Chapter 9

A Language with Flow

Here, explicit and implicit models of flow control are contrasted.

The former uses primitive selection and iteration constructs, in the manner of C++, whereas the latter relies on dynamic binding alone, in the manner of Smalltalk. Primitive if() and while() constructs are introduced, for which models and types may be found in our basic calculus. It is then shown how selection can be replaced by dynamic binding and iteration by mapping operations, at the cost of an increase in type complexity. Selection plays a role in storage and type recovery. Due consideration is given to implementation support issues.

9.1 Selection and Iteration

No language is complete without a set of conditional and branching statements. Control flow is handled in widely different ways in current object-oriented languages. Whereas *C++* provides three branching constructs: *if*, *switch* and *?:* and three looping constructs: *for*, *while* and *do* [ES90, Stro91], partly to maintain back-compatibility with *C* and partly because its whole philosophy is to offer the programmer a wide range of stylistic options, *Eiffel* strives after minimalism with binary *if*, multibranch *inspect* and a single general *loop* [Meye92]. Meyer argues strongly against multibranch selection in [Meye88, p24, 35], recognising that this is often used in situations where dynamic binding is more appropriate. A *case*-like statement fixes the number of branches and prevents code extension, whereas dynamic binding allows unforeseen types of behaviour to be added.

9.1.1 Primitive Selection

In chapter 7, an *n*-place compound expression was translated into a binder applied to *n* expressions. It is possible to model an *n*-place selection using a similar technique. Here, the lazy evaluation of λ-calculus expressions is assumed. Using an *(n+1)*-place expression binder, it is possible to delay the evaluation of *n* sub-expressions, subject to their selection from the closure using a projection function. An example with boolean logic and binary selection will show this:

$$\textbf{if}\ (x > y)\ \textbf{then}\ x\ \textbf{else}\ y \Leftrightarrow (\lambda a.\lambda b.\lambda f.(f\ a\ b)\ \ x\ y\ (x > y))$$
$$\Rightarrow (\lambda f.(f\ x\ y)\ (x > y))$$

Boolean switching involves 2-place selection, so the sub-expressions *x* and *y* of the selection are bound with a 3-place binder. The last abstraction λf delays the release and evaluation of any sub-expression. This is exactly the same as the tuple-building strategy we used in chapter 3. There, values were released from tuples by applying the tuple to a projection function. The release of *x* or *y* depends on the outcome of the test (x > y). Assuming this test returns a boolean value, *true* and *false* should then be encoded as the first and second 2-place projections:

$$\text{true} = \lambda a.\lambda b.a \qquad \text{false} = \lambda a.\lambda b.b$$

This approach can be generalised to handle ternary and quaternary logics. For example, a 3-place selection is given by:

$$(\lambda a.\lambda b.\lambda c.\lambda f.(f\ a\ b\ c)\ \ p\ q\ r) \Rightarrow \lambda f.(f\ p\ q\ r)$$

where *p, q* and *r* are sub-expressions. To release the second sub-expression, this closure must be applied to the second 3-place projection:

$$(\lambda f.(f\ p\ q\ r)\ \ \lambda a.\lambda b.\lambda c.b) \Rightarrow (\lambda a.\lambda b.\lambda c.b\ \ p\ q\ r) \Rightarrow q$$

In this model, the *select* function is implicit, since values are released according to the structure of the projections. The *n* different *n*-place projections form the elements of an *n*-valued logic.

9.1.2 Binary Valued Logic

Our primitive branching construct is based on this idea. Branching of a general nature is handled using *if()*, where the tested expression in parentheses is followed by a tagged block. A *tagged block* is a compound expression, made up of sub-expressions, each of which is tagged with a symbolic value from a logic. The tag identifier is separated from the rest of the sub-expression by a colon:

```
if (x > y)
{   true  :  out.put( "x is greater", \endline);
    false  :  out.put( "x is lesser or equal", \endline);
}
```

The tag represents a guard, or entry condition to the sub-expression. Only one of the sub-expressions (at most) will be executed. If none of the guards is satisfied, *if()* terminates and control passes to the next expression. For convenience's sake, we do not require all branches to be present, nor to appear in any particular order. Single-branch tests are therefore possible using a single guard. Where branches are absent, an implicit translation into the form of an ordered *n*-place selection is assumed, in which the missing branches are restored as tagged sub-expressions returning the trivial value.

```
if (x = 0)
{  true  :  if (y = 0)
   {          true  :  out.put( "both x and y are zero", \endline);
              false  :  out.put( "x only is zero", \endline);
   }
   false  :  if (y = 0)
   {          true  :  out.put( "y only is zero", \endline);
              false  :  out.put( "neither x nor y are zero", \endline);
   }
}
```

Figure 9.1 Binary Decision Tree

A tagged block differs from a normal compound expression in that it represents a selection rather than a sequence. Whereas all expressions in a sequence are evaluated, only one is evaluated in a selection. The idea of tagged blocks is appealing, since this avoids the accretion of reserved words in the style: **if** ... **then** ... **else**; and yet marks out each sub-expression according to the dispatching value. Decision trees like that in figure 9.1 produce clearly nested structures in which the combinations of *true* and *false* outcomes are more clearly flagged for each terminal branch than in some languages. This layout is only inconvenient in the case of repeated **else** ... **if** style conditions. Nested decision trees can sometimes be avoided by dispatching on a different logic having more than the binary *true* and *false* values of boolean logic.

9.1.3 Multivalued Logics

The use of *if()* may be extended to ternary and quaternary logics, since it is not restricted to binary selection. Richer multivalued logics are theoretically interesting in their own right, but also hide behind some common programming idioms. Consider the three-valued *strcmp()* from *C* which returns +1, 0 or -1 depending on the total lexical ordering of two strings. This is striving after a ternary logic. Figure 9.2 illustrates an intuitionistic logic having a third value *fail* meaning that truth or falsity is not proven. This is more flexible than the negation-as-failure strategy adopted in most languages.

```
if (accused.guilty)
{ true  :     {         accused.jail(sentence);
                        accused.pay(legalFees);
   }
  false  :    {         accused.free;
                        plaintiff.pay(legalFees);
   }
  fail  :     {         accused.free;
                        accused.pay(legalFees / 2);
                        plaintiff.pay(legalFees / 2);
   }
}
```

Figure 9.2: Ternary Decision Tree

In figure 9.3, it is probably more useful to think of the general comparison operator <?> as mapping into a four-valued space since this encompasses all the relationships between elements of partial and total orders. Dispatching on multi-valued logics often eliminates the need for nested decision-trees and this makes the flow of control easier to follow.

```
if (setX <?> setY)
{  lesser  :  out.put( "setX is included in setY", \endline);
   greater  :  out.put( "setX includes setY", \endline);
   equal  :  out.put( "setX and setY are isomorphic", \endline);
   unlike  :  out.put( "setX and setY are incommensurable", \endline);
}
```

Figure 9.3: Quaternary Decision Tree

9.1.4 General Selection

It should be clear from these examples that *if()* is exactly like a multi-branch selection function, such as *case()* in Pascal. To give an extra degree of flexibility in those rare cases where arbitrary multi-branching is desired, multiply-tagged sub-expressions are provided and also a default tag *else* for marking a sub-expression to execute if no other guard is satisfied:

```
isVowel : BOOLEAN                    ...in class CHARACTER
{  if (self)
   {          'a', 'e', 'i', 'o', 'u'  :  true
              else  :  false
   }
}
```

Figure 9.4: Selection with Default

In terms of the primitive model, multiply-tagged sub-expressions with *p* tags are considered shorthand for an expansion into *p* branches containing identical sub-expressions. The default expression with the *else* tag is likewise considered a shorthand for a multiply-tagged expression having all the remaining n - Σp tags not present in any other tagged sub-expression, where n is the arity of the logic.

Techniques exist for optimising binary branching and multi-branch selection where all branches are ordered according to their dispatching value. Any type which has a finite, enumerable set of values may be used as the dispatching logic of *if()*. This allows boolean, ternary, quaternary, character and other subrange types to be used. It is anticipated that enumerations will often be used for the logics given to *if()*; compilers will therefore be able to exploit the ordering of such values to re-arrange the branches into a logical dispatching order.

9.1.5 Selection and Type

Since all expressions have a value in our language, the type of a selection must also be considered. The most important principle to observe is that all branches of an *if()* must return the same type or class. In most of the examples given above, branches of *if()* returned no value, which is interpreted in our semantics as the element of the trivial type UNIT. However, the example of the *isVowel()* method illustrates a quite common case where different values from the same type are returned in different branches. An important consequence of our rules for interpreting selections with missing branches is that they are only correctly typed if they return no value. This is because automatic completion would restore other branches having the UNIT type. The *else* guard is therefore often useful in covering the remaining cases of a selection which *must* return a typed value.

9.1.6 Primitive Iteration

Primitive iteration is modelled in the λ-calculus by recursive functions with a selection that tests for continuation:

```
while (x > 0)          ⇔     (Y  λf.λx.if (x > 0)
{   g(x);                              then (λy.λz.z g(x) f(x-1))
    x := x-1; }                        else unit)
```

Here, a *while()* loop tests the value of x and performs some computation *g(x)* as long as $x > 0$. We transform the loop into a recursive function *f(x)* using the technique of abstracting over the point of recursion λf and fixing with Y.

The function takes as its argument the iteration value *x* and the body of *f()* is a selection testing $x > 0$, which is assumed to be modelled in the style presented above. The selection has two cases: the **then** branch is the recursive continuation of *f()*; the **else** branch terminates with the trivial value. The continuation is a sequence, modelled using a 2-place binder, the first sub-expression of which is *g(x)* and the second calls *f()* recursively with the new value for *x*. Since lazy evaluation is assumed elsewhere for selection, we must insist on eager evaluation for sequences, otherwise *g(x)* is not forced to be evaluated. In a fully lazy λ-calculus, the body of the *while()* would have to be constructed as a single expression *g(x)* returning the next value for *x*.

A simple binary selection is used to return control to the entry-point of the body of the iteration, or to signal termination. There is no theoretical reason why multi-branch selection should not be used. This gives rise to the curious possibility of loops that have more than one continuation condition.

9.1.7 General Iteration

General iteration is handled in our language using *while()*. The tested expression appears in parentheses, followed by a tagged block. Continuation of the loop can be made contingent on any single condition, expressed by a logical tag:

```
while (in.atEnd)
{  false  :  { in.get(word, \separator);
               out.put( "Word read from input was:", \space);
               out.put(word, \endline);
               }
}
```

Figure 9.5: Single Continuation Condition

Figure 9.5 illustrates how testing for *false* is sometimes clearer than inverting the truth condition of the test. Alternatively, continuation can be made contingent on a number of conditions, illustrated in figure 9.6:

```
while (table.at(index) <?> word)
{  lesser  :  index := index + 1;
   equal  :  { out.put( "Entry found at:", \space);
               out.put(index.asString, \endline);
               index := index + 1;
               }
}
```

Figure 9.6: Multiple Continuation Conditions

Since any logic may be used, not just binary boolean logic, it is possible to specify more than one guard in the tagged block. In this case, testing for continuation is combined with branching within the body of the loop. The above example will continue to iterate so long as the entry found in the *table* is lexically *lesser* or *equal* to the tested *word* string; and will terminate if the comparison <?> returns *greater* or *unlike*. The *index* is incremented in each of the tagged sub-expressions, to ensure that a different *table* position is searched on each iteration, no matter which sub-expression was executed last. Care should be taken to ensure that loops terminate. If all possible outcomes of a selection are covered in the guards, then a loop will not terminate. A *while()* that contains an *else* guard will loop forever. Complementary, or symmetrical updates to the iteration variables will sometimes result in non-termination:

```
while (table.at(index) <?> word)
{  lesser  :  index := index + 1;
   greater  :  index := index -1;
}
out.put( "Entry found at:", \space);
out.put(index.asString, \endline);
```

Figure 9.7: Semi-terminating Iteration

Here, if no entry for *word* is found in the *table*, the *index* will oscillate forever about the position where the word was expected. This is simply bad algorithmic design, equivalent to the *while not (state = found)* style of programming in Pascal, and not a fault of our approach. To combat unintended infinite loops, a compiler may issue warnings in two cases: where all the values of the tested logic are found in the guards such that no exit-case is possible; and where no change is made to any of the objects participating in the test, for some guarded sub-expression.

9.1.8 Iteration and Type

The type of a *while()* expression is always UNIT; in other words, *while()* returns no value. This is because the semantic interpretation of a missing guard is a tagged expression whose value is *unit*. Since loops terminate when no guard is satisfied, the trivial value will always be returned. Other guarded expressions catching the continuation cases are interpreted as sequences containing a recursive call to *while()*, which eventually will yield the trivial case upon termination. To inculcate a sense of syntactic uniformity, good programming style will insist that the branches of a *while()* are written with the semicolon ";" terminating the final expression in each sequence.

9.2 Dynamic Dispatch and Mapping

Smalltalk provides no built-in primitives for control, replacing static branching instructions by dynamic dispatch [GR83]. Its *Boolean* subclasses *True* and *False* define opposite method responses for the single-branch *ifTrue:* and *ifFalse:* messages and the for the binary branching *ifTrue:ifFalse:* message. This behaviour is generalised through recursion with the *whileTrue:* and *whileFalse:* messages supporting loops. The approach is based on a higher-order treatment of code blocks as first-class objects, which are passed as arguments to the flow switching methods. A block wraps up a compiled sequence of instructions whose evaluation is delayed until it receives a *value* message. Classes *True* and *False* simply evaluate alternate blocks in their flow switching methods. Aficionados of this style admire it for its apparent object-oriented purity, since it needs no other primitive selection constructs to support it. However, this claim to object-oriented purity can be challenged on the grounds that blocks hide other non-object-oriented mechanisms.

9.2.1 Selection and the Object Model

By seeking to take selection into the object model, *Smalltalk* inadvertently creates another concept which does not fit the object model. For though both blocks and objects can be explained as closures, there are no constraints on the form of a block, whereas objects follow the pattern of a generator. Blocks are instances of the class *Context* [Digi92]; however, they may be bound over any number of free variables and contain any number of sub-expressions. The class *Context* could effectively model any other class, making blocks the most powerful and least disciplined concept in the language. The first example in figure 9.8 illustrates the general pattern for selection in Smalltalk. The blocks are the expressions in brackets [ ], which we can treat formally as closures whose evaluation is delayed by abstraction over a *unit* value:

```
| a b max |
a := 4 factorial.               "selection blocks closed over a, b, max"
b := 5 squared.
a > b ifTrue: [max := a] ifFalse: [max := b].


                                "iteration block with leading block argument"


'Here we go again' select: [:char | char isVowel].
```

Figure 9.8: Selection and Mapping in Smalltalk

Furthermore, iterating methods require blocks to have one or two *block arguments*, iteration variables which bind to elements of collections when the block is evaluated. The second example in figure 9.8 illustrates a block whose argument *char* is mapped over character elements of the leading character string. Such a block is more like a free-standing function with a bound argument. Free-standing functions conflict with the notion of methods. Further problems arise in *Smalltalk* through blocks having dynamic extent (they are allocated on heap memory) and dynamic scope, due to their sharing of non-local variables [Wolc88]: consider the consequences of substituting object references for the local variables *a*, *b* and *max* above and then returning a block for later evaluation out of its defining context. Our language insists on static scope and extent, for the sake of safety and memory management.

Smalltalk iterates by mapping closures over elements of collections. We have suggested techniques for restricting mapped functions to existing methods in an earlier project [Blac92]. Here, objects may only access the surrounding context through an activation record passed to the method, or through call-back to the object driving the iteration. This works well, but is strictly less expressive than allowing arbitrary closures. A further drawback is the need to return to library classes to add new mapped methods destined for use in a particular application; this breaks the principle of closed modules. In any case, the particular mapping or filtering operation required by the application does not depend in general on the class of element, but on some higher design. Recognising this, many advocate the design of special-purpose *iterator* classes [GHJV95] which control access to collections.

9.2.2 Integrating with Primitive Selection

Figure 9.9 illustrates a simple #BOOLEAN[ ] class, shadowing the enumerated **values** {false, true} and designed using our primitive *if()*. It follows the normal pattern of a recursive type, which is contrasted later with figure 9.10.

```
class #BOOLEAN [B]  values {false, true}
(self : @B) is (super : #OBJECT [B]) with
{  private
    value : B (false);
   public
    not : B
    {           if(self) { true : false  false : true }
    }
    and (other : B) : B
    {           if(self) { true : other  false : false }
    }
    or (other : B) : B
    {           if(self) { true : true  false : other }
    }
    implies (other : B) : B
    {           if(self) { true : other  false : true }
    }
}
```

Figure 9.9: A Boolean Class using Primitives

The **values** keyword notifies the compiler that the special symbols *false* and *true* are to be treated as self-identifying instances of the #BOOLEAN[ ] class. A compiler may choose any appropriate physical representation for enumerated values, such as a short bit-pattern, or an integer. Formally, an enumeration is considered to list the set of constructors for a class; in this view *false* and *true* are functions creating instances of the class. The keyword **private** introduces the single state variable storing the whole state of the class; by default this is initialised to *false*. The keyword **public** introduces the exported methods of the #BOOLEAN[ ] class, which inherits further basic methods from #OBJECT[ ], since we wish to be able to assign, alias and copy BOOLEAN objects. For simplicity's sake, our language maps boolean methods onto the primitive control flow function *if()* described above; this is as much for the sake of efficiency, since the current compilation model translates expressions in our language into a portable pseudo-assembler having primitive branching and iteration (see chapter 9). Our compiler inlines all occurrences of these boolean methods, since they are never redefined; in turn the primitive *if()* is compiled to basic jump instructions in machine code.

This approach is to be commended for its efficiency. Nonetheless, since our language's type system can handle methods passed as arguments, it would be possible to implement the Smalltalk method mapping approach in a statically scoped and bound way. This is explored below.

9.2.3 Removing Primitive Selection

In order to support selection by dynamic binding, the #BOOLEAN[ ] class must be specialised into disjoint child classes #FALSE[ ] and #TRUE[ ]. The aim of this is to provide alternative versions of selection methods, such as *ifTrueFalse()*, in the child classes, which dispatch on the dynamic type of objects found in variables with the type $\forall(\beta \subseteq \Phi\text{BOOLEAN }[\beta])$. This means that the parent #BOOLEAN[ ] class must provide deferred signatures for selection methods. The typing of their signatures is complicated further by the need to preserve nondeterministic polymorphism:

```
class #BOOLEAN [B] uses #BOOLEAN [O], #BOOLEAN [R]
(self : @B) is (super : #OBJECT [B]) with
{  public
   not : R {}
   and (other : O) : R {}
   or (other : O) : R {}
   implies (other : O) : R {}
}
```

Figure 9.10: A Boolean Class for Dynamic Dispatch

Figure 9.10 illustrates the changes from the class presented in figure 9.9. The class deliberately omits introducing **values** since the type system must now identify *false* and *true* with disjoint types. Here, different unresolved polymorphic types O and R are given to the argument and result of each method. Neither of these types may be linked to the type B of *self*, since in general the objects instantiating *self*, *other* and *result* have unconnected types.

The strangeness of this indicates that perhaps *Smalltalk*'s treatment of *False* and *True* as subclasses of a *Boolean* class is somewhat forced. The class design does not follow the normal pattern of recursion in the *self*-type, in the way that an integer subrange type is structurally homomorphic with the base integer type. It would be more straightforward to assert that *False* and *True* were plain subtypes of *Boolean*, since they partition a value set {*false, true*}. In this case *Smalltalk* would have to admit dispatching on individual *values* as well as on *types*, since the inherited methods would all be in the monomorphic type *Boolean*. This gives another reason why a primitive *if()* construction should be preferred for testing the values of a type, seeing this as distinct from the kind of dynamic dispatch obtained through polymorphism.

Nonetheless, the polymorphic model will be developed here to its conclusion. Figure 9.11 illustrates the alternative implementations of methods in the #FALSE[ ] and #TRUE[ ] classes. Each class introduces a single **values** element, which is identified with its most specific class. Again, this style relies on unresolved polymorphic type variables to preserve a nondeterminism in the methods' argument and result types.

```
class #FALSE [F]  values {false} uses #BOOLEAN [O], #BOOLEAN [R]
(self : @F) is (super : #BOOLEAN [F]) with
{  private
    value : F (false);
   public
    not : R { true }
    and (other : O) : R { self }
    or (other : O) : R { other }
    implies (other : O) : R { true }
}


class #TRUE [T]  values {true} uses #BOOLEAN [O], #BOOLEAN [R]
(self : T) is (super : #BOOLEAN [T]) with
{  private
    value : T (true);
   public
    not : R { false }
    and (other : O) : R { other }
    or (other : O) : R { self }
    implies (other : O) : R { other }
}
```

Figure 9.11: True and False Classes

At first, the reader may think it legitimate to narrow the types of redefined methods down to a single class, or type. For example, the #FALSE[ ] method for *not()* returns a value whose type is apparently $\forall(t \subseteq \Phi$TRUE [t]), or even the simple type TRUE. This would allow the static propagation of type information leading to static binding and the possible compile-time evaluation of certain boolean expressions. This may not easily be done, however, since the effect would be to make the formal translations of the redefined signatures type incompatible with the methods they replace. Methods with unresolved return-values are formally protected by an extra type abstraction. This is incompatible with a method whose return-value has a simple type, since the argument list of one method is longer than the other.

Example translations of the #BOOLEAN[ ] method signatures for *and()* and *or()* show how the **uses** declaration is really a shorthand for the introduction of multiple type parameters inside the scope of the *self* recursion variable:

```
uses #BOOLEAN [O], #BOOLEAN [R]
{... and (other : O) : R {}
     or (other : O) : R {}
...}                              ⇔

{... and ↦ ∀(o ⊆ ΦBOOLEAN [o]).∀(r ⊆ ΦBOOLEAN [r]).
                 λ(other : o). ⊥:r,
     or ↦ ∀(o ⊆ ΦBOOLEAN [o]).∀(r ⊆ ΦBOOLEAN [r]).
                 λ(other : o). ⊥:r,
...}
```

in which it is clearer how unresolved polymorphic type variables have a scope restricted to individual methods. The **uses** keyword introduces as many variables as is necessary to preserve the distinct types required within a single method. In the formal translation, these become type arguments to the methods concerned.

The signatures of the redefined *and()* and *or()* methods in the #FALSE[ ] and #TRUE[ ] classes must respect the two inserted type arguments to remain type compatible with those of the parent class #BOOLEAN[ ]. The translation of the class #TRUE's *and()* and *or()* methods is given by:

```
uses #BOOLEAN [O], #BOOLEAN [R]
{... and (other : O) : R { other }
     or (other : O) : R { self }
...}                              ⇔

{... and ↦ ∀(o ⊆ ΦBOOLEAN [o]).∀(r ⊆ ΦBOOLEAN [r]).
                 λ(other : o). other:r,
     or ↦ ∀(o ⊆ ΦBOOLEAN [o]).∀(r ⊆ ΦBOOLEAN [r]).
                 λ(other : o). self:r,
...}
```

in which it is clearer how the method signatures are compatible. As it stands, this appears to lose type information, since in *and()*, *other* and the *result* should have the same type; likewise in *or()*, the *result* should have the type of *self*. For the sake of dynamic binding, the same argument pattern as #BOOLEAN[ ]'s methods must be retained in both child classes. In cases of static binding, the system may optimise by distributing known types to type-arguments before run-time. This achieves the benefit that was desired above: some boolean expressions may be precomputed. In the remaining cases, the system must distribute the unknown type $\perp$ and then check the type of the result at run-time.

9.2.4 Selection Methods

Further methods may now be supplied for the #BOOLEAN[ ] class to perform selection by dynamic binding. The following examples seek to preserve a pure object-oriented style and at the same time disallow the passing of arbitrary closures as arguments. In consequence, any selection method must accept an object and two further arguments which are methods owned by that object. The idea is for one method to be invoked in the *true*-case and the other in the *false*-case. These will be called the *then* and *else* methods. On the assumption that these methods both return a result, the *ifTrueFalse()* selection method may be written with the type signature shown in figure 9.12:

```
class #BOOLEAN [B] uses #OBJECT [O], #OBJECT [R], ...
{...
   ifTrueFalse (object : O; then : @(O : R); else : @(O : R)) : R {}
....}

class #FALSE [F] uses #OBJECT [O], #OBJECT [R], ...
{...
   ifTrueFalse (object : O; then : @(O : R); else : @(O : R)) : R
   { object.else }
....}

class #TRUE [T] uses #OBJECT [O], #OBJECT [R], ...
{...
   ifTrueFalse (object : O; then : @(O : R); else : @(O : R)) : R
   { object.then }
....}
```

Figure 9.12: A Selection Method

The method *ifTrueFalse()* accepts any kind of *object*, so long as the following *then* and *else* arguments are methods having the signature O → R, and it returns a result in the same type R. The types of arguments that are methods are represented using the conventions:

@(T, P, Q : R) - method of class T accepting P and Q, returning R

in which the first argument must be the class owning the method. The method *ifTrueFalse()* is applied in the following way:

... (3 < 4).ifTrueFalse(3, isLesser, isGreater);

This has the advantage of a strong type constraint on the methods *isLesser()* and *isGreater()*, which must both have the type: INTEGER → UNIT. However, it has the disadvantage of requiring anecdotal methods for printing messages in the class #INTEGER[ ], or one of its ancestors (figure 9.13).

```
isLesser : J                                    ...method of #INTEGER[J]
{  out.put(self.asString, \space);
   out.put( "is smaller", \endline);
   self
}
```

Figure 9.13: Trivial Dispatched Method

An alternative approach would be to provide a hierarchy of #CONTEXT[ ] classes, whose instances were passed as arguments to *ifTrueFalse()* and which supplied varieties of alternate method pairs to be used in response to boolean selection. An instance of some #CONTEXT[ ] subclass would behave like an activation record, encapsulating a number of values. This would be similar to binding values in a closure, but with the advantage that the record of values followed some explicit class pattern.

The necessary constraints on the type returned by primitive *if()* were described above. Identical conditions must apply to any type returned by the *ifTrueFalse()* method and similar selection methods. Either all branches must return the same static type, or the result of *ifTrueFalse()* must be typed in the least upper F-bound of all types returned in each branch.

9.2.5 Mapping Methods

Mapping and filtering methods may be provided for the collection classes in a similar vein. Mapping transforms iteration into recursion, using methods with names like *collect()* and *select()* in the *Smalltalk* idiom. Figure 9.14 illustrates the pattern for a homogeneous #LIST[ ] class with a mapping method *collect()*.

```
class #LIST [L[O]] uses #OBJECT [O], #OBJECT [R]
(self : @L[O]) is (super : #SEQUENCE [L[O]]) with
{  private
    head : O;
    tail : @L[O];
   public
    head : O { head }
    tail : @L[O] { tail }
    add (elem : O) : @L[O]
    {         cell : @L[O];
              cell.create(elem, self)
    }
    collect : (fun : @(O : R)) : @L[R]
    {         ...                          ...pattern of a mapping method
    }
...}
```

Figure 9.14: A Homogeneous List with Mapping

The higher-order semantic translation is used, in which L stands for a type function. The list is homogeneous, since all type applications in *self* are of the form L[O]. However, mapping over a list typically generates a list like *self*, but having a different parameterisation. The unpredetermined nature of the resulting element type is indicated by introducing an unresolved polymorphic type R which is bound by type application when the mapping method is invoked.

Again, to preserve object-oriented purity, the functions mapped over collections must be methods owned by the particular type of element against which they are invoked. A mapping method *collect()* for homogeneous lists typed L[O] should accept a transformer method typed O $\rightarrow$ R owned by elements of type O and return a list of type L[R] collecting the results typed R of applying the transformer to each element of *self*. The type of *collect()* is therefore:

$$\begin{aligned}\text{collect} : &\ \forall(s\ \mathbf{<:}\ \Phi\text{LIST}\ [s]).\forall(t \subseteq \Phi\text{OBJECT}\ [t]).\\ &\ \forall(r \subseteq \Phi\text{OBJECT}\ [r]).s[t] \rightarrow ((t \rightarrow r) \rightarrow s[r])\end{aligned}$$

The body of the method *collect()* has two cases: it should halt if the list is empty and continue mapping otherwise. The only way to do this in a purist dynamic dispatching manner is to have auxiliary functions that are passed to a method like *ifTrueFalse()*. The *collect()* method breaks down into two auxiliary methods to handle the trivial and recursive cases of mapping, illustrated in figure 9.15:

```
collect : (fun : @(O : R)) : @L[R]
{  self.isEmpty.ifHalt(self, fun, collectEnd, collectMore)
}


collectEnd : (fun : @(O : R)) : @L[R]             ...trivial version
{  link : @L[R]                                   ...returns void alias
}


collectMore : (fun : @(O : R)) : @L[R]            ...recursive version
{  self.tail.collect(fun).add(self.head.fun)      ...adds one element
}
```

Figure 9.15: Trivial and Recursive Mapping Cases

The main *collect()* method tests for the emptiness of a list and dispatches one out of *collectEnd()* or *collectMore()* on the boolean result. Here, the internal dispatching method is given the name *ifHalt()*, since it decides whether to terminate the mapping, or continue. It is different from *ifTrueFalse()* in that it accepts the list object, the element transformer method and two alternative list mapping methods to dispatch. The type signature for *ifHalt()* is involved, since it accepts as arguments methods which themselves accept method arguments. Essentially, the alternative #FALSE[ ] and #TRUE[ ] versions of *ifHalt()* simply invoke *collectMore()* and *collectEnd()* respectively, as illustrated in figure 9.16.

Mapping methods should really be defined for a more general class of collections than the #LIST[ ] class shown here, since this would allow iteration over other types of object to be handled in the same way. In this case, the #BOOLEAN[ ] classes would depend on a polymorphic #COLLECTION[C[O]], whose type would be resolved every time mapping were invoked.

```
class #FALSE [F] uses #OBJECT [O], #OBJECT [R], #LIST [L[O]]...
{...          ifHalt (list : L[O];  fun : @(O : R);
                       then : @(L[O], @(O : R) : L[R]);
                       else : @(L[O], @(O : R) : L[R]))  :  L[R]
   { list.else(fun) }
....}

class #TRUE [T] uses #OBJECT [O], #OBJECT [R], #LIST [L[O]]...
{...
   ifHalt (list : L[O];  fun : @(O : R);
                       then : @(L[O], @(O : R) : L[R]);
                       else : @(L[O], @(O : R) : L[R]))  :  L[R]
   { list.then(fun) }
....}
```

Figure 9.16: Testing for Termination

The solution given above respects the role played by boolean classes in determining whether mapping should continue or halt. Yet another way of achieving the dynamic dispatching effect would be to partition a #LIST[ ] class into #EMPTY_LIST[ ] and #NONEMPTY_LIST[ ] variants, adopting an approach similar to dynamic reclassification (see chapter 3). The base #LIST[ ] class would defer the *collect()* method; the empty child would define the trivial terminating version and the nonempty child would define the recursive version. This bypasses the boolean classes at the expense of introducing dynamic typing for all #LIST[ ] variables, which must be considered a serious loss.

Although this approach succeeds in providing iteration through pure dynamic dispatching, the increased levels of type complexity, class dependency and numbers of auxiliary methods make it obvious why selection and iteration primitives are much to be preferred. The development of the dynamic dispatching and mapping model has been fruitful, since it has helped to reveal the true underlying complexity of the model used by *Smalltalk*.

9.3 Scope and Type Nondeterminism

Finally, some areas of program nondeterminism are addressed in which selection plays a role. Apart from the general introduction of polymorphism through parameterised structures, choice-points in algorithms often give rise to the creation of objects having different types, leading to dynamic binding later. In a slightly different vein, the freedom to allocate objects in different ways requires careful managment of storage. On various occasions, the system needs to recover the formal type and the kind of storage used for objects.

9.3.1 Scope Nondeterminism

Two special logics are used by the system model: one describes the created states of variables, the other describes the closed set of types at a dynamic dispatching site. For example, the system *destroy()* method ensures that an alias variable was allocated on the heap before attempting to deallocate it:

```
destroy
{  if (self.scope)
   { heap  :  self.deallocate; }              ...primitive system instruction
}
```

Figure 9.17: Testing Storage Allocation

The method *scope()* is part of the system model and shadows a primitive operation for inspecting flag bits set in pointers. The set of created states is the type ALLOCATION = {*static*, *void, stack, heap, store*}. All ordinary variables are *static*, whereas alias variables may be *void* references, or refer to data allocated on the *stack* or on the *heap*. The *store* value represents a forward-looking extension to handle objects allocated on remote or persistent storage devices.

The alias mechanism is designed to be as flexible as possible, yet secure. Alias variables with *heap* or *stack* allocation must be treated carefully. Clearly, the system cannot decide for itself when to deallocate *heap* data - it has an unpredetermined extent and may legally be passed back outside the scope in which it was created. Instead, the primitive *destroy()* instruction is made available as a method for the sake of those classes whose instances need to clean up internal dynamic data storage. As described in chapter 7, *create()* and *destroy()* are called automatically when static data comes into scope and goes out of scope, in order to handle the allocation and deallocation of internal dynamic data storage, in the manner of *C*++ constructors and destructors.

The reasons for marking *stack* aliases apart is so that the system can keep track of the scope of the aliased data. It is reasonable to expect a compile-time checker to rule out attempts to return aliases to locally-declared data storage, but perhaps unreasonable to expect it to trace all alias assignments and argument passing with a view to determining when the aliased data goes out of scope. This would require some kind of global flow analysis. Instead, an integer counter is attached to alias variables, whose value is initially zero, representing the number of nested levels of scope in which the alias is being used relative to the scope in which it was bound to the static data. Passing data by reference to a method increments the counter in the method's alias argument and decrements the counter in the method's alias return value. Alias assignment copies the scope counter. In this way, accessing aliased *stack* data is made contingent on having a non-negative scope counter. To avoid inefficiency, this check is made once when methods return alias values to their callers.

9.3.2 Type Nondeterminism

Subject to certain conditions, objects of different types may be created or selected within different branches of *if()* and returned to the caller. Such an expression is only correctly typed if there exists a *least upper bound* on all the types returned. This means a polymorphic F-bound which all the types returned in each branch must satisfy. Fortunately, the class denoting the least upper bound is typically indicated in the calling expression:

```
... uses #GRAPHIC [G] ...

newGraphic (sel : CHARACTER) : @G
{  c : @CIRCLE;
   s : @SQUARE;
   if (sel)
   {         'c'  :  c.create                 ...return a CIRCLE
             's'  :  s.create                 ...return a SQUARE
   ... }
}
```

Figure 9.18: Returning Multiple Types

Here, the type of *if()* is also the type of the function *newGraphic()*. It creates an instance of either a CIRCLE or a SQUARE, depending on the selection-value supplied. The function's designer indicates the return type to be G, a parameter which stands for the polymorphic class $\forall(\tau \subseteq \Phi\text{GRAPHIC}\ [\tau])$ by virtue of some introductory declaration: **uses** #GRAPHIC [G]. Provided each branch returns a type satisfying the bound, the compiler will accept this as the return type of *if()*. As it stands, the function is not yet type-correct, since there are missing branches for 254 other CHARACTER values than 'c' and 's', indicated by the ellipsis. To avoid rejection, a default selection case must be supplied. It is more usual to design this kind of function using *g* : @G, a polymorphic local alias variable parameterised by the same type as the result:

```
... uses #GRAPHIC [G] ...

newGraphic (sel : CHARACTER) : @G
{  g : @G;
   if (sel)
   {         'c'  :  g[CIRCLE].create             ...return a CIRCLE
             's'  :  g[SQUARE].create             ...return a SQUARE
             else : g                             ...void object
   }
}
```

Figure 9.19: Instantiating Returned Types

and to create objects of specific types in different branches by type-application, indicated by the brackets *g*[ ]. Each local alias variable in the old design would be implemented as one smart pointer[1] on a stack frame. Clearly, the new design reduces this to a single smart pointer, wasting no space. The default selection case here returns a void alias, to be tested by the caller.

9.3.3 Run-Time Type Recovery

The caller of *newGraphic()* cannot statically determine the exact type of object it receives. This is one case where polymorphism remains unresolved until run-time and gives rise appropriately to situations where dynamic binding is used to select methods. Programs sometimes need to recover the exact type of an object at run-time, usually in situations where the result of a polymorphic function is assigned to a variable with a monomorphic type.

```
... uses #GRAPHIC [G] ...

addGraphic (g : @G)                          ...polymorphic #GRAPHIC[ ]
{  if (g.type)
   {        CIRCLE  :  circles.add(g);       ...recovered as CIRCLE
            SQUARE :  squares.add(g);        ...recovered as SQUARE
   ... }
}
```

Figure 9.20: Run-Time Type Recovery

Consider a #DRAWING[ ] class, which allows the addition of polymorphic #GRAPHIC[ ] picture elements using the exported function *addGraphic()*, illustrated in figure 9.20. For efficiency's sake, it stores picture elements in homogeneously-typed collections of elements. To do this, it must recover the type of each picture element added. Our language provides run-time type identification through the system function *type()* which returns the type of a variable. This may be tested in selection expressions. In figure 9.20, the guards on sub-expressions are the names of types. The *addGraphic()* method uses this information to dispatch one or other branch appropriately.

It is generally difficult to think of situations where type recovery is preferable to dynamic binding. Meyer has rightly campaigned against *typecase()*-like statements in object-oriented languages [Meye88, p24], since this fixes the number of types on which dispatching is performed, whereas dynamic binding leaves this open. Nonetheless, languages without explicit run-time type identification offer type recovery by more devious means. Type recovery is performed in *C++* using the dangerous technique of *downcasting* in the inheritance hierarchy [Meys92, p135-142]; elsewhere programmers use shared strings as run-time type tags. In *Eiffel*, type recovery is handled through the *reverse assignment attempt*, in which the target of assignment has a more restricted *static* type than the source [TW95, p355-356]; however, if the

[1] This is a primitive pointer with the flag bits and scope counter described above.

dynamic types do not conform at run-time, then the result of a reverse assignment is a void reference. The reference must be tested before further operations can be carried out on the variable, making this a clumsy way to perform type recovery. In our language, the system model already has the facility to inspect types (this is needed for dynamic dispatch), so there is no reason not to offer this to the programmer, with appropriate health warnings. A technique for modelling values with type tags in the λ-calculus is given in Appendix 1.

This chapter has considered two alternative mechanisms for handling flow-control. One of the main determining factors in preferring the primitive selection model over the dynamic dispatching model was in order to simplify the typing and binding issues. Our type model reveals how using dynamic dispatch to handle flow control requires designing functions deliberately with large amounts of unresolved polymorphism. This eventually proves less habitable than having functions with simple or resolveable polymorphic types and admitting selection primitives. A technique for recovering type was also presented. Type propagation is an important concern in our language, since polymorphic expressions may be used in a context where static type information is available. The resolution of polymorphism is considered in more detail in the following chapter, as part of a general optimisation process.