Agents, iteration and introspection

27.1 OVERVIEW

Objects represent information equipped with operations. These are clearly defined concepts; no one would mistake an operation for an object.

For some applications — graphics, numerical computation, iteration, writing contracts, building development environments, “reflection” (a system’s ability to explore its own properties) — you may find the operations so interesting that you will want to define objects to represent them, and pass these objects around to software elements, which can use these objects to execute the operations whenever they want. Because this separates the place of an operation’s definition from the place of its execution, the definition can be incomplete, since you can provide any missing details at the time of any particular execution.

You can create agent objects to describe such partially or completely specified computations. Agents combine the power of higher-level functionals — operations acting on other operations — with the safety of Eiffel’s static typing system.

27.2 A QUICK PREVIEW

Why do we need agents? Here are a few examples. This preview skips many details but will give you an idea of the power of the mechanism; any apparent mystery will soon be cleared as you read further into the chapter.

Let’s start with a typical need of graphical user interface (GUI) programming. Using EiffelVision, the multi-platform graphical library of Eiffel Software, you may write

\[
\text{your\_button.\ click\_actions.\ extend (agent your\_routine)}
\]

to add your_routine — a routine of your application, executing appropriate operations — to the list of actions triggered by a mouse click on your_button. This is all you need to set up the application’s response to such an event.
The argument to `extend, agent your_routine`, is an **agent expression**. The keyword `agent` avoids confusion with an actual routine call: when calling `extend`, you don’t want to call `your_routine` yet! Instead you pass to `extend` an “agent”, which `extend` adds to the `click_actions` list for `your_button`, enabling EiffelVision to call `your_routine` for every subsequent occurrence of a click event on the button. The agent includes any context information that `your_routine` may need: cursor position, button number, pressure.

Now a numerical example. Over the interval [0, 1], you want to integrate a function \( g(x): \text{REAL} \to \text{REAL} \). With `your_integrator` of a suitable type `INTEGRATOR` (detailed later), just use the expression

\[
\text{your_integrator. integral (agent } \ g(?), 0.0, 1.0) \]

Again this doesn’t call the routine \( g \), but enables `integral` to call \( g \) when it pleases, as often as it pleases, on whatever values it pleases. We must tell `integral` where to substitute such values for \( x \) at the places where its algorithm needs to evaluate \( g \) to approximate the integral. This is the role of the question mark `?`, replacing the argument to \( g \).

You may use the same scheme in

\[
\text{your_integrator. integral (agent } \ h(u, ?, v), 0.0, 1.0) \]

to compute the integral \( \int_0^1 h(u, x, v) \, dx \), where \( h \) is a three-argument function \( h(a: T1; x: \text{REAL}; b: T2): \text{REAL} \) and \( u \) and \( v \) are arbitrary values. As before you will use a question mark at the “open” position, corresponding to the integration variable \( x \). Two “closed” positions show actual values \( u \) and \( v \).

Note the flexibility of the mechanism: it allows you to use the same routine, `integral`, to integrate a one-argument function such as \( f \) as well as functions such as \( h \) involving extra values.

You can rely on a similar structure to provide **iteration** mechanisms on data structures such as lists. Assume a class `CC` with an attribute

\[
\text{intlist: LINKED_LIST [INTEGER]} \]

and a function

\[
\text{integer_property (i: INTEGER): BOOLEAN} \]

returning true or false depending on a property involving \( i \). You may write

\[
\text{intlist, for_all (agent } \ \text{integer_property (?))} \]

to denote a boolean value, true if and only if every integer in the list `intlist` satisfies `integer_property`. This expression might be useful, for example, in a class invariant. It will work for any kind of `integer_property`, even if this function involves arbitrary features of the current object.

Now assume that in `CC` you also have a list of employees:

\[
\text{emplist: LINKED_LIST [EMPLOYEE]} \]

and that class `EMPLOYEE` has a function `is_married: BOOLEAN` with no argument, telling us about the current employee’s marital status. Then you may also write in `CC` the boolean expression
to find out whether all employees in the list are married. The argument to `for_all` is imitated from a normal feature call `some_employee.is_married`, but instead of specifying a particular employee we just give the type `{EMPLOYEE}`, to indicate where `for_all` must evaluate `is_married` for successive targets taken from the list.

The `{EMPLOYEE}` notation replaces the question mark of the previous examples. Those examples used an argument as the open operand — the place where the routine will be evaluated — as in `integer_property(?)`, where the argument type is clear from the declaration of `integer_property`. But with `is_married` the open operand is the target, so we need to specify the type: many classes may have a function called `is_married`.

Note again the flexibility of the iteration mechanism and its adaptation to the object-oriented form of computation: you can use the same iteration routine, here `for_all` from `LINKED_LIST`, to iterate actions applying to either:

- The **target** of a feature, as with `is_married`, a feature of class `EMPLOYEE`, to be applied to its `EMPLOYEE` target.

- The **actual argument** of a feature, as with `integer_property` which evaluates a property of its argument `i` — and may or may not, in addition, involve properties of its target, an object of type `CC`.

It seems mysterious that a single iterator mechanism can handle both cases equally well. We will see how to write `for_all` and other iterators accordingly. The trick is that they simply work on their open operands; when calling them, you choose what to leave open: either the argument as with `integer_property` and `integral`, or the target as with `is_married`.

Now assume that you want to pass to some object the mechanisms needed to execute the cursor resetting and advance operations, `start` and `forth`, on a particular list. Here nothing is left open: you fix the list, and the operations have no arguments. You may write

```
emplist.for_all( agent {EMPLOYEE}.is_married )
```

All operands — target and arguments — of the agents passed to `object` are “closed”; so `object` can execute call operations on such objects without providing any further information.

At the other extreme, you might leave an agent expression fully open, as in

```
object.operation( agent your_list.start, agent your_list.forth )
```

so that `object`, when it desires to apply a call operation, will have to provide both a linked list and an actual argument to execute `extend`. When as here all the arguments are open, you may omit the argument list, writing just `agent {LINKED_LIST [T]}.extend(?)`. Such an agent is a “**routine object**”: an object representing the routine `extend` from `LINKED_LIST`, such as could be used by browsing tools or other reflection facilities.
To use an agent, a routine such as `operation` can apply to it the procedure `call`, passing a tuple of values for the open operands. This will have the same effect as an execution of the original feature — `f, h, integer_property, is_married, start, forth, extend` — on all the operands, closed and open.

The notation provides an extra degree of flexibility by letting you define inline agents, which instead of referring to a feature of the class define a routine text as part of the agent declaration. Inline agents have the same form as a Routine body, as in

\[
\begin{align*}
(\text{agent} (i: \text{INTEGER}): \text{BOOLEAN} \text{ do Result} & := \text{integer_property} (i))
\end{align*}
\]

-- Means the same as: \(\text{agent} \text{ integer_property} (?)\)

\[
\begin{align*}
(\text{agent} (e: \text{EMPLOYEE}): \text{BOOLEAN} \text{ do Result} & := e.\text{is_married})
\end{align*}
\]

-- Means the same as: \(\text{agent} \{\text{EMPLOYEE}\}, \text{is_married}\)

In these examples the previous forms were simpler and shorter, but inline agents are useful when you want to express the computation just for the agent, without making it a routine of the enclosing class. For example you may define the inline agent

\[
\begin{align*}
(\text{agent} (i: \text{INTEGER}): \text{BOOLEAN} \text{ do Result} & := (\text{item} (i) = a.\text{item} (i) + b.\text{item} (i))) \text{ end})
\end{align*}
\]

which could be useful in a postcondition

\[
\begin{align*}
\text{summed}: (\text{lower} [, , .. | \text{upper}), \text{for all}}
\end{align*}
\]

\[
\begin{align*}
((\text{agent} (i: \text{INTEGER}): \text{BOOLEAN} \text{ do Result} & := (\text{item} (i) = a.\text{item} (i) + b.\text{item} (i))) \text{ end}))
\end{align*}
\]

This states that for every element \(i\) of the interval \([\text{lower} , , .. | \text{upper})\) the value of the item at position \(i\) (in a structure such as an array or list) is the sum of the corresponding values in \(a\) and \(b\). To obtain the same semantics without agent arguments, you would need to express the agent as `agent is_sum_of (\(?, a, b\))` and define a function `is_sum_of` such that \(is\_sum\_of (i, x, y)\) is true if and only if \(\text{item} (i) = x.\text{item} (i) + y.\text{item} (i)\).

The semantics is the same, but if you have many properties of this kind — for example in contracts — the inline form avoids introducing many specialized functions such as `is_sum_of`.

In this example the agent represents a function, with an expression as its body: `item (i) = a.\text{item} (i) + b.\text{item} (i)`. It is also possible to use an inline form for a procedure agent, as in

\[
\begin{align*}
\text{emplist.} \text{do_all}((\text{agent} (e: \text{EMPLOYEE}) \text{ do sum} := \text{sum} + e.\text{salary}\text{ end}) \text{))}
\end{align*}
\]

where `do_all` applies its agent argument to all successive elements in a list; this increases `sum` by the total of all employees’ salaries.

For an agent involving a single routine such as `integer_property`, `integral`, `is_married`, `extend` and the other previous examples, the original non-inline form is shorter, more abstract, and usually preferable.

You may wonder how this can all work in a type-safe fashion. So it is time to stop this preview and cut to the movie.
27.3 FROM CALLS TO AGENTS

Feature calls and their operands

First we should remind ourselves of the basic properties of feature calls. When programming with Eiffel we rely all the time on this fundamental mechanism of object-oriented computation. We write things like

\[
[q] \quad a0.f(a1, a2, a3)
\]

to mean: call feature \( f \) on the object attached to \( a0 \), with actual arguments \( a1, a2, a3 \). In Eiffel this is all governed by type rules, checkable statically: \( f \) must be a feature of the base class of the type \( a0 \); and the types of \( a1 \) and the other actuals of the call must all conform to the types specified for the corresponding formals in the declaration of \( f \).

In a frequent special case \( a0 \), the target of the call, is just \textit{Current}, denoting the current object. Then we may omit the dot and the target altogether, writing the call as just

\[
[u] \quad f(a1, a2, a3)
\]

which assumes that \( f \) is a feature of the class in which this call appears. The first form, with the dot, is a qualified call; the second form is unqualified (hence the names \[Q\] and \[U\] given to our two examples).

In either form the call is syntactically an expression if \( f \) is a function or an attribute, and an instruction if \( f \) is a procedure. If \( f \) has been declared with no formals (as in the case of a function without arguments, or an attribute) we omit the list of actuals, \((a1, a2, a3)\).

The effect of executing such a call is to apply feature \( f \) to the target object, with the actuals given if any. If \( f \) is a function or an attribute, the value of the call expression is the result returned by this application.

To execute properly, the call needs the value of the target and the actuals, for which this chapter needs a collective name:

\begin{quote}
**Operands of a call**

The operands of a call include its target (explicit in a qualified call, implicit in an unqualified call), and its arguments if any.
\end{quote}

In the examples the operands are \( a0 \) (or \textit{Current} in the unqualified version \[U\]), \( a1, a2 \) and \( a3 \). Also convenient is the notion of position of an operand:

\begin{quote}
**Operand position**

The target of a call has position 0. The \( i \)-th actual argument, for any applicable \( i \), has position \( i \).
\end{quote}
Positions, then, range from 0 to the number of arguments declared for the feature. Position 0, the target position, is always applicable.

**Delaying calls**

For a call such as the above, we expect the effect just discussed to occur as a direct result of executing the call instruction or expression: the computation is immediate. In some cases, however, we might want to write an expression that only describes the calls intended computation, and to execute that description later on, at a time of our own choosing, or someone else’s. This is the purpose of agent expressions, which may be described as delayed calls.

Why would we delay a call in this way? Here are some typical cases:

A •We might want the call to be applied to all the elements of a certain structure, such as a list. In such a case we will specify the agent expression once, and then execute it many times without having to re-specify it in the software text. The software element that will repeatedly execute the same call on different objects is known as an **iterator**. Function `for_all`, used earlier, was an example of iterator.

B •In an iterator-like scheme for numerical computation, we might use a mechanism that applies a call to various values in a certain interval, for example to approximate the integral of a function over that interval. The first example in this chapter relied on such an **integral** function.

C •We might want the call to be executed by another software element: passing an agent object to that element is a way to give it the right to operate on some of our own data structures, at a time of its own choosing. This was illustrated with the calls passing to `object` some agent expressions representing operations applicable to `your_list`. GUI examples also belong to that category: to state that a certain action must be executed whenever a certain event (such as mouse click) occurs on a certain graphical object (such as a button), we add an agent representing the action to a list of agents associated with the object and the event.

D •We might want to ensure that the call is executed only when and if needed, and then only once for any particular object. This would give us a “once per object” mechanism along the lines of “once functions” (which are executed once per system).

E •Finally, we may be interested in the agent as a way to gain information about the feature itself, whether or not we ever intend to execute the call. This may be part of the more general goal of providing **introspective** capabilities: ways to enable a software system to explore and manipulate information about its own properties.

Once functions see “**ROUTINE BODY**”, §5, page 218. The once per object mechanism using agents is described below. **Introspection** is also called **reflection**, but the first term appears more appropriate.
These examples illustrate one of the differences between an agent expression and a plain feature call: to execute a feature call we need the value of all its operands (target and actuals); but for an agent expression we may want to leave some of the operands open for later filling-in. This is clearly necessary for cases A and B, in which the iteration or integration mechanism will need to apply the feature repeatedly, using different operands each time. In an integration

\[ \int_{x=a}^{x=b} g(x) \, dx \]

we will need to apply \( g \) to successive values of the interval \([a, b]\).

**Agents and their operands**

For an agent we need to distinguish between two moments:

- **Construction time, call time**

  The **construction time** of an agent object is the time of evaluation of the agent expression defining it. Its **call time** is when a call to its associated operation is executed.

Since the only way to obtain an agent initially is through **agent expressions**, as specified next, it is meaningful to talk about the “agent expression defining it”.

For a normal call the two moments are the same. For an agent we will have one construction time (zero if the expression is never evaluated), and zero or more call times. At construction time, we may leave some operands unspecified; they they will be called the **open** operands. At call time, however, the execution needs all operands, so the call will need to specify values for the open operands. These values may be different for different executions (different call times) of the same agent expression (with a single construction time).

There is no requirement to make all operands open at construction time: you may provide some operands, which will be closed, and leave some others open. In the example of computing, for some values \( u \) and \( v \), the integral

\[ \int_{x=a}^{x=b} h(u, v, x) \, dx \]

where \( h \) is a three-argument function, we pass to the integration mechanism an agent that is closed on its first and last operands, \( u \) and \( v \), but open on \( x \).
Nothing forces you, on the other hand, to leave any operand open. An agent with all operands closed corresponds to the kind of application called C above, in which we don’t want to execute the call ourselves but let another software element carry it out when it is ready. We choose the construction time, and package the call completely, including all the information needed to carry it out; the other software element chooses the call time. This style is used by iterators in the C++ STL library.

At the other extreme, an agent with all operands open has no information about the target and actuals, but includes all the relevant information about the feature. This is useful in application E: passing around information about a feature for introspection purposes, enabling a system to deliver information about its own components.

27.4 AGENT TYPES

A normal call is a syntactical component — instruction or expression — meant only for one thing: immediate execution. If it is an expression (because the feature is a function), it has a value, computed by the execution, and so it denotes an object.

An agent expression has a different status. Since construction time is separate from call time, the agent expression can only denote an object. That object (an agent) contains all the information needed to execute the call later, at various call times. This includes in particular:

- Information about the routine itself and its base type.
- The values of all the closed operands.
What is the type of an agent expression? Four Kernel Library classes are used to describe such types: `ROUTINE`, `PROCEDURE`, `FUNCTION` and `PREDICATE`. Their class headers start as follows:

```
defered class ROUTINE [BASE, OPEN -> TUPLE]
class PROCEDURE [BASE, OPEN -> TUPLE] inherit ROUTINE [BASE, OPEN]
class FUNCTION [BASE, OPEN -> TUPLE, RES] inherit ROUTINE [BASE, OPEN]
class PREDICATE [BASE, OPEN -> TUPLE] inherit FUNCTION [BASE, OPEN, BOOLEAN]
```

In the actual class texts, the formal generic matters have names `BASE_TYPE`, `OPEN_ARGS` and `RESULT_TYPE` to avoid conflicts with programmer-chosen class names. This chapter uses shorter names for simplicity.

If the associated feature is a procedure the agent will be an instance of `PROCEDURE`; for a function or attribute, we get an instance of `PREDICATE` when the result is boolean, of `FUNCTION` with any other type. Here for ease of reference is a picture of the inheritance hierarchy:

### Agent classes

![Inheritance Hierarchy]

The role of the formal generic parameters is:

- `BASE`: type (class + generics if any) to which the feature belongs.
- `OPEN`: tuple of the types of open operands, if any.
- `RES`: result type for a function.

One of the fundamental features of class `ROUTINE` is

```
call (v: OPEN)
    -- Call feature with all its operands, using v for the open operands.
```

→ A.6.30 to A.6.32 in the ELKS chapter, starting on page 1001.
In addition, \textit{FUNCTION} and \textit{PREDICATE} have the feature

\begin{verbatim}
 last_result: RES
  -- Function result returned by last call to \textit{call}, if any
\end{verbatim}

and, for convenience, the function \textit{item} combining \textit{call} and \textit{last_result}, with the following specification:

\begin{verbatim}
 item (v: like open_operands): RES
  -- Result of calling feature with all its operands,
  -- using v for the open operands.
  -- (Uses \textit{call} for the call.)
 ensure
  set_by_call: Result = last_result
\end{verbatim}

The formal generic parameters for \textit{ROUTINE, PROCEDURE, FUNCTION} and \textit{PREDICATE} provide what we need to make the agent mechanism statically type-safe. \textit{OPEN}, a tuple type, gives the exact list of open operand types; since the argument to \textit{call} and \textit{item} is of type \textit{OPEN}, it is possible from the software text to check that the actual arguments to \textit{call} will at call time be of the proper types, conforming to the original feature’s formal argument types at the open positions. The actuals at closed positions are set at construction time, again with type checking. So the combination of open and closed actuals will be type-valid for the feature.

\textit{ROUTINE, PROCEDURE, FUNCTION} and \textit{PREDICATE} have more features than listed above; in particular, they provide introspection facilities, describing properties of the associated routines and discussed below. For a complete interface specification, see the corresponding sections in the presentation of Kernel Library classes.

\section*{27.5 CALL AGENTS}

How do we obtain agent objects? The most common construct is a \textit{call agent} expression. (We will see the other case, \textit{inline agents}, in a later section.)

The basic form of a call agent is very simple: just add the keyword \textit{agent} at the beginning of a normal feature call. This yields an agent with operands all closed. To specify open operands, you may:

\begin{itemize}
  \item Use a question mark \texttt{?} in lieu of an argument.
  \item Use a type in braces, \texttt{\{TYPE\}}, in lieu of the target.
  \item Omit the argument list altogether, to make all arguments open.
\end{itemize}

Let’s examine these variants and the associated semantics.
§27.5 CALL AGENTS

All-closed agents

If you start from a valid call, either qualified or unqualified

\[
\begin{align*}
\text{[Q]} & & a_0, f(a_1, a_2, a_3) \\
\text{[U]} & & f(a_1, a_2, a_3)
\end{align*}
\]

you get an agent expression in each case by adding the keyword agent:

\[
\begin{align*}
\text{agent } a_0 & . f(a_1, a_2, a_3) \\
\text{agent } f(a_1, a_2, a_3)
\end{align*}
\]

Such an agent expression is not a call (instruction or expression) any more, but an expression of a new syntactic kind, Feature_agent, denoting an agent, of a PROCEDURE type if \( f \) is a procedure, and a FUNCTION or PREDICATE type if \( f \) is a function. Both of these examples have no arguments, so they are closed on all operands; we will start adding arguments soon.

You can do with an agent expression all you are used to do with other expressions. You can assign it to an entity of the appropriate type; assuming \( f \) is a procedure of a class \( CC \), you may write, in class \( CC \) itself:

\[
\begin{align*}
p: \text{PROCEDURE } [CC, \text{TUPLE}] \\
\ldots \\
p := \text{agent } a_0 & . f(a_1, a_2, a_3) \\
\ldots \\
p & . \text{call } ([])
\end{align*}
\]

Since all operands are closed — we have specified the target \( a_0 \) and all the arguments \( a_1, a_2, a_3 \) — the second formal generic is just TUPLE, and the call to call takes an empty tuple \([\] \).

More commonly than assigning a call expression to an entity as here, you will pass it as actual argument to a routine, as in

\[
\begin{align*}
\text{object}, \text{do_something} (\text{agent } a_0 & . f(a_1, a_2, a_3))
\end{align*}
\]

where do_something, in the corresponding class, takes a formal \( p \) declared as

\[
\begin{align*}
p: \text{PROCEDURE } [CC, \text{TUPLE}]
\end{align*}
\]

or just

\[
\begin{align*}
p: \text{PROCEDURE } [\text{ANY}, \text{TUPLE}]
\end{align*}
\]

presumably to call call on \( p \) at some later stage, as we will shortly learn to do. This was the scheme called C in the presentation of example applications; passing a completely closed agent to another component of the system, to let it execute the call when it chooses to. For example you can pass agent your_list.start or agent your_list.extend(some_value).
Keeping operands open

The examples just seen are still of limited interest because all their operands are closed. But you may want to keep some operands open for latter filling-in at call time, for example by an iteration or integration mechanism.

To specify an open target, you will replace the target by its type in braces, \( [\text{TARGET\_TYPE}] \). This is the brace convention. To specify an open argument, you will use the question mark convention: just replace the target by a question mark ?.

Here are some examples, obtained by starting from the call \( a_0.f(a_1, a_2, a_3) \) and opening the target or some arguments.

```
-- Start with an agent closed on all operands:
s := agent a_0.f(a_1, a_2, a_3)

-- Next, individually open the target and each successive argument:
t := agent \{T_0\}.f(a_1, a_2, a_3)
u := agent a_0.f(? , a_2, a_3)
v := agent a_0.f(a_1, ?, a_3)
w := agent a_0.f(a_1, p, ?)

-- An example with two open arguments, target closed:
x := agent a_0.f(a_1, ?, ?)

-- Arguments all open, target still closed:
y := agent a_0.f(? , ?, ?)

-- Finally, open everything:
z := agent \{T_0\}.f(? , ?, ?)
```

The respective types of these call expressions are, assuming that \( f \) is a procedure declared in the base class of \( T_0 \), having formals declared of types \( T_1, T_2 \) and \( T_3 \):

```
s: PROCEDURE [T_0, TUPLE]
t: PROCEDURE [T_0, TUPLE [T_0]]
u: PROCEDURE [T_0, TUPLE [T_1]]
v: PROCEDURE [T_0, TUPLE [T_2]]
w: PROCEDURE [T_0, TUPLE [T_3]]
x: PROCEDURE [T_0, TUPLE [T_2, T_3]]
y: PROCEDURE [T_0, TUPLE [T_1, T_2, T_3]]
z: PROCEDURE [T_0, TUPLE [T_0, T_1, T_2, T_3]]
```

If \( f \) were a function, the types would use FUNCTION instead of PROCEDURE, with an extra generic parameter representing the result type (except for a boolean-valued function, which would use PREDICATE).
The first generic parameter, \( T_0 \) in all of these examples, represents the current type (class with generic parameters if any) of the underlying feature. Here we assume for simplicity that \( f \) comes from a non-generic class \( T_0 \).

The second generic parameter, a tuple type, represent the sequence of types of open operands. For the first example, \( t \), it’s just \( \text{TUPLE} \) with no parameters, since the agent has no open operands. For the other examples the parameters of the \( \text{TUPLE} \) type represent the types of the open operands. They indicate what argument types are permissible in calls to \( \text{call} \) (or \( \text{item} \) for a function) on the corresponding agents.

Here indeed are examples of valid uses of \( \text{call} \) on the previous agent examples. For each of them, the comment on the next line shows how we would have obtained the same effect through a normal call (call time same as construction time, not using agents).

```haskell
val_0: T0; val_1: T1; val_2: T2; val_3: T3
... Assign values to \( val_0, val_1, val_2, val_3 \)...
s. call ([]) -- Note empty tuple: no open operands
   -- \( a0.f(a1, a2, a3) \)
t. call ([val_0])
   -- \( \text{val}_0.f(a1, a2, a3) \)
u. call ([val_1])
   -- \( a0.f(val_1, a2, a3) \)
v. call ([val_2])
   -- \( a0.f(a1, \text{val}_2, a3) \)
w. call ([val_3])
   -- \( a0.f(a1, a2, \text{val}_3) \)
x. call ([val_2, val_3])
   -- \( a0.f(a1, \text{val}_2, \text{val}_3) \)
y. call ([val_1, val_2, val_3])
   -- \( a0.f(val_1, \text{val}_2, \text{val}_3) \)
z. call ([val_0, val_1, val_2, val_3]) -- Must provide all operands
   -- \( \text{val}_0.f(val_1, \text{val}_2, \text{val}_3) \)
```

It should be clear by now how mechanisms such as \( \text{for\_all} \) can manage to work on operations that work on their target, such as \( \text{is\_married} \), as well as others that work on an argument, such as \( \text{is\_positive} \). The type of an agent only describes, through the \( \text{OPEN} \) parameter, the tuple of types of operands. It doesn’t make any difference whether these open operands come from a target or an argument.
For example, both of the following boolean expressions

\[
\text{emplist.} \text{for\_all} (\text{agent \{EMPLOYEE\}.is\_married})
\]

\[
\text{emplist.} \text{for\_all} (\text{agent object.is\_married (?))}
\]

will be valid if:

- Class \text{EMPLOYEE} has, as previously assumed, a feature \text{is\_married}: \text{BOOLEAN}.

- \text{object} is of type \text{SOME\_TYPE}, and \text{SOME\_TYPE} has a feature \text{is\_married (e: BOOLEAN)}.

The brace convention

Two of the examples used the brace convention to keep the target open:

\[
t := \text{agent \{T0\}.f(a1, a2, a3)}
\]

\[
z := \text{agent \{T0\}.f(?, ?, ?)}
\]

-- Also expressible (see below) as just: \text{agent \{T0\}.f}

For the target, as noted, the question mark convention is not applicable, since the feature name does not suffice to identify the target type: many classes may have a feature called \text{f}.

For arguments we have no such problem since once we know \text{f} and its class we know the declared type of each of \text{f}'s formal arguments. This justifies the question mark convention for arguments.

Omitting the argument list

A further simplification of the notation is available when \text{all} arguments are open, as in \text{agent a0.f (?, ?, ?)}. Then you may omit the parenthesized argument list, as in

\[
\text{agent a0.f}
\]

-- Abbreviates \text{agent a0.f (?, ?, ?)}

A call of the form \text{a0.f} would be invalid, since \text{f} always requires three actual arguments. But with an \text{agent} expression the convention of omitting arguments creates no ambiguity; it simply means that we consider an agent built from \text{f} with all arguments open.
This fully abbreviated form has the advantage of conveying the idea that the denoted agent is a true “feature object”, carrying properties of the feature in its virginal state, not tainted by any particular choice of actual argument. The last two variants shown do not even name a target. This is the kind of object that we need for such introspective applications as writing a system that enables its users to browse through its own classes.

A summary of the possibilities

As a summary of the preceding examples, here is a summary of the ways to build a call agent:

**Syntactical forms for a call agent**

A call agent is of the form

```
agent agent_body
```

where `agent_body` is a Call, qualified (as in `x.r (...)`) or unqualified (as in `f (...)`) with the following possible variants:

- You may replace any argument by a question mark `?`, making the argument open.
- You may replace the target, by `{TYPE}` where `TYPE` is the name of a type, making the target open.
- You may remove the argument list `(...)` altogether, making all arguments open.

This is not a formal syntax definition, but a summary of the available forms permitted by the syntax and validity rules that follow.

27.6 USING AGENTS

Although we have studied only one of the two syntactical forms of agents, call agents (the other is inline agents), and not yet taken the trouble to look at the syntax, validity rules and precise semantics, we have enough background to explore applications of agents, starting with the examples sketched at the very beginning of this chapter, which we can now revisit and extend. We’ll see how to make them work in practice: not just the client side — registering an action to be executed for a certain GUI event, integrating a function, iterating an operation — but the suppliers too: the event processing, the integrator, the iterators.
GUI programming: establishing a direct connection to the Business Model

The first example illustrated the EiffelVision style of GUI programming. We wrote

```
your_button, click_actions, extend (agent your_routine)
```

to specify that `your_routine` must be executed whenever the button press event occurs on `your_button` during execution. Here is how things work. In your application, `your_button` denotes a graphical object, variously known as a “control” (the Windows terminology), a “widget” (the X Windows terminology) or a “context”; `click` denotes one of the events that may occur on this control. The list `your_button, click_actions` contains agents, representing the actions to be executed when the event occurs on the control. This is a plain list (from the EiffelBase library), to which we may, as here, apply the procedure `extend`, adding a new item at the end.

When EiffelVision detects that the event has occurred on the button, it will execute, for every element `item` of the list of agents, a call such as

```
item, call ([])
```

For the list `item` that represents `your_routine`, this will produce what we wanted: a call to `your_routine` in response to the event.

This setup assumes that `your_routine` is a routine without arguments. In reality, a routine to be executed as a result of a mouse event, such as a click, may need the `x, y` mouse coordinates of the event. Let’s call it `your_routine2`. What EiffelVision actually executes is

```
item, call ([mouse_horizontal, mouse_vertical])
```

using as arguments the cursor coordinates, part of the event’s information recorded in the event. This assumes of course that `your_routine2` can deal with these arguments. If `your_routine2` indeed takes two real values as arguments, the previous form of registering the agent

```
your_button, click_actions, extend (agent your_routine2)
```

is still applicable; as you will remember, it is a shortcut for

```
your_button, click_actions, extend (agent your_routine(?, ?))
```
Now assume that your_routine is a routine from the “Business Model” part of your application, meaning the part of the software that takes care of doing the real processing, independently of any GUI. The \( x \) and \( y \) values might be only some of the arguments that your_routine needs. For example your_routine might be the procedure

\[
\text{compute_stats (country: COUNTRY; year: INTEGER; x, y: REAL)}
\]

which, in a cartographical application, computes statistics for a certain year for the city closest to positions \( x \) and \( y \) on the map for a certain country. When loading the map for that country you may register compute_stats:

\[
\text{your_button\_click\_actions\_extend}
\]

\[
\text{agent compute_stats (Usa, 2002, ?, ?))}
\]

The beauty of the notion of closed and open arguments is that you can set some values (here the country and the year) at construction time, and leave others (here the mouse coordinates) to be filled in at call time.

To the EiffelVision mechanism, there is no difference between this case using compute_stats — a routine with four arguments, two of which we have closed at construction time — and the previous one involving your_routine2 and its two open arguments. The call executed by the EiffelVision side, shown above as

\[
\text{item\_call ([mouse\_horizontal, mouse\_vertical])}
\]

works properly in both cases.

This scheme, relying on open and closed arguments, has crucial practical consequences for the programming of GUI applications. Following the MVC model introduced by Smalltalk, it is often stated that GUI applications should include three components:

- **Model** (the acronym’s M), called the **Business Model** above: this is the part that does the actual computation, data manipulation and processing. A routine such as compute_stats, describing some important operation of the Business Model, belongs to this part of the system.

- **View** (the “V”): the purely graphical part of the application, taking care of presenting information visually and interacting with users. Notions such as buttons, other controls and events belong to that part.

- **Controller** (the “C”): software elements that connect the model with the view, by specifying what operations from the model must be executed in response to what user interface events.
Without agents, the Controller part, serving as glue between Model and View, can take up a significant amount of code, based for example on command classes. As the last example indicates, using agents can bring the need for such glue code down to a minimum, or even remove it altogether. The only Controller element that we used in this example to connect the button and event to the routine `compute_stats` from our model was the agent `agent compute_stats (Usa, 2002, ?, ?)`. You don’t have to write any other code: no new class, not even any special instructions.

This is one of the great benefits of agents for GUI programming, as used extensively in EiffelVision: you directly connect elements from the Business Model to elements from the User Interface, without requiring any “glue code”. The notion of open and closed operands gives us remarkable flexibility: as long as a routine from the Business Model, such as `compute_stats`, takes arguments representing the coordinates, it doesn’t matter what positions these arguments have in the routine, and what others it may have. Just leave the $x$ and $y$ arguments open when you connect the routine to the interface.

This ability to plug elements of the Business Model directly into the user interface is one of the principal attractions of the agent model.

One of the uses of command classes is to support undoing and redoing in an interactive system. It is easy to see how to provide this through agents too: just pass two agents, one representing the “do” operation and the other representing the “undo”. This technique — whose details the reader is invited to spell out — is used in many of ISE’s interactive products supporting undo and redo.

### Integrating a function

The next set of examples was about integration. We assumed functions

```eiffel
g (x: REAL): REAL
h (x: REAL; a: T1; b: T2): REAL
```

and wanted to integrate them over a real interval such as $[0, 1]$, that is to say, approximate the two integrals

$$
\int_{x=0}^{x=1} g (x) \, dx \quad \text{and} \quad \int_{x=0}^{x=1} h (x, u, v) \, dx
$$

We declare

`your_integrator: INTEGRATOR`
and, with the proper definition of function \texttt{integral} in class \texttt{INTEGRATOR},
to be seen shortly, we will obtain the integrals through the expressions

\begin{verbatim}
your_integrator. integral (agent g (?), 0.0, 1.0)
your_integrator. integral (agent h (? u, v), 0.0, 1.0)
\end{verbatim}

The question mark indicates, in each case, the open argument: the place where \texttt{integral} will substitute various real values for \( x \) when evaluating \( g \) or \( h \).

Note that if we wanted in class \texttt{D} to integrate a real-valued function from class \texttt{REAL}, such as \texttt{abs} which is declared in \texttt{REAL} as

\begin{verbatim}
abs: REAL
   -- Absolute value
do ... end
\end{verbatim}

we would obtain it simply through

\begin{verbatim}
your_integrator. integral (agent \{REAL\}.abs, 0.0, 1.0)
\end{verbatim}

Let us now see how to write function \texttt{integral} to make all these uses possible. We use a primitive algorithm — this is not a treatise on numerical methods — but what matters is that any integration technique will have the same overall form, requiring it to evaluate \( f \) for various values in the given interval. Here class \texttt{INTEGRATOR} will have a real attribute \texttt{step} representing the integration step, with an invariant clause stating that \texttt{step} is positive. Then we may write \texttt{integral} as:

\begin{verbatim}
The boxed expression is where the algorithm needs to evaluate the function \( f \) passed to \texttt{integral}. Remember that \texttt{item}, as defined in class \texttt{FUNCTION}, calls the associated function, substituting any operands (here \( x \)) at the open positions, and returning the function’s result. The argument of \texttt{item} is a tuple (of type \texttt{OPEN}, the second generic parameter of \texttt{FUNCTION}); this is why we need to enclose \( x \) in brackets, giving a one-argument tuple: \([x]\).

In the first two example uses, \texttt{agent g (?)} and \texttt{agent h (?, u, v)}, this argument corresponds to the question mark operands to \( g \) and \( h \). In the last example the call expression passed to \texttt{integral} was \texttt{agent \{REAL\}.abs}, where the open operand is the target, represented by \{\texttt{REAL}\}, and successive calls to \texttt{item} in \texttt{integral} will substitute successive values of \( x \) as targets for evaluating \texttt{abs}.

In the case of \( h \), the closed operands \( u \) and \( v \) are evaluated at the time of the evaluation of the expression \texttt{agent h (?, u, v)}, and so they remain the same for every successive call to \texttt{item} within a given execution of \texttt{integral}. 
Note the type \( \text{FUNCTION} [\text{ANY}, \text{TUPLE} [\text{REAL}, \text{REAL}]]; \) declared in \text{integral} for the argument \( f \). It means that the corresponding actual must be a call expression describing a function from any class (hence the first actual generic parameter, \text{ANY}) that has one open operand of type \text{REAL} (hence \text{TUPLE} [\text{REAL}]) and returns a real result (hence \text{REAL}). Each of the three example functions \( g, h \) and \( \text{abs} \) can be made to fit this bill through a judicious choice of open operand position.

**Iteration examples**

The next set of initial examples covered iteration. In a class \( \text{CC} \) we want to manipulate both a list of integers and a list of employees and apply the same function \( \text{for\_all} \) to both cases:

\[
\text{if intlist.\text{for\_all} (agent is\_positive (?)) then … end}
\]
\[
\text{if intlist.\text{for\_all} (agent over\_threshold (?)) then … end}
\]
\[
\text{if emplist.\text{for\_all} (agent \{\text{EMPLOYEE}\}, is\_married) then … end}
\]
The function `for_all` is one of the iterators defined in class `TRAVERSABLE` of EiffelBase, and available as a result in all descendant classes describing traversable structures, such as `TREE` and `LINKED_LIST`. This boolean-valued function determines whether a certain property holds for every element of a sequential structure. The property is passed as argument to `for_all` in the form of a call expression with one open argument.

Our examples use three such properties of a very different nature. The first two are functions of the client class `CC`, assessing properties of their integer argument. The result of the first depends only on that argument:

```eiffel
is_positive (i: INTEGER): BOOLEAN
-- Is i positive?
do Result := (i > 0) end
```

Alternatively the property may, as in the second example, involve other aspects of `CC`, such as an integer attribute `threshold`:

```eiffel
over_threshold (i: INTEGER): BOOLEAN
-- Is i greater than threshold?
do Result := (i > threshold) end
```

Here `over_threshold` compares the value of `i` to a field of the current object. Surprising as it may seem at first, function `for_all` will work just as well in this case; the key is that the call expression `agent over_threshold (?)`, open on its argument, is closed on its target, the current object; so the agent object it produces has the information it needs to access the `threshold` field.

In the third case, the argument to `for_all` is `agent {EMPLOYEE}, is_married`; this time we are not using a function of `CC` but a function `is_married` from another class `EMPLOYEE`, declared there as

```eiffel
is_married: BOOLEAN is do ... end
```

Unlike the previous two, this function takes no argument since it assesses a property of its target; We can still, however, pass it to `for_all`: it suffices to make the target open.

The types of the call expressions are the following:

```
PREDICATE [CC, TUPLE [INTEGER]]
-- In first two examples (is_positive and over_threshold)
PREDICATE [EMPLOYEE, TUPLE [EMPLOYEE]]
-- In the is_married example
```

This assumes again that `CC` is non-generic, so that it is both a class and a type. Remember that a PREDICATE is a FUNCTION with a BOOLEAN result type.
You may also apply \texttt{for\_all} to functions with an arbitrary number of arguments, as long as you leave only one operand (target or argument) open, and it is of the appropriate type. You may for example write the expressions

\begin{verbatim}
intlist, for_all (agent some_criterion (e1, ?, e2, e3))
emplist, for_all (agent {EMPLOYEE}, some_function (e4, e5))
\end{verbatim}

assuming in \texttt{CC} and \texttt{EMPLOYEE}, respectively, the functions

\begin{verbatim}
some_criterion (a1: T1; i: INTEGER; a2: T2; a3: T3) -- In CC
some_function (a4: T4; a5: T5) -- In EMPLOYEE
\end{verbatim}

for arbitrary types $T_1, \ldots, T_5$. Since operands $e_1, \ldots, e_5$ are closed in the calls, these types do not in any way affect the types of the call expressions, which remain as above: \texttt{PREDICATE [CC, TUPLE [INTEGER]]} and \texttt{PREDICATE [EMPLOYEE, TUPLE [EMPLOYEE]]}.

Let us now see how to write the iterator mechanisms themselves, such as \texttt{for\_all}. They should be available in all classes representing traversable structures, so they must be introduced in a high-level class of EiffelBase, \texttt{TRAVERSABLE [G]}. Some of the iterators are unconditional, such as

\begin{verbatim}
do_all (action: ROUTINE [ANY, TUPLE [G]])
  -- Apply action to every item of the structure in turn.
  require
    \[\ldots\] Appropriate preconditions \[\ldots\]
  do
    from start until off loop
      action, call ([item])
      forth
  end
end
\end{verbatim}
This uses the four fundamental iteration facilities, all declared in the most
general form possible as deferred features in TRAVERSABLE: start to
position the iteration cursor at the beginning of the structure; forth to
advance the cursor to the next item in the structure; off to tell us if we have
exhausted all items (not off is a precondition of forth); and item to return
the item at cursor position.

The argument action is declared as ROUTINE [ANY, TUPLE [G]],
meaning that we expect a routine with an arbitrary base type, with an open
operand of type G, the formal generic parameter of TRAVERSABLE,
representing the type of the elements of the traversable structure. Feature
item indeed returns a result of type G (representing the element at cursor
position), so that it is valid to pass as argument the one-argument tuple
[item] in the call action, call ([item]) that the loop repeatedly executes.

We normally expect action to denote a procedure, so its type could be more
accurately declared as PROCEDURE [ANY, TUPLE [G]]. Using ROUTINE
leaves open the possibility of passing a function, even though the idea of
treating a function as an action does not conform to the Command-Query
Separation principle of the Eiffel method.

Where do_all applies action to all elements of a structure, other iterators
provide conditional iteration, selecting applicable items through another
call expression argument, test. Here is the “while” iterator:

```
while_do
  (action: ROUTINE [ANY, TUPLE [G]]
  test: PREDICATE [ANY, TUPLE [G]])
  -- Apply action to every item of structure up to,
  -- but not including, first one not satisfying test.
  -- If all satisfy test, apply to all items and move off.
  require
    … Appropriate preconditions …
  do
    from start until
    off or else not action, test ([item])
    loop
      action, call ([item])
    forth
    end
  end
```

Descendants of TRAVERSABLE effect these features in various ways
to provide iteration mechanisms on lists, hash tables, trees and
many other structures.
Note how the algorithm applies call to action, representing a routine (normally a procedure), and item to test, representing a boolean-valued function. In both cases the argument is the one-element tuple [item].

The iterators of TRAVERSABLE cover common control structures: \texttt{while\_do}; \texttt{do\_while} (same as \texttt{while\_do} but with “test at the end of the loop”, that is to say, apply action to all items up to and including first one satisfying test); \texttt{until\_do}; \texttt{do\_until}; \texttt{do\_if}.

Yet another iterator of TRAVERSABLE is \texttt{for\_all}, used in earlier examples. It is easy to write a \texttt{for\_all} loop algorithm similar to the preceding ones. Here is another possible definition, in terms of \texttt{while\_do}:

\begin{verbatim}
for_all (test: PREDICATE [G, TUPLE [G]]): BOOLEAN
  -- Do all items satisfy test?
  require
    ... Appropriate preconditions ...
  do
    while_do (agent nothing (?), test)
    Result := off
  end
\end{verbatim}

using a procedure \texttt{nothing} (x: \texttt{G}) which has no effect (but needs an argument \texttt{x} for typing reasons, since the first argument of \texttt{while\_do} must be of type \texttt{ROUTINE [ANY, TUPLE [G]]}). It is trivial to define \texttt{nothing} in terms of \texttt{procedure do\_nothing}, from class \texttt{ANY}. We apply \texttt{nothing} as long as \texttt{test} is true of successive items; if we find ourselves \texttt{off}, we return true; otherwise we have found an element not satisfying the \texttt{test}.

Assuming a proper definition of \texttt{do\_until}, the declaration of \texttt{exists}, providing the second basic quantifier of predicate calculus, is nicely symmetric with \texttt{for\_all}:

\begin{verbatim}
exists (test: PREDICATE [G, TUPLE [G]]): BOOLEAN
  -- Does at least one item satisfy test?
  require
    ... Appropriate preconditions ...
  do
    do_until (agent nothing (?), test)
    Result := not off
  end
\end{verbatim}

\texttt{do\_nothing} is cited in 35.6, page 920.

It is possible to avoid defining a procedure \texttt{nothing} by using an inline agent.
Before moving on to the last details of the agent mechanism, let’s gain further appreciation for its power and versatility by looking at two interesting applications, error processing and “once per object” (followed in the next section by examples of the inline form).

Error processing without the mess

The first example addresses a frequent situation in which we perform a sequence of actions, each of which might encounter an anomaly that prevents continuing as hoped. The problem here is that it’s difficult to avoid a complex, deeply nested control structure, since we may have to get out at any step. The straightforward implementation will look like this:

```
action1
  if ok1 then
    action2
      if ok2 then
        action3
          ... More processing, more nesting ...
      end
  end
... More processing, more nesting ...
end
```

For example, we may want to do something with a file of name `path_name`. We first test that that `path_name` is not void. Then that the string is not empty. Then that the directory exists. Then that the file exists. Then that it is readable. Then that it contains what we need. And so on. A negative answer at any step along the way must lead to reporting an error situation and aborting the whole process.

The problem is not so much the nesting itself; after all, some algorithms are by nature complex. But often the normal processing is not complicated at all; it’s the error processing that messes everything up, hiding the “useful” processing in a few islands lost in an ocean of error handling. If the error processing is different in each case (not ok1, not ok2 and so on) we can’t do much about it. But if it is always of the form: “Record the error source and terminate the whole thing”, then the above structure may seem too complicated. Although we may address this issue through exceptions, they are often overkill.

An agent-based technique is useful in some cases. It assumes that you write the various actions — action1 ... action3 above — as procedures, each with a body of the form

```
...Try to do what’s needed...
controlled_check (execution_ok, "...Appropriate message...")
```

with `execution_ok` representing the condition that must be satisfied for the processing to continue. Then you can rewrite the processing above as just:
This linear structure is much simpler than the original.

The features whose names start with `controlled_` come from the EiffelBase class `CONTROLLED_EXECUTION`, of which the class containing the above scheme should be a descendant. These procedures are not difficult to write; for example `controlled_check` sets `controlled_glitch` and `controlled_glitch_message`, and `controlled_execute` looks like this:

```eiffel
if controlled_glitch then
  warning (controlled_glitch_message)
end
```

This routine as it appears in the library has a few extra instructions to record the glitch step and, on option, raise an exception.

```eiffel
controlled_execute ([:
  agent action1,
  agent action2 (...),
  agent action3 (...)
])
```

```eiffel
if controlled_glitch then
  warning (controlled_glitch_message)
end
```

---

**Once per object**

The second example, also supported by an EiffelBase class, provides a “once per object” mechanism.

You know, of course, Eiffel’s “once routines”, executed only once per system execution. They define a “once per class” mechanism: all instances of a class share the result of a once function. (All these concepts are applicable to procedures, but for this discussion we restrict ourselves to functions.) Now assume you need functions that compute a result specific to each instance of the class, and computed just once for that instance, the first time it’s requested — if at all.

```eiffel
controlled_execute ([:
  agent action1, 
  agent action2 (...),
  agent action3 (...)
])
```

```eiffel
controlled_execute ([:
  agent action1, 
  agent action2 (...),
  agent action3 (...)
])
```

---

For an introduction to once routines see "ROUTINE BODY", §5, page 218.
A typical application would be large pieces of information associated with objects of a certain type, but stored in a database; for example each instance of a class `COMPANY` may have `stock_history` information, of type `HISTORY`, which may be huge. We only want to retrieve the information on demand; given the size of the information and the number of instances of the class, it is not acceptable to load everything ahead of time. Even if an instance of `COMPANY` is in memory, we want to retrieve the associated `HISTORY` from the database only when and if we need access to the company's `stock_history`.

Agents provide us with a general solution to all problems of this kind. In class `COMPANY` you will simply declare

```
stock_history: ONCE_PER_OBJECT [HISTORY]
```

and obtain the value, when and if needed, as

```
stock_history.item (agent retrieved_history)
```

Here `retrieved_history` is the function that computes the needed result — the one that you want to call once for each object. That's all you have to do! Note that this scheme allows you to have as many “once per object” functions as you like in any given class. It relies on a general-purpose EiffelBase class `ONCE_PER_OBJECT` of the following form:

```eiffel
expanded class
  ONCE_PER_OBJECT [G]
feature -- Access
  item (f: FUNCTION [ANY, TUPLE, G]): G
    -- Value of f, computed once for each object;
    -- subsequent calls return same value for same object.
    do
      if not computed then
        internal_result := f.item ([ ])
        computed := True
      end
      Result := internal_result
    end
feature {NONE} -- Implementation
  computed: BOOLEAN
    -- Has item already been requested?
  internal_result: G
    -- Result, if already computed
end
```
27.8 USING INLINE AGENTS

The agents seen so far are of the Call_agent kind, relying on class features, such as \( f \) and \( g \) (integration examples), integer_property and is_married (iterator examples), compute_stats (EiffelVision example) and others.

Sometimes, the only reason for writing a certain computation is to define an agent from it. To avoid adding a feature that will make the enclosing class more complicated, you may write the algorithm within the agent. The syntactical construct for this inline case, previewed at the beginning of this chapter, mirrors the definition of a routine — although, like any other agent construct, it is syntactically an expression. Here are some examples of inline agents, all to be used as expressions:

\[
\begin{align*}
\text{agent} & (i: \text{INTEGER}) : \text{BOOLEAN} \text{ do } \text{Result} := \text{is_positive} (i) \text{ end} \\
\text{agent} & (e: \text{EMPLOYEE}) : \text{BOOLEAN} \text{ do } \text{Result} := e.\text{is_married} \text{ end} \\
\text{agent} & (e, f: \text{EMPLOYEE}) : \text{BOOLEAN} \text{ do } \text{Result} := (e.\text{salary} > f.\text{salary}) \text{ end} \\
\text{agent} & (e, f: \text{EMPLOYEE}; p: \text{POSITION}) : \text{BOOLEAN} \text{ do } \text{Result} := (e.\text{job} = p) \text{ and } (f.\text{job} = p) \text{ end}
\end{align*}
\]

As noted in the comments, the first two of these examples have Call_agent equivalents, since they directly rely on existing routines of some class. But in the last two cases, there are no such routines.

The third agent (for example) denotes an object representing a boolean-valued operation that, for two objects of type EMPLOYEE, returns true if and only if the query salary yields a higher result for the first than for the second.

It is still possible to use a Call_agent in these cases, but this requires adding features to the enclosing class:

\[
\begin{align*}
\text{higher_salary} & (e, f: \text{EMPLOYEE}) : \text{BOOLEAN} \\
& \text{ do } \text{Result} := (e.\text{salary} > f.\text{salary}) \text{ end} \\
\text{same_job} & (e, f: \text{EMPLOYEE}; pos: \text{POSITION}) : \text{BOOLEAN} \\
& \text{ do } \text{Result} := ((e.\text{job} = pos) \text{ and } (f.\text{job} = pos)) \text{ end}
\end{align*}
\]
to enable rewriting the calls as **agent** higher_salary (abbreviating, as usual, **agent** higher_salary (? , ?)) and **agent** same_job. But if the only use of the given little algorithms is to define the corresponding agents, for example to pass them to some iterators, then you may want to avoid burdening the enclosing class with such routines, using inline agents instead.

The inline agents shown so far denote functions (**FUNCTION** or **PREDICATE**). Here is an example that passes an inline procedure agent to an iterator, to raise by 50 percent the salary of every employee called “Tina”:

```
emplist.do_all
((agent e: EMPLOYEE)
  require
    employee_exists: e /= Void
  do
    if equal (e.first_name, once "Tina") then
      e.set_salary (1.5 * e.salary)
    end
  end)
```

The **require** … **do** … **end** part is a specimen of **Routine**; an inline agent indeed uses exactly the same **Routine** construct as the declaration of a routine in a class; so it can have all the applicable clauses, such as **Precondition** here, but also **Local_declarations**, **Postcondition** and **Rescue**.

We can use an inline agent to simplify the earlier definition of **for_all** in terms of **while_do**, which required a function **nothing** (x: G) because **do_nothing** from **ANY**, with no argument, has the wrong signature. An inline agent avoids this:

```
for_all (test: PREDICATE [G, TUPLE [G]]): BOOLEAN
  -- Do all items satisfy test?
  do
    while_do ((agent (x: G) do do_nothing end), test)
    Result := off
  end
```

Inline agents do not give us anything fundamentally new, since we can always use call agents instead. They are useful if you want to avoid features such as **same_job** and **nothing** whose only purpose is to define agents.

The **methodological advice** is clear: if the computation becomes complex, it is usually better to add a feature to the class. The agent passed as argument to **do_all** in the last example is already complex enough to justify writing a separate function instead.

The inline form is particularly useful to express advanced contract specifications. Here is an example. Assume that in a class describing sequential structures (such as **LIST** [G] in EiffelBase) you write a procedure that appends an element. It might include this postcondition:
In the last postcondition clause — the one of interest for this discussion — $1 \ldots \text{old count}$ is the interval from 1 to \text{old count}, to whose items \text{for_all} applies the agent property on the next line. The property expresses that the item at position $i$, for arbitrary $i$, is equal to the original item at that position (more precisely, to the item at position $i$ in \text{old twin}, a copy of the list taken on entry to the procedure). This is typical of how agents enable us to express non-trivial postcondition or invariant properties, stating that a whole set of items have not changed, or have a certain association with the corresponding set of items in another structure.

We could restate the inline agent (the argument to \text{for_all}) in non-inline form as \texttt{agent equal\_item (\texttt{old twin}, \_)}, but this assumes a function

```plaintext
extend (x: G)
  -- Add x at end; keep other items
  require
  ...
  do ...
  ensure
    one\_more: count = old count + 1
    added\_at\_end: item (count) = x
    others\_unchanged:
      (1 \ldots \text{old count}), \text{for_all}
      (\text{agent} (i: \text{INTEGER}): \text{BOOLEAN}
        \text{do Result :=equal (item (i), (\text{old twin}). item (i)) end})
  end
```

If you want to specify your software completely — expressing not only straightforward properties such as \texttt{item (count) = x}, but also those involving entire substructures — you may end up writing many such functions. Although they add interesting information, one may also feel that, being only used for assertions, they needlessly complicate the class. They may destabilize the software since any effort at better specification may cause the addition of a whole set of new features, used only in the assertions and of no other interest to clients of the class. Inline agents solve this problem.
Here is another example application. The agents described in this chapter represent delayed calls; you may have wondered whether we also need an expression construct to denote delayed object creation, perhaps something like `agent create {SOME_TYPE}, make (a1,?)`. The answer is no, since we can achieve the intended effect (assuming we need it) by using a creation expression as part of an inline agent in

```
(agent (b1: B) do create {SOME_TYPE}, make (a1, b1) end)
```

where `B` is the type of `make`'s second argument.

You may view inline agents as anonymous routines, similar to anonymous classes (tuple types) and anonymous objects (tuples). This is particularly clear in the Routine case `(…) … do … end`, which has exactly the same form as a routine declaration:

```
r (…) is … do … end
```

(with, as noted, the possibility of including all relevant clauses, such as precondition, postcondition, rescue, local variable declarations). The only difference is that the inline agent doesn’t use a routine name `r` — it doesn’t need one. When such a routine is used with the sole purpose of being passed as argument to a routine expecting an agent, the anonymous form avoids cluttering the class with a full-status routine.

### 27.9 ACCESSING FEATURE PROPERTIES

Class `ROUTINE` and its descendants provide a starting point for many of the introspection needs that Eiffel applications may need.

The first introspection mechanism is a simple way, through class `ROUTINE` and its descendants, to gain access to the precondition and postcondition of a routine:

```
precondition (args: OPEN): BOOLEAN
  -- Do args satisfy routine’s precondition in present state?

postcondition (args: OPEN): BOOLEAN
  -- Does current state satisfy routine’s postcondition
  -- for operands args?
```

This enables you to check the precondition before you apply an agent, as in

```
if your_agent.precondition (your_operands) then
  your_agent.call (your_operands)
end
```
where \( \text{your}_\text{agent} \) is an agent expression and \( \text{your}_\text{operands} \) is a valid tuple of operands for that agent.

There is, as will be seen next, a similar facility for class invariants.

### 27.10 THE BASE CLASS AND TYPE

Introspection support is also one of the concerns behind the first generic parameter of \( \text{ROUTINE, PROCEDURE, FUNCTION} \) and \( \text{PREDICATE} \).

The specification

\[
\text{ROUTINE} \left[ \text{BASE, OPEN } \rightarrow \text{TUPLE} \right]
\]

includes, as first generic parameter, the type \( \text{BASE} \) representing the type (class with generic parameters) to which an agent’s feature belongs. This is the type of the target expected by the feature.

The examples seen so far do not use \( \text{BASE} \) at all, because procedure \( \text{call} \) does not need it. If the agent is closed on its target, as in

\[
y := \text{agent} \ a0. \ f(a1, \?, \?)
\]

then it includes, here through \( a0 \), the target information that a later call to \( \text{call} \) may require. In the other case — open target — as in

\[
t := \text{agent} \ \{ T0 \}. \ f(a1, a2, \?)
\]

then the target type is specified, here \( T0 \), and provides the information needed to determine the right version of \( f \). In this case the \( \text{BASE} \) generic parameter is in fact redundant, since it is identical to the first component of the tuple type corresponding to \( \text{OPEN} \); the type of \( t \), for example, is

\[
\text{ROUTINE} \left[ T0, \text{TUPLE} \left[ T0, T3 \right] \right]
\]

where the two tuple components correspond to the two open operands: the target, and the last argument.

In both the closed target and open target cases, then, we don’t need the \( \text{BASE} \) generic parameter if all we do with agents is execute \( \text{call} \) on them.

\( \text{BASE} \) is useful for other purposes. Without \( \text{BASE} \) a call closed on its target, as with \( y \) above, could not contain any information about the class (and associated type) where the call’s associated feature is defined. To open the gate to full introspection services — enabling a system to explore its own properties — class \( \text{ROUTINE} \) uses a feature

\[
\text{base}_\text{type}: \text{TYPE} \left[ \text{BASE} \right]
\]

that yields the type to which the agent’s feature belongs. Class \( \text{TYPE} \left[ G \right] \) from the Kernel Library provides information about a type \( G \) and its base class.

Class \( \text{TYPE} \) is, even more fundamentally than \( \text{ROUTINE} \) and its heirs, the starting place for introspection. Example features include:

- \( \text{name}: \text{STRING} \), the upper name of the type’s base class.
§27.11 AGENT SYNTAX

- **generics**: ARRAY [TYPE [ANY]], the actual generic parameters, if any, used in the type’s derivation.

- **routines**: ARRAY [ROUTINE [ANY, TUPLE]], the routines of a class, each an instance of **PROCEDURE, FUNCTION** or **PREDICATE**.

- **attributes**: ARRAY [FUNCTION [ANY]], the attributes.

- **invariant** (obj: G): BOOLEAN, telling us whether an instance obj satisfies the invariant.

Class **ANY** has a feature

```
generator: TYPE [like Current]
```

which yields an object describing the type of the current object.

So within a class of which f is a feature, generator has the same value as (agent f) • base_type; if a is of type T and f is a feature of T, then a • generator has the same value as (agent [T] • f) • base_type.

A more complete interface specification of **TYPE** appears in the description of the Kernel Library classes.

Thanks to the presence of **BASE** among the generic parameters of **ROUTINE** and its descendants, we can give a proper type to base_type, and as a result gain access to a whole library of introspection mechanisms.

### 27.11 AGENT SYNTAX

The rest of this chapter gives the precise syntax, validity and semantics of agent expressions. There will be no fundamentally new concept, so the hurried reader may skip to the next chapter.

The new construct is **Agent**, a variant of **Expression**:

```
Agents
```

```
Agent ≜ Call_agent | Inline_agent
Call_agent ≜ agent Call_agent_body
Inline_agent ≜ agent [Formal_arguments]
AGENTS [Type_mark]
 AGENTS [Attribute_or_routine]
 AGENTS [Agent_actuals]
```

The two variants are call agents and inline agents. In both cases the keyword **agent** signals the use of an agent expression. **Call_agent** is the keyword **agent** followed by a **Call_agent_body**, similar to a call but with the possibility of using a question mark ? in lieu of an argument or a type in braces **[TYPE]** in lieu of the target. An **Inline_agent** is like an inline routine declaration. Let’s detail both cases in turn.
Syntax of call agents

We have encountered numerous examples of Call_agent, such as

\begin{verbatim}
agent f(a1, a2, a3)
agent f
agent {T0}, f(a1, ?, ?)
\end{verbatim}

A Call_agent starts with the keyword agent. The part that follows, called a Call_agent_body, closely resembles a Call; we can’t just use that earlier construct, however, since we must allow for the question mark and brace conventions, which have no equivalent in normal calls:

\begin{verbatim}
Call_agent_body \triangleq Agent_qualified | Agent_unqualified
Agent_qualified \triangleq Agent_target "." Agent_unqualified
Agent_unqualified \triangleq Feature_name [Agent_actuals]
Agent_target \triangleq Entity | Parenthesized | Manifest_type
Agent_actuals \triangleq "(" Agent_actual_list ")"
Agent_actual_list \triangleq {Agent_actual "", "..."}*
Agent_actual \triangleq Expression | Placeholder
Placeholder \triangleq "?"
\end{verbatim}

--- FIX --- Manifest_type in the syntax of Multi_branch as

An Agent_target may be of three kinds: Entity, Parenthesized and Manifest_type. The third (used in the last example) enables you to specify an open target by listing a type in braces. For an actual argument, you can use, besides an actual value, a Placeholder (question mark).

The possibility of using a Manifest_type or Placeholder to specify open operands is the principal difference between agent calls and normal calls. There is another difference, not immediately obvious from the syntax. In a Call_agent as well as a normal Call, the argument list, Agent_actuals, is optional. But omitting it doesn’t have the same effect. If f is declared as having one or more arguments, a call of the form a, f, or its unqualified variant f, are invalid since they violate the Argument rule: you must always specify actual arguments, as in a0, f(a1, a2, a3). For an agent call, however, corresponding forms such as

\begin{verbatim}
a0,f
f
{T0}, f
\end{verbatim}
are valid; they are simply convenience abbreviations to indicate that all arguments are open, meaning respectively the same as

\[
\begin{align*}
\text{agent } a0, f(? , ? , ?) \\
f(? , ? , ?) \\
\{ T0 \}, f(? , ? , ?)
\end{align*}
\]

You may in such a case omit the argument list, to indicate that all arguments (if any) are open. The Agent Call rule, introduced later in this chapter, explicitly allows this. It causes no ambiguity and (unless you prefer a fully explicit style) lets you avoid cluttering your class text with question marks.

A final note on call agent syntax. You may build a Call_agent not only from identifier features as in these examples, but also from an operator or bracket feature. Just designate the feature by its identifier, in conformance with the Feature Identifier principle: with a feature your_name alias "§" for some binary operator §, use the identifier to build agent expressions such as

\[
\begin{align*}
\text{agent } a \text{ your_name (} b \text{)} & \quad \text{-- All closed} \\
\text{agent } \text{ your_name (}? \text{) } & \quad \text{-- Open on argument} \\
\text{agent } \{ T \} \text{ your_name (}? \text{) } & \quad \text{-- Open on target and argument}
\end{align*}
\]

and similarly with a feature other_name alias "[]".

**Syntax of inline agents**

We have seen that an Inline_agent is like a routine declaration, but given inline, without a name, as in

\[
\begin{align*}
\text{(agent (} e, f \text{: EMPLOYEE): BOOLEAN} \\
& \quad \text{-- Is the cumulated salary of } e \text{ and } f \text{ higher than threshold?} \\
& \quad \text{require} \\
& \quad \text{first_exists: } e \neq \text{ Void} \\
& \quad \text{second_exists: } e \neq \text{ Void} \\
& \quad \text{local} \\
& \quad \text{salary_sum: REAL} \\
& \quad \text{do} \\
& \quad \text{salary_sum := } e \text{. salary } + f \text{. salary} \\
& \quad \text{Result := (salary_sum > threshold)} \\
& \quad \text{end)
\end{align*}
\]

This is reflected by the syntax given on the previous page, which specifies:

- An optional Formal_arguments list, as \((e, f: \text{EMPLOYEE})\).
- An optional Type_mark, as in : BOOLEAN. If this part is present, the associated routine is a function; otherwise it is a procedure.
- A Routine, with all the possible trappings, including Precondition, Local_declarations,
As already noted, it is not recommended to have such extensive computations in inline agents: after all, an Agent is an expression, meant for example to be passed as argument to a routine. But this is just methodological advice; the whole Routine syntax is available if you wish to use it, including the optional Precondition, Local_declarations, Postcondition and Rescue clauses; even Header_comment and Obsolete. The only restriction (stated in the validity constraint given next) is that the routine must not be deferred.

27.12 AGENT VALIDITY

We may now add the validity rules. It is convenient to deal separately with Call_agent and Inline_agent cases.

Validity of call agents

For call agents, it is useful first to define the notion of target type:

<table>
<thead>
<tr>
<th>Target type of an agent call</th>
</tr>
</thead>
<tbody>
<tr>
<td>The target type of a Call_agent is:</td>
</tr>
<tr>
<td>1 • If there is no Agent_target, the current type.</td>
</tr>
<tr>
<td>2 • If there is an Agent_target and it is an Entity or Parenthesized, its type.</td>
</tr>
<tr>
<td>3 • If there is an Agent_target and it is a Manifest_type, the type that it lists (in braces).</td>
</tr>
</tbody>
</table>

The validity rule follows:

<table>
<thead>
<tr>
<th>Call Agent rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Call_agent involving a Feature_name ( fn ), appearing in a class ( C ), with target type ( T_0 ), is valid if and only if it satisfies the following conditions:</td>
</tr>
<tr>
<td>1 • ( fn ) is the name of a feature ( f ) of ( T_0 ).</td>
</tr>
<tr>
<td>2 • If there is an Agent_target, ( f ) is export-valid for ( T_0 ) in ( C ).</td>
</tr>
<tr>
<td>3 • If the Agent_actuals part is present, the number of elements in its Agent_actual_list is equal to the number of formals of ( f ).</td>
</tr>
<tr>
<td>4 • Any Agent_actual of the Actual kind is of a type conforming to the type of the corresponding formal in ( f ).</td>
</tr>
</tbody>
</table>

--- REMOVED CONDITION ON SEPARATE --- The rule’s phrasing makes certain forms of the construct automatically valid:

- If any Agent_actual is of the Placeholder kind, represented simply by a question mark, clause 4 does not apply, so the argument raises no type validity problem. This is as expected, since such an argument is left open for future filling-in.
If there is no `Agent_actuals` part, clauses 3 and 4 do not apply. If \( f \) has no formals, we are calling an argumentless feature with no actuals, as we should. If \( f \) has one or more formal arguments, we view the absence of explicit actuals of an abbreviation for actuals that are all of the `Placeholder` kind (question marks): assuming \( f \) takes three arguments, `agent a0, f` is simply an abbreviation for `agent a0, f(?, ?, ?)`. In this case the implicit arguments are all open, and hence automatically valid.

Clause 3 differs from its counterpart for normal calls, which required actual argument list to match the formal list if any. Instead we explicitly allow omitting actuals altogether, to signify that all arguments are open.

--- Clause --- is a consistency condition for concurrent computation, and parallels a similar clause discussed in the chapter on normal calls.

**Validity of inline agents**

To define the validity of inline agents (also their semantics), it is convenient to consider this case as equivalent to the previous one, `Call_agent`, by treating any inline agent as equivalent to `agent f(...)` where \( f \) is a fictitious routine added to the class. Here is the definition of this equivalence:

**Definition**

**Associated feature of an inline agent**

Every inline agent \( ia \) of a class \( C \) has an associated feature, defined as a fictitious routine of \( C \), such that:

1. The name of \( f \) is chosen not to conflict with any other feature name in \( C \) and its descendants.
2. The formal arguments of \( f \) are those of \( ia \).
3. \( f \) is secret (available for call to no class).
4. The Routine of \( f \) is defined by the Routine part of \( ia \).
5. \( f \) is a function if \( ia \) has a Type_mark (its return type being given by the Type in that Type_mark), a procedure otherwise.

Clause 2 lists, as arguments to \( f \), not only the arguments to the inline agent but also the local variables of the enclosing routine. The local variables will indeed serve as closed arguments; this will be specified in the semantics given in the next section.

The validity rule follows:

**Inline Agent rule**

An Inline_agent \( a \) of associated feature \( f \), is valid in the text of a class \( C \) if and only if it satisfies the following conditions:

1. \( f \), if added to \( C \), would be valid.
2. \( f \) is not deferred.
There is no other condition, since in particular the Routine part must be valid on its own; in particular, the Entity rule states that any entity appearing in the Agent_body must be a formal argument of the inline agent itself, such as other and i in

\[
\begin{align*}
(\text{other}, \text{like Current}; i: \text{INTEGER}) \\
\text{do Result} := (\text{item}(i) = \text{other}.\text{item}(i)) \text{ end}
\end{align*}
\]

or a local variable of an enclosing agent or routine, or a feature of the enclosing class.

Here are some properties following from the Inline Agent rule:

\[
\text{Inline Agent Requirements}
\]

An Inline_agent \(a\) must satisfy the following conditions:

1. No formal argument or local variable of \(a\) has the same name as a feature of the enclosing class.
2. Every entity appearing in the Routine part of \(a\) is the name of one of: a formal argument of \(a\); a local variable of \(a\); a feature of the enclosing class; Current.
3. The Feature_body of \(a\)'s Routine is not of the Deferred form.

These conditions are stated as another validity rule permitting compilers to issue more understandable error messages. It is not in the usual “if and only if” form (since the preceding rule, the more official one, takes care of this), but the requirements given cover the most obvious possible errors.

27.13 AGENT SEMANTICS

(Like the previous two, this section may be skipped on first reading.) The final part of the specification addresses the semantics of agents. It is organized in three parts:

- Call-agent equivalent of an inline agent (enabling the next two parts to restrict themselves to the Call_agent part).
- Open and closed operands.
- Type and value of an agent.

Call-agent equivalent of an inline agent

To define the validity of an inline agent \(a\), it was convenient to define its associated feature. Then \(a\) itself can be viewed as if it were a Call_agent:
This allows a simple specification for the semantics of inline agents:

**Call-agent equivalent of an inline agent**

An inline agent $ia$ with $n$ formal arguments ($n \geq 0$) has a call-agent equivalent defined as the Call_agent

$$\text{Call_agent} f (\ldots, a_1, a_2, \ldots, a_m)$$

using $n$ question marks, where $a_1, a_2, \ldots, a_m$ ($m \geq 0$) are the formal arguments and local variables of the enclosing routine (if any) and any enclosing agents, and $f$ is the associated feature of $ia$. (If both $n$ and $m$ are 0, the Call_agent is just $\text{agent } f$.)

Thanks to this rule, we can focus on call agents when defining the type and execution effect of agents.

Note how the formal arguments and local variables of the enclosing routine if any, and of any enclosing agents, serve as closed arguments to the agent. In reading earlier discussions of inline agents, you may have pondered two as yet unanswered questions:

- Is it permitted for an inline agent to refer to a local variable of the enclosing routine, and, if so, what does that mean?
- Call agents may have both closed and open operands. We have seen how to give an inline agent open operands: just specify them as arguments to the agent. But is there a way to give it closed operands too?

The rule just given answers both questions at once by giving a status to local variables of the enclosing routine: treat them as closed operands.

So a routine of the form

```
local n: INTEGER
do
  n := 1
  your_list, do_all ((i: INTEGER) do print (i + n) end)
end
```
will print the successive values in your_list (assumed to be of type LIST [INTEGER]) all incremented by 1, the value of the local variable n at construction time. As specified by the last rule, this is the same effect as if the call were

```
do_all (agent print_incremented_value (?, n))
```

where print_incremented_value is the “fictitious routine” introduced by the definition of “associated feature” of an inline agent:

```
print_incremented_value (i: INTEGER; n: INTEGER)
  do
    print (i + n)
  end
```

In examining the above definition of call-agent equivalent, note that the validity rule on inline agents guarantees that there can be no name clash between the formal arguments and local variables of any enclosing agents and of the enclosing routine if any. (Nesting inline agents doesn’t seem a desirable use of the mechanism, but no rule disallows it.)

The semantics of inline agents also requires a specific rule on the meaning of Result. An inline agent may be embedded in a function of the class, or even in another function agent, causing a potential ambiguity. We decide that Result always refers to the result of the innermost agent:

```
Use of Result in an inline function agent
```

In an agent of the Inline_routine form denoting a function, the local variable Result denotes the result of the agent itself.

This is a rather specific case and another approach would be to disallow function agents within functions or other function agents, or to use a special notation to remove the ambiguity. The rule as given seems preferable. If you need to refer to an outer Result, you may assign its value to a local variable and use that local variable in the innermost agent scope. This causes a little extra work, but only in a rare and special case.

**Open and closed operands**

It is useful to define precisely what “open” and “closed” mean for the operands of an agent expression:

From the definition of call-agent equivalent form we deduce that for an inline agent:

- The open operands are the agent’s formal arguments, if any.
Open and closed operands

The open operands of a Call_agent include:

1. Any Agent_actual that is a Placeholder.
2. The Agent_target if it is present and is a Manifest_type.

The closed operands include all non-open operands.

The closed operands are the local variables and formal arguments of the enclosing routine and any enclosing agents.

An earlier definition also introduced the notion of operand position, which we can now extend to a definition of open and closed positions:

Open and closed operand positions

The open operand positions of a Feature_agent are the operand positions of its open operands, and the closed operand positions those of its closed operands.

Type and value of an agent expression

The preceding definitions enable us to specify the semantics of an agent expression. It suffices to give it for a Call_agent:

Type of an agent expression

Consider a Call_agent \(a\), whose associated feature \(f\) has a generating type \(T_0\). Let \(i_1, ..., i_m (m \geq 0)\) be its open operand positions, if any, and let \(T_{i_1}, ..., T_{i_m}\) be the types of \(f\)’s formals at positions \(i_1, ..., i_m\) (taking \(T_{i_1}\) to be \(T_0\) if \(i_1 = 0\)).

The type of \(d\) is:

- **PROCEDURE** \([T_0, \text{TUPLE}[T_{i_1}, ..., T_{i_m}]]\) if \(f\) is a procedure;
- **FUNCTION** \([T_0, \text{TUPLE}[T_{i_1}, ..., T_{i_m}], R]\) if \(f\) is a function of result type \(R\) other than **BOOLEAN**;
- **PREDICATE** \([T_0, \text{TUPLE}[T_{i_1}, ..., T_{i_m}]]\) if \(f\) is a function of result type **BOOLEAN**.

→ The operands of a call were defined on page 715 as including its target, and its arguments if any.

→ “Operand position” was defined on page 715: the target position is 0, and the argument positions start at 1.
Although this will be an implicit consequence of the last rule, it doesn’t hurt to state explicitly what some of the information in D0 is good for: enabling calls on agent objects.

Agent Expression semantics

The value of an agent expression \( a \) at a certain construction time yields a reference to an instance D0 of the type of \( a \), containing information identifying:

- The associated feature of \( a \).
- Its open operand positions.
- The values of its closed operands at the time of evaluation.

Effect of executing call on an agent

Let D0 be an agent object with associated feature \( f \) and open positions \( i1, \ldots, im \) \((m \geq 0)\). The information in D0 enables a call to the procedure call, executed at any call time posterior to D0’s construction time, with target D0 and (if required) actual arguments \( a_{i1}, \ldots, a_{im} \), to perform the following:

- Produce the same effect as a call to \( f \), using the closed operands at the closed operand positions and \( a_{i1}, \ldots, a_{im} \), evaluated at call time, at the open operand positions.
- In addition, if \( f \) is a function, setting the value of the query \texttt{last_result} for D0 to the result returned by such a call.