Y=:24    case {Choose 2}
              of 1 then X=4 {FD.distribute [Y]}
C= X+Y=:10    [] 2 then X\=:4 {FD.distribute Root}
   X=<:Y      end
       Root=sol(X Y)   X::4#6   Y::4#6
```

Merging a space is necessary to access the solution:

- *Access by merging.* A thread cannot see the variables of a child space, unless the child space is *merged* with its parent. Space merging is an explicit program operation. It causes the child space to disappear and all the child's content to be added to the parent space.

And `Merge` is defined by

- `{Merge S Y}` binds `Y` to the root variable of space `S` and discards the space.

Space failure

Suppose now that the search would continue. This would be the case if the first alternative had no solution. The search engine would then execute `{Commit C 2} L={DFE C}`. The statement `{Commit C 2}` causes `{Choose 2}` to return `2`, which makes the space `C` evolve to

```
C= X*Y=:24  X+Y=:10  X=<:Y  {FD.distribute Root}
       Root=sol(X Y)   X::5#6   Y::4#6
```

As we have seen, the action of the propagators lead to inconsistencies. For instance, `X*Y=:24` propagates the constraints `X=6` and `Y=4`. The propagator `X=<:Y` cannot be satisfied with those values, which makes the space `C` fail:

```
C=<failed>
```

In the search engine, the call to `{Ask C}` would return `failed`. This means that `C` contains no solution. The search would then return `nil` in that case.

Failures, stateful operations, and interaction with the external world are encapsulated in computation spaces in the following way.

- *Exceptions and failures.* A thread that tries to add an inconsistent basic constraint to its constraint store will raise a failure exception. What happens then in the top level space is implementation-dependent. If the exception occurs in a child space and is not caught, then the space fails. A failure happening in a propagator immediately results in its space's failure, because propagators are threads by themselves.
- *Stateful operations.* Operations on stateful entities across spaces are forbidden. For instance, a thread cannot read or change the value of a cell that belongs to its space's parent. A consequence is that only the top level space can interact with the external world.

Injecting a computation into a space

There is one primitive operation that we have not used, namely `Inject`. This operation is however useful, because it permits to add constraints to an existing space. For instance, you can constrain the solution of a space to be "better" than an already known solution. The definition of "better" is problem-dependent, of course. Here is the definition of `Inject`:

- `{Inject S P}` is similar to space creation except that it uses an existing space `S`. It creates a new thread in the space and invokes `{P R}` in the thread, where `R` is the space's root variable. This makes a stable space not stable again. Adding constraints to an existing space is necessary for some distribution strategies such as branch-and-bound and saturation [172, 169].

12.5 Implementing the relational computation model

We end this brief introduction to constraint programming by connecting with the relational computation model of Chapter 9. The relational model extends the declarative model with **`choice`** and **`fail`** statements and with a `Solve` operation to do encapsulated search. We can now show how to program these operations with computation spaces. We have already showed how to do **`fail`**; it remains to implement **`choice`** and `Solve`. Their implementation is independent of the constraint domain. It will work for finite domain constraints. It will also work for the single-assignment store used in the rest of the book, since it is also a constraint system.

12.5.1 The `choice` statement

We can define the **`choice`** statement in terms of the `Choose` operation. The following statement:

```
choice ⟨s⟩1 [] ⟨s⟩2 [] ... [] ⟨s⟩n end
```

is a linguistic abstraction that is defined as follows:

```
case {Choose N}
of 1 then ⟨s⟩1
[] 2 then ⟨s⟩2
...
[] N then ⟨s⟩n
end
```

This creates a choice point and then executes the statement corresponding to the choice made by the search engine.

12.5.2 Implementing the `Solve` function

Figure 12.6 shows the implementation of the `Solve` function. It is an all-solution search engine that uses both computation spaces and laziness. The reader should pay attention to *where* laziness occurs. It is important because of the stateful nature of spaces. For instance, in the **`else`** clause of `SolveLoop`, a clone of `S` must be created *before* any attempt to `Commit` on `S`. Because of the lazy nature of `SolveLoop`, we could actually have declared `C` and `NewTail` in reverse order:

```
...
   NewTail={SolveLoop S I+1 N SolTail}
   C={Space.clone S}
...
```

This works because the value of `NewTail` is not needed before `C` is committed.

12.6 Exercises

1. **Cryptarithmetic**. Write a program to solve all puzzles of the form "*Word1* plus *Word2* equals *Word3*". The words should be input interactively. Use the solution to the *Send+More=Money* problem given in Section 12.2.1 as a guide. The user should be able to stop the search process if it is taking too long. Use the `Solve` function to enumerate the solutions.

```
% Returns the list of solutions of Script given by a lazy
% depth-first exploration
fun {Solve Script}
   {SolveStep {Space.new Script} nil}
end

% Returns the list of solutions of S appended with SolTail
fun {SolveStep S SolTail}
   case {Space.ask S}
   of failed          then SolTail
   [] succeeded       then {Space.merge S}|SolTail
   [] alternatives(N) then {SolveLoop S 1 N SolTail}
   end
end

% Lazily explores the alternatives I through N of space S,
% and returns the list of solutions found, appended with
% SolTail
fun lazy {SolveLoop S I N SolTail}
   if I>N then
      SolTail
   elseif I==N then
      {Space.commit S I}
      {SolveStep S SolTail}
   else
      C={Space.clone S}
      NewTail={SolveLoop S I+1 N SolTail}
   in
      {Space.commit C I}
      {SolveStep C NewTail}
   end
end
```

Figure 12.6: Lazy all-solution search engine `Solve`

Part IV

Semantics

Chapter 13

Language Semantics

"This is the secret meaning of the runes; I hid here magic-runes, undisturbed by evil witchcraft. In misery shall he die by means of magic art who destroys this monument."
– Runic inscription, Björketorp Stone

For all the computation models of the previous chapters, we gave a formal semantics in terms of a simple abstract machine. For the declarative model, this abstract machine contains two main parts: a single-assignment store and a semantic stack. For concurrency, we extended the machine to have multiple semantic stacks. For lazy execution we added a trigger store. For explicit state we added a mutable store. For read-only views we added a read-only store.

This chapter brings all these pieces together. It defines an operational semantics for all the computation models of the previous chapters.[1] We use a different formalism than the abstract machine of the previous chapters. The formalism of this chapter is more compact and easier to reason with than the abstract machine definitions. It has three principal changes with respect to the abstract machine of Chapter 2:

- It uses a concise notation based on reduction rules. The reduction rules follow the abstract syntax, i.e., there are one or more rules for each syntactic construct. This approach is called Structural Operational Semantics, or SOS for short. It was pioneered by Gordon Plotkin [208].

- It uses substitutions instead of environments. We saw that statements, in order to be reducible, must define bindings for their free identifiers. In the abstract machine, these bindings are given by the environment in the semantic statement. In this chapter, the free identifiers are directly substituted by references into the store. We have the invariant that in a reducible statement, all free identifiers have been replaced by store references.

[1] This chapter was co-authored with Raphaël Collet.

- It represents the single-assignment store as a logical formula. This formula is a conjunction of basic constraints, each of which represents a single variable binding. Activation conditions are replaced by logical conditions such as entailment and disentailment.

The chapter is structured as follows:

- Section 13.1 is the main part. It gives the semantics of the shared-state concurrent model.
- Section 13.2 gives a formal definition of declarative concurrency, which is an important property of some subsets of the shared-state concurrent model.
- Section 13.3 explains how subsets of this semantics cover the different computation models of the previous chapters.
- Section 13.4 explains how the semantics covers the different programming abstractions and concepts seen in previous chapters.
- Section 13.5 briefly summarizes the historical development of the shared-state concurrent model and its relative, the message-passing concurrent model.

This chapter is intended to be self-contained. It can be understood independently of the previous chapters. However, its mathematical content is much higher than the previous chapters. To aid understanding, we therefore recommend that you connect it with the abstract machine that was defined before.

13.1 The shared-state concurrent model

This section gives a structural operational semantics for the shared-state concurrent model. We also call this the *general computation model*, since it is the most general model of the book. It covers all the computation models of the book except for the relational and constraint-based models. The semantics of each earlier model, e.g., the declarative, declarative concurrent, and stateful models, can be obtained by taking just the rules for the language constructs that exist in those models. A configuration in the shared-state concurrent model consists of several tasks connected to a shared *store*:

$$\begin{array}{ccc} \textit{task} & \cdots & \textit{task} \\ & \searrow \; \swarrow & \\ & \textit{store} & \end{array}$$

A *task*, also called *thread*, is the basic unit of sequential calculation. A computation consists of a sequence of computation steps, each of which transforms a configuration into another configuration. At each step, a task is chosen among all reducible tasks. The task then does a single reduction step. The execution of the different tasks is therefore interleaved. We say that the model has an *interleaving* semantics. Concurrency is modeled by reasoning about all possible interleavings.

13.1.1 The store

The store consists of two parts: a single-assignment store and a predicate store:

- The *single-assignment store* (also called *constraint store*) contains variables and their bindings. The constraint store is monotonic: variables and bindings can be added, but never changed or removed.

- The *predicate store* contains the additional information that is needed for the execution of certain statements. The predicate store consists of the procedure store (containing procedure values), the mutable store (containing cells), the trigger store (containing by-need triggers), and the read-only store (containing read-only views). Some of these stores are nonmonotonic. These stores are introduced in step-by-step fashion as we define the reduction rules that need them.

All reduction rules are carefully designed so that task reduction is monotonic: once a task is reducible, then it stays reducible even if information is added to the constraint store or the predicate store is changed.

13.1.2 The single-assignment (constraint) store

The constraint store is a repository of information about the program variables. For instance, the store can contain the information "x is bound to 3 and x is equal to y", which is written $x{=}3 \wedge x{=}y$. Such a set of bindings is called a *constraint*. It has a logical semantics, which is explained in Chapter 9. This is why we also call this store the *constraint store*. For this chapter we use just a small part of the logical semantics, namely logical conjunction (adding new information to the store, i.e., doing a binding) and entailment (checking whether some information is in the store).

The constraint store *entails* information. For example, the store $x{=}3 \wedge x{=}y$ entails $y{=}3$, even though that information is not directly present as a binding. We denote the store by σ and we write this as $\sigma \models y{=}3$. We also use another relation called *disentailment*. If β is a constraint, then we say that σ *disentails* β if σ entails the negation of β, i.e., $\sigma \models \neg\beta$. For example, if σ contains $x{=}3$ then it disentails $x{=}4$.

Entailment and disentailment are the general relations we use to *query* the store. They are both forms of logical implication. We assume that the implementation uses an efficient algorithm for checking them. Such an algorithm is given in Section 2.7.2.

The constraint store is *monotonic*, i.e., information can be added but not changed or removed. Consequently, both entailment and disentailment are monotonic too: when the store entails some information or its negation, this stays true forever.[2] The constraint store provides two primitive operations to the program-

[2] Note that "σ disentails β" is *not* the same as "it is not true that σ entails β". The former is monotonic while the latter is not.

mer, called *tell* and *ask*:

- **Tell**. The *tell* operation is a mechanism to add information to the store. A task telling the information β to store σ updates the store to $\sigma \wedge \beta$, **provided** that the new store is consistent. For instance, a task may not tell y=7 to the store x=3 $\wedge$ x=y. It may however tell y=3, which is consistent with the store. An inconsistent tell leaves the store unchanged. It is signaled with some mechanism, typically by raising an exception.

- **Ask**. The *ask* operation is a mechanism to query the store for the presence of some information. A task asking store σ for information β becomes reducible when σ entails either β or its negation $\neg\beta$. For instance, with the store x=3 $\wedge$ x=y, asking for y=3 will give an affirmative answer (the information is present). Asking for y=4 will give a negative answer (the information will never be present). An affirmative answer corresponds to an entailment and a negative answer corresponeds to a disentailment. The task will not reduce until either an affirmative or negative answer is possible. Therefore the *ask* operation is a synchronization mechanism. The task doing the *ask* is said to *synchronize* on β, which is called its *guard*.

Monotonicity of the store implies a strong property: *task reduction is monotonic*. Assume that a task waits for the store to contain some information, i.e., the task becomes reducible when the store entails some information. Then, once the task is reducible, it stays reducible even if other tasks are reduced before it. This is an excellent basis for dataflow concurrency, where tasks synchronize on the availability of data.

13.1.3 Abstract syntax

Figure 13.1 defines the abstract syntax for the kernel language of the shared-state concurrent model. Here S denotes a statement, C, P, X, Y denote variable identifiers, k denotes an integer constant, and n is an integer such that $n \geq 0$. In the *record* $f(l_1{:}X_1 \cdots l_n{:}X_n)$, the *label* f denotes an atom, and each one of the *features* l_i denotes an atom or integer constant. We use $\equiv$ to denote equality between semantic objects, in order to avoid confusion with $=$ in the equality statement.

We assume that in any statement defining a lexical scope for a list of variable identifiers, the identifiers in the list are pairwise distinct. To be precise, in the three statements

$$\texttt{local}\ X_1 \cdots X_n\ \texttt{in}\ S\ \texttt{end}$$
$$\texttt{case}\ X\ \texttt{of}\ f(l_1{:}X_1 \cdots l_n{:}X_n)\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end}$$
$$\texttt{proc}\ \{P\ X_1 \cdots X_n\}\ S\ \texttt{end}$$

we must have $X_i \not\equiv X_j$ for $i \neq j$. We further assume that all identifiers (including X) are distinct in the record tell X=$f(l_1{:}X_1 \cdots l_n{:}X_n)$. These conditions on

S	::=	**skip**	*empty statement*
	|	$S_1\ S_2$	*sequential composition*
	|	**thread** S **end**	*thread introduction*
	|	**local** $X_1 \cdots X_n$ **in** S **end**	*variable introduction* ($n \geq 1$)
	|	X=Y	*imposing equality (tell)*
	|	X=k	
	|	X=$f(l_1{:}X_1 \cdots l_n{:}X_n)$	
	|	**if** X **then** S_1 **else** S_2 **end**	*conditional statements (ask)*
	|	**case** X **of** $f(l_1{:}X_1 \cdots l_n{:}X_n)$ **then** S_1 **else** S_2 **end**	
	|	`{NewName` X`}`	*name introduction*
	|	**proc** `{`$P\ X_1 \cdots X_n$`}` S **end**	*procedural abstraction*
	|	`{`$P\ X_1 \cdots X_n$`}`	
	|	`{IsDet` $X\ Y$`}`	*explicit state*
	|	`{NewCell` $X\ C$`}`	
	|	`{Exchange` $C\ X\ Y$`}`	
	|	`{ByNeed` $P\ X$`}`	*by-need trigger*
	|	Y=`!!`X	*read-only variable*
	|	**try** S_1 **catch** X **then** S_2 **end**	*exception handling*
	|	**raise** X **end**	
	|	`{FailedValue` $X\ Y$`}`	

Figure 13.1: The kernel language with shared-state concurrency

pairwise distinctness are important to ensure that statements are truly primitive, i.e., that there are no hidden tells of the form $X = Y$.

13.1.4 Structural rules

The system advances by successive reduction steps. A possible reduction step is defined by a *reduction rule* of the form

$$\frac{\mathcal{T}}{\sigma} \Bigg\| \frac{\mathcal{T}'}{\sigma'} \text{ if } C$$

stating that the computation makes a transition from a multiset of tasks $\mathcal{T}$ connected to a store σ, to a multiset of tasks $\mathcal{T}'$ connected to a store σ'. We call the pair $\mathcal{T}/\sigma$ a *configuration*. The rule can have an optional boolean condition C, which has to be **true** for the rule to reduce. In this notation, we assume that the left-hand side of a rule (the initial configuration $\mathcal{T}/\sigma$) may have *patterns* and that an empty pattern matches anything. For the rule to reduce, the pattern must be matched in the obvious way.

We use a very light notation for multisets of tasks: the multiset is named by a letter in calligraphic style, disjoint union is denoted by a white space, and singletons are written without curly braces. This allows to write "$T_1\ \mathcal{T}\ T_2$" for $\{T_1\} \uplus \mathcal{T} \uplus \{T_2\}$. Any confusion with a sequence of statements is avoided because of the thread syntax (see later). We generally write "σ" to denote a store, leaving implicit the set of its variables, say $\mathcal{V}$. If need be, we can make the set explicit by writing the store with $\mathcal{V}$ as a subscript: $\sigma_{\mathcal{V}}$.

We use two equivalent notations to express that a rule has the entailment condition $\sigma \models \beta$. The condition can be written as a pattern on the left-hand side or as an explicit condition:

$$\begin{array}{c|c} \mathcal{T} & \mathcal{T}' \\ \hline \sigma \wedge \beta & \sigma \wedge \beta \end{array} \qquad \text{or} \qquad \begin{array}{c|c} \mathcal{T} & \mathcal{T}' \\ \hline \sigma & \sigma \end{array} \text{ if } \sigma \models \beta$$

In the definitions that follow, we use whichever notation is the most convenient.

We assume the semantics has the following two rules, which express model properties that are independent of the kernel language.

$$\begin{array}{c|c} \mathcal{T}\ \mathcal{U} & \mathcal{T}'\ \mathcal{U} \\ \hline \sigma & \sigma' \end{array} \text{ if } \begin{array}{c|c} \mathcal{T} & \mathcal{T}' \\ \hline \sigma & \sigma' \end{array} \qquad\qquad \begin{array}{c|c} \mathcal{T} & \mathcal{T} \\ \hline \sigma & \sigma' \end{array} \text{ if } \sigma \text{ and } \sigma' \text{ are equivalent}$$

The first rule expresses concurrency: a subset of the threads can reduce without directly affecting or depending on the others. The second rule states that the store can be replaced by an equivalent one. The second rule can also be written as

$$\begin{array}{c|c} & \\ \hline \sigma & \sigma' \end{array} \text{ if } \sigma \text{ and } \sigma' \text{ are equivalent}$$

(using an empty pattern instead of $\mathcal{T}$).

Equivalent stores

A store σ consists of a constraint store σ_c and a predicate store σ_p. We denote this as $\sigma = \sigma_c \wedge \sigma_p$. We say that two stores σ and σ' are *equivalent* if (1) their constraint stores entail one another, that is, $\sigma_c \models \sigma'_c$ and $\sigma'_c \models \sigma_c$, and (2) their stores entail the other's predicate store, that is, $\sigma \models \sigma'_p$ and $\sigma' \models \sigma_p$.

We define *entailment* for the predicate store σ_p as follows. We consider σ_p as a multiset of items called *predicates*. A predicate can be considered as a tuple of variables, e.g., $\mathit{trig}(x, y)$ is a predicate. We say that $\sigma \models p'_1 \wedge \cdots \wedge p'_n$ if there exists a subset $\{p_1, \ldots, p_n\}$ of σ_p such that for all i, p_i and p'_i have the same labels and number of arguments, and the corresponding arguments of p_i and p'_i are equal in σ_c. For example, if $\sigma \equiv x{=}x' \wedge \mathit{trig}(x, y)$ then $\sigma \models \mathit{trig}(x', y)$.

This definition of equivalence is a form of *logical equivalence*. It is possible because entailment makes the store independent of its representation: if σ and σ' are equivalent, then $\sigma \models \gamma$ if and only if $\sigma' \models \gamma$.

13.1.5 Sequential and concurrent execution

A thread is a sequence of statements $S_1\ S_2 \cdots S_n$ that we write in a head-tail fashion with angle brackets, i.e., $\langle S_1\ \langle S_2\ \langle \cdots \langle S_n\ \langle\rangle\rangle \cdots \rangle\rangle\rangle$. The abstract syntax of threads is

$$T ::= \langle\rangle \mid \langle S\ T\rangle.$$

A terminated thread has the form $\langle\rangle$. Its reduction simply leads to an empty set of threads. A non-terminated thread has the form $\langle S\ T\rangle$. Its reduction replaces its topmost statement S by its reduction S':

$$\frac{\langle\rangle \;\Big\|\;}{\sigma \;\Big\|\; \sigma} \qquad\qquad \frac{\langle S\ T\rangle \;\Big\|\; \langle S'\ T\rangle}{\sigma \;\Big\|\; \sigma'} \text{ if } \frac{S \;\Big\|\; S'}{\sigma \;\Big\|\; \sigma'}$$

(We extend the reduction rule notation to allow statements in addition to multisets of tasks.) The empty statement, sequential composition, and thread introduction are intimately tied to the notion of thread. Their reduction needs a more specific definition than the one given above for S:

$$\frac{\langle \texttt{skip}\ T\rangle \;\Big\|\; T}{\sigma \;\Big\|\; \sigma} \qquad \frac{\langle (S_1\ S_2)\ T\rangle \;\Big\|\; \langle S_1\ \langle S_2\ T\rangle\rangle}{\sigma \;\Big\|\; \sigma} \qquad \frac{\langle \texttt{thread}\ S\ \texttt{end}\ T\rangle \;\Big\|\; \langle T\rangle\ \langle S\ \langle\rangle\rangle}{\sigma \;\Big\|\; \sigma}$$

The empty statement **skip** is removed from the thread's statement sequence. A sequence $S_1\ S_2$ makes S_1 the thread's first statement, while **thread** S **end** creates a new thread with statement S, that is, $\langle S\ \langle\rangle\rangle$.

13.1.6 Comparison with the abstract machine semantics

Now that we have introduced some reduction rules, let us briefly compare them with the abstract machine. For example, let us consider the semantics of sequential composition. The abstract machine semantics defines sequential composition as follows (taken from Section 2.4):

The semantic statement is

$(\langle \mathsf{s}\rangle_1\ \langle \mathsf{s}\rangle_2, E)$

Execution consists of the following actions:

- Push $(\langle \mathsf{s}\rangle_2, E)$ on the stack.
- Push $(\langle \mathsf{s}\rangle_1, E)$ on the stack.

The reduction rule semantics of this chapter defines sequential composition as follows (taken from the previous section):

$$\frac{\langle (S_1\ S_2)\ T\rangle \;\Big\|\; \langle S_1\ \langle S_2\ T\rangle\rangle}{\sigma \;\Big\|\; \sigma}$$

It pays dividends to compare carefully these two definitions. They say exactly the same thing. Do you see why this is? Let us go over it systematically. In the reduction rule semantics, a thread is given as a sequence of statements. This sequence corresponds exactly to the semantic stack of the abstract machine. The rule for sequential composition transforms the list from $\langle (S_1\ S_2)\ T\rangle$ to $\langle S_1\ \langle S_2\ T\rangle\rangle$. This transformation can be read operationally: first pop $(S_1\ S_2)$ from the list, then push S_2, and finally push S_1.

The reduction rule semantics is nothing other than a precise and compact notation for the English-language definition of the abstract machine with substitutions.

13.1.7 Variable introduction

The **local** statement does variable introduction: it creates new variables in the store and replaces the free identifiers by these variables. We give an example to understand how the **local** statement executes. In the following statement, the identifier `Foo` in S_2 refers to a different variable from the one referred to by `Foo` in S_1 and S_3:

$$\left.\begin{array}{l}\textbf{local}\ \texttt{Foo Bar}\ \textbf{in} \\ \quad S_1 \\ \quad \textbf{local}\ \texttt{Foo}\ \textbf{in}\ S_2\ \textbf{end} \\ \quad S_3 \\ \textbf{end}\end{array}\right\} \equiv S_4$$

The outermost **local** replaces the occurrences of `Foo` in S_1 and S_3 but not those in S_2. This gives the following reduction rule:

$$\frac{\textbf{local}\ X_1 \cdots X_n\ \textbf{in}\ S\ \textbf{end} \quad \Big\| \quad S\{X_1 \rightarrow x_1, \ldots, X_n \rightarrow x_n\}}{\sigma_{\mathcal{V}} \quad \Big\| \quad \sigma_{\mathcal{V} \cup \{x_1, \ldots, x_n\}}} \ \text{if } x_1, \ldots, x_n \text{ fresh variables}$$

In this rule, as in subsequent rules, we use "x" to denote a variable and "X" to denote an identifier. A variable is *fresh* if it is different from all existing variables in the store. So the condition of the rule states that all the variables x_i are distinct and not in $\mathcal{V}$.

The notation $S\{X_1 \rightarrow x_1, \ldots, X_n \rightarrow x_n\}$ stands for the simultaneous substitution of the *free* occurrences of X_1 by x_1, X_2 by x_2, ..., X_n by x_n. For instance, the substitution of `Foo` by x and `Bar` by y in the statement S_4 defined above gives

$$\begin{array}{rcl} S_4\{\texttt{Foo} \rightarrow x, \texttt{Bar} \rightarrow y\} & \equiv & S_1\{\texttt{Foo} \rightarrow x, \texttt{Bar} \rightarrow y\} \\ & & \textbf{local}\ \texttt{Foo}\ \textbf{in}\ S_2\{\texttt{Bar} \rightarrow y\}\ \textbf{end} \\ & & S_3\{\texttt{Foo} \rightarrow x, \texttt{Bar} \rightarrow y\} \end{array}$$

A substitution is actually an environment that is used as a function. Since variables and identifiers are in disjoint sets, the substitution $S\{X_1 \rightarrow x_1, \ldots, X_n \rightarrow x_n\}$ is equivalent to the composition of single substitutions $S\{X_1 \rightarrow x_1\} \cdots \{X_n \rightarrow x_n\}$. The substitution operation $S\{X \rightarrow x\}$ is defined formally in Section 13.1.17.

13.1.8 Imposing equality (tell)

According to Section 13.1.7, a variable introduced by **local** has no initial value. The variable exists but the store simply has no information about it. Adding information about the variable is done by the *tell* operation. Let β denote a statement imposing equality. This statement has three possible forms:

$$\beta ::= x{=}y \mid x{=}z \mid x{=}f(l_1{:}x_1 \cdots l_n{:}x_n).$$

This states that x is equal to either another variable y, an integer or name z, or a record with label f, features (i.e., field names) l_i, and fields x_i. Doing a tell operation adds the information in β to the store, provided that it does not lead to an inconsistent store. This is also called *binding* the variable x.

It is possible that the new information in β conflicts with what the store already knows about x. We say that β is *inconsistent* with σ. This happens whenever $\beta \wedge \sigma \leftrightarrow$ **false**. For example, take $\beta \equiv x{=}10$ and $\sigma \equiv x{=}20$. Instead of adding β to the store, we signal this as an error, e.g., by raising an exception. Therefore the store is always consistent.

In practice, most tell operations are very simple: telling β just binds one variable, x, without binding any others. For example, telling $x{=}23$ where σ has no binding for x. But the tell operation is actually much more general. It can cause many bindings to be done. For example, take $\sigma \equiv x{=}f(x_1\ x_2) \wedge y{=}f(y_1\ y_2)$. Then telling $x = y$ does three bindings: $x{=}y$, $x_1{=}y_1$, and $x_2{=}y_2$.

Naive semantics of tell

The following two rules decide whether to add β to the store.

$$\frac{\beta \;\|\; \texttt{skip}}{\sigma \;\|\; \sigma \wedge \beta} \text{ if } \sigma \wedge \beta \text{ is consistent}$$

$$\frac{\beta \;\|\; \texttt{fail}}{\sigma \;\|\; \sigma} \text{ if } \sigma \wedge \beta \text{ is inconsistent}$$

(Note that β is used to denote both a statement and a constraint.) We could implement tell to follow these rules. However, such an implementation would be complicated and hard to make efficient. The Mozart system uses a slightly more elaborate semantics that can be implemented efficiently. The tell operation is a good example of the trade-off between simple semantics and efficient implementation.

Realistic semantics of tell

We have seen that one tell operation can potentially add many bindings to the store. This generality has an important consequence for inconsistent tells. For example, take $\beta \equiv x{=}y$ and $\sigma \equiv x{=}f(x_1\ x_2) \wedge y{=}f(y_1\ y_2) \wedge x_2{=}\texttt{a} \wedge y_2{=}\texttt{b}$. The tell is inconsistent. Does the tell add $x_1{=}y_1$ to the store? It would be nice if the tell did nothing at all, i.e., σ is unchanged afterwards. This is the naive semantics. But this is very expensive to implement: it means the tell operation would be a *transaction*, which is rolled back if an inconsistency is detected. The system would have to do a transaction for each variable binding. It turns out that implementing tell as a transaction is not necessary. If $\beta \wedge \sigma$ is inconsistent, practical experience shows that it is perfectly reasonable that some bindings remain in place after the inconsistency is detected.

For the semantics of a tell operation we therefore need to distinguish a binding that implies no other bindings (which we call a *basic* binding) and a binding that implies other bindings (which we call a *nonbasic* binding). In the above example, $x{=}y$ is nonbasic and $x_1{=}y_1$ is basic.

Bindings implied by β

To see whether β is a basic binding, we need to determine the extra bindings that happen as part of a tell operation, i.e., the bindings of other variables than x. For a store σ, we write $\beta \xrightarrow{\sigma} \gamma$ to say that the binding β involves the extra binding γ. The relation $\xrightarrow{\sigma}$ is defined as the least *reflexive transitive* relation satisfying

$$\begin{array}{rll} x{=}f(l_1{:}y_1 \cdots l_n{:}y_n) \xrightarrow{\sigma} x_i{=}y_i & \text{if} & \sigma \models x{=}f(l_1{:}x_1 \cdots l_n{:}x_n) \\ x{=}y \xrightarrow{\sigma} x_i{=}y_i & \text{if} & \sigma \models x{=}f(l_1{:}x_1 \cdots l_n{:}x_n) \wedge y{=}f(l_1{:}y_1 \cdots l_n{:}y_n) \end{array}$$

We can now define $\mathit{subbindings}_\sigma(\beta)$, the set of bindings strictly involved by β and not yet entailed by σ, as

$$\mathit{subbindings}_\sigma(\beta) = \left\{ \gamma \;\middle|\; \beta \xrightarrow{\sigma} \gamma \text{ and } \gamma \not\xrightarrow{\sigma} \beta \text{ and } \sigma \not\models \gamma \right\}.$$

Rules for basic bindings

We refine the naive semantics to allow some nonbasic bindings to remain when the tell is inconsistent. We first give the rules for the basic bindings. They decide whether to add β to the store, in the simple case where β just binds one variable.

$$\frac{\beta \;\|\; \texttt{skip}}{\sigma \;\|\; \sigma \wedge \beta} \quad \text{if } \mathit{subbindings}_\sigma(\beta) = \emptyset \text{ and } \sigma \wedge \beta \text{ is consistent}$$

$$\frac{\beta \;\|\; \texttt{fail}}{\sigma \;\|\; \sigma} \quad \text{if } \mathit{subbindings}_\sigma(\beta) = \emptyset \text{ and } \sigma \wedge \beta \text{ is inconsistent}$$

If only basic bindings are done, then these rules are sufficient. In that case, the naive semantics and the realistic semantics coincide. On the other hand, if there are nonbasic bindings, we need one more rule, which is explained next.

Rule for nonbasic bindings

The following rule applies when β involves other bindings. It allows β to be decomposed into basic bindings, which can be told first.

$$\frac{\beta}{\sigma} \Bigg\| \frac{\gamma\ \beta}{\sigma} \quad \text{if } \gamma \in \mathit{subbindings}_{\sigma}(\beta)$$

With the three binding rules, we can now completely explain how a realistic tell operation works. Telling β consists of two parts. If β is basic, then the two basic binding rules explain everything. If β is nonbasic, then the nonbasic binding rule is used to "peel off" basic bindings, until the tell is reduced to basic bindings only. The rule allows basic bindings to be peeled off in any order, so the implementation is free to choose an order that it can handle efficiently.

This rule handles the fact that some bindings may be done even if β is inconsistent with the store. The inconsistency will eventually be noticed by a basic binding, but some previously peeled-off basic bindings may have already been done by then.

13.1.9 Conditional statements (ask)

There is a single conditional statement that does an ask operation, namely the **`if`** statement. The reduction of an **`if`** statement depends on its condition variable:

$$\frac{\texttt{if } x \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ end}}{\sigma \wedge x{=}\texttt{true}} \Bigg\| \frac{S_1}{\sigma \wedge x{=}\texttt{true}}$$

$$\frac{\texttt{if } x \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ end}}{\sigma \wedge x{=}\texttt{false}} \Bigg\| \frac{S_2}{\sigma \wedge x{=}\texttt{false}}$$

This statement synchronizes on the value of the variable x. The first rule applies when the store entails x=**`true`** and the second rule applies when the store entails x=**`false`**. The value of x can be determined by a boolean function, as in x=(y<z) (Section 13.1.11). What happens if x is different from the atoms **`true`** and **`false`** is explained later.

The **`if`** statement only becomes reducible when the store entails sufficient information to decide whether x is **`true`** or **`false`**. If there is not enough information in the store, then neither rule can reduce. The **`if`** statement is said to do *dataflow synchronization*. Because store variables are the basis for dataflow execution, they are called *dataflow variables*.

The `case` statement

The **`case`** statement is a linguistic abstraction for pattern matching that is built on top of **`if`**. Its semantics can be derived from the semantics of **`if`**, **`local`**, and the record operations `Arity` and `Label`. Because pattern matching is such an interesting concept, though, we prefer to give the semantics of **case** directly as reduction rules:

$$\frac{\begin{array}{c}\textbf{case}\ x\ \textbf{of}\ f(l_1{:}X_1 \cdots l_n{:}X_n)\\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}\end{array}}{\sigma \wedge x{=}f(l_1{:}x_1 \cdots l_n{:}x_n)} \Bigg\| \frac{S_1\{X_1{\rightarrow}x_1, \ldots, X_n{\rightarrow}x_n\}}{\sigma \wedge x{=}f(l_1{:}x_1 \cdots l_n{:}x_n)}$$

$$\frac{\begin{array}{c}\textbf{case}\ x\ \textbf{of}\ f(l_1{:}X_1 \cdots l_n{:}X_n)\\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}\end{array}}{\sigma} \Bigg\| \frac{S_2}{\sigma} \quad \begin{array}{l}\text{if } \sigma \models x{\neq}f(l_1{:}x_1 \cdots l_n{:}x_n)\\ \quad\text{for any variables } x_1, \ldots, x_n\end{array}$$

The semantics of pattern matching uses entailment. We say that x *matches* the pattern $f(l_1{:}X_1 \cdots l_n{:}X_n)$ if there exist x_1, ..., x_n such that the store entails $x{=}f(l_1{:}x_1 \cdots l_n{:}x_n)$. If the match is successful, then the **case** statement reduces to S_1 where the identifiers X_i are replaced by the corresponding x_i. This implies that the lexical scope of the X_i covers the whole statement S_1. Otherwise, if we can deduce that the match will never succeed, the **case** reduces to S_2. If there is not enough information to decide one way or another, then neither rule can reduce. This is the dataflow behavior of **case**.

Determined variables and the `Wait` statement

We say that a variable is *determined* if it is bound to an integer, a name, or a record. We say an equality *determines* a variable if it results in the variable becoming determined. We define the predicate $det(x)$ which is entailed by the store when the given variable x is determined.

$$\begin{array}{lrll}\sigma \models det(x) & \text{iff} & \sigma \models x{=}z & \text{for some integer or name } z\\ & \text{or} & \sigma \models x{=}f(l_1{:}x_1 \ldots l_n{:}x_n) & \text{for some } f, l_i, x_i \text{ with } n \geq 0\end{array}$$

It is useful to introduce a statement that blocks until a variable is determined. We call this the `Wait` statement. Its semantics is extremely simple: it reduces to **skip** when its argument is determined.

$$\frac{\texttt{\{Wait}\ x\texttt{\}}}{\sigma} \Bigg\| \frac{\textbf{skip}}{\sigma} \ \text{if } \sigma \models det(x)$$

`Wait` is a form of ask; like the **case** statement it can be defined in terms of **if**:

```
proc {Wait X}
   if X==unit then skip else skip end
end
```

That is, `{Wait X}` waits until it can be decided whether `X` is the same as or different from **unit**. This reduces when *anything definite*, no matter what, is known about `X`.

13.1.10 Names

Names are unforgeable constants, similar to atoms but without a print representation. They are used in the semantics to give a unique identity to procedures and cells (see Sections 13.1.11 and 13.1.12). But their usefulness goes much beyond this semantic role. They behave as first-class *rights*, because they do not have a concrete representation and cannot be forged. A thread cannot guess a name value: a thread can know a name only if it references it via one of its variables. We therefore provide names to the programmer as well as using them in the semantics.

There are just two operations on a name: creation and equality test. A name is equal only to itself. New names can be created at will. We use the metavariable ξ to denote a name, and we extend the equality statement for names:

$$\beta ::= \cdots \mid x{=}\xi.$$

This statement cannot be typed directly by the programmer, but only created indirectly through the `NewName` operation, which creates a new name:

$$\frac{\texttt{\{NewName } x\texttt{\}} \;\Big\|\; x{=}\xi}{\sigma \;\Big\|\; \sigma} \quad \text{if } \xi \text{ fresh name}$$

The `NewName` operation is not needed for the semantics of procedures and cells.

13.1.11 Procedural abstraction

A procedure is created by the execution of a **proc** statement. This puts a *procedure value* **proc** {\$ $X_1 \cdots X_n$} S **end** in the procedure store. This value is almost the same as a λ-expression in the λ-calculus. The difference is a matter of detail: a true λ expression returns a result when applied, whereas a procedure value binds its arguments when applied. This means that a procedure value can return any number of results including none. When the procedure is applied, its procedure value is pushed on the semantic stack and its argument identifiers X_i reference its effective arguments. The procedure value must of course contain no free occurrence of any identifier. This can be proved as a property of the reduction rule semantics.

We associate a procedure to a variable by giving the procedure a *name*. Names are globally unique constants; they were introduced in the previous section. We pair the name ξ with the procedure value, giving ξ:**proc** {\$ $X_1 \cdots X_n$} S **end**, which is put in the procedure store. The procedure store consists of pairs *name*:*value* which define a mapping from names to procedure values. A variable that refers to the procedure is bound to ξ in the constraint store.

$$\frac{\textbf{proc}\ \{x_p\ X_1 \cdots X_n\}\ S\ \textbf{end} \;\Big\|\; x_p{=}\xi}{\sigma \;\Big\|\; \sigma \wedge \xi{:}\textbf{proc}\ \{\$\ X_1 \cdots X_n\}\ S\ \textbf{end}} \quad \text{if } \xi \text{ fresh name}$$

$$\frac{\{x_p\ x_1 \cdots x_n\}}{\sigma \wedge x_p{=}\xi \wedge \xi{:}\textbf{proc}\ \{\$\ X_1 \cdots X_n\}\ S\ \textbf{end}} \Bigg\| \frac{S\{X_1{\rightarrow}x_1, \ldots, X_n{\rightarrow}x_n\}}{\sigma \wedge x_p{=}\xi \wedge \xi{:}\textbf{proc}\ \{\$\ X_1 \cdots X_n\}\ S\ \textbf{end}}$$

It is interesting to see the dataflow behavior of the procedure call. The invocation statement $\{x_p\ x_1 \cdots x_n\}$ synchronizes on the value of x_p. So the procedure can be created in a concurrent thread, provided that no other thread binds x_p to a value.

Where is the contextual environment?

In the abstract machine, a procedure value consists of two parts: the procedure's source definition and a contextual environment that gives its external references. Where does the contextual environment appear in the procedure value ξ:**proc** {$ $X_1 \cdots X_n$} S **end**? It is very simple: the contextual environment appears in the procedure body S. When a **local** statement (or another statement that creates variables) executes, it substitutes identifiers by variables in all the statements that it encompasses, including procedure bodies. Take for example:

```
local Add N in
   N=3
   proc {Add A B} B=A+N end
end
```

When the procedure is defined, it creates the value ξ:**proc** {$ A B} B=A+n **end**, where n is the variable that was substituted for `N`. The contextual environment is $\{n\}$.

Built-in procedures

A practical implementation of the shared-state concurrent model has to define built-in procedures, such as arithmetic operators, comparisons, etc. For instance, the sum operation can be written as $x{=}x_1 + x_2$, which is actually a shorthand for the procedure call {Add x_1 x_2 x} that is defined by

$$\frac{\{\texttt{Add}\ x_1\ x_2\ x\}}{\sigma \wedge x_1{=}k_1 \wedge x_2{=}k_2} \Bigg\| \frac{x{=}k}{\sigma \wedge x_1{=}k_1 \wedge x_2{=}k_2}\ \text{if}\ k = k_1 + k_2$$

Another built-in procedure is the equality test, which is often used in conjunction with an **if** statement. Equality test is the general form of the ask operation defined in Section 13.1.2. It is usually written as a boolean function in infix notation, as in $x{=}(x_1{\texttt{==}}x_2)$ which is shorthand for {Equal x_1 x_2 x}.

$$\frac{\{\texttt{Equal}\ x_1\ x_2\ x\}}{\sigma} \Bigg\| \frac{x{=}\textbf{true}}{\sigma}\ \text{if}\ \sigma \models x_1{=}x_2$$

$$\frac{\{\texttt{Equal}\ x_1\ x_2\ x\}}{\sigma} \Bigg\| \frac{x{=}\textbf{false}}{\sigma}\ \text{if}\ \sigma \models x_1{\neq}x_2$$

An algorithm to implement the `Equal` operation is given in Section 2.7.2. Notice that both `Add` and `Equal` have dataflow behavior.

13.1.12 Explicit state

There are two forms of explicit state in the model. First, there is the boundness check of dataflow variables, which is a weak form of state. Then there are cells, which is a true explicit state. We explain them in turn. The relationship between the two is explored in an exercise.

Boundness check

The boundness check `IsDet` lets us examine whether variables are determined or not, without waiting. This lets us examine the instantaneous status of a dataflow variable. It can be defined with the following rules:

$$\frac{\texttt{\{IsDet}\ x\ y\texttt{\}} \;\Big\|\; y\text{=}\textbf{\texttt{true}}}{\sigma \;\Big\|\; \sigma} \ \text{if } \sigma \models det(x)$$

$$\frac{\texttt{\{IsDet}\ x\ y\texttt{\}} \;\Big\|\; y\text{=}\textbf{\texttt{false}}}{\sigma \;\Big\|\; \sigma} \ \text{if } \sigma \models \neg det(x)$$

The first rule, checking whether x is determined, is similar to the rule for `Wait`. It is the second rule that introduces something new: it allows to give a definite result, y = **`false`**, for a *negative* test. This was not possible up to now. This is the first rule in our semantics that has a nonmonotonic condition, i.e., if the rule is reducible then adding more information to the store can make the rule no longer reducible.

Cells

All the statements introduced up to now define a language that calculates with the constraint store and procedure store, both of which are monotonic. We have now arrived at a point where we need a nonmonotonic store, which we call the *mutable store*. The mutable store contains entities called *cells*, which implement explicit state. This is important for reasons of modularity (see Section 4.7). It greatly increases the model's expressive power, allowing object-oriented programming, for instance. The reverse side of the coin is that reasoning about programs and testing them become harder.

A cell is named in the same way as a procedure: when the cell is created, a fresh name ξ is associated with it. A pair $\xi{:}x$ is put in the mutable store, where the variable x defines the current value of the cell. One changes a cell's value to y by replacing the pair $\xi{:}y$ in the mutable store by $\xi{:}y$. Cells need two primitive operations only, namely cell creation and exchange:

$$\frac{\texttt{\{NewCell}\ x\ x_c\texttt{\}} \;\Big\|\; x_c\text{=}\xi}{\sigma \;\Big\|\; \sigma \wedge \xi{:}x} \ \text{if } \xi \text{ fresh name}$$

$$\frac{\texttt{\{Exchange}\ x_c\ x_{old}\ x_{new}\texttt{\}} \;\Big\|\; x_{old}\text{=}x}{\sigma \wedge x_c\text{=}\xi \wedge \xi{:}x \;\Big\|\; \sigma \wedge x_c\text{=}\xi \wedge \xi{:}x_{new}}$$