

There exist garbage collection algorithms, called *real-time* garbage collectors, that can run continuously, interleaved with the program execution. They can be used in cases, such as hard real-time programming, in which there must not be any pauses.

### Garbage collection is not magic

Having garbage collection lightens the burden of memory management for the developer, but it does not eliminate it completely. There are two cases that remain the developer's responsibility: avoiding memory leaks and managing external resources.

**Avoiding memory leaks** It is the programmer's responsibility to avoid memory leaks. If the program continues to reference a data structure that it no longer needs, then that data structure's memory will never be recovered. The program should be careful to lose all references to data structures no longer needed.

For example, take a recursive function that traverses a list. If the list's head is passed to the recursive call, then list memory will not be recovered during the function's execution. Here is an example:

```
L=[1 2 3 ... 1000000]

fun {Sum X L1 L}
  case L1 of Y|L2 then {Sum X+Y L2 L}
  else X end
end

{Browse {Sum 0 L L}}
```

Sum sums the elements of a list. But it also keeps a reference to L, the original list, even though it does not need L. This means L will stay in memory during the whole execution of Sum. A better definition is as follows:

```
fun {Sum X L1}
  case L1 of Y|L2 then {Sum X+Y L2}
  else X end
end

{Browse {Sum 0 L}}
```

Here the reference to L is lost immediately. This example is trivial. But things can be more subtle. For example, consider an active data structure S that contains a list of other data structures D1, D2, ..., Dn. If one of these, say Di, is no longer needed by the program, then it should be removed from the list. Otherwise its memory will never be recovered.

A well-written program therefore has to do some "cleanup" after itself: making sure that it no longer references data structures that it no longer needs. The

cleanup can be done in the declarative model, but it is cumbersome.<sup>9</sup>

**Managing external resources** A Mozart program often needs data structures that are external to its operating system process. We call such a data structure an *external resource*. External resources affect memory management in two ways. An internal Mozart data structure can refer to an external resource and vice versa. Both possibilities need some programmer intervention. Let us consider each case separately.

The first case is when a Mozart data structure refers to an external resource. For example, a record can correspond to a graphic entity in a graphics display or to an open file in a file system. If the record is no longer needed, then the graphic entity has to be removed or the file has to be closed. Otherwise, the graphics display or the file system will have a memory leak. This is done with a technique called *finalization*, which defines actions to be taken when data structures become unreachable. Finalization is explained in Section 6.9.2.

The second case is when an external resource needs a Mozart data structure. This is often straightforward to handle. For example, consider a scenario where the Mozart program implements a database server that is accessed by external clients. This scenario has a simple solution: never do automatic reclaiming of the database storage. Other scenarios may not be so simple. A general solution is to set aside a part of the Mozart program to represent the external resource. This part should be active (i.e., have its own thread) so that it is not reclaimed haphazardly. It can be seen as a “proxy” for the resource. The proxy keeps a reference to the Mozart data structure as long as the resource needs it. The resource informs the proxy when it no longer needs the data structure. Section 6.9.2 gives another technique.

### The Mozart garbage collector

The Mozart system does automatic memory management. It has both a local garbage collector and a distributed garbage collector. The latter is used for distributed programming and is explained in Chapter 11. The local garbage collector uses a *copying dual-space* algorithm.

The garbage collector divides memory into two spaces, which each takes up half of available memory space. At any instant, the running program sits completely in one half. Garbage collection is done when there is no more free memory in that half. The garbage collector finds all data structures that are reachable from the root set and copies them to the other half of memory. Since they are copied to one contiguous memory block this also does compaction.

The advantage of a copying garbage collector is that its execution time is proportional to the active memory size, not to the total memory size. Small programs will garbage collect quickly, even if they are running in a large memory space. The two disadvantages of a copying garbage collector are that half the

---

<sup>9</sup>It is more efficiently done with explicit state (see Chapter 6).

memory is unusable at any given time and that long-lived data structures (like system tables) have to be copied at each garbage collection. Let us see how to remove these two disadvantages. Copying long-lived data can be avoided by using a modified algorithm called a *generational* garbage collector. This partitions active memory into generations. Long-lived data structures are put in older generations, which are collected less often.

The memory disadvantage is only important if the active memory size approaches the maximum addressable memory size of the underlying architecture. Mainstream computer technology is currently in a transition period from 32-bit to 64-bit addressing. In a computer with 32-bit addresses, the limit is reached when active memory size is 1000 MB or more. (The limit is usually not 4000 MB due to limitations in the operating system.) At the time of writing, this limit is reached by large programs in high-end personal computers. For such programs, we recommend to use a computer with 64-bit addresses, which has no such problem.

## 2.5 From kernel language to practical language

The kernel language has all the concepts needed for declarative programming. But trying to use it for practical declarative programming shows that it is too minimal. Kernel programs are just too verbose. It turns out that most of this verbosity can be eliminated by judiciously adding syntactic sugar and linguistic abstractions. This section does just that:

- It defines a set of syntactic conveniences that give a more concise and readable full syntax.
- It defines an important linguistic abstraction, namely *functions*, that is useful for concise and readable programming.
- It explains the interactive interface of the Mozart system and shows how it relates to the declarative model. This brings in the **declare** statement, which is a variant of the **local** statement designed for interactive use.

The resulting language is used in Chapter 3 to explain the programming techniques of the declarative model.

### 2.5.1 Syntactic conveniences

The kernel language defines a simple syntax for all its constructs and types. The full language has the following conveniences to make this syntax more usable:

- Nested partial values can be written in a concise way.
- Variables can be both declared and initialized in one step.

- Expressions can be written in a concise way.
- The **if** and **case** statements can be nested in a concise way.
- The new operators **andthen** and **orelse** are defined as conveniences for nested **if** statements.
- Statements can be converted into expressions by using a nesting marker.

The nonterminal symbols used in the kernel syntax and semantics correspond as follows to those in the full syntax:

Kernel syntax	Full syntax
$\langle x \rangle, \langle y \rangle, \langle z \rangle$	$\langle \text{variable} \rangle$
$\langle s \rangle$	$\langle \text{statement} \rangle, \langle \text{stmt} \rangle$

### Nested partial values

In Table 2.2, the syntax of records and patterns implies that their arguments are variables. In practice, many partial values are nested deeper than this. Because nested values are so often used, we give syntactic sugar for them. For example, we extend the syntax to let us write `person(name:"George" age:25)` instead of the more cumbersome version:

```
local A B in A="George" B=25 X=person(name:A age:B) end
```

where `x` is bound to the nested record.

### Implicit variable initialization

To make programs shorter and easier to read, there is syntactic sugar to bind a variable immediately when it is declared. The idea is to put a bind operation between **local** and **in**. Instead of `local x in x=10 {Browse x} end`, in which `x` is mentioned three times, the short-cut lets one write `local x=10 in {Browse x} end`, which mentions `x` only twice. A simple case is the following:

```
local X= $\langle \text{expression} \rangle$  in  $\langle \text{statement} \rangle$  end
```

This declares `x` and binds it to the result of  $\langle \text{expression} \rangle$ . The general case is:

```
local  $\langle \text{pattern} \rangle$ = $\langle \text{expression} \rangle$  in  $\langle \text{statement} \rangle$  end
```

where  $\langle \text{pattern} \rangle$  is any partial value. This declares all the variables in  $\langle \text{pattern} \rangle$  and then binds  $\langle \text{pattern} \rangle$  to the result of  $\langle \text{expression} \rangle$ . In both cases, the variables occurring on the *left-hand side* of the equality, i.e., `x` or the variables in  $\langle \text{pattern} \rangle$ , are the ones declared.

Implicit variable initialization is convenient for taking apart a complex data structure. For example, if `T` is bound to the record `tree(key:a left:L right:R value:1)`, then just one equality is enough to extract all four fields:

$ \begin{aligned} \langle \text{expression} \rangle &::= \langle \text{variable} \rangle \mid \langle \text{int} \rangle \mid \langle \text{float} \rangle \mid \\ &\mid \langle \text{expression} \rangle \langle \text{evalBinOp} \rangle \langle \text{expression} \rangle \\ &\mid \text{'('} \langle \text{expression} \rangle \langle \text{evalBinOp} \rangle \langle \text{expression} \rangle \text{'}' \\ &\mid \text{'{'} \langle \text{expression} \rangle \{ \langle \text{expression} \rangle \} \text{'}' \\ &\mid \dots \\ \langle \text{evalBinOp} \rangle &::= \text{'+'} \mid \text{'-'} \mid \text{'*'} \mid \text{'/'} \mid \text{div} \mid \text{mod} \mid \\ &\mid \text{'=='} \mid \text{'\='} \mid \text{'<'} \mid \text{'=<'} \mid \text{'>'} \mid \text{'>='} \mid \dots \end{aligned} $
---

Table 2.4: Expressions for calculating with numbers

```

local
  tree(key:A left:B right:C value:D)=T
in
  ⟨statement⟩
end

```

This is a kind of pattern matching.  $T$  must have the right structure, otherwise an exception is raised. This does part of the work of the **case** statement, which generalizes this so that the programmer decides what to do if the pattern is not matched. Without the short-cut, the following is needed:

```

local A B C D in
  {Label T}=tree
  A=T.key
  B=T.left
  C=T.right
  D=T.value
  ⟨statement⟩
end

```

which is both longer and harder to read. What if  $T$  has more than four fields, but we want to extract just four? Then we can use the following notation:

```

local
  tree(key:A left:B right:C value:D ...)=T
in
  ⟨statement⟩
end

```

The “...” means that there may be other fields in  $T$ .

## Expressions

An *expression* is syntactic sugar for a sequence of operations that returns a value. It is different from a statement, which is also a sequence of operations but does not return a value. An expression can be used inside a statement whenever a value is needed. For example,  $11 * 11$  is an expression and  $x = 11 * 11$  is a statement. Semantically, an expression is defined by a straightforward translation into kernel

<pre> &lt;statement&gt; ::= <b>if</b> &lt;expression&gt; <b>then</b> &lt;inStatement&gt;                 { <b>elseif</b> &lt;expression&gt; <b>then</b> &lt;inStatement&gt; }                 [ <b>else</b> &lt;inStatement&gt; ] <b>end</b>                   ... &lt;inStatement&gt; ::= [ { &lt;declarationPart&gt; }+ <b>in</b> ] &lt;statement&gt; </pre>
--

Table 2.5: The **if** statement

<pre> &lt;statement&gt; ::= <b>case</b> &lt;expression&gt;                 <b>of</b> &lt;pattern&gt; [ <b>andthen</b> &lt;expression&gt; ] <b>then</b> &lt;inStatement&gt;                 { <b>^</b> [ <b>^</b> ] <b>^</b> &lt;pattern&gt; [ <b>andthen</b> &lt;expression&gt; ] <b>then</b> &lt;inStatement&gt; }                 [ <b>else</b> &lt;inStatement&gt; ] <b>end</b>                   ... &lt;pattern&gt; ::= &lt;variable&gt;   &lt;atom&gt;   &lt;int&gt;   &lt;float&gt;                   &lt;string&gt;   <b>unit</b>   <b>true</b>   <b>false</b>                   &lt;label&gt; <b>^</b> ( <b>^</b> { [ &lt;feature&gt; <b>^</b> : <b>^</b> ] &lt;pattern&gt; } [ <b>^</b> ... <b>^</b> ] <b>^</b> ) <b>^</b>                   &lt;pattern&gt; &lt;consBinOp&gt; &lt;pattern&gt;                   <b>^</b> [ <b>^</b> { &lt;pattern&gt; }+ <b>^</b> ] <b>^</b> &lt;consBinOp&gt; ::= <b>^</b> # <b>^</b>   <b>^</b>   <b>^</b> </pre>
--

Table 2.6: The **case** statement

syntax. So `x=11*11` is translated into `{Mul 11 11 x}`, where `Mul` is a three-argument procedure that does multiplication.<sup>10</sup>

Table 2.4 shows the syntax of expressions that calculate with numbers. Later on we will see expressions for calculating with other data types. Expressions are built hierarchically, starting from basic expressions (e.g., variables and numbers) and combining them together. There are two ways to combine them: using operators (e.g., the addition `1+2+3+4`) or using function calls (e.g., the square root `{Sqrt 5.0}`).

### Nested **if** and **case** statements

We add syntactic sugar to make it easy to write **if** and **case** statements with multiple alternatives and complicated conditions. Table 2.5 gives the syntax of the full **if** statement. Table 2.6 gives the syntax of the full **case** statement and its patterns. (Some of the nonterminals in these tables are defined in Appendix C.) These statements are translated into the primitive **if** and **case** statements of the kernel language. Here is an example of a full **case** statement:

```

case Xs#Ys
of nil#Ys then <s>1

```

<sup>10</sup>Its real name is `Number.^*^`, since it is part of the `Number` module.

```

[ ] Xs#nil then <s>2
[ ] (X|Xr)#(Y|Yr) andthen X=<Y then <s>3
else <s>4 end

```

It consists of a sequence of alternative cases delimited with the “[ ]” symbol. The alternatives are often called *clauses*. This statement translates into the following kernel syntax:

```

case Xs of nil then <s>1
else
  case Ys of nil then <s>2
  else
    case Xs of X|Xr then
      case Ys of Y|Yr then
        if X=<Y then <s>3 else <s>4 end
      else <s>4 end
    else <s>4 end
  end
end

```

The translation illustrates an important property of the full **case** statement: clauses are tested sequentially starting with the first clause. Execution continues past a clause only if the clause’s pattern is inconsistent with the input argument.

Nested patterns are handled by looking first at the outermost pattern and then working inwards. The nested pattern  $(X|Xr)\#(Y|Yr)$  has one outer pattern of the form  $A\#B$  and two inner patterns of the form  $A|B$ . All three patterns are tuples that are written with infix syntax, using the infix operators  $\text{‘}\#\text{’}$  and  $\text{‘}\mid\text{’}$ . They could have been written with the usual syntax as  $\text{‘}\#\text{’}(A\ B)$  and  $\text{‘}\mid\text{’}(A\ B)$ . Each inner pattern  $(X|Xr)$  and  $(Y|Yr)$  is put in its own primitive **case** statement. The outer pattern using  $\text{‘}\#\text{’}$  disappears from the translation because it occurs also in the **case**’s input argument. The matching with  $\text{‘}\#\text{’}$  can therefore be done at translation time.

### The operators **andthen** and **orelse**

The operators **andthen** and **orelse** are used in calculations with boolean values. The expression:

```
<expression>1 andthen <expression>2
```

translates into:

```
if <expression>1 then <expression>2 else false end
```

The advantage of using **andthen** is that  $\langle\text{expression}\rangle_2$  is not evaluated if  $\langle\text{expression}\rangle_1$  is **false**. There is an analogous operator **orelse**. The expression:

```
<expression>1 orelse <expression>2
```

translates into:

```
if <expression>1 then true else <expression>2 end
```

$\langle \text{statement} \rangle$	$::=$	<b>fun</b> $\{ \langle \text{variable} \rangle \{ \langle \text{pattern} \rangle \} \}$ $\langle \text{inExpression} \rangle$ <b>end</b>
		...
$\langle \text{expression} \rangle$	$::=$	<b>fun</b> $\{ \langle \text{variable} \rangle \{ \langle \text{pattern} \rangle \} \}$ $\langle \text{inExpression} \rangle$ <b>end</b>
		<b>proc</b> $\{ \langle \text{variable} \rangle \{ \langle \text{pattern} \rangle \} \}$ $\langle \text{inStatement} \rangle$ <b>end</b>
		$\{ \langle \text{expression} \rangle \{ \langle \text{expression} \rangle \} \}$
		<b>local</b> $\{ \langle \text{declarationPart} \rangle \} +$ <b>in</b> $\langle \text{expression} \rangle$ <b>end</b>
		<b>if</b> $\langle \text{expression} \rangle$ <b>then</b> $\langle \text{inExpression} \rangle$
		$\{ \text{elseif } \langle \text{expression} \rangle \text{ then } \langle \text{inExpression} \rangle \}$
		$[ \text{else } \langle \text{inExpression} \rangle ]$ <b>end</b>
		<b>case</b> $\langle \text{expression} \rangle$
		<b>of</b> $\langle \text{pattern} \rangle [ \text{andthen } \langle \text{expression} \rangle ]$ <b>then</b> $\langle \text{inExpression} \rangle$
		$\{ \langle \text{pattern} \rangle [ \text{andthen } \langle \text{expression} \rangle ] \text{ then } \langle \text{inExpression} \rangle \}$
		$[ \text{else } \langle \text{inExpression} \rangle ]$ <b>end</b>
		...
$\langle \text{inStatement} \rangle$	$::=$	$[ \{ \langle \text{declarationPart} \rangle \} +$ <b>in</b> $] \langle \text{statement} \rangle$
$\langle \text{inExpression} \rangle$	$::=$	$[ \{ \langle \text{declarationPart} \rangle \} +$ <b>in</b> $] [ \langle \text{statement} \rangle ] \langle \text{expression} \rangle$

Table 2.7: Function syntax

That is,  $\langle \text{expression} \rangle_2$  is not evaluated if  $\langle \text{expression} \rangle_1$  is **true**.

### Nesting markers

The nesting marker “\$” turns any statement into an expression. The expression’s value is what is at the position indicated by the nesting marker. For example, the statement  $\{P \ x1 \ x2 \ x3\}$  can be written as  $\{P \ x1 \ \$ \ x3\}$ , which is an expression whose value is  $x2$ . This makes the source code more concise, since it avoids having to declare and use the identifier  $x2$ . The variable corresponding to  $x2$  is hidden from the source code.

Nesting markers can make source code more readable to a proficient programmer, while making it harder for a beginner to see how the code translates to the kernel language. We will use them only when they greatly increase readability. For example, instead of writing:

```
local X in {Obj get(X)} {Browse X} end
```

we will instead write  $\{Browse \ \{Obj \ get(\$)\}\}$ . Once you get used to nesting markers, they are both concise and clear. Note that the syntax of procedure values as explained in Section 2.3.3 is consistent with the nesting marker syntax.

### 2.5.2 Functions (the **fun** statement)

The declarative model provides a linguistic abstraction for programming with functions. This is our first example of a linguistic abstraction, as defined in



Section 2.1.2. We define the new syntax for function definitions and function calls and show how they are translated into the kernel language.

### Function definitions

A function definition differs from a procedure definition in two ways: it is introduced with the keyword **fun** and the body must end with an expression. For example, a simple definition is:

```
fun {F X1 ... XN} <statement> <expression> end
```

This translates to the following procedure definition:

```
proc {F X1 ... XN ?R} <statement> R=<expression> end
```

The extra argument *R* is bound to the expression in the procedure body. If the function body is an **if** statement, then each alternative of the **if** can end in an expression:

```
fun {Max X Y}
  if X>=Y then X else Y end
end
```

This translates to:

```
proc {Max X Y ?R}
  R = if X>=Y then X else Y end
end
```

We can further translate this by transforming the **if** from an expression to a statement. This gives the final result:

```
proc {Max X Y ?R}
  if X>=Y then R=X else R=Y end
end
```

Similar rules apply for the **local** and **case** statements, and for other statements we will see later. Each statement can be used as an expression. Roughly speaking, whenever an execution sequence in a procedure ends in a *statement*, the corresponding sequence in a function ends in an *expression*. Table 2.7 gives the complete syntax of expressions. This table takes all the statements we have seen so far and shows how to use them as expressions. In particular, there are also function values, which are simply procedure values written in functional syntax.

### Function calls

A function call {F X1 ... XN} translates to the procedure call {F X1 ... XN R}, where *R* replaces the function call where it is used. For example, the following nested call of *F*:

```
{Q {F X1 ... XN} ... }
```

is translated to:

```

local R in
  {F X1 ... XN R}
  {Q R ... }
end

```

In general, nested functions are evaluated *before* the function in which they are nested. If there are several, then they are evaluated in the order they appear in the program.

### Function calls in data structures

There is one more rule to remember for function calls. It has to do with a call inside a data structure (record, tuple, or list). Here is an example:

```
Ys={F X}|{Map Xr F}
```

In this case, the translation puts the nested calls *after* the bind operation:

```

local Y Yr in
  Ys=Y|Yr
  {F X Y}
  {Map Xr F Yr}
end

```

This ensures that the recursive call is last. Section 2.4.6 explains why this is important for execution efficiency. The full Map function is defined as follows:

```

fun {Map Xs F}
  case Xs
  of nil then nil
  [] X|Xr then {F X}|{Map Xr F}
  end
end

```

Map applies the function F to all elements of a list and returns the result. Here is an example call:

```
{Browse {Map [1 2 3 4] fun {$ X} X*X end}}
```

This displays [1 4 9 16]. The definition of Map translates as follows to the kernel language:

```

proc {Map Xs F ?Ys}
  case Xs of nil then Ys=nil
  else case Xs of X|Xr then
    local Y Yr in
      Ys=Y|Yr
      {F X Y}
      {Map Xr F Yr}
    end
  end end
end

```

$ \begin{aligned} \langle \text{interStatement} \rangle &::= \langle \text{statement} \rangle \\ &\quad   \text{declare } \{ \langle \text{declarationPart} \rangle \} + [ \langle \text{interStatement} \rangle ] \\ &\quad   \text{declare } \{ \langle \text{declarationPart} \rangle \} + \text{in } \langle \text{interStatement} \rangle \\ \langle \text{declarationPart} \rangle &::= \langle \text{variable} \rangle \mid \langle \text{pattern} \rangle \text{ '=' } \langle \text{expression} \rangle \mid \langle \text{statement} \rangle \end{aligned} $
--

Table 2.8: Interactive statement syntax

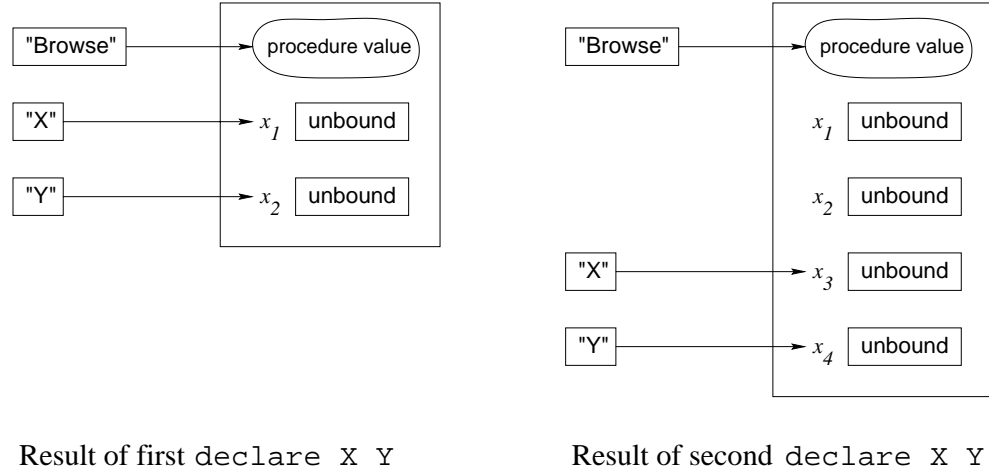


Figure 2.19: Declaring global variables

The dataflow variable `Yr` is used as a “placeholder” for the result in the recursive call `{Map Xr F Yr}`. This lets the recursive call be the last call. In our model, this means that the recursion executes with the same space and time efficiency as an iterative construct like a **while** loop.

### 2.5.3 Interactive interface (the **declare** statement)

The Mozart system has an interactive interface that allows to introduce program fragments incrementally and execute them as they are introduced. The fragments have to respect the syntax of interactive statements, which is given in Table 2.8. An *interactive statement* is either any legal statement or a new form, the **declare** statement. We assume that the user feeds interactive statements to the system one by one. (In the examples given throughout this book, the **declare** statement is often left out. It should be added if the example declares new variables.)

The interactive interface allows to do much more than just feed statements. It has all the functionality needed for software development. Appendix A gives a summary of some of this functionality. For now, we assume that the user just knows how to feed statements.

The interactive interface has a single, global environment. The **declare** statement adds new mappings to this environment. It follows that **declare** can

*only* be used interactively, not in standalone programs. Feeding the following declaration:

```
declare X Y
```

creates two new variables in the store,  $x_1$  and  $x_2$ . and adds mappings from **X** and **Y** to them. Because the mappings are in the global environment we say that **X** and **Y** are *global variables* or *interactive variables*. Feeding the same declaration a second time will cause **X** and **Y** to map to two other new variables,  $x_3$  and  $x_4$ . Figure 2.19 shows what happens. The original variables,  $x_1$  and  $x_2$ , are still in the store, but they are no longer referred to by **X** and **Y**. In the figure, **Browse** maps to a procedure value that implements the browser. The **declare** statement adds new variables and mappings, but leaves existing variables in the store unchanged.

Adding a new mapping to an identifier that already maps to a variable may cause the variable to become inaccessible, if there are no other references to it. If the variable is part of a calculation, then it is still accessible from within the calculation. For example:

```
declare X Y
X=25
declare A
A=person(age:X)
declare X Y
```

Just after the binding **X**=25, **X** maps to 25, but after the second **declare X Y** it maps to a new unbound variable. The 25 is still accessible through the global variable **A**, which is bound to the record `person(age:25)`. The record contains 25 because **X** mapped to 25 when the binding `A=person(age:X)` was executed. The second **declare X Y** changes the mapping of **X**, but not the record `person(age:25)` since the record already exists in the store. This behavior of **declare** is designed to support a modular programming style. Executing a program fragment will not cause the results of any previously-executed fragment to change.

There is a second form of **declare**:

```
declare X Y in <stmt>
```

which declares two global variables, as before, and then executes `<stmt>`. The difference with the first form is that `<stmt>` declares no variables (unless it contains a **declare**).

## The Browser

The interactive interface has a tool, called the *Browser*, which allows to look into the store. This tool is available to the programmer as a procedure called **Browse**. The procedure **Browse** has one argument. It is called as `{Browse <expr>}`, where `<expr>` is any expression. It can display partial values and it will update the display whenever the partial values are bound more. Feeding the following:

```
{Browse 1}
```

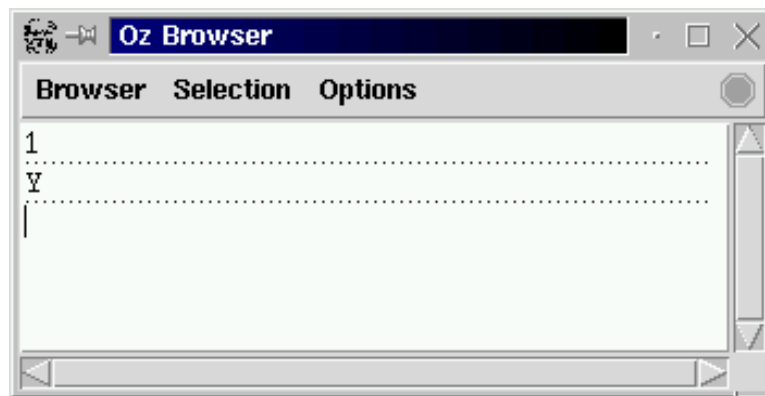


Figure 2.20: The Browser

displays the integer 1. Feeding:

```
declare Y in
  {Browse Y}
```

displays just the name of the variable, namely `Y`. No value is displayed. This means that `Y` is currently unbound. Figure 2.20 shows the browser window after these two operations. If `Y` is bound, e.g., by doing `Y=2`, then the browser will update its display to show this binding.

### Dataflow execution

We saw earlier that declarative variables support dataflow execution, i.e., an operation waits until all arguments are bound before executing. For sequential programs this is not very useful, since the program will wait forever. On the other hand, it is useful for concurrent programs, in which more than one instruction sequence can be executing at the same time. An independently-executing instruction sequence is called a *thread*. Programming with more than one thread is called *concurrent programming*; it is introduced in Chapter 4.

All examples in this chapter execute in a single thread. To be precise, each program fragment fed into the interactive interface executes in its own thread. This lets us give simple examples of dataflow execution in this chapter. For example, feed the following statement:

```
declare A B C in
  C=A+B
  {Browse C}
```

This will display nothing, since the instruction `C=A+B` blocks (both of its arguments are unbound). Now, feed the following statement:

```
A=10
```

This will bind `A`, but the instruction `C=A+B` still blocks since `B` is still unbound. Finally, feed the following:

B=200

This displays 210 in the browser. *Any* operation, not just addition, will block if it does not get enough input information to calculate its result. For example, comparisons can block. The equality comparison  $x==y$  will block if it cannot decide whether or not  $x$  is equal to or different from  $y$ . This happens, e.g., if one or both of the variables are unbound.

Programming errors often result in dataflow suspensions. If you feed a statement that should display a result and nothing is displayed, then the probable cause of the problem is a blocked operation. Carefully check all operations to make sure that their arguments are bound. Ideally, the system's debugger should detect when a program has blocked operations that cannot continue.

## 2.6 Exceptions

How do we handle exceptional situations within a program? For example, dividing by zero, opening a nonexistent file, or selecting a nonexistent field of a record? These errors do not occur in a correct program, so they should not encumber normal programming style. On the other hand, they do occur sometimes. It should be possible for programs to manage these errors in a simple way. The declarative model cannot do this without adding cumbersome checks throughout the program. A more elegant way is to extend the model with an *exception-handling mechanism*. This section does exactly that. We give the syntax and semantics of the extended model and explain what exceptions look like in the full language.

### 2.6.1 Motivation and basic concepts

In the semantics of Section 2.4, we speak of “raising an error” when a statement cannot continue correctly. For example, a conditional raises an error when its argument is a non-boolean value. Up to now, we have been deliberately vague about exactly what happens next. Let us now be more precise. We would like to be able to detect these errors and handle them from within a running program. The program should not stop when they occur. Rather, it should in a controlled way transfer execution to another part, called the *exception handler*, and pass the exception handler a value that describes the error.

What should the exception-handling mechanism look like? We can make two observations. First, it should be able to *confine* the error, i.e., quarantine it so that it does not contaminate the whole program. We call this the *error confinement principle*:

Assume that the program is made up of interacting “components” organized in hierarchical fashion. Each component is built of smaller components. We put “component” in quotes because the language does not need to have a component concept. It just needs to be

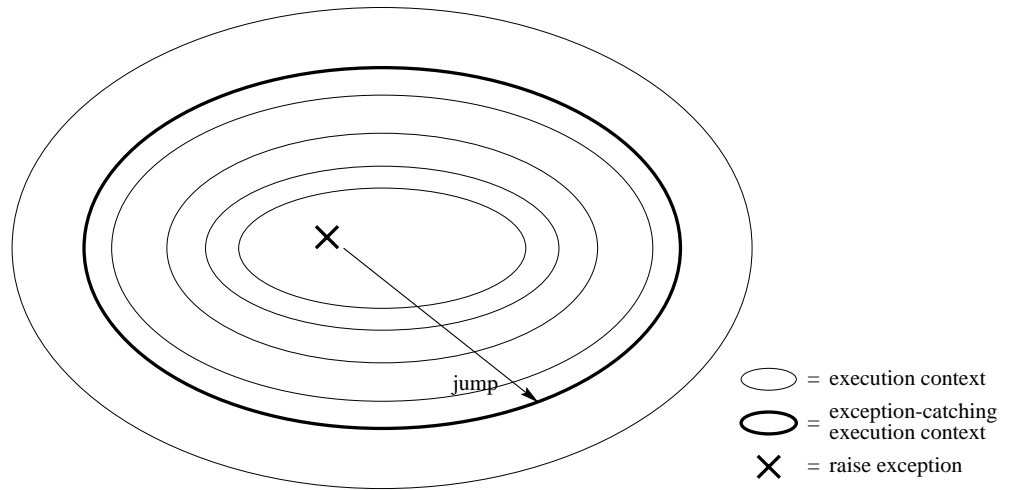


Figure 2.21: Exception handling

*compositional*, i.e., programs are built in layered fashion. Then the error confinement principle states that an error in a component should be catchable at the component boundary. Outside the component, the error is either invisible or reported in a nice way.

Therefore, the mechanism causes a “jump” from inside the component to its boundary. The second observation is that this jump should be a single operation. The mechanism should be able, in a single operation, to exit from arbitrarily many levels of nested context. Figure 2.21 illustrates this. In our semantics, a *context* is simply an entry on the semantic stack, i.e., an instruction that has to be executed later. Nested contexts are created by procedure calls and sequential compositions.

The declarative model cannot jump out in a single operation. The jump has to be coded explicitly as little hops, one per context, using boolean variables and conditionals. This makes programs more cumbersome, especially since the extra coding has to be added everywhere that an error can possibly occur. It can be shown theoretically that the only way to keep programs simple is to extend the model [103, 105].

We propose a simple extension to the model that satisfies these conditions. We add two statements: the **try** statement and the **raise** statement. The **try** statement creates an exception-catching context together with an exception handler. The **raise** statement jumps to the boundary of the innermost exception-catching context and invokes the exception handler there. Nested **try** statements create nested contexts. Executing **try**  $\langle s \rangle$  **catch**  $\langle x \rangle$  **then**  $\langle s \rangle_1$  **end** is equivalent to executing  $\langle s \rangle$ , if  $\langle s \rangle$  does not raise an exception. On the other hand, if  $\langle s \rangle$  raises an exception, i.e., by executing a **raise** statement, then the (still ongoing) execution of  $\langle s \rangle$  is aborted. All information related to  $\langle s \rangle$  is popped from the semantic stack. Control is transferred to  $\langle s \rangle_1$ , passing it a reference to the exception in  $\langle x \rangle$ .

Any partial value can be an exception. This means that the exception-handling mechanism is extensible by the programmer, i.e., new exceptions can be defined as they are needed by the program. This lets the programmer foresee new exceptional situations. Because an exception can be an unbound variable, raising an exception and determining what the exception is can be done concurrently. In other words, an exception can be raised (and caught) before it is known which exception it is! This is quite reasonable in a language with dataflow variables: we may at some point know that there *exists* a problem but not know yet *which* problem.

### An example

Let us give a simple example of exception handling. Consider the following function, which evaluates simple arithmetic expressions and returns the result:

```
fun {Eval E}
  if {IsNumber E} then E
  else
    case E
    of   plus(X Y) then {Eval X}+{Eval Y}
    []   times(X Y) then {Eval X}*{Eval Y}
    else raise illFormedExpr(E) end
    end
  end
end
```

For this example, we say an expression is *ill-formed* if it is not recognized by Eval, i.e., if it contains other values than numbers, plus, and times. Trying to evaluate an ill-formed expression E will raise an exception. The exception is a tuple, `illFormedExpr(E)`, that contains the ill-formed expression. Here is an example of using Eval:

```
try
  {Browse {Eval plus(plus(5 5) 10)}}
  {Browse {Eval times(6 11)}}
  {Browse {Eval minus(7 10)}}
catch illFormedExpr(E) then
  {Browse `*** Illegal expression `#E#` ***`}
end
```

If any call to Eval raises an exception, then control transfers to the **catch** clause, which displays an error message.

### 2.6.2 The declarative model with exceptions

We extend the declarative computation model with exceptions. Table 2.9 gives the syntax of the extended kernel language. Programs can use two new statements, **try** and **raise**. In addition, there is a third statement, **catch** <x> **then**



$\langle s \rangle ::=$	
<b>skip</b>	Empty statement
$\langle s \rangle_1 \langle s \rangle_2$	Statement sequence
<b>local</b> $\langle x \rangle$ <b>in</b> $\langle s \rangle$ <b>end</b>	Variable creation
$\langle x \rangle_1 = \langle x \rangle_2$	Variable-variable binding
$\langle x \rangle = \langle v \rangle$	Value creation
<b>if</b> $\langle x \rangle$ <b>then</b> $\langle s \rangle_1$ <b>else</b> $\langle s \rangle_2$ <b>end</b>	Conditional
<b>case</b> $\langle x \rangle$ <b>of</b> $\langle \text{pattern} \rangle$ <b>then</b> $\langle s \rangle_1$ <b>else</b> $\langle s \rangle_2$ <b>end</b>	Pattern matching
$\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$	Procedure application
<b>try</b> $\langle s \rangle_1$ <b>catch</b> $\langle x \rangle$ <b>then</b> $\langle s \rangle_2$ <b>end</b>	<b>Exception context</b>
<b>raise</b> $\langle x \rangle$ <b>end</b>	<b>Raise exception</b>

Table 2.9: The declarative kernel language with exceptions

$\langle s \rangle$  **end**, that is needed internally for the semantics and is not allowed in programs. The **catch** statement is a “marker” on the semantic stack that defines the boundary of the exception-catching context. We now give the semantics of these statements.

### The **try** statement

The semantic statement is:

$$(\text{try } \langle s \rangle_1 \text{ catch } \langle x \rangle \text{ then } \langle s \rangle_2 \text{ end}, E)$$

Execution consists of the following actions:

- Push the semantic statement (**catch**  $\langle x \rangle$  **then**  $\langle s \rangle_2$  **end**,  $E$ ) on the stack.
- Push  $(\langle s \rangle_1, E)$  on the stack.

### The **raise** statement

The semantic statement is:

$$(\text{raise } \langle x \rangle \text{ end}, E)$$

Execution consists of the following actions:

- Pop elements off the stack looking for a **catch** statement.
  - If a **catch** statement is found, pop it from the stack.
  - If the stack is emptied and no **catch** is found, then stop execution with the error message “Uncaught exception”.
- Let (**catch**  $\langle y \rangle$  **then**  $\langle s \rangle$  **end**,  $E_c$ ) be the **catch** statement that is found.

$\langle \text{statement} \rangle$	$::=$	<b>try</b> $\langle \text{inStatement} \rangle$ $[$ <b>catch</b> $\langle \text{pattern} \rangle$ <b>then</b> $\langle \text{inStatement} \rangle$ $\{ \text{ ' [ ] ' } \langle \text{pattern} \rangle$ <b>then</b> $\langle \text{inStatement} \rangle \}$ $]$ $[$ <b>finally</b> $\langle \text{inStatement} \rangle$ $]$ <b>end</b> $ $ <b>raise</b> $\langle \text{inExpression} \rangle$ <b>end</b> $ $ ...
$\langle \text{inStatement} \rangle$	$::=$	$[ \{ \langle \text{declarationPart} \rangle \}^+ \text{in} ] \langle \text{statement} \rangle$
$\langle \text{inExpression} \rangle$	$::=$	$[ \{ \langle \text{declarationPart} \rangle \}^+ \text{in} ] [ \langle \text{statement} \rangle ] \langle \text{expression} \rangle$

Table 2.10: Exception syntax

- Push  $(\langle s \rangle, E_c + \{ \langle y \rangle \rightarrow E(\langle x \rangle) \})$  on the stack.

Let us see how an uncaught exception is handled by the Mozart system. For interactive execution, an error message is printed in the Oz emulator window. For standalone applications, the application terminates and an error message is sent on the standard error output of the process. It is possible to change this behavior to something else that is more desirable for particular applications, by using the System module `Property`.

### The **catch** statement

The semantic statement is:

$(\text{catch } \langle x \rangle \text{ then } \langle s \rangle \text{ end}, E)$

Execution is complete after this pair is popped from the semantic stack. I.e., the **catch** statement does nothing, just like **skip**.

### 2.6.3 Full syntax

Table 2.10 gives the syntax of the **try** statement in the full language. It has an optional **finally** clause. The **catch** clause has an optional series of patterns. Let us see how these extensions are defined.

### The **finally** clause

A **try** statement can specify a **finally** clause which is *always* executed, whether or not the statement raises an exception. The new syntax:

**try**  $\langle s \rangle_1$  **finally**  $\langle s \rangle_2$  **end**

is translated to the kernel language as:

**try**  $\langle s \rangle_1$   
**catch**  $x$  **then**  
 $\langle s \rangle_2$

```

    raise X end
end
⟨s⟩2

```

(where an identifier  $x$  is chosen that is not free in  $\langle s \rangle_2$ ). It is possible to define a translation in which  $\langle s \rangle_2$  only occurs once; we leave this to the reader.

The **finally** clause is useful when dealing with entities that are external to the computation model. With **finally**, we can guarantee that some “cleanup” action gets performed on the entity, whether or not an exception occurs. A typical example is reading a file. Assume  $F$  is an open file<sup>11</sup>, the procedure `ProcessFile` manipulates the file in some way, and the procedure `CloseFile` closes the file. Then the following program ensures that  $F$  is always closed after `ProcessFile` completes, whether or not an exception was raised:

```

try
  {ProcessFile F}
finally {CloseFile F} end

```

Note that this **try** statement does not catch the exception; it just executes `CloseFile` whenever `ProcessFile` completes. We can combine both catching the exception and executing a final statement:

```

try
  {ProcessFile F}
catch X then
  {Browse `*** Exception `#X#` when processing file ***`}
finally {CloseFile F} end

```

This behaves like two nested **try** statements: the innermost with just a **catch** clause and the outermost with just a **finally** clause.

### Pattern matching

A **try** statement can use pattern matching to catch only exceptions that match a given pattern. Other exceptions are passed to the next enclosing **try** statement. The new syntax:

```

try ⟨s⟩
catch ⟨p⟩1 then ⟨s⟩1
  [ ] ⟨p⟩2 then ⟨s⟩2
  ...
  [ ] ⟨p⟩n then ⟨s⟩n
end

```

is translated to the kernel language as:

```

try ⟨s⟩
catch X then
  case X

```

---

<sup>11</sup>We will see later how file input/output is handled.

```

of ⟨p⟩1 then ⟨s⟩1
  [ ] ⟨p⟩2 then ⟨s⟩2
  ...
  [ ] ⟨p⟩n then ⟨s⟩n
else raise X end
end
end

```

If the exception does not match any of the patterns, then it is simply raised again.

### 2.6.4 System exceptions

The Mozart system itself raises a few exceptions. They are called *system exceptions*. They are all records with one of the three labels `failure`, `error`, or `system`:

- **failure**: indicates an attempt to perform an inconsistent bind operation (e.g., `1=2`) in the store (see Section 2.7.2).
- **error**: indicates a runtime error in the program, i.e., a situation that should not occur during normal operation. These errors are either type or domain errors. A *type error* occurs when invoking an operation with an argument of incorrect type, e.g., applying a nonprocedure to some argument (`{foo 1}`, where `foo` is an atom), or adding an integer to an atom (e.g., `x=1+a`). A *domain error* occurs when invoking an operation with an argument that is outside of its domain (even if it has the right type), e.g., taking the square root of a negative number, dividing by zero, or selecting a nonexistent field of a record.
- **system**: indicates a runtime condition occurring in the environment of the Mozart operating system process, e.g., an unforeseeable situation like a closed file or window or a failure to open a connection between two Mozart processes in distributed programming (see Chapter 11).

What is stored inside the exception record depends on the Mozart system version. Therefore programmers should rely only on the label. For example:

```

fun {One} 1 end
fun {Two} 2 end
try {One}={Two}
catch
  failure(...) then {Browse caughtFailure}
end

```

The pattern `failure(...)` catches any record whose label is `failure`.

## 2.7 Advanced topics

This section gives additional information for deeper understanding of the declarative model, its trade-offs, and possible variations.

### 2.7.1 Functional programming languages

Functional programming consists of defining functions on complete values, where the functions are true functions in the mathematical sense. A language in which this is the only possible way to calculate is called a *pure functional language*. Let us examine how the declarative model relates to pure functional programming. For further reading on the history, formal foundations, and motivations for functional programming we recommend the survey article by Hudak [85].

#### The $\lambda$ calculus

Pure functional languages are based on a formalism called the  $\lambda$  *calculus*. There are many variants of the  $\lambda$  calculus. All of these variants have in common two basic operations, namely defining and evaluating functions. For example, the function value **fun** { $\$$   $x$ }  $x*x$  **end** is identical to the  $\lambda$  expression  $\lambda x. x * x$ . This expression consists of two parts: the  $x$  before the dot, which is the function's argument, and the expression  $x * x$ , which is the function's result. The **Append** function, which appends two lists together, can be defined as a function value:

```
Append=fun { $\$$  Xs Ys}
  if {IsNil Xs} then Xs
  else {Cons {Car Xs} {Append {Cdr Xs} Ys}}
  end
end
```

This is equivalent to the following  $\lambda$  expression:

$$append = \lambda xs, ys. \text{ if } isNil(xs) \text{ then } ys \\ \text{ else } cons(car(xs), append(cdr(xs), ys))$$

The definition of **Append** uses the following helper functions:

```
fun {IsNil X} X==nil end
fun {IsCons X} case X of _|_ then true else false end end
fun {Car H|T} H end
fun {Cdr H|T} T end
fun {Cons H T} H|T end
```

#### Restricting the declarative model

The declarative model is more general than the  $\lambda$  calculus in two ways. First, it defines functions on partial values, i.e., with unbound variables. Second, it uses a procedural syntax. We can define a pure functional language by putting

two syntactic restrictions on the declarative model so that it always calculates functions on complete values:

- Always bind a variable to a value immediately when it is declared. That is, the **local** statement always has one of the following two forms:

```
local  $\langle x \rangle = \langle v \rangle$  in  $\langle s \rangle$  end
local  $\langle x \rangle = \{ \langle y \rangle \ \langle y \rangle_1 \ \dots \ \langle y \rangle_n \}$  in  $\langle s \rangle$  end
```

- Use only the function syntax, not the procedure syntax. For function calls inside data structures, do the nested call before creating the data structure (instead of after, as in Section 2.5.2). This avoids putting unbound variables in data structures.

With these restrictions, the model no longer needs unbound variables. The declarative model with these restrictions is called the *(strict) functional model*. This model is close to well-known functional programming languages such as Scheme and Standard ML. The full range of higher-order programming techniques is possible. Pattern matching is possible using the **case** statement.

### Varieties of functional programming

Let us explore some variations on the theme of functional programming:<sup>12</sup>

- The functional model of this chapter is dynamically typed like Scheme. Many functional languages are statically typed. Section 2.7.3 explains the differences between the two approaches. Furthermore, many statically-typed languages, e.g., Haskell and Standard ML, do type inferencing, which allows the compiler to infer the types of all functions.
- Thanks to dataflow variables and the single-assignment store, the declarative model allows programming techniques that are not found in most functional languages, including Scheme, Standard ML, Haskell, and Erlang. This includes certain forms of last call optimization and techniques to compute with partial values as shown in Chapter 3.
- The declarative concurrent model of Chapter 4 adds concurrency while still keeping all the good properties of functional programming. This is possible because of dataflow variables and the single-assignment store.
- In the declarative model, functions are *eager* by default, i.e., function arguments are evaluated before the function body is executed. This is also called *strict* evaluation. The functional languages Scheme and Standard ML are strict. There is another useful execution order, *lazy* evaluation, in which

<sup>12</sup>In addition to what is listed here, the functional model does not have any special syntactic or implementation support for currying. Currying is a higher-order programming technique that is explained in Section 3.6.6.

$\langle \text{statement} \rangle$	$::=$	$\langle \text{expression} \rangle \text{ ' = ' } \langle \text{expression} \rangle \mid \dots$
$\langle \text{expression} \rangle$	$::=$	$\langle \text{expression} \rangle \text{ ' == ' } \langle \text{expression} \rangle$ $\mid \langle \text{expression} \rangle \text{ ' \= ' } \langle \text{expression} \rangle \mid \dots$
$\langle \text{binaryOp} \rangle$	$::=$	$\text{ ' = ' } \mid \text{ ' == ' } \mid \text{ ' \= ' } \mid \dots$

Table 2.11: Equality (unification) and equality test (entailment check)

function arguments are evaluated only if their result is needed. Haskell is a lazy functional language.<sup>13</sup> Lazy evaluation is a powerful flow control technique in functional programming [87]. It allows to program with potentially infinite data structures without giving explicit bounds. Section 4.5 explains this in detail. An eager declarative program can evaluate functions and then never use them, thus doing superfluous work. A lazy declarative program, on the other hand, does the absolute minimum amount of work to get its result.

## 2.7.2 Unification and entailment

In Section 2.2 we have seen how to bind dataflow variables to partial values and to each other, using the equality ( $\text{' = '}$ ) operation as shown in Table 2.11. In Section 2.3.5 we have seen how to compare values, using the equality test ( $\text{' == '}$  and  $\text{' \= '}$ ) operations. So far, we have seen only the simple cases of these operations. Let us now examine the general cases.

Binding a variable to a value is a special case of an operation called *unification*. The unification  $\langle \text{Term1} \rangle = \langle \text{Term2} \rangle$  makes the partial values  $\langle \text{Term1} \rangle$  and  $\langle \text{Term2} \rangle$  equal, if possible, by adding zero or more bindings to the store. For example,  $f(x\ y) = f(1\ 2)$  does two bindings:  $x=1$  and  $y=2$ . If the two terms cannot be made equal, then an exception is raised. Unification exists because of partial values; if there would be only complete values then it would have no meaning.

Testing whether a variable is equal to a value is a special case of the *entailment check* and *disentailment check* operations. The entailment check  $\langle \text{Term1} \rangle == \langle \text{Term2} \rangle$  (and its opposite, the disentailment check  $\langle \text{Term1} \rangle \backslash = \langle \text{Term2} \rangle$ ) is a two-argument boolean function that blocks until it is known whether  $\langle \text{Term1} \rangle$  and  $\langle \text{Term2} \rangle$  are equal or not equal.<sup>14</sup> Entailment and disentailment checks never do any binding.

<sup>13</sup>To be precise, Haskell is a non-strict language. This is identical to laziness for most practical purposes. The difference is explained in Section 4.9.2.

<sup>14</sup>The word “entailment” comes from logic. It is a form of logical implication. This is because the equality  $\langle \text{Term1} \rangle == \langle \text{Term2} \rangle$  is true if the store, considered as a conjunction of equalities, “logically implies”  $\langle \text{Term1} \rangle == \langle \text{Term2} \rangle$ .

### Unification (the = operation)

A good way to conceptualize unification is as an operation that adds information to the single-assignment store. The store is a set of dataflow variables, where each variable is either unbound or bound to some other store entity. The store's information is just the set of all its bindings. Doing a new binding, for example `X=Y`, will add the information that `X` and `Y` are equal. If `X` and `Y` are already bound when doing `X=Y`, then some other bindings may be added to the store. For example, if the store already has `X=foo(A)` and `Y=foo(25)`, then doing `X=Y` will bind `A` to `25`. Unification is a kind of “compiler” that is given new information and “compiles it into the store”, taking account the bindings that are already there. To understand how this works, let us look at some possibilities.

- The simplest cases are bindings to values, e.g., `X=person(name:X1 age:X2)`, and variable-variable bindings, e.g., `X=Y`. If `X` and `Y` are unbound, then these operations each add one binding to the store.
- Unification is symmetric. For example, `person(name:X1 age:X2)=X` means the same as `X=person(name:X1 age:X2)`.
- Any two partial values can be unified. For example, unifying the two records:

```
person(name:X1 age:X2)
person(name:"George" age:25)
```

This binds `X1` to `"George"` and `x2` to `25`.

- If the partial values are already equal, then unification does nothing. For example, unifying `X` and `Y` where the store contains the two records:

```
X=person(name:"George" age:25)
Y=person(name:"George" age:25)
```

This does nothing.

- If the partial values are incompatible then they cannot be unified. For example, unifying the two records:

```
person(name:X1 age:26)
person(name:"George" age:25)
```

The records have different values for their `age` fields, namely `25` and `26`, so they cannot be unified. This unification will raise a `failure` exception, which can be caught by a `try` statement. The unification might or might not bind `X1` to `"George"`; it depends on exactly when it finds out that there is an incompatibility. Another way to get a unification failure is by executing the statement `fail`.



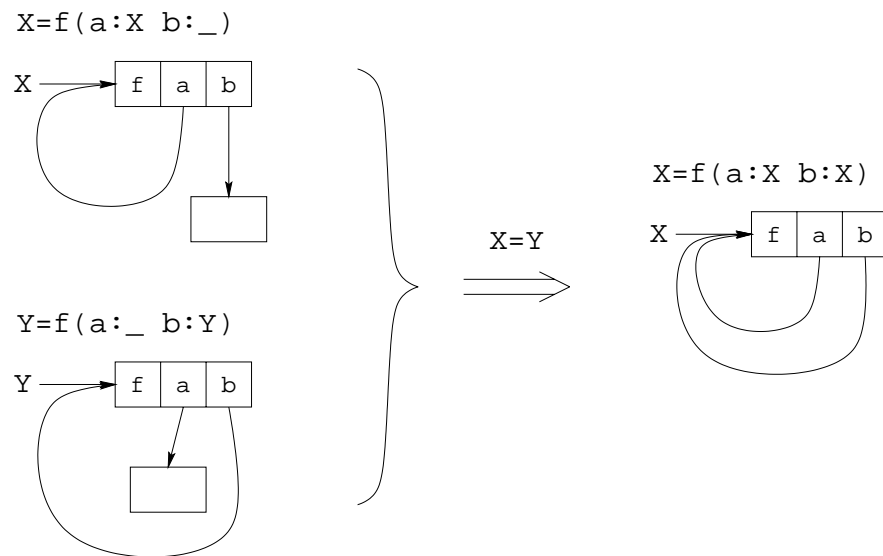


Figure 2.22: Unification of cyclic structures

- Unification is symmetric in the arguments. For example, unifying the two records:

```
person(name:"George" age:X2)
person(name:X1 age:25)
```

This binds `x1` to "George" and `x2` to 25, just like before.

- Unification can create cyclic structures, i.e., structures that refer to themselves. For example, the unification `x=person(grandfather:x)`. This creates a record whose `grandfather` field refers to itself. This situation happens in some crazy time-travel stories.
- Unification can bind cyclic structures. For example, let's create two cyclic structures, in `X` and `Y`, by doing `X=f(a:X b:_)` and `Y=f(a:_ b:Y)`. Now, doing the unification `X=Y` creates a structure with *two* cycles, which we can write as `X=f(a:X b:X)`. This example is illustrated in Figure 2.22.

### The unification algorithm

Let us give a precise definition of unification. We will define the operation  $\text{unify}(x, y)$  that unifies two partial values  $x$  and  $y$  in the store  $\sigma$ . Unification is a basic operation of logic programming. When used in the context of unification, store variables are called *logic variables*. Logic programming, which is also called relational programming, is discussed in Chapter 9.

**The store** The store consists of a set of  $k$  variables,  $x_1, \dots, x_k$ , that are partitioned as follows:

- Sets of unbound variables that are equal (also called *equivalence sets* of variables). The variables in each set are equal to each other but not to any other variables.
- Variables bound to a number, record, or procedure (also called *determined* variables).

An example is the store  $\{x_1 = \text{foo}(\text{a}:x_2), x_2 = 25, x_3 = x_4 = x_5, x_6, x_7 = x_8\}$  that has eight variables. It has three equivalence sets, namely  $\{x_3, x_4, x_5\}$ ,  $\{x_6\}$ , and  $\{x_7, x_8\}$ . It has two determined variables, namely  $x_1$  and  $x_2$ .

**The primitive bind operation** We define unification in terms of a primitive bind operation on the store  $\sigma$ . The operation binds all variables in an equivalence set:

- $\text{bind}(ES, \langle v \rangle)$  binds all variables in the equivalence set  $ES$  to the number or record  $\langle v \rangle$ . For example, the operation  $\text{bind}(\{x_7, x_8\}, \text{foo}(\text{a}:x_2))$  modifies the example store so that  $x_7$  and  $x_8$  are no longer in an equivalence set but both become bound to  $\text{foo}(\text{a}:x_2)$ .
- $\text{bind}(ES_1, ES_2)$  merges the equivalence set  $ES_1$  with the equivalence set  $ES_2$ . For example, the operation  $\text{bind}(\{x_3, x_4, x_5\}, \{x_6\})$  modifies the example store so that  $x_3, x_4, x_5$ , and  $x_6$  are in a single equivalence set, namely  $\{x_3, x_4, x_5, x_6\}$ .

**The algorithm** We now define the operation  $\text{unify}(x, y)$  as follows:

1. If  $x$  is in the equivalence set  $ES_x$  and  $y$  is in the equivalence set  $ES_y$ , then do  $\text{bind}(ES_x, ES_y)$ . If  $x$  and  $y$  are in the same equivalence set, this is the same as doing nothing.
2. If  $x$  is in the equivalence set  $ES_x$  and  $y$  is determined, then do  $\text{bind}(ES_x, y)$ .
3. If  $y$  is in the equivalence set  $ES_y$  and  $x$  is determined, then do  $\text{bind}(ES_y, x)$ .
4. If  $x$  is bound to  $l(l_1 : x_1, \dots, l_n : x_n)$  and  $y$  is bound to  $l'(l'_1 : y_1, \dots, l'_m : y_m)$  with  $l \neq l'$  or  $\{l_1, \dots, l_n\} \neq \{l'_1, \dots, l'_m\}$ , then raise a failure exception.
5. If  $x$  is bound to  $l(l_1 : x_1, \dots, l_n : x_n)$  and  $y$  is bound to  $l(l_1 : y_1, \dots, l_n : y_n)$ , then for  $i$  from 1 to  $n$  do  $\text{unify}(x_i, y_i)$ .

**Handling cycles** The above algorithm does not handle unification of partial values with cycles. For example, assume the store contains  $x = \text{f}(\text{a}:x)$  and  $y = \text{f}(\text{a}:y)$ . Calling  $\text{unify}(x, y)$  results in the recursive call  $\text{unify}(x, y)$ , which is identical to the original call. The algorithm loops forever! Yet it is clear that  $x$  and  $y$  have exactly the same structure: what the unification *should* do is

add exactly zero bindings to the store and then terminate. How can we fix this problem?

A simple fix is to make sure that  $\text{unify}(x, y)$  is called at most once for each possible pair of two variables  $(x, y)$ . Since any attempt to call it again will not do anything new, it can return immediately. With  $k$  variables in the store, this means at most  $k^2$  unify calls, so the algorithm is guaranteed to terminate. In practice, the number of unify calls is much less than this. We can implement the fix with a table that stores all called pairs. This gives the new algorithm  $\text{unify}'(x, y)$ :

- Let  $M$  be a new, empty table.
- Call  $\text{unify}''(x, y)$ .

This needs the definition of  $\text{unify}''(x, y)$ :

- If  $(x, y) \in M$  then we are done.
- Otherwise, insert  $(x, y)$  in  $M$  and then do the original algorithm for  $\text{unify}(x, y)$ , in which the recursive calls to unify are replaced by calls to  $\text{unify}''$ .

This algorithm can be written in the declarative model by passing  $M$  as two extra arguments to  $\text{unify}''$ . A table that remembers previous calls so that they can be avoided in the future is called a *memoization table*.

### Displaying cyclic structures

We have seen that unification can create cyclic structures. To display these in the browser, it has to be configured right. In the browser's **Options** menu, pick the **Representation** entry and choose the **Graph** mode. There are three display modes, namely **Tree** (the default), **Graph**, and **Minimal Graph**. **Tree** does not take sharing or cycles into account. **Graph** correctly handles sharing and cycles by displaying a graph. **Minimal Graph** shows the smallest graph that is consistent with the data. We give some examples. Consider the following two unifications:

```

local X Y Z in
  f(X b)=f(a Y)
  f(Z a)=Z
  {Browse [X Y Z]}
end

```

This shows the list `[a b R14=f(R14 a)]` in the browser, if the browser is set up to show the **Graph** representation. The term `R14=f(R14 a)` is the textual representation of a cyclic graph. The variable name `R14` is introduced by the browser; different versions of Mozart might introduce different variable names. As a second example, feed the following unification when the browser is set up for **Graph**, as before:

```

declare X Y Z in
  a(X c(Z) Z)=a(b(Y) Y d(X))
  {Browse X#Y#Z}

```

Now set up the browser for the **Minimal Graph** mode and display the term again. How do you explain the difference?

### Entailment and disentanglement checks (the == and \= operations)

The entailment check  $x==y$  is a boolean function that tests whether  $x$  and  $y$  are equal or not. The opposite check,  $x\backslash=y$ , is called a disentanglement check. Both checks use essentially the same algorithm.<sup>15</sup> The entailment check returns **true** if the store implies the information  $x=y$  in a way that is verifiable (the store “entails”  $x=y$ ) and **false** if the store will never imply  $x=y$ , again in a way that is verifiable (the store “disentails”  $x=y$ ). The check blocks if it cannot determine whether  $x$  and  $y$  are equal or will never be equal. It is defined as follows:

- It returns the value **true** if the graphs starting from the nodes of  $x$  and  $y$  have the same structure, i.e., all pairwise corresponding nodes have identical values or are the same node. We call this *structure equality*.
- It returns the value **false** if the graphs have different structure, or some pairwise corresponding nodes have different values.
- It blocks when it arrives at pairwise corresponding nodes that are different, but at least one of them is unbound.

Here is an example:

```

declare L1 L2 L3 Head Tail in
  L1=Head|Tail
  Head=1
  Tail=2|nil

  L2=[1 2]
  {Browse L1==L2}

  L3=^(1:1 2:^(2 nil))
  {Browse L1==L3}

```

All three lists  $L1$ ,  $L2$ , and  $L3$  are identical. Here is an example where the entailment check cannot decide:

```

declare L1 L2 X in
  L1=[1]
  L2=[X]
  {Browse L1==L2}

```

---

<sup>15</sup>Strictly speaking, there is a single algorithm that does both the entailment and disentanglement checks simultaneously. It returns **true** or **false** depending on which check calls it.

Feeding this example will not display anything, since the entailment check cannot decide whether `L1` and `L2` are equal or not. In fact, both are possible: if `x` is bound to 1 then they are equal and if `x` is bound to 2 then they are not. Try feeding `x=1` or `x=2` to see what happens. What about the following example:

```
declare L1 L2 X in
  L1=[X]
  L2=[X]
  {Browse L1==L2}
```

Both lists contain the same unbound variable `x`. What will happen? Think about it before reading the answer in the footnote.<sup>16</sup> Here is a final example:

```
declare L1 L2 X in
  L1=[1 a]
  L2=[X b]
  {Browse L1==L2}
```

This will display **false**. While the comparison `1==X` blocks, further inspection of the two graphs shows that there is a definite difference, so the full check returns **false**.

### 2.7.3 Dynamic and static typing

“The only way of discovering the limits of the possible is to venture a little way past them into the impossible.”

– Clarke’s Second Law, *Arthur C. Clarke* (1917–)

It is important for a language to be *strongly-typed*, i.e., to have a type system that is enforced by the language. (This is contrast to a *weakly-typed* language, in which the internal representation of a type can be manipulated by a program. We will not speak further of weakly-typed languages in this book.) There are two major families of strong typing: dynamic typing and static typing. We have introduced the declarative model as being dynamically typed, but we have not yet explained the motivation for this design decision, nor the differences between static and dynamic typing that underlie it. In a dynamically-typed language, variables can be bound to entities of any type, so in general their type is known only at run time. In a statically-typed language, on the other hand, all variable types are known at compile time. The type can be declared by the programmer or inferred by the compiler. When designing a language, one of the major decisions to make is whether the language is to be dynamically typed, statically typed, or some mixture of both. What are the advantages and disadvantages of dynamic and static typing? The basic principle is that static typing puts restrictions on what programs one can write, reducing expressiveness of the language in return for giving advantages such as improved error catching ability, efficiency, security, and partial program verification. Let us examine this closer:

---

<sup>16</sup>The browser will display **true**, since `L1` and `L2` are equal no matter what `X` might be bound to.

- Dynamic typing puts no restrictions on what programs one can write. To be precise, all syntactically-legal programs can be run. Some of these programs will raise exceptions, possibly due to type errors, which can be caught by an exception handler. Dynamic typing gives the widest possible variety of programming techniques. The increased flexibility is felt quite strongly in practice. The programmer spends much less time adjusting the program to fit the type system.
- Dynamic typing makes it a trivial matter to do separate compilation, i.e., modules can be compiled without knowing anything about each other. This allows truly open programming, in which independently-written modules can come together at run time and interact with each other. It also makes program development scalable, i.e., extremely large programs can be divided into modules that can be compiled individually without recompiling other modules. This is harder to do with static typing because the type discipline must be enforced across module boundaries.
- Dynamic typing shortens the turnaround time between an idea and its implementation. It enables an incremental development environment that is part of the run-time system. It allows to test programs or program fragments even when they are in an incomplete or inconsistent state.
- Static typing allows to catch more program errors at compile time. The static type declarations are a partial specification of the program, i.e., they specify part of the program's behavior. The compiler's type checker verifies that the program satisfies this partial specification. This can be quite powerful. Modern static type systems can catch a surprising number of semantic errors.
- Static typing allows a more efficient implementation. Since the compiler has more information about what values a variable can contain, it can choose a more efficient representation. For example, if a variable is of boolean type, the compile can implement it with a single bit. In a dynamically-typed language, the compiler cannot always deduce the type of a variable. When it cannot, then it usually has to allocate a full memory word, so that any possible value (or a pointer to a value) can be accommodated.
- Static typing can improve the security of a program. Secure ADTs can be constructed based solely on the protection offered by the type system.

Unfortunately, the choice between dynamic and static typing is most often based on emotional ("gut") reactions, not on rational argument. Adherents of dynamic typing relish the expressive freedom and rapid turnaround it gives them and criticize the reduced expressiveness of static typing. On the other hand, adherents of static typing emphasize the aid it gives them for writing correct and efficient programs and point out that it finds many program errors at compile time. Little