3

An Imperative Sequential Calculus

The ASP calculus starts from an imperative sequential object calculus *à la* Abadi and Cardelli. Only a few characteristics have been changed between the original **imp**ς-calculus and ASP sequential calculus.

- Because arguments passed to an active object method will play a particular role, a parameter is added to every method as in [123]: in addition to the "self" argument of the methods (noted x_j and representing the object on which the method is invoked – self), an argument representing a parameter object can be sent to the method (y_j in the syntax below).
- Method update is not included in the ASP calculus because we do not find it necessary, and it is possible to express updatable methods in ASP calculus anyway (e.g., updatable fields containing lambda expressions). Moreover, adding updatable methods would not raise any theoretical problems.[1] Section 10.3 (page 141) will further discuss the expression of method update in ASP.
- As in [79], during the reduction, *locations* (reference to objects in a store) can be part of terms in order to simplify the semantics. The locations should not appear in source terms.

3.1 Syntax

The abstract syntax of the ASP calculus is defined in Table 3.1:

- l_i are field names,
- m_j are method names,
- ς is a binder for method parameters,
- a location ι is an entry in the store defined below,

[1] But, in the parallel case, updating methods would unnaturally modify the meaning of requests that have already been sent but are not executed yet.

- in the following, l_i, $i \in 1..n$, range over fields names,[2] and m_j, $j \in 1..m$, over method names. In practice, there is one integer n and one integer m for each object, but as there is no ambiguity we will simply denote all these numbers by n and m.

$a, b \in L ::= x$	variable
$\mid [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]_{j\in 1..m}^{i\in 1..n}$	object definition
$\mid a.l_i$	field access
$\mid a.l_i := b$	field update
$\mid a.m_j(b)$	method call
$\mid clone(a)$	superficial copy
$\mid \iota$	location (not in source terms)

Table 3.1. Syntax of ASP sequential calculus

As an example, a point object could be defined in the following way:

$$Point \triangleq [x = 0,\ y = 0,\ color = [R = 0, G = 0, B = 0; print = \ldots];\\ getX = \varsigma(s,p)s.x,\ setX = \varsigma(s,p)s.x := p, getColor = \varsigma(s,p)s.color, \ldots]$$

An object without fields will be denoted by $[; m = \varsigma(z, x)\ldots]$.

let $x = a$ *in* b and sequence $a; b$ can be easily expressed in this sequential calculus and will be used in the following:

$$let\ x = a\ in\ b \triangleq [; m = \varsigma(z, x)b].m(a)$$

$$a; b \triangleq [; m = \varsigma(z, z')b].m(a)$$

Lambda expressions can be encoded as follows (strongly inspired by [3]):

$$\lambda x.b \triangleq [arg = [], val = \varsigma(x, y)b\{\!\{x \leftarrow x.arg\}\!\}]\\ (b\ a) \triangleq (clone(b).arg := a).val([])$$

A simple way of expressing (mutually) recursive *let* in the case where a and b are objects (e.g., lambda abstractions) is defined below:

$$\begin{aligned} let\ x = a\ and\ y = b\ in\ c \triangleq\ & let\ o = [x = [], y = []]\ in\\ & let\ x = a\{\!\{x \leftarrow o.x, y \leftarrow o.y\}\!\}\ in\\ & let\ y = b\{\!\{x \leftarrow o.x, y \leftarrow o.y\}\!\}\ in\\ & o.x := x; o.y := y; c \end{aligned}$$

This solution only works if mutual references are only accessed by methods of a and b, and a and b are not active,[3] but this simplified expression will be

[2] $i \in 1..n$ classically represents $i \in \mathbb{N} \cap [1, n]$.

[3] Another encoding will be provided for mutually dependent active objects in Chap. 5.

sufficient for the sequential terms used in the rest of this book. Note that this expression also allows to modify the value stored in x: $x := a'$.

The general definition of (mutually) recursive *let* is much more complex (see for example [36] for a typing analysis of recursive objects).

For example, we will neither use nor try to give a semantics to the following term:

$$let\ x\ =\ [f = y]\ \ and\ y = [g = x]\ \ in\ \ \ldots$$

Moreover, methods with zero and more than one argument are also easy to encode in the sequential calculus and will also be used in this study. For example, $foo = \varsigma(s)\ldots$ and $o.foo()$ will respectively denote the definition and the invocation of a method foo with zero argument.

Finally, we also use in the following integers and boolean and their associated operations. For example we will use $if\ldots then\ldots else\ldots$ statements, integer comparisons ($<$, $>$, ...) and classical operations on integers and booleans ($+$, not, ...). Integers and booleans can be either added to the sequential calculus or encoded into the sequential calculus (e.g., Church integers).

3.2 Semantic Structures

Let $locs(a)$ be the set of locations occurring in a and $fv(a)$ the set of variables occurring free in a. The *source terms* (initial expressions) are *closed* terms ($fv(a) = \emptyset$) without any location ($locs(a) = \emptyset$); such terms are also called *static terms*. Locations appear when objects are put in the store.

3.2.1 Substitution

The substitution of b by c in a is written: $a\{\!\{b \leftarrow c\}\!\}$. Substitutions are denoted by $\theta ::= \{\!\{b \leftarrow c\}\!\}$. Multiple substitutions are applied from left to right: $a\theta\theta' = (a\theta)\theta'$.

In method calls (INVOKE), substitution is applied in a classical way on bounded variables: formal parameter x is replaced by the location of the argument without replacing inside binders $\varsigma(x, z)$ or $\varsigma(z, x)$.

An *injective* substitution of some locations by other locations that do not appear in the involved term will also be called a *renaming*. A renaming is in fact an alphaconversion of locations. Renamings will be useful to define equivalence relations between terms.

3.2.2 Store

Reduced objects are objects with all fields reduced to a location:

$$o ::= [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]_{j\in 1..m}^{i\in 1..n}$$

A *store* σ is a finite map from locations to reduced objects:

$$\sigma ::= \{\iota_i \mapsto o_i\}$$

The domain of σ, $dom(\sigma)$, is the set of locations defined by σ.

Let $\sigma :: \sigma'$ append two stores with disjoint locations (*store append*). When the domains are not disjoint, $\sigma + \sigma'$ updates the values defined in σ' by those defined in σ (*store update*). It is defined on $dom(\sigma) \cup dom(\sigma')$ by:

$$(\sigma + \sigma')(\iota) = \begin{array}{l} \sigma(\iota) \text{ if } \iota \in dom(\sigma) \\ \sigma'(\iota) \text{ otherwise} \end{array}$$

The operator + will be used, for example, to update the value associated to a location.

Note that $\sigma :: \sigma'$ is equal to $\sigma + \sigma'$ but specifies that $dom(\sigma) \cap dom(\sigma') = \emptyset$.

3.2.3 Configuration

Let a *configuration* (a, σ) be a pair (expression, store): the store σ associates values to some locations that can appear in a. $\vdash (a, \sigma)$ OK denotes a *well-formed configuration* (no free variable and σ defines every useful location):

Definition 3.1 (Well-formed sequential configuration)

$$\vdash (a, \sigma) \text{ OK} \Leftrightarrow \begin{cases} locs(a) \subseteq dom(\sigma) \wedge fv(a) = \emptyset \\ \forall \iota \in dom(\sigma),\ locs(\sigma(\iota)) \subseteq dom(\sigma) \ \wedge \ fv(\sigma(\iota)) = \emptyset \end{cases}$$

Let $\equiv$ be the equality between configurations modulo renaming of locations:

Definition 3.2 (Equivalence on sequential configurations)

$$(a, \sigma) \equiv (a', \sigma') \Leftrightarrow \exists \theta,\ (a\theta, \sigma\theta) = (a', \sigma')$$

3.3 Reduction

Table 3.2 defines a small-step, substitution-based operational semantics ($\rightarrow_S$) for the sequential calculus. It gives reduction rules for:

- object creation (STOREALLOC),
- field access (FIELD),
- method invocation (INVOKE),

- field update (UPDATE),
- and shallow clone (CLONE).

This semantics is very close to the one defined in [80]. Table 3.2 applies one rule on the point of reduction represented by the unique occurrence of $\bullet$ in the following *reduction contexts*:

$$\begin{aligned}\mathcal{R} ::= \bullet \mid [l_i = \iota_i, l_k = \mathcal{R}, l_{k'} = b_{k'}; m_j = \varsigma(x_j, y_j)a_j]_{j\in 1..m}^{i\in 1..k-1, k'\in k+1..n} \\ \mid \mathcal{R}.m_i \mid \mathcal{R}.m_j(b) \mid \iota.m_j(\mathcal{R}) \mid \mathcal{R}.l_i := b \mid \iota.l := \mathcal{R} \mid clone(\mathcal{R})\end{aligned}$$

The reduction contexts allow us to specify the order of reductions inside the sequential terms. For example, to evaluate a field update, the object to be updated is first evaluated, then the new value of the field is also evaluated, and finally the field update is performed.

$\mathcal{R}[a]$ denotes the substitution inside a reduction context, that is to say $\mathcal{R}[a]$ is the term obtained by replacing the only hole of $\mathcal{R}$ by the sub-term a:

$$\mathcal{R}[a] = \mathcal{R}\{\!\{\bullet \leftarrow a\}\!\}$$

In $\mathcal{R}[a]$, a is the sub-term that has to be reduced by the next elementary sequential reduction, and $\mathcal{R}$ is the rest of the term (the context) that is neither useful nor modified by the next elementary reduction.

STOREALLOC:

$$\frac{\iota \notin dom(\sigma)}{(\mathcal{R}[o], \sigma) \rightarrow_S (\mathcal{R}[\iota], \{\iota \mapsto o\} :: \sigma)}$$

FIELD:

$$\frac{\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]_{j\in 1..m}^{i\in 1..n} \quad k \in 1..n}{(\mathcal{R}[\iota.l_k], \sigma) \rightarrow_S (\mathcal{R}[\iota_k], \sigma)}$$

INVOKE:

$$\frac{\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]_{j\in 1..m}^{i\in 1..n} \quad k \in 1..m}{(\mathcal{R}[\iota.m_k(\iota')], \sigma) \rightarrow_S (\mathcal{R}[a_k\{\!\{x_k \leftarrow \iota, y_k \leftarrow \iota'\}\!\}], \sigma)}$$

UPDATE:

$$\frac{\begin{array}{c}\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]_{j\in 1..m}^{i\in 1..n} \quad k \in 1..n \\ o' = [l_i = \iota_i, l_k = \iota', l_{k'} = \iota_{k'}; m_j = \varsigma(x_j, y_j)a_j]_{j\in 1..m}^{i\in 1..k-1, k'\in k+1..n}\end{array}}{(\mathcal{R}[\iota.l_k := \iota'], \sigma) \rightarrow_S (\mathcal{R}[\iota], \{\iota \rightarrow o'\} + \sigma)}$$

CLONE:

$$\frac{\iota' \notin dom(\sigma)}{(\mathcal{R}[clone(\iota)], \sigma) \rightarrow_S (\mathcal{R}[\iota'], \{\iota' \mapsto \sigma(\iota)\} :: \sigma)}$$

Table 3.2. Sequential reduction

3.4 Properties

Initial Configuration

To evaluate a source term a, we create an *initial configuration* $(a, \emptyset)$ containing this term and an empty store. Then, this configuration can be evaluated following the reduction $\rightarrow_S$.

Well-formedness

As a first correctness property, it is easy to show that reduction preserves well-formedness.

Property 3.3 (Well-formed sequential reduction)

$$\vdash (a, \sigma)\ \text{OK} \wedge (a, \sigma) \rightarrow_S (b, \sigma') \implies \vdash (b, \sigma')\ \text{OK}$$

This property is proved by case analysis on the applied sequential rule and checking that every referenced location exists in the store. It is necessary to ensure that every accessed object exists in the store; for example, when evaluating $\iota.l_0$, Property 3.3 ensures that the accessed object referenced by ι exists in the store.

Sequential Determinism

A first result toward determinism is to ensure that a sequential reduction is deterministic. Indeed, the reduction contexts fully specify the order of reduction. Consequently, a sequential reduction is deterministic up to the choice of freshly allocated locations:

Property 3.4 (Determinism)

$$c \rightarrow_S d \wedge c \rightarrow_S d' \Rightarrow d \equiv d'$$

In fact, at most one reduction can be made on each configuration. The only choice is the name of locations created by the rules STOREALLOC and CLONE.

4

Asynchronous Sequential Processes

We introduce here the ASP calculus which is based on *activities*, each one including a single *active object*. After providing the main principles of ASP, we present two new syntactic constructs: *Active* and *Serve*. We conclude this chapter with a detailed informal semantics.

4.1 Principles

Each ASP object is either *active* or *passive*. There is one active object at the root of each activity. Activities execute instructions concurrently (potentially in parallel), and interact only through method calls. Method calls toward active objects are always *asynchronous*. Synchronization is due to *wait-by-necessity* on the result of an asynchronous method call (data-driven synchronization).

An *activity* is a single process (a single execution thread) associated with a set of objects put in a store. Among them one is *active* and every *request* (method call) sent to the activity is actually sent to this object. The activity serves/executes each of the received requests one after another. As such, an activity also contains the pending requests (requests that have been received and are still pending) and the responses to the requests for which the execution is finished (values of the results). *Passive* (non-active) objects are only referenced by objects belonging to the same activity, but any object can reference active objects: ASP activities do not share memory. Figure 4.1 shows an example of an objects topology in a configuration containing four activities.

The principles of asynchronous method calls is the following: when an object sends a request to an activity, it is stored in a request queue and a *future* is associated to this request. Such a request is called *pending*. Later on this request will be *served* (i.e., taken into the request queue in order to be evaluated); it becomes a *current request* . When the service is finished, a value is associated to the result of this request and the association between the future corresponding to this request and the calculated value is stored in

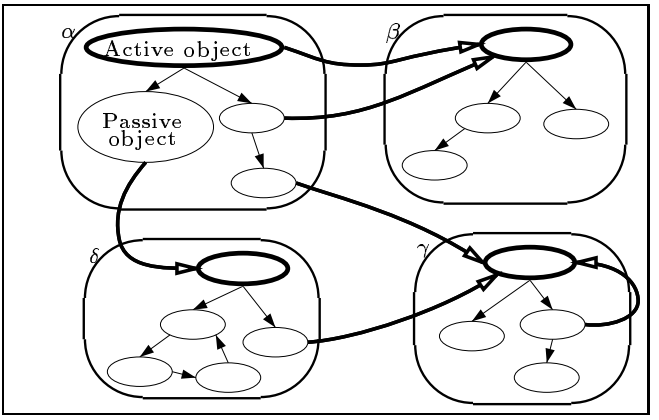

Fig. 4.1. Objects and activities topology

a *future values list*. Such requests are called *served requests*. Afterward, the distant reference to the future may be *updated* by the calculated value. A *future* represents the result of a method call to an active object that has not yet been returned.

The activation of a given object a ($Active(a, m)$) creates a new activity whose active object is a copy of a. If a method m is provided, it specifies the active object activity (a sort of main), else, by default, when $m = \emptyset$ the object activity is a FIFO service of requests. $Serve(M)$ performs a blocking service of requests targeted at a method belonging to M and received by the current active object.

For example, with the point object defined in Sect. 3.1, $Point.getColor()$ will perform a classical method call with synchronous semantics. In the term $let\ p = Active(Point, \emptyset)\ in\ let\ col = p.getColor()\ in\ p.setX(2); col.print()$, every method call to object p will be asynchronous. $p.setX(2)$ will trigger the execution of the method in the activity of p and continues the local evaluation in parallel. Execution will be blocked when one tries to perform a strict operation on the result of an asynchronous method before the end of its execution. Such blocking states are called *wait-by-necessity*. In the preceding example, $col.print()$ is a strict operation on the object col ; if the result of *getcolor* has not been returned by the active object p, then the local activity is stuck until this result is replied.

Unlike many other concurrent calculi based on ς-calculus (e.g., Obliq [45]), in ASP, the requests are not executed by the process that performs the method call, but by the processes associated to the destination of the request.

4.2 New Syntax

In Table 4.1, we extend the sequential calculus by adding the possibility to create an active object and to serve a request.

$a, b \in L ::= \ldots$		
	$\mid Active(a, m_j)$	Activates object: deep copy + activity creation, m_j is the activity method or $\emptyset$ for FIFO service
	$\mid Serve(M)$	Serves a request among a set of method labels,
	$\mid a \Uparrow f, b$	a with continuation b (not in source terms)

where M is a set of method labels used to specify the request that has to be served:

$$M = m_{k_1}, \ldots, m_{k_h}$$

Table 4.1. Syntax of ASP parallel primitives

A parallel configuration is a set of activities; each activity contains several fields that will be introduced informally in Sect. 4.3 just below, and formally defined in Chap. 6:

$$P, Q ::= \alpha[a_\alpha; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[\ldots] \parallel \ldots$$

To summarize, the whole syntax of the ASP source terms is given in Table 4.2.

$a, b \in L ::= x$		variable,
	$\mid [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]_{j \in 1..m}^{i \in 1..n}$	object definition
	$\mid a.l_i$	field access
	$\mid a.l_i := b$	field update
	$\mid a.m_j(b)$	method call
	$\mid clone(a)$	superficial copy
	$\mid Active(a, m_j)$	activity creation
	$\mid Serve(m_{k_j})^{j \in 1..h}$	service primitive

Table 4.2. Syntax of ASP calculus

4.3 Informal Semantics

In every activity α, a *current term* a_α represents the current computation. Every activity has its own *store* σ_α which contains one active and many passive objects. It also contains *pending requests* which store the pending method calls, and a *future list* which stores the result of finished requests.

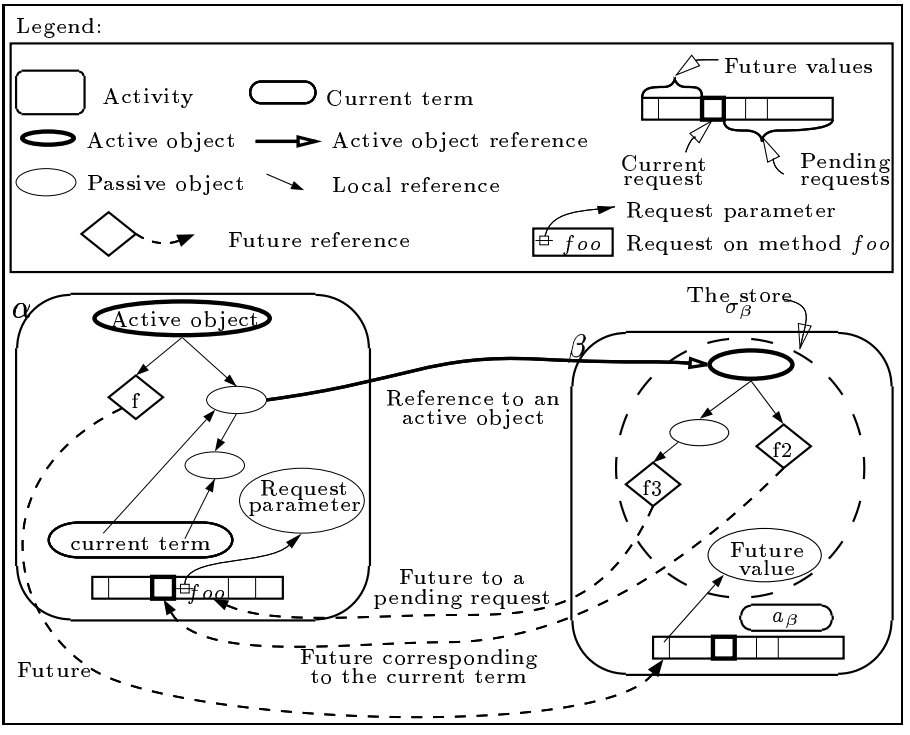

Fig. 4.2. Example of a parallel configuration

Figure 4.2 shows a representation of a configuration consisting of two activities (α and β). It contains three references to futures: one calculated (f), one current (f_2), and one pending (f_3). The active objects are bold ellipses; the futures references are diamonds; the futures values, the current future, and the pending requests are merged in the bottom rectangles: calculated future values are on the left, the current future is represented by a bold rectangle, and pending requests are on the right. The continuation does not appear in the diagrams.

4.3.1 Activities

The *Active* operator ($Active(a, m_j)$) creates a new activity α with the object a at its root. The object a is copied as well as all its dependencies (deep copy) in the new activity, to prevent distant references to passive objects. The second argument of the *Active* operator is the name of a method which will be called as soon as the object is activated. This method is called the *service method* as it usually specifies the order of requests that the activity should serve. If no service method is specified, a *FIFO* service will be performed: the requests will be served in the order they arrived in the activity. Note that in Fig. 4.2, in

case of a FIFO service, the current request (bold square) progresses from left to right in the queue. When the service method terminates, no more requests are treated, and there is no more intrinsic activity. The activity conceptually ends.

The remote references to the active object of activity α will be denoted by $AO(\alpha)$. $AO(\alpha)$ acts as a proxy for the active object of activity α.

4.3.2 Requests

The communications between activities are due to method calls on active objects and returns of corresponding results. A method call on an active object ($Active(o, \emptyset).foo()$) consists in atomically adding an entry to the *pending requests* of the callee, and associating a *future* (an identifier representing the result of the request) to the response. From a practical point of view, this atomicity is guaranteed by a *rendezvous* mechanism (the request sender waits for an acknowledgment before continuing its execution). The arguments of requests and the values of futures are deeply copied when they are transmitted between activities; this prevents sharing, i.e., distant reference to passive objects. Active objects are transmitted with a reference semantics.

4.3.3 Futures

A *future* is a unique identifier representing the reply to a request between the moment when the request is sent and put in a queue, and the moment when the result of the request is sent and updated. A *future update* consists in replacing the reference to the future by a copy of the future value. Of course, a deep copy occurs, again to prevent sharing.

Futures are generalized references that can be manipulated classically while no strict operation is performed on the object they represent. In Fig. 4.2, the futures f_2 and f_3 denote pointers to not yet computed requests while f is a future pointing to a value already computed by the activity β.

An operation on an object is *strict* if it needs to access the content of the object: the only strict operations are field and method access, field update, and clone. For example, transmitting an object as a method parameter is not a strict operation, including in a request.

A wait-by-necessity occurs when one tries to perform a strict operation on a future. This wait-by-necessity can only be released by *updating* the future.

The fact that futures are first-class entities is sometimes called "automatic continuation." We avoid using this terminology in this book as it may be confusing.

4.3.4 Serving Requests

The primitive *Serve* can appear at any point in source code. Its execution stops the current activity until a matching request is found in the pending

requests. A *matching request* is a request on one of the method labels specified as parameter of the *Serve* primitive. For semantics specification reasons, we introduced the operator $\Uparrow$ which allows us to save the continuation of the request we are currently serving while we serve another one.

Note that with such a mechanism there can be several requests being served at the same time. More precisely, an activity always serves at most one request if *Serve* operations are only performed by the activity method or the method recursively called by this method, because in that case, no $Serve(M)$ is performed while a request is being served.

When the execution of a request is finished, the corresponding future is associated to the newly calculated value (*future value*). Then, the execution continues by restoring the stored continuation. The term that had served the finished request continues its execution (it becomes the current term). The *future list* maps futures to their values within the activities that computed them. As a future value can contain references to other futures, a future value is called *partial* if its dependencies contain futures references.

Note that a field access on an active object is forbidden (it would nearly always be non-deterministic) and an activity trying to access a field of an active object is irreversibly stuck (like an access to a non-existing field).

However, one can syntactically transform a field access (field update) into a call to a getter method (setter method). Provided such a call is treated as a normal request, inserted in the pending queue and normally served, this allows one to deal adequately with remote fields.

Springer

http://www.springer.com/978-3-540-20866-2

A Theory of Distributed Objects
Asynchrony - Mobility - Groups - Components
Caromel, D.; Henrio, L.
2005, XXXII, 352 p., Hardcover
ISBN: 978-3-540-20866-2