Having just one operation to use cells, `Exchange`, is rather minimal. It is often convenient to assume that two other operations exist, namely $x_c$`:=`$x$ (assignment) and $x$`=@`$x_c$ (access). Since we can define them in terms of `Exchange`, no additional rules are needed for them.

It is interesting to see the dataflow behavior of `Exchange`. It blocks until its first argument references a cell. It never blocks on the second or third arguments. This allows it to manipulate the cell's contents even before they are determined.

#### Example of a stream

Using cells and dataflow variables together permits some remarkable programming techniques. We give a small example that uses a stream. Assume that the cell `C` contains the tail of a stream. Then the following statement adds the atom `one` to the stream:

```
local X Old New in
   {Exchange C Old New}
   X=one
   Old=X|New
end
```

The three instructions inside this **`local`** statement can be executed in any order and the final result is exactly the same. What's more, several threads can independently add elements to the stream by each executing this **`local`** statement. The order of the elements on the stream is determined by the order in which the `Exchange` statements are executed.

### 13.1.13 By-need triggers

The by-need trigger is the basic concept used to define demand-driven execution. Its semantics is carefully designed so that the demand-driven concurrent model is still declarative. We define the semantics in two steps. We first define the $need_\sigma(S, x)$ relation that says when statement $S$ "needs" variable $x$. We then define the semantics of the *ByNeed* operation. For this, we add two predicates to the store, namely $need(x)$ and $trig(p, x)$. We can view these as sitting in a new store called the *trigger store*.

The semantics of Section 4.5.1 is correct according to this section, but the semantics of this section is more general (it allows more executions). Section 4.5.1 is more restricted to make it easier to implement.

The by-need semantics of this section is designed so that the demand-driven concurrent model of Chapter 4 is declarative. In particular, the semantics is designed so that the $need(x)$ predicate is monotonic, reduction rules for by-need triggers introduce no nondeterminism, and unification never blocks because of by-need execution.

### The $need_\sigma(S, x)$ relation

The relation $need_\sigma(S, x)$ holds between a statement $S$, a store $\sigma$, and a variable $x$ if and only if three conditions hold:

- No reduction is possible for $S$ with store $\sigma$.
- There exists a constraint $c$ (a set of variable bindings) such that $\sigma \wedge c$ is consistent and a reduction is possible for $S$ with store $\sigma \wedge c$.
- It is true that $\sigma \models \neg det(x)$ and for all constraints $c$ that satisfy the previous condition, we have $\sigma \wedge c \models det(x)$.

The first condition says that $S$ is suspended. The second condition says that $S$ can be made reducible by adding bindings to the store. The third condition says that these added bindings also make $x$ determined, i.e., making $x$ determined is a necessary condition on the added bindings.

### Rules for $need(x)$

We use the $need_\sigma(S, x)$ relation to decide when to add the $need(x)$ predicate to the trigger store. The first rule implements this idea:

$$\frac{S \;\|\; S}{\sigma \;\|\; \sigma \wedge need(x)} \text{ if } need_\sigma(S, x) \text{ and } \sigma \not\models need(x)$$

We need a second rule:

$$\frac{\;\|\;}{\sigma \;\|\; \sigma \wedge need(x)} \text{ if } \sigma \models det(x) \text{ and } \sigma \not\models need(x)$$

This rules says that even if no statement needs $x$, the mere fact of $x$ being determined is enough to make it needed. This ensures that the $need(x)$ predicate is *monotonic*. We can use this fact to show that the demand-driven model is declarative.

### Rules for by-need trigger

The following rule defines the creation of a by-need trigger:

$$\frac{\texttt{\{ByNeed}\ x_p\ x\texttt{\}} \;\|\; \textbf{skip}}{\sigma \;\|\; \sigma \wedge trig(x_p, x)}$$

The following rule defines the activation of a by-need trigger:

$$\frac{\;\|\; \langle \texttt{\{}x_p\ x\texttt{\}}\ \langle\rangle\rangle}{\sigma \wedge trig(x_p, x) \;\|\; \sigma} \text{ if } \sigma \models need(x)$$

These two rules can be seen as a variation of the semantics of **`thread {`**$x_p\ x$**`} end`**, where the existence of $need(x)$ is used to decide whether or not to execute `{`$x_p\ x$`}`. The predicate $trig(x_p, x)$ can be seen as a kind of suspended thread.

### Lazy functions

A lazy function is implemented by attaching a by-need trigger to the variable that will contain the function result. The "`lazy`" annotation is a syntactic short-cut for this technique. Any lazy function, e.g.,

```
fun lazy {F X1 ... Xn} ⟨expr⟩ end
```

behaves as if it were defined by:

```
fun {F X1 ... Xn}
   {ByNeed fun {$} ⟨expr⟩ end}
end
```

When written in full, this becomes:

```
proc {F X1 ... Xn X}
   local P in
      proc {P X} X=⟨expr⟩ end
      {ByNeed P X}
   end
end
```

### The `WaitQuiet` statement

It is possible to define a variation of `Wait`, called `WaitQuiet`, that has a different behavior with by-need execution:

$$\frac{\texttt{\{WaitQuiet } x\texttt{\}} \;\|\; \texttt{skip}}{\sigma \;\|\; \sigma} \text{ if } \sigma \models det(x)$$

This rule is identical with the rule for `Wait`. The difference between the two appears when the variable's value is computed by need. *By definition*, we stipulate that arguments of `WaitQuiet` are not recognized by the $need_\sigma(S, x)$ relation. This means that `Wait` requests the computation of the value, while `WaitQuiet` does not. `WaitQuiet` is used in the Mozart system to implement the Browser.

## 13.1.14 Read-only variables

A read-only variable is a restricted version of a dataflow variable that cannot be made determined by binding it. Any such attempt will block. A read-only variable $y$ is always linked to another variable $x$ that does not have this restriction. When $x$ becomes determined then $y$ is bound to the same partial value. Any blocked bindings of $y$ can then continue.

To define the semantics of read-only variables, we first add the predicate $future(x, y)$ to the store. This states that $y$ is a read-only view of $x$. We can view these predicates as sitting in a new store called the *read-only store*. Once $x$ is

determined, the predicate is removed from the store and replaced by the binding $x{=}y$.

Four rules are needed: one each for the creation and removal of the read-only view, one to block any attempted bindings, and one to handle by-need synchronization. A read-only view is created by the procedure `{ReadOnly` $x$ $x_r$`}`, which binds $x_r$ to a read-only view of $x$. To be compatible with Mozart syntax, which uses the prefix operator "`!!`", we will always write this procedure as a function call $x_r{=}$`!!`$x$,

$$\frac{x_r{=}\texttt{!!}x \;\|\; x_r{=}y}{\sigma \;\|\; \sigma \wedge \mathit{future}(x,y)} \text{ if } x \text{ fresh variable}$$

This creates $y$, a read-only variable for $x$, and a *future* predicate that associates them. The second rule removes the *future* predicate when $x$ is determined.

$$\frac{}{\sigma \wedge \mathit{future}(x,y) \;\|\; \sigma \wedge x{=}y} \text{ if } \sigma \models \mathit{det}(x)$$

A third rule is needed to block any attempt to make $y$ determined by binding it. This rule replaces the first basic binding rule given in Section 13.1.8. It adds one new condition to the basic binding rule.

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

Here $\mathit{prevent}_\sigma(\beta)$ prevents a binding in two cases: (1) the variable to be bound is read-only and would be made determined, and (2) two read-only variables would be bound together. We define it as follows:

$$\begin{aligned}
\mathit{prevent}_\sigma(\beta) &\equiv \mathit{pre1}_\sigma(\beta) \vee \mathit{pre2}_\sigma(\beta) \\
\mathit{pre1}_\sigma(\beta) &\equiv \exists y.\ \sigma \models \mathit{future}(_, y) \text{ and } \sigma \wedge \beta \models \mathit{det}(y) \\
\mathit{pre2}_\sigma(\beta) &\equiv \exists y, y'.\ \sigma \models \mathit{future}(_, y) \wedge \mathit{future}(_, y') \text{ and } \beta \equiv y{=}y'
\end{aligned}$$

A final rule is needed for by-need synchronization. Read-only views are used to protect dataflow variables used in abstractions, but the dataflow variables should still be effective in lazy calculations. This implies that if $y$ is needed, the need should be propagated to $x$.

$$\frac{}{\sigma \;\|\; \sigma \wedge \mathit{need}(x)} \text{ if } \sigma \models \mathit{need}(y) \wedge \mathit{future}(x,y) \text{ and } \sigma \not\models \mathit{need}(x)$$

It is possible to add a "quiet" version of the `!!` operation which is opaque to the need condition. The quiet version would not need this final rule.

### 13.1.15 Exception handling

The exception mechanism is closely bound to sequential composition. Indeed, raising an exception modifies the sequence of statements in the thread where it

has been thrown. It skips every statement inside the scope defined by the most enclosing **try**/**catch** block.

The following rule for the **try**/**catch** statement is a first attempt towards its semantics:

$$\frac{\texttt{try}\ S_1\ \texttt{catch}\ X\ \texttt{then}\ S_2\ \texttt{end} \;\|\; \sigma}{\texttt{try}\ S_1'\ \texttt{catch}\ X\ \texttt{then}\ S_2\ \texttt{end} \;\|\; \sigma'} \text{ if } \frac{S \;\|\; \sigma}{S' \;\|\; \sigma'}$$

It defines the reduction of the nested statement. We then just need to add two rules for the cases where $S_1$ is **skip** and **raise** $x$ **end**. But this definition is not complete: it does not handle thread creation inside the **try**/**catch** block.

So let us try another approach. We "unfold" the **try**/**catch** statement, in order to match the reduction of the nested statement with the usual rules for sequence and thread creation. We say that the statement

$$\texttt{try}\ S_1\ \texttt{catch}\ X\ \texttt{then}\ S_2\ \texttt{end}$$

unfolds to a sequence of two statements, the first one being $S_1$, and the second one a "**catch**" statement:

$$S_1\ (\texttt{catch}\ X\ \texttt{then}\ S_2\ \texttt{end})$$

The new **catch** statement is for semantic use only: it is a marker that stops a **raise** statement exactly at the place it must. The unfolding technique works even when **try**/**catch** blocks are nested. For instance, the statement:

$$\left.\begin{array}{l}\texttt{try}\\ \quad \texttt{try}\ S_1\\ \quad \texttt{catch}\ X\ \texttt{then}\ S_2\ \texttt{end}\\ \quad S_3\\ \texttt{catch}\ Y\ \texttt{then}\ S_4\ \texttt{end}\end{array}\right\} \equiv \begin{array}{l}\text{scope of outer}\\ \textbf{try}/\textbf{catch}\end{array}$$

$$S_5$$

when put in a thread unfolds to:

$$\overbrace{\underbrace{S_1}\langle \texttt{catch}\ X\ \texttt{then}\ S_2\ \texttt{end}\ \langle S_3}^{\begin{array}{c}\text{scope of outer}\\ \textbf{try}/\textbf{catch}\end{array}}\rangle\ \langle \texttt{catch}\ Y\ \texttt{then}\ S_4\ \texttt{end}\ \langle S_5\ \langle\rangle\rangle\rangle\rangle\rangle$$

scope of nested **try**/**catch**

The following two rules define the unfolding of a **try**/**catch** statement and the simplification of a **catch** statement when no exception is raised:

$$\frac{\texttt{try}\ S_1\ \texttt{catch}\ X\ \texttt{then}\ S_2\ \texttt{end} \;\|\; \sigma}{S_1\ (\texttt{catch}\ X\ \texttt{then}\ S_2\ \texttt{end}) \;\|\; \sigma}$$

$$\frac{\texttt{catch}\ X\ \texttt{then}\ S_2\ \texttt{end} \;\|\; \sigma}{\texttt{skip} \;\|\; \sigma}$$

We now define the behavior of a **raise** statement. As we said earlier, it should skip every statement following it, except a **catch** statement. As the following statements reside in the current thread's tail, we must use a "thread-level" reduction:

$$\frac{\langle \textbf{raise}\ x\ \textbf{end}\ \langle S\ T\rangle\rangle \;\|\; \langle \textbf{raise}\ x\ \textbf{end}\ T\rangle}{\sigma \qquad\qquad\quad \sigma} \text{ if } S \not\equiv \textbf{catch}\ldots\textbf{end}$$

$$\frac{\langle \textbf{raise}\ x\ \textbf{end}\ \langle S\ T\rangle\rangle \;\|\; \langle S_2\{X{\rightarrow}x\}\ T\rangle}{\sigma \qquad\qquad\quad \sigma} \text{ if } S \equiv \textbf{catch}\ X\ \textbf{then}\ S_2\ \textbf{end}$$

What happens if the thread's statement sequence is done (i.e., there is only the termination symbol)? The behavior in this case is implementation dependent. The implementation should have a rule like this one:

$$\frac{\langle \textbf{raise}\ x\ \textbf{end}\ \langle\rangle\rangle \;\|\; \ldots}{\sigma \qquad\qquad\quad \ldots}$$

The Mozart system has a rule that halts the process with an error message ("`Uncaught exception`").

### Sources of exceptions

Exceptions can have three origins: explicitly by executing a **raise**, implicitly through a language operation that is impossible, and implicitly through an event external to the system. This section defines the implicit exceptions that come from language operations. Several statements can raise an exception when their reduction will never be possible. The first case is imposing an equality that would lead to an inconsistent store. This means that **fail** is replaced by **raise** in the second basic binding rule:

$$\frac{\beta \;\|\; \textbf{raise}\ \texttt{failure(...)}\ \textbf{end}}{\sigma \;\|\; \sigma} \text{ if } \mathit{subbindings}_\sigma(\beta) = \emptyset \text{ and } \sigma{\wedge}\beta \text{ is inconsistent}$$

where the `...` stands for some debugging information that is not specified here.[3]

The second case is a type inconsistency. This is defined with the following rules. An exception is raised when the condition variable of an **if** statement is not a boolean:

$$\frac{\textbf{if}\ x\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end} \;\|\; \textbf{raise}\ \texttt{error(...)}\ \textbf{end}}{\sigma \qquad\qquad\quad \sigma} \text{ if } \sigma \models \mathit{det}(x) \wedge x \notin \{\textbf{true}, \textbf{false}\}$$

[3]The **raise** statement in this rule is shorthand for **local** `X` **in** `X=failure(...)` **raise** `X` **end end**.

An exception is raised if a procedure application is invoked on something that is not a procedure or is a procedure with a wrong number of arguments:

$$\frac{\{x_p\ x_1 \cdots x_n\} \quad \| \quad \texttt{raise error(...) end}}{\sigma \quad \| \quad \sigma} \text{ if } \sigma \models det(x_p) \wedge (x_p \text{ is not a procedure})$$

$$\frac{\{x_p\ x_1 \cdots x_n\} \quad \| \quad \texttt{raise error(...) end}}{\sigma \quad \| \quad \sigma} \text{ if } \sigma \models x_p{=}\xi \wedge \xi{:}\lambda X_1 \cdots X_m.S \text{ and } m \neq n$$

An exception is raised if `Exchange` is executed on something that is not a cell:

$$\frac{\{\texttt{Exchange}\ x_c\ x_{old}\ x_{new}\} \quad \| \quad \texttt{raise error(...) end}}{\sigma \quad \| \quad \sigma} \text{ if } \sigma \models det(x_c) \wedge (x_c \text{ is not a cell})$$

We can add analogous rules for the built-in procedures.

### 13.1.16 Failed values

The semantics of failed values is defined by four rules. The first rule creates a failed value:

$$\frac{\{\texttt{FailedValue}\ x\ x_f\} \quad \| \quad x_f{=}y}{\sigma \quad \| \quad \sigma \wedge y{=}\mathit{failed}(x)} \text{ if } y \text{ fresh variable}$$

The entity $\mathit{failed}(x)$ represents a failed value that encapsulates the variable $x$. A failed value is not a value, i.e., it is not a member of the set of possible values. It follows that a rule that needs a value to reduce will not reduce with a failed value. However, we allow a failed value to be bound to an unbound variable. This means it can be passed to and from a procedure and it can be embedded in a data structure. The second rule ensures that needing a failed value raises an exception:

$$\frac{S \quad \| \quad \texttt{raise}\ x\ \texttt{end}}{\sigma \quad \| \quad \sigma} \text{ if } \mathit{need}_{\sigma_y}(S, y) \text{ and } \sigma \models y{=}\mathit{failed}(x)$$

Here $\sigma_y = \sigma \setminus \{y{=}\mathit{failed}(x)\}$, i.e., $y$ is unbound in $\sigma_y$. This allows correct calculation of the $\mathit{need}_\sigma$ relation. The third rule handles the case of `IsDet`. This case is not handled correctly by the second rule because it does a test on a variable being *not* determined. We therefore have to handle it separately:

$$\frac{\{\texttt{IsDet}\ x_f\ y\} \quad \| \quad \texttt{raise}\ x\ \texttt{end}}{\sigma \quad \| \quad \sigma} \text{ if } \sigma \models x_f{=}\mathit{failed}(x)$$

This assumes that neither of the rules of Section 13.1.12 will reduce for a failed value. The fourth rule ensures that attempting to bind a failed value to a nonvariable raises an exception:

$$\frac{\beta \quad \| \quad \texttt{raise}\ x\ \texttt{end}}{\sigma \quad \| \quad \sigma} \text{ if } \mathit{subbindings}_\sigma(\beta) = \emptyset \text{ and } \mathit{failconflict}_\sigma(\beta, x)$$

This rule is added to the two basic binding rules. We define the condition $\mathit{failconflict}_\sigma(\beta, x)$ to be true in two cases. First, if $\beta \models \mathit{det}(y)$ and $\sigma \models y{=}\mathit{failed}(x)$. Second, if $\beta \equiv y{=}y'$ and at least one of $y$ or $y'$ is bound to a failed value of the form $\mathit{failed}(x)$ and the other is a failed value or determined.

### 13.1.17 Variable substitution

This section defines the substitution of identifiers by variables in a statement. The notation $S\theta$, where $\theta = \{X_1{\rightarrow}x_1, \ldots, X_n{\rightarrow}x_n\}$, stands for the substitution of $X_1$ by $x_1$, ..., $X_n$ by $x_n$ in the statement $S$. For convenience, we first define substitutions for variables and identifiers. Let $\chi$ denote an identifier or a variable, i.e., $\chi ::= X \mid x$.

$$\chi\theta = \begin{cases} \theta(\chi) & \text{if } \chi \in \mathrm{dom}(\theta) \\ \chi & \text{otherwise} \end{cases}$$

The following substitutions do not involve lexical scoping, so their definition is easy.

$$\begin{aligned}
(\texttt{skip})\theta &\equiv \texttt{skip} \\
(S_1\ S_2)\theta &\equiv S_1\theta\ S_2\theta \\
(\texttt{thread}\ S\ \texttt{end})\theta &\equiv \texttt{thread}\ S\theta\ \texttt{end} \\
(\chi_1{=}\chi_2)\theta &\equiv \chi_1\theta = \chi_2\theta \\
(\chi{=}z)\theta &\equiv \chi\theta = z \\
(\chi{=}f(l_1{:}\chi_1 \cdots l_n{:}\chi_n))\theta &\equiv \chi\theta = f(l_1{:}\chi_1\theta \cdots l_n{:}\chi_n\theta) \\
(\texttt{if}\ \chi\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end})\theta &\equiv \texttt{if}\ \chi\theta\ \texttt{then}\ S_1\theta\ \texttt{else}\ S_2\theta\ \texttt{end} \\
(\{\chi\ \chi_1 \cdots \chi_n\})\theta &\equiv \{\chi\theta\ \chi_1\theta \cdots \chi_n\theta\} \\
(\texttt{raise}\ \chi\ \texttt{end})\theta &\equiv \texttt{raise}\ \chi\theta\ \texttt{end}
\end{aligned}$$

We assume that `NewName`, `IsDet`, `NewCell`, `Exchange`, `ByNeed`, `ReadOnly`, and `FailedValue` are handled by the procedure application case. The remaining substitutions deal with lexical scoping. The notation $\theta_{\{X_1,\ldots,X_n\}}$ stands for the removal of the mappings of $X_1$, ..., $X_n$ from $\theta$, i.e.,

$$\theta_{\{X_1,\ldots,X_n\}} = \left\{ X{\rightarrow}x \in \theta \;\middle|\; X \notin \{X_1, \ldots, X_n\} \right\}.$$

$$\begin{aligned}
(\texttt{local}\ X_1 \cdots X_n\ \texttt{in}\ S\ \texttt{end})\theta &\equiv \texttt{local}\ X_1 \cdots X_n\ \texttt{in}\ S\theta_{\{X_1,\ldots,X_n\}}\ \texttt{end} \\
\begin{pmatrix}\texttt{case}\ \chi\ \texttt{of}\ f(l_1{:}X_1 \ldots l_n{:}X_n) \\ \quad\texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end}\end{pmatrix}\theta &\equiv \begin{array}{l}\texttt{case}\ \chi\theta\ \texttt{of}\ f(l_1{:}X_1 \ldots l_n{:}X_n) \\ \quad\texttt{then}\ S_1\theta_{\{X_1,\ldots,X_n\}}\ \texttt{else}\ S_2\theta\ \texttt{end}\end{array} \\
(\texttt{proc}\ \{\chi\ X_1 \cdots X_n\}\ S\ \texttt{end})\theta &\equiv \texttt{proc}\ \{\chi\theta\ X_1 \cdots X_n\}\ S\theta_{\{X_1,\ldots,X_n\}}\ \texttt{end} \\
(\texttt{try}\ S_1\ \texttt{catch}\ X\ \texttt{then}\ S_2\ \texttt{end})\theta &\equiv \texttt{try}\ S_1\theta\ \texttt{catch}\ X\ \texttt{then}\ S_2\theta_{\{X\}}\ \texttt{end}
\end{aligned}$$

## 13.2 Declarative concurrency

In Section 4.1.4 we gave an informal definition of the concept of declarative concurrency. Let us now make this definition formal. Recall how we define a reduction step:

$$\frac{\mathcal{T} \;\|\; \mathcal{T}'}{\sigma \;\|\; \sigma'}$$

Here $\mathcal{T}$ is a multiset of threads in execution (i.e., statement sequences) and $\sigma$ is a set of bindings (a store). Let us assume for this section that $\sigma$ has no cells. We call $\mathcal{T}$ a *program in execution*, or *program*, for short, if there is no risk of confusion with the meaning of program as a source text.

### Partial and total termination

We say that the configuration $\mathcal{T}/\sigma$ is *partially terminated* if it cannot be further reduced (no reduction rule applies). The termination is partial since adding bindings to $\sigma$ might allow some rules to apply and execution to continue. Although it is not needed for defining declarative concurrency, we can also define *total termination*: no matter what bindings are added to $\sigma$, the configuration cannot be reduced further.

We can also consider failed computations as partially terminated if we introduce the following two reduction rules.

$$\frac{\langle \texttt{raise}\ x\ \texttt{end}\ \langle\rangle\rangle \;\|\;}{\sigma \;\|\; \texttt{false}} \qquad\qquad \frac{\mathcal{T} \;\|\;}{\texttt{false} \;\|\; \texttt{false}}$$

With those rules, any uncaught exception eventually lead to the *failure configuration* $\emptyset/\texttt{false}$.

### Logical equivalence

We define logical equivalence between stores as we did in the beginning of the chapter. We extend this to logical equivalence between configurations. Let $\mathcal{V}$ be a set of variables. Two configurations $\mathcal{T}/\sigma$ and $\mathcal{T}'/\sigma'$ are *logically equivalent with respect to* $\mathcal{V}$ if there exists a bijection $r$ on variables and names such that

- for all $x$ in $\mathcal{V}$, $r(x) = x$,
- $r(\sigma) \equiv \sigma'$ and $\sigma \equiv r^{-1}(\sigma')$,
- $r(\mathcal{T}) = \mathcal{T}'$ and $\mathcal{T} = r^{-1}(\mathcal{T}')$, where $r$ and $r^{-1}$ are used as substitutions.

The mapping $r$ makes the correspondence between variables and names that are not in a common set $\mathcal{V}$.

### Declarative concurrency

We say that a program $\mathcal{T}$ is *declarative concurrent* if for all $\sigma_\mathcal{V}$,

- $\mathcal{T}/\sigma$ always reduces after a finite number of reduction steps to a partially terminated configuration and all these configurations are logically equivalent with respect to $\mathcal{V}$ ;
- for every partial termination $\mathcal{T}'/\sigma'$ of $\mathcal{T}/\sigma$, $\sigma'$ entails $\sigma$ (monotonicity).

Those two statements also hold for failure configurations. The failed store **`false`** entails all the other stores.

In general, we say that a computation model is declarative concurrent if all its programs are declarative concurrent. Intuitively, we can consider a declarative concurrent program as calculating a partial function $b = f_\mathcal{T}(a)$, where $a = \sigma$ and $b = \sigma'$. The function is determined by the program $\mathcal{T}$.

The execution of a declarative concurrent program can always be separated into a sequence of alternating input and output "phases": adding a set of bindings (input phase) and executing until partial termination (output phase).

We can prove that all the declarative concurrent models of Chapter 4 are declarative concurrent according to the above definition. In particular, the most general model (which contains both threads and by-need triggers) is declarative concurrent.

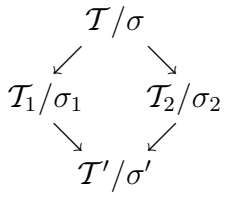
From the viewpoint of foundational calculi, this result means that the declarative concurrent model can be seen as an interesting intermediate step between functional calculi such as the $\lambda$ calculus and process calculi such as the $\pi$ calculus. The $\lambda$ calculus is a model of functional programming. This has nice properties such as confluence (see Section 4.9.2). The $\pi$ calculus is a model of concurrent programming: it is not functional but it is able to express many concurrent computations. The declarative concurrent model is both functional and concurrent. It restricts the expressiveness of concurrency compared to the $\pi$ calculus in such a way that computations become functional again like in the $\lambda$ calculus.

### Confluence property

The above definition of declarative concurrency only considers partial terminations. Nothing is said about infinite executions. Here we propose another way to express declarative concurrency which takes all kinds of computations into account. We use the notation $\mathcal{T}/\sigma \longrightarrow \mathcal{T}'/\sigma'$ to say that there exists a finite execution that begins with configuration $\mathcal{T}/\sigma$ and ends with configuration $\mathcal{T}'/\sigma'$. Partial termination is not required for $\mathcal{T}'/\sigma'$.

A program $\mathcal{T}$ is *declarative concurrent* if for all $\sigma_\mathcal{V}$, and for all executions $\mathcal{T}/\sigma \longrightarrow \mathcal{T}_1/\sigma_1$ and $\mathcal{T}/\sigma \longrightarrow \mathcal{T}_2/\sigma_2$, there exist two further executions $\mathcal{T}_1/\sigma_1 \longrightarrow \mathcal{T}_1'/\sigma_1'$ and $\mathcal{T}_2/\sigma_2 \longrightarrow \mathcal{T}_2'/\sigma_2'$ such that the configurations $\mathcal{T}_1'/\sigma_1'$ and $\mathcal{T}_2'/\sigma_2'$ are equivalent with respect to $\mathcal{V}$.

The property can be depicted by the following diagram, where the configuration $\mathcal{T}'/\sigma'$ is given "up to equivalence with respect to $\mathcal{V}$."




This property is useful for infinite executions. It states that all finite executions of a neverending declarative program must be consistent with each other. For instance, consider a program $\mathcal{T}$ that binds $x$ to an infinite list. If $x$ is bound to `1|2|3|...` during one execution, and to `2|4|6|...` during another execution, then the program is not declarative.

## 13.3 Eight computation models

The previous section gives the semantics of the shared-state concurrent model, which is the most expressive general-purpose model of the book. This semantics is factorized so that the semantics of most of the earlier models are subsets of it. To make these subsets easy to understand, we distinguish three properties: concurrency, state, and laziness. Each of these properties is defined by a part of the semantics:

- **Concurrency** is introduced by the **`thread`** statement. Having concurrency implies that there is a multiset of tasks.
- **State** is introduced by the `NewCell` operation and handled by the `Exchange` operation. Having state implies that there is a mutable store. We assume that having *ports* is equivalent to having state.
- **Laziness** is introduced by the `ByNeed` operation. Having laziness implies that there is a trigger store.

Each of the three properties can be left out of the model by removing its statements. This gives *eight* useful models of varying degrees of expressiveness (!). Table 13.1 lists these eight models. All of these models are practical and most have been used in real programming languages. Table 13.1 also situates a number of real languages with respect to the model that in our opinion best fits the intended use of each language. In this table, C means concurrency, L means laziness, and S means state. An $\times$ means the property is in the model, a blank means it is not.

In the shared-state concurrent model, the three properties are all *explicit*. That is, the programmer controls whether or not they are used by means of explicit commands. This is not true of all the languages mentioned. For example,

| C | L | S | Description |
|---|---|---|---|
| | | | Declarative model (Chapters 2 &3, Mercury, Prolog). |
| | | × | Stateful model (Chapters 6 & 7, Scheme, Standard ML, Pascal). |
| | × | | Lazy declarative model (Haskell). |
| | × | × | Lazy stateful model. |
| × | | | Eager concurrent model (Chapter 4, dataflow). |
| × | | × | Stateful concurrent model (Chapters 5 & 8, Erlang, Java, FCP). |
| × | × | | Lazy concurrent model (Chapter 4, demand-driven dataflow). |
| × | × | × | Stateful concurrent model with laziness (Oz). |

Table 13.1: Eight computation models

laziness is implicit in Haskell and concurrency is implicit in FCP (Flat Concurrent Prolog).

Languages can be based on the same computation model and yet "feel" very differently to the programmer:

- Scheme, Standard ML, and Pascal are all based on the stateful model. Pascal is a simple imperative language. Scheme and Standard ML are "mostly-functional" languages. By "mostly" we mean that state is intended to be used in a limited way.
- Erlang, Java, and FCP are all based on the stateful concurrent model, either of the shared-state variety or of the message-passing variety. Erlang is based on port objects that are programmed in a functional model and communicate with asynchronous message passing. Java is based on passive objects referenced by threads and that communicate through shared monitors. FCP is based on the process model of logic programming, with predicates in Horn clause syntax that communicate through shared streams.

Whether a language is dynamically or statically typed is independent of its place in the table. Scheme, Prolog, Erlang, FCP, and Oz are dynamically typed. Haskell, Standard ML, Mercury, Java, and Pascal are statically typed.

The table does not give the semantics of the relational computation model of Chapter 9 (the declarative model with search). We delay this until we give the semantics of constraint programming in Chapter 12. The logical semantics of Prolog and Mercury are closely related to the relational computation model.

## 13.4 Semantics of common abstractions

We have seen many programming abstractions throughout this book. For example, some of the more general ones are:

- Loop abstractions such as the **`for`** loop.

- Software components (functors) and their instances (modules).
- Stream objects and declarative concurrency.
- Coroutines (non-preemptive threads).
- Lazy functions and list comprehensions.
- Secure abstract data types, wrappers, and revocable capabilities.
- Incremental definition of abstract data types (classes) and their instances (objects).
- Ports (communication channels) and port objects.
- Concurrent components (port objects and their compositions).
- Active objects, both asynchronous and synchronous.
- Active objects with mailboxes (as used in Erlang).
- Locks, reentrant locks, monitors, and transactions.
- Tuple spaces (similar to the Linda concept).

We showed how to implement these abstractions using the shared-state concurrent model or a subset of this model. When taken together with this chapter, these implementations can be seen as formal semantic definitions of the abstractions. The choice of which concepts are primitive and which are derived is often a matter of judgement. For example, Chapter 5 defines a port as a primitive concept and gives its semantics directly.

For some of the abstractions, we have defined new syntax, thus making them into linguistic abstractions. For the semantics, it is almost irrelevant whether or not an abstraction has syntactic support. We say "almost" because the syntax can guarantee that the abstraction is not used in an incorrect way, which is important when reasoning about programs.

## 13.5 Historical notes

The computation model of this chapter was developed over many years. We briefly summarize its history. In the late 1980's, a new model of computation known as the *concurrent constraint model* was developed by Michael Maher and Vijay Saraswat out of concurrent logic programming and constraint logic programming [163, 117, 90]. All computation models of the book are ultimately based on this model.

The concurrent constraint model led to the AKL language [93, 92] and subsequently to Oz 1 [179, 180], a precursor of the language used in this book. AKL

adds stateful data (in the form of ports) and encapsulated search to the basic concurrent constraint model. Oz 1 further adds higher-order procedures, a compositional syntax (instead of the Horn clause syntax of AKL), stateful abstractions including an object system, and computation spaces for encapsulated search. Like AKL, Oz 1 has implicit concurrency: when a statement blocks it is put into its own thread that contains only that statement. The direct successor of Oz 1, called Oz 2, replaces implicit concurrency by explicit thread creation, which allows an improved object system and makes it easier to reason about programs.

The kernel languages used in this book are subsets of Oz 3, which this book calls simply *Oz*. Oz 3 extends and simplifies Oz 2 in many ways. It adds by-need execution (an early version is given in [121]), first-class software components called *functors* [50], and a distributed computation model [72]. It has a simple formal semantics that can be implemented efficiently. The formal semantics of this chapter completes and corrects the semantics given in earlier publications, notably regarding by-need execution and read-only variables.

## 13.6 Exercises

1. **The `case` statement**. Let us investigate the **`case`** statement, whose semantics is defined in Section 13.1.9.

   (a) Show how the semantic rules of **`case`** can be derived from the rules for **`local`** and **`if`**.

   (b) In the first rule for the **`case`**, we could have explicitly introduced variables for the $X_i$ by:

   | | |
   |---|---|
   | **`case`** $x$ **`of`** $f(l_1{:}X_1 \ldots l_n{:}X_n)$ **`then`** $S_1$ **`else`** $S_2$ **`end`** | **`local`** $X_1 \cdots X_n$ **`in`** $X_1{=}x_1 \ \cdots \ X_n{=}x_n \ S_1$ **`end`** |
   | $\sigma \wedge x{=}f(l_1{:}x_1 \ldots l_n{:}x_n)$ | $\sigma \wedge x{=}f(l_1{:}x_1 \ldots l_n{:}x_n)$ |

   Do the rules lead to the same possible executions? What are the differences (if any)?

   (c) It is possible to write an **`if`** statement in terms of a **`case`** statement. How? This implies that **`case`** could have been put in the kernel language instead of **`if`**, and **`if`** could have been defined as a linguistic abstraction.

2. **Lexically-scoped closures**. The rules for procedural abstraction in Section 13.1.11 are designed to follow lexical scoping, i.e., procedure introduction creates a lexically-scoped closure. Let us look more closely to see how this works:

   - Write the consecutive computation steps (rule reductions) for the execution of the `ForAll` and `MakeAdder` definitions in Section 13.1.11.

- Procedure introduction creates the $\lambda$ expression $\xi{:}\lambda X_1 \cdots X_n.S$ in the store. Explain how the contextual environment is stored in this $\lambda$ expression.

3. **Implementing cells with `IsDet`**. Section 13.1.12 explains the `IsDet` operation, which can be used to check the status of a dataflow variable. For this exercise, let us examine the expressive power of `IsDet`.

   - Define the operations `NewCell` and `Exchange` in the declarative model extended with `IsDet`. The semantics of these operations should be identical to their semantics with cells, as given in this chapter. It is straightforward to define a solution, albeit an inefficient one, that works in a sequential model. Hint: use the function `LastCons`, defined as:

     ```
     fun {LastCons Xs}
        case Xs of X|Xr then
           if {IsDet Xr} then {LastCons Xr} else Xs end
        [] nil then nil end
     end
     ```

     Using `LastCons` let us gets around the monotonicity of the store. The idea is to build incrementally a list with unbound tail and use `IsDet` to get its latest known element.

   - Does the above solution work in a concurrent model, i.e., when exchanges on the same cell are done concurrently? Is such a solution possible? In the light of this result, comment on the relationship between `IsDet` and explicit state.

4. **Reading and writing a cell**. Section 13.1.12 mentions that two more cell operations can be added for programming convenience, namely $x_{old}$`=@`$x_c$ to read the content and $x_c$`:=`$x_{new}$ to update the content. Define the semantics of these two operations.

5. **Dataflow streams**. Section 13.1.12 gives an example of a **`local`** statement that adds an element to a stream. Prove that executing two of these statements in different threads always gives exactly the same final result as if they were executed sequentially in the same thread in one order or another.

6. **Stateful streams**. Define a stream datatype that does not use dataflow variables. That is, it is a list in which each tail is a cell whose content points to the rest of the list. The last cell contains a marker saying that the stream is not yet complete, e.g., the atom `incomplete`. (This is not the same as the atom `nil` which means that the stream is complete.) There is a global cell `C` whose contents is always the last cell in the stream. Write an operation that adds an element to the stream and that works in a concurrent setting.

*Hint:* assume that there exists a statement **lock** $S$ **end** such that only one thread at a time can be executing $S$; all others suspend if needed to make this true. Can you do it without using a **lock** statement? Compare your solution to that of the previous exercise. Which is simpler?

7. **Needing a variable**. Section 13.1.13 gives a definition of what it means to *need* a variable. Because this need relation is monotonic, we can show that the demand-driven concurrent model is declarative. However, there are other ways to define the need relation that also result in declarative models. For this exercise, try to find at least one such definition.

8. **Exceptions with a finally clause**. Section 13.1.15 defines the **try**/**catch** statement:

   **try** $S_1$ **catch** $X$ **then** $S_2$ **end**

   which executes $S_2$ if an exception is raised in $S_1$. Another, very useful statement relating to exceptions is the **try**/**finally** statement:

   **try** $S_1$ **finally** $S_2$ **end**

   which *always* executes $S_2$, whether or not $S_1$ raises an exception. If $S_1$ raises an exception, then this exception is raised again after executing $S_2$. Define the **try**/**finally** statement in terms of the **try**/**catch** statement.

9. *(advanced exercise)* **Lambda calculus**. The book claims that the declarative model and the declarative concurrent model both do functional programming. For this exercise, prove this claim formally. First show that any execution in the declarative model corresponds to an execution in a version of the $\lambda$ calculus. How do dataflow variables show up? Then show that adding concurrency and laziness do not change this result.

10. *(research project)* **Trade-offs in language design**. When designing a language, there are often trade-offs between the programmer's mental model, the language semantics, and the implementation. One would like all three to be simple, but that is often impossible. One of the delicate matters is to find the right balance. To make this concrete, let us see how to provide the concept of binding in a dataflow language. Section 13.1.8 defines two semantics for binding, which it calls the naive semantics and the realistic semantics. There is also a third possibility. Let us summarize all three:

    - The naive semantics does binding atomically, as a transaction. If adding $\beta$ would be inconsistent, then the store is unchanged. This gives a simple mental model and a simple semantics, but the implementation is complex. This semantics was much discussed in the context of concurrent logic programming, but was dropped because of problems implementing it efficiently [177, 190].

- The realistic semantics does binding as an incremental tell. That is, if $\beta$ is inconsistent, then the store might still be changed. This makes the implementation simple, but the semantics somewhat more complex. Experience shows that the mental model is acceptable. This semantics is chosen for the computation models of this book.
- The third semantics is more in line with mainstream programming languages. It jettisons unification in favor of simple bindings only. It allows binding only unbound variables with terms. Variable-variable bindings block and term-term bindings raise an exception. This makes both the implementation and the semantics simple. However, it is less expressive for the programmer. This approach was pioneered, e.g., in dataflow languages with I-structures such as Id and pH [131, 132, 133] and in Multilisp [68] (see Section 4.9.4).

For this exercise, reexamine the trade-offs between these three approaches. Which would you recommend?

# Part V

# Appendices

# Appendix A

# Mozart System Development Environment

"Beware the ides of March."
– Soothsayer to Julius Caesar, *William Shakespeare* (1564–1616)

The Mozart system used in this book has a complete IDE (Interactive Development Environment). To get you started, we give a brief overview of this environment here. We refer you to the system documentation for additional information.

## A.1 Interactive interface

The Mozart system has an interactive interface that is based on the Emacs text editor. The interfactive interface is sometimes called the OPI, which stands for Oz Programming Interface. The OPI is split into several buffers: scratch pad, Oz emulator, Oz compiler, and one buffer for each open file. This interface gives access to several tools: incremental compiler (which can compile any legal program fragment), Browser (visualize the single-assignment store), Panel (resource usage), Compiler Panel (compiler settings and environment), Distribution Panel (distribution subsystem including message traffic), and the Explorer (interactive graphical resolution of constraint problems). These tools can also be manipulated from within programs, e.g., the `Compiler` module allows to compile strings from within programs.

### A.1.1 Interface commands

You can access all the important OPI commands through the menus at the top of the window. Most of these commands have keyboard equivalents. We give the most important ones:

| Command | Effect |
|---|---|
| `CTRL-x CTRL-f` | Read a file into a new editor buffer |
| `CTRL-x CTRL-s` | Save current buffer into its file |
| `CTRL-x i` | Insert file into the current buffer |
| `CTRL-. CTRL-l` | Feed current line into Mozart |
| `CTRL-. CTRL-r` | Feed current selected region into Mozart |
| `CTRL-. CTRL-p` | Feed current paragraph into Mozart |
| `CTRL-. CTRL-b` | Feed current buffer into Mozart |
| `CTRL-. h` | Halt the run-time system (but keep the editor) |
| `CTRL-x CTRL-c` | Halt the complete system |
| `CTRL-. e` | Toggle the emulator window |
| `CTRL-. c` | Toggle the compiler window |
| `CTRL-x 1` | Make current buffer fill the whole window |
| `CTRL-g` | Cancel current command |

The notation "`CTRL-x`" means to hold down the Control key and then press the key `x` once. The `CTRL-g` command is especially useful if you get lost. To *feed* a text means to compile and execute it. A *region* is a contiguous part of the buffer. It can be selected by dragging over it while holding the first mouse button down. A *paragraph* is a set of non-empty text lines delimited by empty lines or by the beginning or end of the buffer.

The emulator window gives messages from the emulator. It gives the output of `Show` and run-time error messages, e.g., uncaught exceptions. The compiler window gives messages from the compiler. It says whether fed source code is accepted by the system and gives compile-time error messages otherwise.

### A.1.2 Using functors interactively

Functors are software component specifications that aid in building well-structured programs. A functor can be instantiated, which creates a module. A module is a run-time entity that groups together any other run-time entities. Modules can contain records, procedures, objects, classes, running threads, and any other entity that exists at run-time.

Functors are compilation units, i.e., their source code can be put in a file and compiled as one unit. Functors can also be used in the interactive interface. This follows the Mozart principle that everything can be done interactively.

- A compiled functor can be loaded interactively. For example, assume that the `Set` module, which can be found on the book's Web site, is compiled in file `Set.ozf`. It will be loaded interactively with the following code:

  ```
  declare
  [Set]={Module.link ["Set.ozf"]}
  ```

  This creates the module `Set`. Other functor manipulations are possible by using the module `Module`.

- A functor is simply a value, like a class. It can be defined interactively with a syntax similar to classes:

  ```
  F=functor $ define skip end
  ```

  This defines a functor and binds `F` to it.

## A.2 Batch interface

The Mozart system can be used from a command line. Oz source files can be compiled and linked. Source files to compile should contain functors, i.e., start with the keyword **`functor`**. For example, assume that we have the source file `Set.oz`, which is available on the book's Web site. We create the compiled functor `Set.ozf` by typing the following command from a command line interface:

```
ozc -c Set.oz
```

We can create a standalone executable `Set` by typing the following:

```
ozc -x Set.oz
```

(In the case of `Set.oz`, the standalone executable does very little: it just defines the set operations.) The Mozart default is to use dynamic linking, i.e., needed modules are loaded and linked at the moment they are needed in an application. This keeps compiled files small. But it is possible to link all imported modules during compilation (static linking) so that no dynamic linking is needed.

# Appendix B

# Basic Data Types

"Wie het kleine niet eert is het grote niet weert."
"*He who does not honor small things is not worthy of great things.*"
– Traditional Dutch proverb.

This appendix explains the most common basic data types in Oz together with some common operations. The types explained are numbers (including integers and floating point numbers), characters (which are represented as small integers), literals (constants of two types, either atoms or names), records, tuples, chunks (records with a limited set of operations), lists, strings (which are represented as lists of characters), and virtual strings (strings represented as tuples).

For each data type discussed in this appendix, there is a corresponding Base module in the Mozart system that defines all operations on the data type. This appendix gives some but not all of these operations. See the Mozart system documentation for complete information [49].

## B.1 Numbers (integers, floats, and characters)

The following code fragment introduces four variables `I`, `H`, `F` and `C`. It binds `I` to an integer, `H` to an integer in hexadecimal notation, `F` to a float, and `C` to the character `t` in this order. It then displays `I`, `H`, `F`, and `C`:

```
declare I H F C in
I = ~5
H = 0xDadBeddedABadBadBabe
F = 5.5
C = &t
{Browse I} {Browse H} {Browse F} {Browse C}
```

Note that ~ (tilde) is the unary minus symbol. This displays the following:

```
~5
1033532870595452951444158
5.5
```

| | | |
|---|---|---|
| ⟨character⟩ | ::= | (any integer in the range 0 ... 255) |
| | | \| ´&´ ⟨charChar⟩ |
| | | \| ´&´ ⟨pseudoChar⟩ |
| ⟨charChar⟩ | ::= | (any inline character except \ and NUL) |
| ⟨pseudoChar⟩ | ::= | ('\' followed by three octal digits) |
| | | \| (´\x´ or ´\X´ followed by two hexadecimal digits) |
| | | \| ´\a´ \| ´\b´ \| ´\f´ \| ´\n´ \| ´\r´ \| ´\t´ |
| | | \| ´\v´ \| ´\\´ \| ´\´´ \| ´\"´ \| ´\`´ \| ´\&´ |

Table B.1: Character lexical syntax

```
116
```

Oz supports binary, octal, decimal, and hexadecimal notation for integers, which can have any number of digits. An octal integer starts with a leading `0` (zero), followed by any number of digits from `0` to `7`. A binary integer starts with a leading `0b` or `0B` (zero followed by the letter `b` or `B`) followed by any number of binary digits, i.e., `0` or `1`. A hexadecimal integer starts with a leading `0x` or `0X` (zero followed by the letter `x` or `X`). The hexadecimal digits from 10 to 15 are denoted by the letters `a` through `f` and `A` through `F`.

Floats are different from integers in that they approximate real numbers. Here are some examples of floats:

```
~3.14159265359  3.5E3  ~12.0e~2  163.
```

Note that Mozart uses ~ (tilde) as the unary minus symbol for floats as well as integers. Floats are internally represented in double precision (64 bits) using the IEEE floating point standard. A float must be written with a decimal point and at least one digit before the decimal point. There may be zero or more digits after the decimal point. Floats can be scaled by powers of ten by appending the letter `e` or `E` followed by a decimal integer (which can be negative with a ´~´).

Characters are a subtype of integers that range from `0` to `255`. The standard ISO 8859-1 coding is used. This code extends the ASCII code to include the letters and accented letters of most languages whose alphabets are based on the Roman alphabet. Unicode is a 16-bit code that extends the ASCII code to include the characters and writing specifics (like writing direction) of most of the alphabets used in the world. It is not currently used, but may be in the future. There are five ways to write characters:

- A character can be written as an integer in the range `0`, `1`, ..., `255`, in accord with the integer syntax given before.
- A character can be written as an ampersand `&` followed by a specific character representation. There are four such representations:
  - Any inline character except for \ (backslash) and the `NUL` character.

| | | |
|---|---|---|
| ⟨expression⟩ | ::= | ⟨expression⟩ ⟨binaryOp⟩ ⟨expression⟩ |
| | \| | ´{´ ⟨expression⟩ { ⟨expression⟩ } ´}´ |
| | \| | ... |
| ⟨binaryOp⟩ | ::= | ´+´ \| ´-´ \| ´*´ \| ´/´ \| **div** \| **mod** \| ... |

Table B.2: Some number operations

Some examples are `&t`, `&` (note the space), and `&+`. Inline control characters are acceptable.

- A backslash \ followed by three octal digits, e.g., `&\215` is a character. The first digit should not be greater than 3.
- A backslash \ followed by the letter `x` or `X`, followed by two hexadecimal digits, e.g., `&\x3f` is a character.
- A backslash \ followed by one of the following characters: `a` (= `\007`, bell), `b` (= `\010`, backspace), `f` (= `\014`, formfeed), `n` (= `\012`, newline), `r` (= `\015`, carriage return), `t` (= `\011`, horizontal tab), `v` (= `\013`, vertical tab), `\` (= `\134`, backslash), `'` (= `\047`, single quote), `"` (= `\042`, double quote), `` ` `` (= `\140`, backquote), and `&` (= `\046`, ampersand). For example, `&\\` is the backslash character, i.e., the integer 92 (the ASCII code for \) .

Table B.1 summarizes these possibilities.

There is no automatic type conversion in Oz, so `5.0 = 5` will raise an exception. The next section explains the basic operations on numbers, including the primitive procedures for explicit type conversion. The complete set of operations for characters, integers, and floats are given in the Base modules `Char`, `Float`, and `Int`. Additional generic operations on all numbers are given in the Base module `Number`. See the documentation for more information.

### B.1.1 Operations on numbers

To express a calculation with numbers, we use two kinds of operations: binary operations, such as addition and subtraction, and function applications, such as type conversions. Table B.2 gives the syntax of these expressions. All numbers, i.e., both integers and floats, support addition, subtraction, and multiplication:

```
declare I Pi Radius Circumference in
I = 7 * 11 * 13 + 27 * 37
Pi = 3.1415926536
Radius = 10.
Circumference = 2.0 * Pi * Radius
```

Integer arithmetic is to arbitrary precision. Float arithmetic has a fixed precision. Integers support integer division (**div** symbol) and modulo (**mod** symbol). Floats

| Operation | Description |
|---|---|
| `{IsChar C}` | Return boolean saying whether `C` is a character |
| `{Char.toAtom C}` | Return atom corresponding to `C` |
| `{Char.toLower C}` | Return lowercase letter corresponding to `C` |
| `{Char.toUpper C}` | Return uppercase letter corresponding to `C` |

Table B.3: Some character operations

support floating division (`/` symbol). Integer division truncates the fractional part. Integer division and modulo satisfy the following identity:

```
A = B * (A div B) + (A mod B)
```

There are several operations to convert between floats and integers.

- There is one operation to convert from an integer to a float, namely `IntToFloat`. This operation finds the best float approximation to a given integer. Because integers are calculated with arbitrary precision, it is possible for an integer to be larger than a representable float. In that case, the float `inf` (infinity) is returned.

- There is one operation to convert from a float to an integer, namely `FloatToInt`. This operation follows the default rounding mode of the IEEE floating point standard, i.e., if there are two possibilities, then it picks the even integer. For example, `{FloatToInt 2.5}` and `{FloatToInt 1.5}` both give the integer `2`. This eliminates the bias that would result by always rounding half integers upwards.

- There are three operations to convert a float into a float that has zero fractional part: `Floor`, `Ceil` (ceiling), and `Round`.

  - `Floor` rounds towards negative infinity, e.g., `{Floor ~3.5}` gives `~4.0` and `{Floor 4.6}` gives `4.0`.
  - `Ceil` rounds towards positive infinity, e.g., `{Ceil ~3.5}` gives `~3.0` and `{Ceil 4.6}` gives `5.0`.
  - `Round` rounds towards the nearest even, e.g., `{Round 4.5}=4` and `{Round 5.5}=6`. `Round` is identical to `FloatToInt` except that it returns a float, i.e., `{Round X} = {IntToFloat {FloatToInt X}}`.

### B.1.2 Operations on characters

All integer operations also work for characters. There are a few additional operations that work only on characters. Table B.3 lists some of them. The Base module `Char` gives them all.

| ⟨expression⟩ ::= **unit** \| **true** \| **false** \| ⟨atom⟩ \| ... |
|---|

Table B.4: Literal syntax *(in part)*

| | |
|---|---|
| ⟨atom⟩ | ::= (lowercase char) { (alphanumeric char) } (except no keyword) |
| | \| ''' { ⟨atomChar⟩ \| ⟨pseudoChar⟩ } ''' |
| ⟨atomChar⟩ | ::= (any inline character except ', \, and NUL) |
| ⟨pseudoChar⟩ | ::= ('\' followed by three octal digits) |
| | \| (´\x´ or ´\X´ followed by two hexadecimal digits) |
| | \| ´\a´ \| ´\b´ \| ´\f´ \| ´\n´ \| ´\r´ \| ´\t´ |
| | \| ´\v´ \| ´\\´ \| ´\´´ \| ´\"´ \| ´\`´ \| ´\&´ |

Table B.5: Atom lexical syntax

## B.2 Literals (atoms and names)

Atomic types are types whose members have no internal structure.[1] The previous section has given one kind of atomic type, namely numbers. In addition to numbers, literals are a second kind of atomic type (see Table B.4 and Table B.5). Literals can be either atoms or names. An *atom* is a value whose identity is determined by a sequence of printable characters. An atom can be written in two ways. First, as a sequence of alphanumeric characters starting with a lowercase letter. This sequence may not be a keyword of the language. Second, by arbitrary printable characters enclosed in single quotes. Here are some valid atoms:

```
a foo ´=´ ´:=´ ´Oz 3.0´ ´Hello World´ ´if´ ´\n,\n ´ a_person
```

There is no confusion between the keyword **if** and the atom ´if´ because of the quotes. The atom ´\n,\n ´ consists of four characters. Atoms are ordered lexicographically, based on the underlying ISO 8859-1 encoding for single characters.

Names are a second kind of literal. A *name* is a unique atomic value that cannot be forged or printed. Unlike numbers or atoms, names are truly atomic, in the original sense of the word: they cannot be decomposed at all. Names have just two operations defined on them: creation and equality comparison. The only way to create a name is by calling the function {NewName}, which returns a new name that is guaranteed to be unique. Note that Table B.4 has no representation for names. The only way to reference a name is through a variable that is bound to the name. As Chapter 3 explains, names play an important role for secure encapsulation in ADTs.

[1] But like physical atoms, atomic values can sometimes be decomposed if the right tools are used, e.g., numbers have a binary representation as a sequence of zeroes and ones and atoms have a print representation as a sequence of characters.

| Operation | Description |
|---|---|
| `{IsAtom A}` | Return boolean saying whether `A` is an atom |
| `{AtomToString A}` | Return string corresponding to atom `A` |
| `{StringToAtom S}` | Return atom corresponding to string `S` |

Table B.6: Some atom operations

| | | |
|---|---|---|
| ⟨expression⟩ | ::= | ⟨label⟩ ´(´ { [ ⟨feature⟩ ´:´ ] ⟨expression⟩ } ´)´ \| ... |
| ⟨label⟩ | ::= | **unit** \| **true** \| **false** \| ⟨variable⟩ \| ⟨atom⟩ |
| ⟨feature⟩ | ::= | **unit** \| **true** \| **false** \| ⟨variable⟩ \| ⟨atom⟩ \| ⟨int⟩ |
| ⟨binaryOp⟩ | ::= | ´.´ \| ⟨consBinOp⟩ \| ... |
| ⟨consBinOp⟩ | ::= | ´#´ \| ... |

Table B.7: Record and tuple syntax (*in part*)

There are three special names that have keywords reserved to them. The keywords are **unit**, **true**, and **false**. The names **true** and **false** are used to denote boolean true and false values. The name **unit** is often used as a synchronization token in concurrent programs. Here are some examples:

```
local X Y B in
   X = foo
   {NewName Y}
   B = true
   {Browse [X Y B]}
end
```

### B.2.1 Operations on atoms

Table B.6 gives the operations in the Base module `Atom` and some of the operations relating to atoms in the Base module `String`.

## B.3 Records and tuples

Records are data structures that allow to group together language references. Here is a record that groups four variables:

```
tree(key:I value:Y left:LT right:RT)
```

It has four components and the label `tree`. To avoid ambiguity, there should be no space between the label and the left parenthesis. Each component consists of an identifier, called *feature*, and a reference into the store. A feature can be either a literal or an integer. Table B.7 gives the syntax of records and tuples.

The above record has four features, `key`, `value`, `left`, and `right`, that identify four language references, `I`, `Y`, `LT`, and `RT`.

It is allowed to omit features in the record syntax. In that case, the feature will be an integer starting from `1` for the first such component and incrementing by `1` for each successive component that does not have a feature. For example, the record `tree(key:I value:Y LT RT)` is identical to `tree(key:I value:Y 1:LT 2:RT)`.

The order of labeled components does not matter; it can be changed without changing the record. We say that these components are unordered. The order of unlabeled components does matter; it determines how the features are numbered. It is as if there were two "worlds": the ordered world and the unordered world. They have no effect on each other and can be interleaved in any way. All the following notations denote the same record:

```
tree(key:I value:Y LT RT)    tree(value:Y key:I LT RT)
tree(key:I LT value:Y RT)    tree(value:Y LT key:I RT)
tree(key:I LT RT value:Y)    tree(value:Y LT RT key:I)
tree(LT key:I value:Y RT)    tree(LT value:Y key:I RT)
tree(LT key:I RT value:Y)    tree(LT value:Y RT key:I)
tree(LT RT key:I value:Y)    tree(LT RT value:Y key:I)
```

Two records are the same if the same set of components is present and the ordered components are in the same order.

It is an error if a feature occurs more than once. For example, the notations `tree(key:I key:J)` and `tree(1:I value:Y LT RT)` are both in error. The error is discovered when the record is constructed. This can be either at compile time or at run time. However, both `tree(3:I value:Y LT RT)` and `tree(4:I value:Y LT RT)` are correct since no feature occurs more than once. Integer features do not have to be consecutive.

### B.3.1 Tuples

If the record has only consecutive integer features starting from `1`, then we call it a *tuple*. All these features can be omitted. Consider this tuple:

```
tree(I Y LT RT)
```

It is exactly the same as the following tuple:

```
tree(1:I 2:Y 3:LT 4:RT)
```

Tuples whose label is `´#´` have another notation using the `#` symbol as an "mixfix" operator (see Appendix C.4). This means that `a#b#c` is a tuple with three arguments, namely `´#´(a b c)`. Be careful not to confuse it with the pair `a#(b#c)`, whose second argument is itself the pair `b#c`. The mixfix notation can only be used for tuples with at least two arguments. It is used for virtual strings (see Section B.7).