```
end

local C={NewCell 0} in
   C:=25
   {Sqr fun {$} C end}
   {Browse @C}
end
```

The argument `A` is evaluated when the result is needed. The local variable `B` stores its result. If the argument is needed again, then `B` is used. This avoids reevaluating the function. In the `Sqr` example this is easy to implement since the result is clearly needed three times. If it is not clear from inspection whether the result is needed, then lazy evaluation can be used to implement call by need directly (see Exercise).

Call by need is exactly the same concept as lazy evaluation. The term "call by need" is more often used in a language with state, where the result of the function evaluation can be the name of a cell (a mutable variable). Call by name is lazy evaluation without memoization. The result of the function evaluation is not stored, so it is evaluated again each time it is needed.

Discussion

Which of these mechanisms (if any) is "right" or "best"? This has been the subject of much discussion (see, e.g., [116]). The goal of the kernel language approach is to factorize programming languages into a small set of programmer-significant concepts. For parameter passing, this justifies using *call by reference* as the primitive mechanism which underlies the other mechanisms. Unlike the others, call by reference does not depend on additional concepts such as cells or procedure values. It has a simple formal semantics and is efficient to implement. On the other hand, this does not mean that call by reference is always the right mechanism for programs. Other parameter passing mechanisms can be coded by combining call by reference with cells and procedure values. Many languages offer these mechanisms as linguistic abstractions.

6.5 Stateful collections

An important kind of ADT is the *collection*, which groups together a set of partial values into one compound entity. There are different kinds of collection depending on what operations are provided. Along one axis we distinguish indexed collections and unindexed collections, depending on whether or not there is rapid access to individual elements (through an index). Along another axis we distinguish extensible or inextensible collections, depending on whether the number of elements is variable or fixed. We give a brief overview of these different kinds of collections, starting with indexed collections.

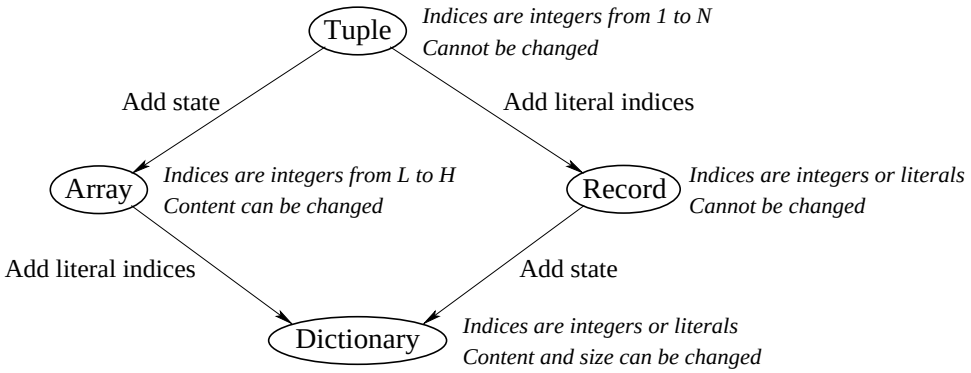

Figure 6.4: Different varieties of indexed collections

6.5.1 Indexed collections

In the context of declarative programming, we have already seen two kinds of indexed collection, namely *tuples* and *records*. We can add state to these two data types, allowing them to be updated in certain ways. The stateful versions of tuples and records are called *arrays* and *dictionaries*.

In all, this gives four different kinds of indexed collection, each with its particular trade-offs between expressiveness and efficiency (see Figure 6.4). With such a proliferation, how does one choose which to use? Section 6.5.2 compares the four and gives advice on how to choose among them.

Arrays

An *array* is a mapping from integers to partial values. The domain is a set of consecutive integers from a lower bound to an upper bound. The domain is given when the array is declared and cannot be changed afterwards. The range of the mapping can be changed. Both accessing and changing an array element are done in constant time. If you need to change the domain or if the domain is not known when you declare the array, then you should use a dictionary instead of an array. The Mozart system provides arrays as a predefined ADT in the `Array` module. Here are some of the more common operations:

- `A={NewArray L H I}` returns a new array with indices from `L` to `H`, inclusive, all initialized to `I`.
- `{Array.put A I X}` puts in `A` the mapping of `I` to `X`. This can also be written `A.I:=X`.
- `X={Array.get A I}` returns from `A` the mapping of `I`. This can also be written as `X=A.I`.
- `L={Array.low A}` returns the lower index bound of `A`.

- `H={Array.high A}` returns the higher index bound of `A`.
- `R={Array.toRecord L A}` returns a record with label `L` and the same items as the array `A`. The record is a tuple only if the lower index bound is 1.
- `A={Tuple.toArray T}` returns an array with bounds between 1 and `{Width T}`, where the elements of the array are the elements of `T`.
- `A2={Array.clone A}` returns a new array with exactly the same indices and contents as `A`.

There is a close relationship between arrays and tuples. Each of them maps one of a set of consecutive integers to partial values. The essential difference is that tuples are stateless and arrays are stateful. A tuple has fixed contents for its fields, whereas in an array the contents can be changed. It is possible to create a completely new tuple differing only on one field from an existing tuple, using the `Adjoin` and `AdjoinAt` operations. These take time and memory proportional to the number of features in the tuple. The `put` operation of an array is a constant time operation, and therefore much more efficient.

Dictionaries

A *dictionary* is a mapping from simple constants (atoms, names, or integers) to partial values. Both the domain and the range of the mapping can be changed. An *item* is a pair of one simple constant and a partial value. Items can be accessed, changed, added, or removed during execution. All operations are efficient: accessing and changing are done in constant time and adding/removal are done in amortized constant time. By *amortized* constant time we mean that a sequence of n add or removal operations is done in total time proportional to n, when n becomes large. This means that each individual operation may not be constant time, since occasionally the dictionary has to be reorganized internally, but reorganizations are relatively rare. The active memory needed by a dictionary is always proportional to the number of items in the mapping. Other than system memory, there are no limits to the number of fields in the mapping. Section 3.7.3 gives some ballpark measurements comparing stateful dictionaries to declarative dictionaries. The Mozart system provides dictionaries as a predefined ADT in the `Dictionary` module. Here are some of the more common operations:

- `D={NewDictionary}` returns a new empty dictionary.
- `{Dictionary.put D LI X}` puts in `D` the mapping of `LI` to `X`. This can also be written `D.LI:=X`.
- `X={Dictionary.get D LI}` returns from `D` the mapping of `LI`. This can also be written `X=D.LI`, i.e., with the same notation as for records.

- `X={Dictionary.condGet D LI Y}` returns from `D` the mapping of `LI`, if it exists. Otherwise, it returns `Y`. This is minor variation of `get`, but it turns out to be extremely useful in practice.
- `{Dictionary.remove D LI}` removes from `D` the mapping of `LI`.
- `{Dictionary.member D LI B}` tests in `D` whether `LI` exists, and binds `B` to the boolean result.
- `R={Dictionary.toRecord L D}` returns a record with label `L` and the same items as the dictionary `D`. The record is a "snapshot" of the dictionary's state at a given moment in time.
- `D={Record.toDictionary R}` returns a dictionary with the same items as the record `R`. This operation and the previous one are useful for saving and restoring dictionary state in pickles.
- `D2={Dictionary.clone D}` returns a new dictionary with exactly the same keys and items as `D`.

There is a close relationship between dictionaries and records. Each of them maps simple constants to partial values. The essential difference is that records are stateless and dictionaries are stateful. A record has a *fixed* set of fields and their contents, whereas in a dictionary the set of fields and their contents can be *changed*. Like for tuples, new records can be created with the `Adjoin` and `AdjoinAt` operations, but these take time proportional to the number of record features. The `put` operation of a dictionary is a constant time operation, and therefore much more efficient.

6.5.2 Choosing an indexed collection

The different indexed collections have different trade-offs in possible operations, memory use, and execution time. It is not always easy to decide which collection type is the best one in any given situation. We examine the differences between these collections to make this decision easier.

We have seen four types of indexed collections: tuples, records, arrays, and dictionaries. All provide constant-time access to their elements by means of indices, which can be calculated at run time. But apart from this commonality they are quite different. Figure 6.4 gives a hierarchy that shows how the four types are related to each other. Let us compare them:

- *Tuples.* Tuples are the most restrictive, but they are fastest and require least memory. Their indices are consecutive positive integers from 1 to a maximum `N` which is specified when the tuple is created. They can be used as arrays when the contents do not have to be changed. Accessing a tuple field is extremely efficient because the fields are stored consecutively.

- *Records.* Records are more flexible than tuples because the indices can be any literals (atoms or names) and integers. The integers do not have to be consecutive. The *record type*, i.e., the label and arity (set of indices), is specified when the record is created. Accessing record fields is nearly as efficient as accessing tuple fields. To guarantee this, records fields are stored consecutively, like for tuples. This implies that creating a *new* record type (i.e., one for which no record exists yet) is much more expensive than creating a new tuple type. A hash table is created when the record type is created. The hash table maps each index to its offset in the record. To avoid having to use the hash table on each access, the offset is cached in the access instruction. Creating new records of an already-existing type is as inexpensive as creating a tuple.

- *Arrays.* Arrays are more flexible than tuples because the content of each field can be changed. Accessing an array field is extremely efficient because the fields are stored consecutively. The indices are consecutive integers from any lower bound to any upper bound. The bounds are specified when the array is created. The bounds cannot be changed.

- *Dictionaries.* Dictionaries are the most general. They combine the flexibility of arrays and records. The indices can be any literals and integers and the content of each field can be changed. Dictionaries are created empty. No indices need to be specified. Indices can be added and removed efficiently, in amortized constant time. On the other hand, dictionaries take more memory than the other data types (by a constant factor) and have slower access time (also by a constant factor). Dictionaries are implemented as dynamic hash tables.

Each of these types defines a particular trade-off that is sometimes the right one. Throughout the examples in the book, we select the right indexed collection type whenever we need one.

6.5.3 Other collections

Unindexed collections

Indexed collections are not always the best choice. Sometimes it is better to use an unindexed collection. We have seen two unindexed collections: lists and streams. Both are declarative data types that collect elements in a linear sequence. The sequence can be traversed from front to back. Any number of traversals can be done simultaneously on the same list or stream. Lists are of finite, fixed length. Streams are incomplete lists; their tails are unbound variables. This means they can always be extended, i.e., they are potentially unbounded. The stream is one of the most efficient extensible collections, in both memory use and execution time. Extending a stream is more efficient than adding a new index to a dictionary and *much* more efficient than creating a new record type.

```
fun {NewExtensibleArray L H Init}
   A={NewCell {NewArray L H Init}}#Init
   proc {CheckOverflow I}
      Arr=@(A.1)
      Low={Array.low Arr}
      High={Array.high Arr}
   in
      if I>High then
         High2=Low+{Max I 2*(High-Low)}
         Arr2={NewArray Low High2 A.2}
      in
         for K in Low..High do Arr2.K:=Arr.K end
         (A.1):=Arr2
      end
   end
   proc {Put I X}
      {CheckOverflow I}
      @(A.1).I:=X
   end
   fun {Get I}
      {CheckOverflow I}
      @(A.1).I
   end
in extArray(get:Get put:Put)
end
```

Figure 6.5: Extensible array (stateful implementation)

Streams are useful for representing ordered sequences of messages. This is an especially appropriate representation since the message receiver will automatically synchronize on the arrival of new messages. This is the basis of a powerful declarative programming style called *stream programming* (see Chapter 4) and its generalization to *message passing* (see Chapter 5).

Extensible arrays

Up to now we have seen two extensible collections: streams and dictionaries. Streams are efficiently extensible but elements cannot be accessed efficiently (linear search is needed). Dictionaries are more costly to extend (but only by a constant factor) and they can be accessed in constant time. A third extensible collection is the extensible array. This is an array that is resized upon overflow. It has the advantages of constant-time access and significantly less memory usage than dictionaries (by a constant factor). The resize operation is amortized constant time, since it is only done when an index is encountered that is greater than the current size.

Extensible arrays are not provided as a predefined type by Mozart. We can

implement them using standard arrays and cells. Figure 6.5 shows one possible version, which allows an array to increase in size but not decrease. The call `{NewExtensibleArray L H X}` returns a secure extensible array `A` with initial bounds `L` and `H` and initial content `X`. The operation `{A.put I X}` puts `X` at index `I`. The operation `{A.get I}` returns the content at index `I`. Both operations extend the array whenever they encounter an index that is out of bounds. The resize operation always at least doubles the array's size. This guarantees that the amortized cost of the resize operation is constant. For increased efficiency, one could add "unsafe" `put` and `get` operations that do no bounds checking. In that case, the responsibility would be on the programmer to ensure that indices remain in bounds.

6.6 Reasoning with state

Programs that use state in a haphazard way are very difficult to understand. For example, if the state is visible throughout the whole program then it can be assigned anywhere. The only way to reason is to consider the whole program at once. Practically speaking, this is impossible for big programs. This section introduces a method, called *invariant assertions*, which allows to tame state. We show how to use the method for programs that have both stateful and declarative parts. The declarative part appears as logical expressions inside the assertions. We also explain the role of abstraction (deriving new proof rules for linguistic abstractions) and how to take dataflow execution into account.

The technique of invariant assertions is usually called *axiomatic semantics*, following Floyd, Hoare, and Dijkstra, who initially developed it in the 1960's and 1970's. The correctness rules were called "axioms" and the terminology has stuck ever since. Manna gave an early but still interesting presentation [118].

6.6.1 Invariant assertions

The method of invariant assertions allows to reason independently about parts of programs. This gets back one of the strongest properties of declarative programming. However, this property is achieved at the price of a rigorous organization of the program. The basic idea is to organize the program as a hierarchy of ADTs. Each ADT can use other ADTs in its implementation. This gives a directed graph of ADTs.

A hierarchical organization of the program is good for more than just reasoning. We will see it many times in the book. We find it again in the component-based programming of Section 6.7 and the object-oriented programming of Chapter 7.

Each ADT is specified with a series of *invariant assertions*, also called *invariants*. An invariant is a logical sentence that defines a relation among the ADT's arguments and its internal state. Each operation of the ADT assumes that some

invariant is true and, when it completes, assures the truth of another invariant. The operation's implementation guarantees this. In this way, using invariants decouples an ADT's implementation from its use. We can reason about each separately.

To realize this idea, we use the concept of *assertion*. An assertion is a logical sentence that is attached to a given point in the program, between two instructions. An assertion can be considered as a kind of boolean expression (we will see later exactly how it differs from boolean expressions in the computation model). Assertions can contain variable and cell identifiers from the program as well as variables and quantifiers that do *not* occur in the program, but are used just for expressing a particular relation. For now, consider a *quantifier* as a symbol, such as $\forall$ ("for all") and $\exists$ ("there exists"), that is used to express assertions that hold true over all values of variables in a domain, not just for one value.

Each operation O_i of the ADT is specified by giving two assertions A_i and B_i. The specification states that, if A_i is true just before O_i, then when O_i completes B_i will be true. We denote this by:

$$\{\ A_i\ \}\ O_i\ \{\ B_i\ \}$$

This specification is sometimes called a *partial correctness assertion*. It is partial because it is only valid if O_i terminates normally. A_i is called the precondition and B_i is called the postcondition. The specification of the complete ADT then consists of partial correctness assertions for each of its operations.

6.6.2 An example

Now that we have some inkling of how to proceed, let us give an example of how to specify a simple ADT and prove it correct. We use the stateful stack ADT we introduced before. To keep the presentation simple, we will introduce the notation we need gradually during the example. The notation is not complicated; it is just a way of writing boolean expressions that allows us to express what we need to. Section 6.6.3 defines the notation precisely.

Specifying an ADT

We begin by specifying the ADT independent of its implementation. The first operation creates a stateful bundled instance of a stack:

```
Stack={NewStack}
```

The function `NewStack` creates a new cell c, which is hidden inside the stack by lexical scoping. It returns a record of three operations, `Push`, `Pop`, and `IsEmpty`, which is bound to `Stack`. So we can say that the following is a specification of `NewStack`:

```
{ true }
Stack={NewStack}
{ @c = nil ∧ Stack = ops(push:Push pop:Pop isEmpty:IsEmpty) }
```

The precondition is **true**, which means that there are no special conditions. The notation @c denotes the content of the cell c.

This specification is incomplete since it does not define what the references `Push`, `Pop`, and `IsEmpty` mean. Let us define each of them separately. We start with `Push`. Executing `{Stack.push X}` is an operation that pushes `X` on the stack. We specify this as follows:

```
{ @c = S }
{Stack.push X}
{ @c = X|S }
```

The specifications of `NewStack` and `Stack.push` both mention the internal cell c. This is reasonable when proving correctness of the stack, but is not reasonable when using the stack, since we want the internal representation to be hidden. We can avoid this by introducing a predicate *stackContent* with following definition:

stackContent(`Stack`, `S`) $\equiv$ @c = `S`

where c is the internal cell corresponding to `Stack`. This hides any mention of the internal cell from programs using the stack. Then the specifications of `NewStack` and `Stack.push` become:

```
{ true }
Stack={NewStack}
{ stackContent(Stack,nil) ∧
  Stack = ops(push:Push pop:Pop isEmpty:IsEmpty) }

{ stackContent(Stack,S) }
{Stack.push X}
{ stackContent(Stack,X|S) }
```

We continue with the specifications of `Stack.pop` and `Stack.isEmpty`:

```
{ stackContent(Stack,X|S) }
Y={Stack.pop}
{ stackContent(Stack,S) ∧ Y = X }

{ stackContent(Stack,S) }
X={Stack.isEmpty}
{ stackContent(Stack,S) ∧ X = (S==nil) }
```

The full specification of the stack consists of these four partial correctness assertions. These assertions do not say what happens if a stack operation raises an exception. We will discuss this later.

Proving an ADT correct

The specification we gave above is how the stack should behave. But does our implementation actually behave in this way? To verify this, we have to check whether each partial correctness assertion is correct for our implementation. Here

is the implementation (to make things easier, we have unnested the nested statements):

```
fun {NewStack}
   C={NewCell nil}
   proc {Push X} S in S=@C C:=X|S end
   fun {Pop} S1 in
      S1=@C
      case S1 of X|S then C:=S X end
   end
   fun {IsEmpty} S in S=@C S==nil end
in
   ops(push:Push pop:Pop isEmpty:IsEmpty)
end
```

With respect to this implementation, we have to verify each of the four partial correctness assertions that make up the specification of the stack. Let us focus on the specification of the `Push` operation. We leave the other three verifications up to the reader. The definition of `Push` is:

```
proc {Push X}
S in
   S=@C
   C:=X|S
end
```

The precondition is **{** *stackContent*(`Stack`, s) **}**, which we expand to **{** `@C` $= s$ **}**, where `C` refers to the stack's internal cell. This means we have to prove:

{ `@C` $= s$ **}**
`S=@C`
`C:=X|S`
{ `@C` $=$ `X|`s **}**

The stack ADT uses the cell ADT in its implementation. To continue the proof, we therefore need to know the specifications of the cell operations `@` and `:=`. The specification of `@` is:

{ P **}**
$\langle$y$\rangle$ = `@`$\langle$x$\rangle$
{ $P \wedge \langle$y$\rangle$ = `@`$\langle$x$\rangle$ **}**

where $\langle$y$\rangle$ is an identifier, $\langle$x$\rangle$ is an identifier bound to a cell, and P is an assertion. The specification of `:=` is:

{ $P(\langle$exp$\rangle)$ **}**
$\langle$x$\rangle$`:=`$\langle$exp$\rangle$
{ $P($`@`$\langle$x$\rangle)$ **}**

where $\langle$x$\rangle$ is an identifier bound to a cell, $P($`@`$\langle$x$\rangle)$ is an assertion that contains `@`$\langle$x$\rangle$, and $\langle$exp$\rangle$ is an expression that is allowed in an assertion. These specifications are also called *proof rules*, since they are used as building blocks in a correctness

proof. When we apply each rule we are free to choose ⟨x⟩, ⟨y⟩, P, and ⟨exp⟩ to be what we need.

Let us apply the proof rules to the definition of `Push`. We start with the assignment statement and work our way backwards: given the postcondition, we determine the precondition. (With assignment, it is often easier to reason in the backwards direction.) In our case, the postcondition is `@C` $=$ `X|`s. Matching this to P(`@`⟨x⟩), we see that ⟨x⟩ is the cell `C` and P(`@C`) $\equiv$ `@C` $=$ `X|`s. Using the rule for `:=`, we replace `@C` by `X|S`, giving `X|S` $=$ `X|`s as the precondition.

Now let us reason forwards from the cell access. The precondition is `@C` $= s$. From the proof rule, we see that the postcondition is (`@C` $= s \wedge$ `S` $=$ `@C`). Bringing the two parts together gives:

```
{ @C = s }
S=@C
{ @C = s ∧ S = @C }
{ X|S = X|s }
C:=X|S
{ @C = X|s }
```

This is a valid proof because of two reasons. First, it strictly respects the proof rules for `@` and `:=`. Second, (`@C` $= s \wedge$ `S` $=$ `@C`) implies (`X|S` $=$ `X|`s).

6.6.3 Assertions

An assertion ⟨ass⟩ is a boolean expression that is attached to a particular place in a program, which we call a *program point*. The boolean expression is very similar to boolean expressions in the computation model. There are some differences because assertions are mathematical expressions used for reasoning, not program fragments. An assertion can contain identifiers ⟨x⟩, partial values x, and cell contents `@`⟨x⟩ (with the operator `@`). For example, we used the assertion `@C` $=$ `X|`s when reasoning about the stack ADT. An assertion can also contain quantifiers and their dummy variables. Finally, it can contain mathematical functions. These can correspond directly to functions written in the declarative model.

To evaluate an assertion it has to be attached to a program point. Program points are characterized by the environment that exists there. Evaluating an assertion at a program point means evaluating using this environment. We assume that all dataflow variables are sufficiently bound so that the evaluation gives **`true`** or **`false`**.

We use the notations $\wedge$ for logical conjunction (and), $\vee$ for logical disjunction (or), and $\neg$ for logical negation (not). We use the quantifiers *for all* ($\forall$) and *there exists* ($\exists$):

$\forall x$.⟨type⟩: ⟨ass⟩	⟨ass⟩ is true when x has any value of type ⟨type⟩.
$\exists x$.⟨type⟩: ⟨ass⟩	⟨ass⟩ is true for at least one value x of type ⟨type⟩.

In each of these quantified expressions, ⟨type⟩ is a legal type of the declarative model as defined in Section 2.3.2.

The reasoning techniques we introduce here can be used in all stateful languages. In many of these languages, e.g., C++ and Java, it is clear from the declaration whether an identifier refers to a mutable variable (a cell or attribute) or a value (i.e., a constant). Since there is no ambiguity, the `@` symbol can safely be left out for them. In our model, we keep the `@` because we can distinguish between the name of a cell (`C`) and its content (`@C`).

6.6.4 Proof rules

For each statement S in the kernel language, we have a proof rule that shows all possible correct forms of $\{ A \} S \{ B \}$. This proof rule is just a specification of S. We can prove the correctness of the rule by using the semantics. Let us see what the rules are for the stateful kernel language.

Binding

We have already shown one rule for binding, in the case $\langle \mathsf{y} \rangle = \texttt{@}\langle \mathsf{x} \rangle$, where the right-hand side is the content of a cell. The general form of a binding is $\langle \mathsf{x} \rangle = \langle \mathsf{exp} \rangle$, where $\langle \mathsf{exp} \rangle$ is a declarative expression that evaluates to a partial value. The expression may contain cell accesses (calls to `@`). This gives the following proof rule:

{ P **}**
$\langle \mathsf{x} \rangle = \langle \mathsf{exp} \rangle$
{ $P \wedge \langle \mathsf{x} \rangle = \langle \mathsf{exp} \rangle$ **}**

where P is an assertion.

Assignment

The following proof rule holds for assignment:

{ $P(\langle \mathsf{exp} \rangle)$ **}**
$\langle \mathsf{x} \rangle$`:=`$\langle \mathsf{exp} \rangle$
{ $P(\texttt{@}\langle \mathsf{x} \rangle)$ **}**

where $\langle \mathsf{x} \rangle$ refers to a cell, $P(\texttt{@}\langle \mathsf{x} \rangle)$ is an assertion that contains $\texttt{@}\langle \mathsf{x} \rangle$, and $\langle \mathsf{exp} \rangle$ is a declarative expression.

Conditional (`if` statement)

The **`if`** statement has the form:

`if` $\langle \mathsf{x} \rangle$ **`then`** $\langle \mathsf{stmt} \rangle_1$ **`else`** $\langle \mathsf{stmt} \rangle_2$ **`end`**

The behavior depends on whether $\langle \mathsf{x} \rangle$ is bound to **`true`** or **`false`**. If we know:

$$\{ P \wedge \langle \mathsf{x} \rangle = \textbf{`true`} \} \langle \mathsf{stmt} \rangle_1 \{ Q \}$$

and also:

$$\{\ P \wedge \langle \mathsf{x} \rangle = \texttt{false}\ \}\ \langle \mathsf{stmt} \rangle_2\ \{\ Q\ \}$$

then we can conclude:

$$\{\ P\ \}\ \texttt{if}\ \langle \mathsf{x} \rangle\ \texttt{then}\ \langle \mathsf{stmt} \rangle_1\ \texttt{else}\ \langle \mathsf{stmt} \rangle_2\ \texttt{end}\ \{\ Q\ \}.$$

Here P and Q are assertions and $\langle \mathsf{stmt} \rangle_1$ and $\langle \mathsf{stmt} \rangle_2$ are statements in the kernel language. We summarize this rule with the following notation:

$$\frac{\begin{array}{c}\{\ P \wedge \langle \mathsf{x} \rangle = \texttt{true}\ \}\ \langle \mathsf{stmt} \rangle_1\ \{\ Q\ \} \\ \{\ P \wedge \langle \mathsf{x} \rangle = \texttt{false}\ \}\ \langle \mathsf{stmt} \rangle_2\ \{\ Q\ \}\end{array}}{\{\ P\ \}\ \texttt{if}\ \langle \mathsf{x} \rangle\ \texttt{then}\ \langle \mathsf{stmt} \rangle_1\ \texttt{else}\ \langle \mathsf{stmt} \rangle_2\ \texttt{end}\ \{\ Q\ \}}$$

In this notation, the premises are above the horizontal line and the conclusion is below it. To use the rule, we first have to prove the premises.

Procedure without external references

Assume the procedure has the following form:

```
proc {⟨p⟩ ⟨x⟩1 ... ⟨x⟩n}
   ⟨stmt⟩
end
```

where the only external references of $\langle \mathsf{stmt} \rangle$ are $\{\langle \mathsf{x} \rangle_1, ..., \langle \mathsf{x} \rangle_n\}$. Then the following rule holds:

$$\frac{\{\ P(\overline{\langle \mathsf{x} \rangle})\ \}\ \langle \mathsf{stmt} \rangle\ \{\ Q(\overline{\langle \mathsf{x} \rangle})\ \}}{\{\ P(\overline{\langle \mathsf{y} \rangle})\ \}\ \{\ \langle \mathsf{p} \rangle\ \langle \mathsf{y} \rangle_1\ ...\ \langle \mathsf{y} \rangle_n\ \}\ \{\ Q(\overline{\langle \mathsf{y} \rangle})\ \}}$$

where P and Q are assertions and the notation $\overline{\langle \mathsf{x} \rangle}$ means $\langle \mathsf{x} \rangle_1, ..., \langle \mathsf{x} \rangle_n$.

Procedure with external references

Assume the procedure has the following form:

```
proc {⟨p⟩ ⟨x⟩1 ... ⟨x⟩n}
   ⟨stmt⟩
end
```

where the external references of $\langle \mathsf{stmt} \rangle$ are $\{\langle \mathsf{x} \rangle_1, ..., \langle \mathsf{x} \rangle_n, \langle \mathsf{z} \rangle_1, ..., \langle \mathsf{z} \rangle_k\}$. Then the following rule holds:

$$\frac{\{\ P(\overline{\langle \mathsf{x} \rangle}, \overline{\langle \mathsf{z} \rangle})\ \}\ \langle \mathsf{stmt} \rangle\ \{\ Q(\overline{\langle \mathsf{x} \rangle}, \overline{\langle \mathsf{z} \rangle})\ \}}{\{\ P(\overline{\langle \mathsf{y} \rangle}, \overline{\langle \mathsf{z} \rangle})\ \}\ \{\ \langle \mathsf{p} \rangle\ \langle \mathsf{y} \rangle_1\ ...\ \langle \mathsf{y} \rangle_n\ \}\ \{\ Q(\overline{\langle \mathsf{y} \rangle}, \overline{\langle \mathsf{z} \rangle})\ \}}$$

where P and Q are assertions.

While loops

The previous rules are sufficient to reason about programs that use recursion to do looping. For stateful loops it is convenient to add another basic operation: the **while** loop. Since we can define the **while** loop in terms of the kernel language, it does not add any new expressiveness. Let us therefore define the **while** loop as a linguistic abstraction. We introduce the new syntax:

while ⟨expr⟩ **do** ⟨stmt⟩ **end**

We define the semantics of the **while** loop by translating it into simpler operations:

```
{While fun {$} ⟨expr⟩ end proc {$} ⟨stmt⟩ end}

proc {While Expr Stmt}
   if {Expr} then {Stmt} {While Expr Stmt} end
end
```

Let us add a proof rule specifically for the **while** loop:

$$\frac{\{\ P \wedge \langle \text{expr} \rangle\ \}\ \langle \text{stmt} \rangle\ \{\ P\ \}}{\{\ P\ \}\ \textbf{while}\ \langle \text{expr} \rangle\ \textbf{do}\ \langle \text{stmt} \rangle\ \textbf{end}\ \{\ P \wedge \neg \langle \text{expr} \rangle\ \}}$$

We can prove that the rule is correct by using the definition of the **while** loop and the method of invariant assertions. It is usually easier to use this rule than to reason directly with recursive procedures.

For loops

In Section 3.6.3 we saw another loop construct, the **for** loop. In its simplest form, this loops over integers:

for ⟨x⟩ **in** ⟨y⟩**..**⟨z⟩ **do** ⟨stmt⟩ **end**

This is also a linguistic abstraction, which we define as follows:

```
{For ⟨y⟩ ⟨z⟩ proc {$ ⟨x⟩} ⟨stmt⟩ end}

proc {For I H Stmt}
   if I=<H then {Stmt I} {For I+1 H Stmt} else skip end
end
```

We add a proof rule specifically for the **for** loop:

$$\frac{\forall i . \langle \text{y} \rangle \leq i \leq \langle \text{z} \rangle : \{\ P_{i-1} \wedge \langle \text{x} \rangle = i\ \}\ \langle \text{stmt} \rangle\ \{\ P_i\ \}}{\{\ P_{\langle \text{y} \rangle - 1}\ \}\ \textbf{for}\ \langle \text{x} \rangle\ \textbf{in}\ \langle \text{y} \rangle\textbf{..}\langle \text{z} \rangle\ \textbf{do}\ \langle \text{stmt} \rangle\ \textbf{end}\ \{\ P_{\langle \text{z} \rangle}\ \}}$$

Watch out for the initial index of P! Because a **for** loop starts with ⟨y⟩, the initial index of P has to be $\langle \text{y} \rangle - 1$, which expresses that we have not yet started the loop. Just like for the **while** loop, we can prove that this rule is correct by using the definition of the **for** loop. Let us see how this rule works with a simple example. Consider the following code which sums the elements of an array:

```
local C={NewCell 0} in
   for I in 1..10 do C:=@C+A.I end
end
```

where A is an array with indices from 1 to 10. Let us choose an invariant:

$$P_i \equiv \texttt{@C} = \sum_{j=1}^{i} \texttt{A}_j$$

where $\texttt{A}_j$ is the j-th element of A. This invariant simply defines an intermediate result of the loop calculation. With this invariant we can prove the premise of the **for** rule:

$$\{\ \texttt{@C} = \sum_{j=1}^{i-1} \texttt{A}_j \wedge \texttt{I} = i\ \}$$
```
C:=@C+A.I
```
$$\{\ \texttt{@C} = \sum_{j=1}^{i} \texttt{A}_j\ \}$$

This follows immediately from the assignment rule. Since P_0 is clearly true just before the loop, it follows from the **for** rule that P_{10} is true. Therefore C contains the sum of all array elements.

Reasoning at higher levels of abstraction

The **while** and **for** loops are examples of reasoning at a higher level of abstraction than the kernel language. For each loop, we defined the syntax and its translation into the kernel language, and then gave a proof rule. The idea is to verify the proof rule once and for all and then to use it as often as we like. This approach, defining new concepts and their proof rules, is the way to go for practical reasoning about stateful programs. Always staying at the kernel language level is much too verbose for all but toy programs.

Aliasing

The proof rules given above are correct if there is no aliasing. They need to be modified in obvious ways if aliasing can occur. For example, assume that C and D both reference the same cell and consider the assignment C:=@C+1. Say the postcondition is @C = 5 ∧ @D = 5 ∧ C = D. The standard proof rule lets us calculate the precondition by replacing @C by @C + 1. This gives an incorrect result because it does not take into account that D is aliased to C. The proof rule can be corrected by doing the replacement for @D as well.

6.6.5 Normal termination

Partial correctness reasoning does not say anything about whether or not a program terminates normally. It just says, *if* a program terminates normally, then such and such is true. This makes reasoning simple, but it is only part of the story. We also have to prove termination. There are three ways that a program can fail to terminate normally:

- Execution goes into an infinite loop. This is a programmer error due to the calculation not making progress towards its goal.
- Execution blocks because a dataflow variable is not sufficiently bound inside an operation. This is a programmer error due to overlooking a calculation or a deadlock situation.
- Execution raises an exception because of an error. The exception is an abnormal exit. In this section we consider only one kind of error, a type error, but similar reasoning can be done for other kinds of errors.

Let us see how to handle each case.

Progress reasoning

Each time there is a loop, there is the danger that it will not terminate. To show that a loop terminates, it suffices to show that there is a function that is nonnegative and that always decreases upon successive iterations.

Suspension reasoning

It suffices to show that all variables are sufficiently bound before they are used. For each use of the variable, tracing back all possible execution paths should always come across a binding.

Type checking

To show that there are no type errors, it suffices to show that all variables are of the correct type. This is called *type checking*. Other kinds of errors need other kinds of checking.

6.7 Program design in the large

"Good software is good in the large *and* in the small, in its high-level architecture and in its low-level details."
– Object-oriented software construction, 2nd ed., *Bertrand Meyer* (1997)

"An efficient and a successful administration manifests itself equally in small as in great matters."
– Memorandum, August 8, 1943, *Winston Churchill* (1874–1965)

Programming in the large is programming by a team of people. It involves all aspects of software development that require communication and coordination between people. Managing a team so they work together efficiently is difficult in any area–witness the difficulty of training football teams. For programming,

it is especially difficult since programs are generally quite unforgiving of small errors. Programs demand an exactness which is hard for human beings to satisfy. Programming in the large is often called *software engineering*.

This section builds on Section 3.9's introduction to programming in the small. We explain how to develop software in small teams. We do not explain what to do for larger teams (with hundreds or thousands of members)–that is the topic of another book.

6.7.1 Design methodology

Many things both true and false have been said about the correct design methodology for programming in the large. Much of the existing literature consists of extrapolation based on limited experiments, since rigorous validation is so difficult. To validate a new idea, several otherwise identical teams would have to work in identical circumstances. This has rarely been done and we do not attempt it either.

This section summarizes the lessons we have learned from our own systems-building experience. We have designed and built several large software systems [198, 29, 129]. We have thought quite a bit about how to do good program design in a team and looked at how other successful developer teams work. We try to isolate the truly useful principles from the others.

Managing the team

The first and most important task is to make sure that the team works together in a coordinated fashion. There are three ideas for achieving this:

- *Compartmentalize the responsibility* of each person. For example, each component can be assigned to one person, who is responsible for it. We find that responsibilities should respect component boundaries and not overlap. This avoids interminable discussions about who should have fixed a problem.

- In contrast to responsibility, *knowledge should be freely exchanged*, not compartmentalized. Team members should frequently exchange information about different parts of the system. Ideally, there should be no team member who is indispensable. All major changes to the system should be discussed among knowledgeable team members before being implemented. This greatly improves the system's quality. It is important also that a component's owner have the last word on how the component should be changed. Junior team members should be apprenticed to more senior members to learn about the system. Junior members become experienced by being given specific tasks to do, which they do in as independent a fashion as possible. This is important for the system's longevity.

- Carefully *document each component interface*, since it is also the interface between the component's owner and other team members. Documentation

is especially important, much more so than for programming in the small. A good documentation is a pillar of stability that can be consulted by all team members.

Development methodology

There are many ways to organize software development. The techniques of top-down, bottom-up, and even "middle-out" development have been much discussed. None of these techniques is really satisfactory, and combining them does not help much. The main problem is that they all rely on the system's requirements and specification to be fairly complete to start with, i.e., that the designers anticipated most of the functional and non-functional requirements. Unless the system is very well understood, this is almost impossible to achieve. In our experience, an approach that works well is the *thin-to-thick* approach:

- Start with a *small set of requirements*, which is a subset of the complete set, and build a complete system that satisfies them. The system specification and architecture are "empty shells": they are just complete enough so that a running program can be built, but they do not solve the user problems.
- Then *continuously extend the requirements*, extending the other layers as needed to satisfy them. Using an organic metaphor, we say the application is "grown" or "evolved". This is sometimes called an *evolutionary* approach. At all times, there is a running system that satisfies its specification and that can be evaluated by its potential users.
- *Do not optimize* during the development process. That is, do not make the design more complex just to increase performance. Use simple algorithms with acceptable complexity and keep a simple design with the right abstractions. Do not worry about the overhead of these abstractions. Performance optimization can be done near the end of development, but only if there are performance problems. Profiling should then be used to find those parts of the system (usually very small) that should be rewritten.
- *Reorganize the design as necessary* during development, to keep a good component organization. Components should encapsulate design decisions or implement common abstractions. This reorganization is sometimes called "refactoring". There is a spectrum between the extremes of completely planning a design and completely relying on refactoring. The best approach is somewhere in the middle.

The thin-to-thick approach has many advantages:

- Bugs of all sizes are detected early and can be fixed quickly.
- Deadline pressure is much relieved, since there is always a working application.

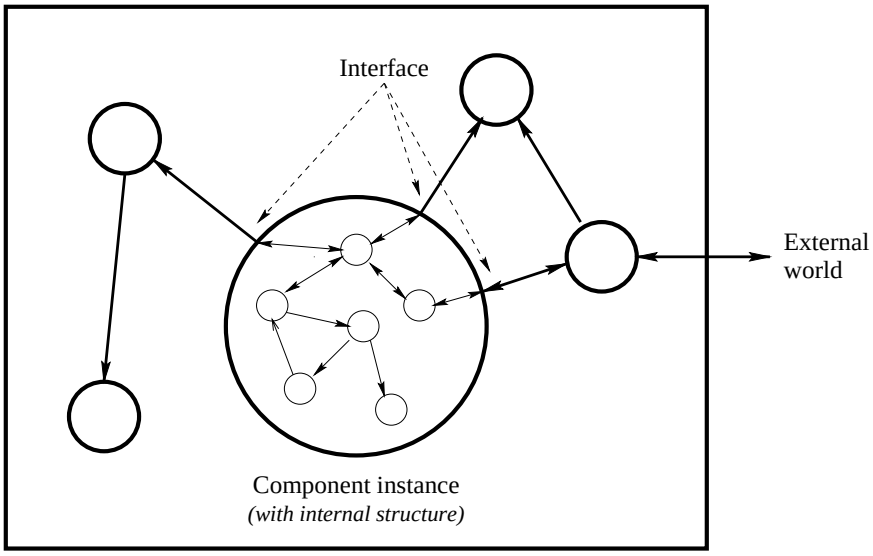

Figure 6.6: A system structured as a hierarchical graph

- Developers are more motivated, because they have quick feedback on their efforts.
- Users are more likely to get what they really need, because they have a chance to use the application during the development process.
- The architecture is more likely to be good, since it can be corrected early on.
- The user interface is more likely to be good, since it is continuously improved during the development process.

For most systems, even very small ones, we find that it is almost impossible to determine in advance what the real requirements are, what a good architecture is to implement them, and what the user interface should be. The thin-to-thick approach works well partly because it makes very few assumptions up front. For a complementary view, we recommend studying *extreme programming*, which is another approach that emphasizes the trade-off between planning and refactoring [185].

6.7.2 Hierarchical system structure

How should the system be structured to support teamwork and thin-to-thick development? One way that works in practice is to structure the application as a hierarchical graph with well-defined interfaces at each level (see Figure 6.6). That is, the application consists of a set of nodes where each node interacts with some other nodes. Each node is a component instance. Each node is itself a

small application, and can itself be decomposed into a graph. The decomposition bottoms out when we reach primitive components that are provided by the underlying system.

Component connection

The first task when building the system is to connect the components together. This has both a static and a dynamic aspect:

- **Static structure**. This consists of the component graph that is known when the application is designed. These components can be linked as soon as the application starts up. Each component instance corresponds roughly to a bundle of functionality that is sometimes known as a *library* or a *package*. For efficiency, we would like each component instance to exist at most once in the system. If a library is needed by different parts of the application, then we want the instances to share this same library. For example, a component may implement a graphics package; the whole application can get by with just one instance.

- **Dynamic structure**. Often, an application will do calculations with components at run time. It may want to link new components, which are known only at run time. Or it may want to calculate a new component and store it. Component instances need not be shared; perhaps several instances of a component are needed. For example, a component may implement a database interface. Depending on whether there are one or more external databases, one or more instances of the component should be loaded. This is determined at run time, whenever a database is added.

Figure 6.7 shows the resulting structure of the application, with some components linked statically and others linked dynamically.

Static structure To support the static structure, it is useful to make components into compilation units stored in files. We call these components *functors* and their instances *modules*. A functor is a component that can be stored in a file. It is a *compilation unit* because the file can be compiled independently of other functors. The functor's dependencies are given as filenames. In order to be accessible by other functors, the functor *must* be stored in a file. This allows other functors to specify that they need it by giving the filename.

A functor has two representations: a source form, which is just a text, and a compiled form, which is a value in the language. If the source form is in a file `foo.oz`, whose entire content is of the form **`functor ... end`**, then it can be compiled to give another file, `foo.ozf`, that contains the compiled form. The content of file *foo.oz* looks like this:

```
functor
import OtherComp1 at File1
```

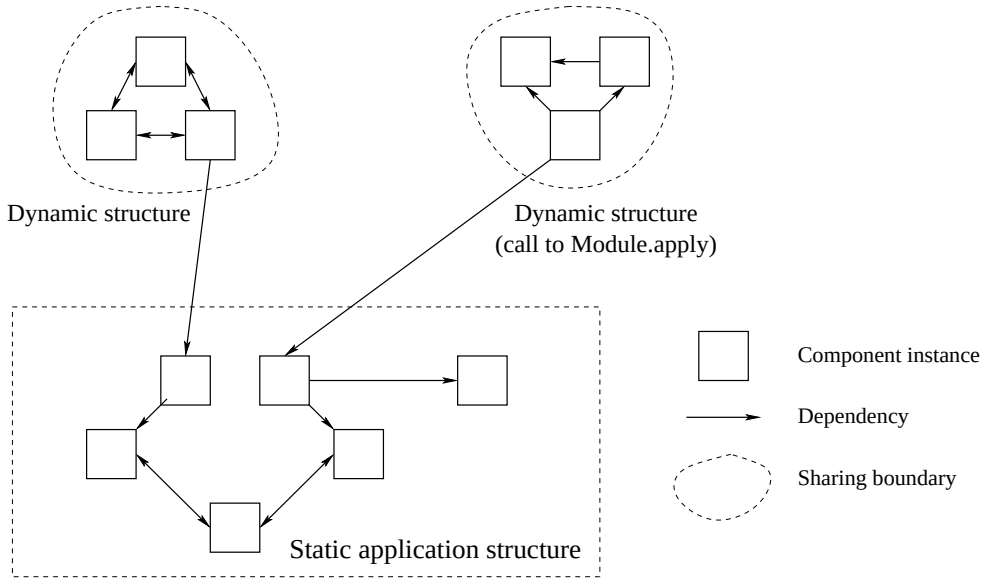

Figure 6.7: System structure – static and dynamic

```
      OtherComp2 at File2
      ...
      OtherCompN at FileN
export op1:X1 op2:X2 ... opK:Xk
define
   % Define X1, ..., Xk
   ...
end
```

This component depends on the other components `OtherComp1`, ..., `OtherCompN`, stored respectively in files `File1`, ..., `FileN`. It defines a module with fields `op1`, ..., `opK`, referenced by `X1`, ..., `Xk` and defined in the functor body.

An application is just one compiled functor. To run the application, all the other compiled functors that it needs, directly or indirectly, must be brought together and instantiated. When the application is run, it loads its components and links them. Linking a component means to evaluate it with its imported modules as arguments and to give the result to the modules that need it. Linking can be done when the application starts up (static linking), or by need, i.e., one by one as they are needed (dynamic linking). We have found that dynamic linking is usually preferable, as long as all the compiled functors are quickly accessible (e.g., exist on the local system). With this default the application starts up quickly and it takes up only as much memory space as it needs.

Dynamic structure In terms of the language, a functor is just another language entity. If a program contains a statement of the form `X=functor ... end` then `X` will be bound to the functor value. Like a procedure, the functor

may have external references. Instantiating a functor gives a module, which is the set of language entities created by initializing the functor. An interface is a record giving the externally-visible language entities of the module. In a convenient abuse of notation, this record is sometimes called the module. There exists a single operation to instantiate functors:

- `Ms={Module.apply Fs}`. Given a list `Fs` of functors (as language entities), this instantiates all the functors and creates a list `Ms` of modules. Within the scope of a `Module.apply` call, functors are shared. That is, if the same functor is mentioned more than once, then it is only linked once.

Each call to `Module.apply` results in a new, fresh set of modules. This operation is part of the module `Module`. If the functor is stored in a file it must be loaded first before calling `Module.apply`.

Component communication

Once the components are connected together, they have to communicate with each other. Here are six of the most popular protocols for component communication. We give them in increasing order of component independence:

1. **Procedure**. The first organization is where the application is sequential and where one component calls the other like a procedure. The caller is not necessarily the only initiator; there can be nested calls where the locus of control passes back and forth between components. But there is only one global locus of control, which strongly ties together the two components.

2. **Coroutine**. A second organization is where two components each evolve independently, but still in a sequential setting. This introduces the concept of a *coroutine* (see Section 4.4.2). A component calls another, which continues where it left off. There are several loci of control, one for each coroutine. This organization is looser than the previous one, but the components are still dependent because they execute in alternation.

3. **Concurrent and synchronous**. A third organization is one in which each component evolves independently of the others and can initiate and terminate communications with another component, according to some protocol that both components agree on. The components execute concurrently. There are several loci of control, called *threads*, which evolve independently (see Chapters 4 and 8). Each component still calls the others synchronously, i.e., each call waits for a response.

4. **Concurrent and asynchronous**. A fourth organization is a set of concurrent components that communicate through asynchronous channels. Each component sends messages to others, but does not have to wait for a response. The channels can have FIFO order (messages received in order of

sending) or be unordered. The channels are called streams in Chapter 4 and ports in Chapter 8. In this organization each component still knows the identity of the component with which it communicates.

5. **Concurrent mailbox**. A fifth organization is a variation of the fourth. The asynchronous channels behave like *mailboxes*. That is, it is possible to do pattern matching to extract messages from the channels, without disturbing the messages that remain. This is very useful for many concurrent programs. It is a basic operation in the Erlang language, which has FIFO mailboxes (see Chapter 5). Unordered mailboxes are also possible.

6. **Coordination model**. A sixth organization is where the components can communicate without the senders and receivers knowing each other's identities. An abstraction called *tuple space* lives at the interface. The components are concurrent and interact solely through the common tuple space. One component can insert a message asynchronously and another can retrieve the message. Section 8.3.2 defines one form of tuple space abstraction and shows how to implement it.

The model independence principle

Each component of the system is written in a particular computation model. Section 4.7.6 has summarized the most popular computation models used to program components. During development, a component's internal structure may change drastically. It is not uncommon for its computation model to change. A stateless component can become stateful (or concurrent, or distributed, etc.), or vice versa. If such a change happens inside a component, then it is not necessary to change its interface. The interface needs to change only if the externally-visible functionality of the component changes. This is an important modularity property of the computation models. As long as the interface is the same, this property guarantees that it is not necessary to change anything else in the rest of the system. We consider this property as a basic design principle for the computation models:

> **Model independence principle**
>
> The interface of a component should be independent of the computation model used to implement the component. The interface should depend only on the externally-visible functionality of the component.

A good example of this principle is *memoization*. Assume the component is a function that calculates its result based on one argument. If the calculation is time consuming, then keeping a cache of argument-result pairs can greatly reduce the execution time. When the function is called, check whether the argument is in the cache. If so, return the result directly without doing the calculation.

If not, do the calculation and add the new argument-result pair to the cache. Section 10.3.2 has an example of memoization. Since the memoization cache is stateful, changing a component to do memoization means that the component may change from using the declarative model to using the stateful model. The model independence principle implies that this can be done without changing anything else in the program.

Efficient compilation versus efficient execution

A component is a compilation unit. We would like to compile a component as quickly and efficiently as possible. This means we would like to compile a component separately, i.e., without knowing about other components. We would also like the final program, in which all components are assembled, to be as efficient and compact as possible. This means we would like to do compile-time analysis, e.g., type checking, Haskell-style type inference, or global optimization.

There is a strong tension between these two desires. If the compilation is truly separate then analysis cannot cross component boundaries. To do a truly global analysis, the compiler must in general be able to look at the whole program at once. This means that for many statically-typed languages, compiling large programs (more than, say, a million lines of source code) requires much time and memory.

There are many clever techniques that try to get the best of both worlds. Ideally, these techniques should not make the compiler too complicated. This is a difficult problem. After five decades of experience in language and compiler design, it is still an active research topic.

Commercial-quality systems span the whole spectrum from completely separate compilation to sophisticated global analysis. Practical application development can be done at any point of the spectrum. The Mozart system is at one extreme of the spectrum. Since it is dynamically typed, components can be compiled without knowing anything about each other. This means that compilation is completely scalable: compiling a component takes the same time, independent of whether it is used in a million-line program or in a thousand-line program. On the other hand, there are disadvantages: less optimization is possible and type mismatches can only be detected at run-time. Whether or not these issues are critical depends on the application and the experience of the developer.

6.7.3 Maintainability

Once the system is built and works well, we have to make sure that it keeps working well. The process of keeping a system working well after it is deployed is called *maintenance*. What is the best way to structure systems so they are maintainable? From our experience, here are some of the most important principles. We look at this from the viewpoint of single components and from the viewpoint of the system.

Component design

There are good ways and bad ways to design components. A bad way is to make a flowchart and carve it up into pieces, where each piece is a component. Much better is to think of a component as an abstraction. For example, assume that we are writing a program that uses lists. Then it is almost always a good idea to gather all list operations into a component, which defines the list abstraction. With this design, lists can be implemented, debugged, changed, and extended without touching the rest of the program. For example, say we want to use the program with lists that are too big for main memory. It is enough to change the list component to store them on files instead of in main memory.

Encapsulate design decisions More generally, we can say that a component should encapsulate a design decision.[6] That way, when the design decision is changed, only that component has to be changed. This a very powerful form of modularity. The usefulness of a component can be evaluated by seeing what changes it accommodates. For example, consider a program that calculates with characters, such as the word frequency example of Section 3.9.4. Ideally, the decision which character format to use (for example, ASCII, Latin-1, or Unicode) should be encapsulated in one component. This makes it simple to change from one format to another.

Avoid changing component interfaces A component can be changed by changing its implementation or by changing its interface. Changing the interface is problematic since all components that depend on the changed interface have to be rewritten or recompiled. Therefore, changing the interface should be avoided. But in practice, interface changes cannot be avoided during the design of a component. All we can do is minimize their frequency. To be precise, the interfaces of often-needed components should be designed as carefully as possible from the start.

Let us give a simple example to make this clear. Consider a component that sorts lists of character strings. It can change its sorting algorithm without changing its interface. This can often be done without recompiling the rest of the program, simply by linking in the new component. On the other hand, if the character format is changed then the component might require a different interface. For example, characters can change size from one to two bytes (if ASCII is replaced with Unicode). This requires recompiling all the components that use the changed component (directly or indirectly), since the compiled code may depend on the character format. Recompilation can be onerous; changing a 10-line component might require recompiling most of the program, if the component is used often.

[6] More romantically, it is sometimes said that the component has a "secret".

System design

Fewest possible external dependencies A component that depends on another, i.e., it requires the other for its operation, is a source of maintenance problems. If the other is changed, then the first must be changed as well. This is a major source of "software rot", i.e., once-working software that stops working. For example, LaTeX is a popular document preparation system in the scientific community that is noted for its high-quality output [108]. A LaTeX document can have links to other files, to customize and extend its abilities. Some of these other files, called *packages*, are fairly standardized and stable. Others are simply local customizations, called *style files*. In our experience, it is very bad for `LaTeX` documents to have links to style files in other, perhaps global, directories. If these are changed, then the documents can often no longer be pageset. To aid maintenance, it is much preferable to have copies of the style files in each document directory. This satisfies a simple invariant: each document is guaranteed to be pagesettable at all times ("working software keeps working"). This invariant is an enormous advantage that far outweighs the two disadvantages: (1) the extra memory needed for the copies and (2) the possibility that a document may use an older style file. If a style file is updated, the programmer is free to use the new version in the document, but only if necessary. Meanwhile, the document stays consistent. A second advantage is that it is easy to send the document from one person to another, since it is self-contained.

Fewest possible levels of indirection This is related to the previous rule. When A points to B, then updating B requires updating A. Any indirection is a kind of "pointer". The idea is to avoid the pointer becoming *dangling*, i.e., its destination no longer makes sense to the source. An action at B may cause A's pointer to become dangling. B doesn't know about A's pointer and so cannot prevent such a thing. A stop-gap is never to change B, but only to make modified copies. This can work well if the system does automatic global memory management.

Two typical examples of problematic pointers are symbolic links in a Unix file system and URLs. Symbolic links are pernicious for system maintenance. They are convenient because they can refer to other mounted directory trees, but in fact they are a big cause of system problems. URLs are known to be extremely flaky. They are often referenced in printed documents, but their lifetime is usually much less than that of the document. This is both because they can quickly become dangling and because the Internet has a low quality of service.

Dependencies should be predictable For example, consider a 'localize' command that is guaranteed to retrieve a file over a network and make a local copy. It has simple and predictable behavior, unlike the "page caching" done by Web browsers. Page caching is a misnomer, because a true cache maintains coherence between the original and the copy. For any such "cache", the replacement policy

should be clearly indicated.

Make decisions at the right level For example, time outs are bad. A time out is an irrevocable decision made at a low level in the system (a deeply-nested component) that propagates all the way to the top level without any way for the intermediate components to intervene. This behavior short-circuits any efforts the application designer may do to mask the problem or solve it.

Documented violations Whenever one of the previous principles is violated, perhaps for a good reason (e.g., physical constraints such as memory limitations or geographic separation force a pointer to exist), then this should be documented! That is, all external dependencies, all levels of indirection, all non-predictable dependencies, and all irrevocable decisions, should be documented.

Simple bundling hierarchy A system should not be stored in a dispersed way in a file system, but should be together in one place as much as possible. We define a simple hierarchy of how to bundle system components. We have found this hierarchy to be useful for documents as well as applications. The easiest-to-maintain design is first. For example, if the application is stored in a file system then we can define the following order:

1. If possible, put the whole application in one file. The file may be structured in sections, corresponding to components.

2. If the above is not possible (for example, there are files of different types or different people are to work on different parts simultaneously), then put the whole application in one directory.

3. If the above is not possible (for example, the application is to be compiled for multiple platforms), put the application in a directory hierarchy with one root.

6.7.4 Future developments

Components and the future of programming

The increased use of components is changing the programming profession. We see two major ways this change is happening. First, components will make programming accessible to application users, not just professional developers. Given a set of components at a high level of abstraction and an intuitive graphical user interface, a user can do many simple programming tasks by himself or herself. This tendency has existed for a long time in niche applications such as statistics packages, signal processing packages, and packages for control of scientific experiments. The tendency will eventually encompass applications for the general public.

Second, programming will change for professional developers. As more and more useful components are developed, the granularity of programming will increase. That is, the basic elements used by programmers will more often be large components instead of fine-grained language operations. This trend is visible in programming tools such as Visual Basic and in component environments such as Enterprise Java Beans. The main bottleneck limiting this evolution is the specification of component behavior. Current components tend to be overcomplicated and have vague specifications. This limits their possible uses. The solution, in our view, is to make components simpler, better factorize their functionalities, and improve how they can be connected together.

Compositional versus noncompositional design

Hierarchical composition may seem like a very natural way to structure a system. In fact, it's not "natural" at all! Nature uses a very different approach, which might be called noncompositional. Let us compare the two. Let us first compare their component graphs. A component graph is one in which each component is a node and there is an edge between components if they know of each other. In a compositional system, the graph is hierarchical. Each component is connected only to its siblings, its children, and its parents. As a result, the system can be decomposed in many ways into independent parts, such that the interface between the parts is small.

In a noncompositional system, the component graph does not have this structure. The graph tends to be bushy and nonlocal. Bushy means that each component is connected to many others. Nonlocal means that each component is connected to widely different parts of the graph. Decomposing the system into parts is more arbitrary. The interfaces between the parts tend to be larger. This makes it harder to understand the components without taking into account their relation with the rest of the system. One example of a noncompositional graph is a Small World graph, which has the property that the graph's diameter is small (each component is within a few hops of any other component).

Let us see why hierarchical composition is suitable for system design by humans. The main constraint for humans is the limited size of human short-term memory. A human being can only keep a small number of concepts in his or her mind simultaneously. A large design must therefore be chopped up into parts that is each small enough to be kept in a single individual's mind. Without external aid, this leads humans to build compositional systems. On the other hand, design by nature has no such limitation. It works through the principle of natural selection. New systems are built by combining and modifying existing systems. Each system is judged as a whole by how well it performs in the natural environment. The most successful systems are those with the most offspring. Therefore natural systems tend to be noncompositional.

It seems that each approach, in its pure form, is a kind of extreme. Human design is goal-oriented and reductionistic. Natural design is exploratory and

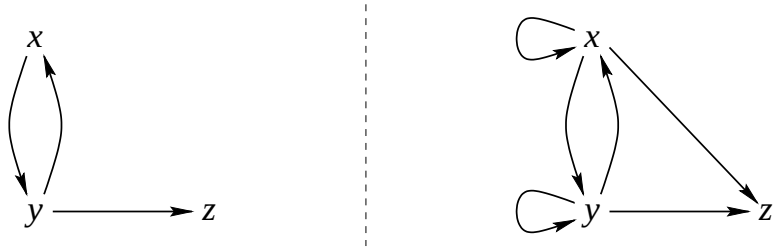

Figure 6.8: A directed graph and its transitive closure

holistic. Is it a meaningful question to try to get the best of both worlds? We can imagine building tools to let human beings use a more "natural" approach to system design. In this book, we do not consider this direction further. We focus on the compositional approach.

6.7.5 Further reading

There are many books on program design and software engineering. We suggest [150, 154] as general texts and [185] for a view on how to balance design and refactoring. *The Mythical Man-Month* by Frederick Brooks dates from 1975 but is still good reading [25, 26]. *Software Fundamentals* is a collection of papers by Dave Parnas that spans his career and is also good reading [144]. *The Cathedral and the Bazaar* by Eric Raymond is an interesting account of how to develop open source software [156].

Component Software: Beyond Object-Oriented Programming

For more information specifically about components, we recommend the book *Component Software: Beyond Object-Oriented Programming* by Clemens Szyperski [187]. This book gives an overview of the state-of-the-art in component technology as of 1997. The book combines a discussion of the fundamentals of components together with an overview of what exists commercially. The fundamentals include the definition of "component", the concepts of interface and polymorphism, the difference between inheritance, delegation, and forwarding, the trade-offs in using composition versus inheritance, and how to connect components together. The three main commercial platforms discussed are the OMG with its CORBA standard, Microsoft with COM and its derivatives DCOM, OLE, and ActiveX, and Sun with Java and JavaBeans. The book gives a reasonably well-balanced technical overview of these platforms.

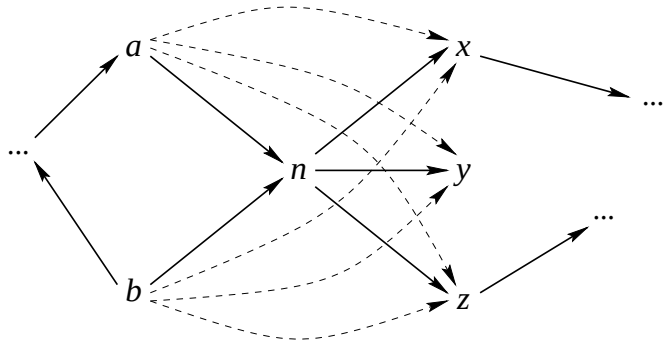

Figure 6.9: One step in the transitive closure algorithm

6.8 Case studies

6.8.1 Transitive closure

Calculating the transitive closure is an example of a graph problem that can be solved reasonably well both with and without state. We define a *directed graph* $G = (V, E)$ as a set of nodes (or vertices) V and a set of edges E represented as pairs (x, y) with $x, y \in V$, such that $(x, y) \in E$ iff there is an edge from x to y. Then we can state the problem as follows:

> Consider any directed graph. Calculate a new directed graph, called the *transitive closure*, that has an edge between two nodes whenever the original graph has a path (a sequence of one or more edges) between those same two nodes.

Figure 6.8 shows an example graph and its transitive closure. We start with an abstract description of an algorithm that is independent of any particular computation model. Depending on how we represent the graph, this description will lead naturally to a declarative and a stateful implementation of the algorithm.

The algorithm successively adds edges according to the following strategy: for each node in the graph, add edges to connect all the node's predecessors to all its successors. Let us see how this works on an example. Figure 6.9 shows part of a directed graph. Here the node n has predecessors a and b and successors x, y, and z. When the algorithm encounters node n, it adds the six edges $a \to x$, $a \to y$, $a \to z$, $b \to x$, $b \to y$, and $b \to z$. After the algorithm has treated all nodes in this fashion, it is finished. We can state this algorithm as follows:

```
For each node x in the graph G:
   for each node y in pred(x, G):
      for each node z in succ(x, G):
         add the edge (y, z) to G.
```

We define the function $\text{pred}(x, G)$ as the set of predecessor nodes of x, i.e., the nodes with edges finishing in x, and the function $\text{succ}(x, G)$ as the set of successor nodes of x, i.e., the nodes with edges starting in x.