Alma Mater Studiorum · Università di Bologna

**SCUOLA DI SCIENZE**
**Corso di Laurea in Informatica**

# A STUDY OF TWO LANGUAGES FOR ACTIVE OBJECTS WITH FUTURES

**Tesi di Laurea in Analisi Statica di Programmi**

**Relatore:**
**Chiar.mo Prof.**
**COSIMO LANEVE**

**Correlatore:**
**Dott.ssa**
**ELENA GIACHINO**

**Presentata da:**
**MARTIN KLAPEŽ**

**Sessione 1**
**Anno Accademico 2012/2013**

# Contents

**Introduzione** 4

**Introduction** 6

**1 A Informal Introduction to the Deadlock Analysis Model** 8

**2 The coreABS$^{--}$ language** 12
- 2.1 Generalities . . . 12
- 2.2 Syntax . . . 14
- 2.3 Semantics . . . 15
- 2.4 Type System and Contracts . . . 18
- 2.5 A Deadlock Example in coreABS$^{--}$ . . . 23

**3 The ASP Calculus** 25
- 3.1 Generalities . . . 25
- 3.2 Syntax . . . 28
- 3.3 Semantics . . . 29
- 3.4 A Deadlock Example in ASP . . . 37

**4 Different ways to approach the analysis** 39
- 4.1 A Type System for ASP . . . 39
- 4.2 A Different coreABS$^{--}$ . . . 47

**Conclusions** 50

**References** 52

# Introduzione

Questo lavoro di tesi é sviluppato interamente attorno allo studio di *deadlock*; informalmente, un *deadlock* é una situazione in cui l'attore X si ritrova indefinitamente bloccato perché in attesa di una risorsa mantenuta dall'attore Y, il quale si ritrova a sua volta indefinitamente bloccato perché in attesa di una risorsa mantenuta dall'attore X.
Di conseguenza, siccome entrambi gli attori sono completamente bloccati, nessuno di essi sará mai in grado di rilasciare il controllo della risorsa di cui l'altro attore necessita per proseguire la propria computazione.

Come al lettore potrebbe essere balenato in mente, un deadlock puó essere inteso come essere analogo al famoso problema di determinare se "é nato prima l'uovo o la gallina", nel senso che diviene impossibile raggiungere l'esito desiderato - la gallina - finché una necessaria precondizione non viene soddisfatta - l'uovo -, mentre perché si verifichi tale precondizione - l'uovo - é obbligatorio che l'esito desiderato - la gallina - si sia giá verificato.

É importante chiarire che, in ambito informatico, un deadlock puó facilmente comprendere piú di due attori contemporaneamente. Infatti, un deadlock puó insorgere da una situazione in cui un certo numero di *processi* o *thread* entra in uno stato di attesa indefinita, ognuno dei quali aspetta il rilascio di una risorsa mantenuta da qualcun'altro, formando in questo modo una pericolosa impasse.
Come metafora potenzialmente chiarificante, una situazione di deadlock che coinvolge esattamente tre attori puó essere vista come molto simile a uno stallo alla messicana.

I processi potenzialmente interessati da questo fenomeno sono processi che vengono detti *concorrenti*, ossia in competizione tra loro in quanto ognuno di essi usa o potrebbe usare in futuro una o piú risorse appartenenti ad un insieme ad essi comune.

Quando, per prendere possesso di una certa risorsa, almeno due processi concorrenti si ritrovano ad attendere la terminazione dell'altro - e quindi nessuno potrá mai terminare - questi processi vengono detti *in stato di deadlock* o in *deadlock*.
Piú nello specifico, un deadlock avviene quando la terminazione di ciascun processo é *subordinata* al possesso di una risorsa mantenuta da un altro.

Nei sistemi informatici, specificamente nei sistemi multiprocesso, paralleli e distribuiti, un deadlock é un problema sia subdolo - perché difficile da prevenire durante la scrittura del codice di sistema - sia potenzialmente molto pericoloso: un sistema in deadlock é con facilitá completamente bloccato, con conseguenze che variano da semplici scocciature a circostanze che mettono in pericolo vite umane, senza dimenticare la non trascurabile via di mezzo di perdite economiche anche ingenti.

A questo punto é sicuramente sorta spontanea la domanda di come sia possibile risolvere questo problema. Molte possibili soluzioni sono state studiate, proposte e implementate. In questa tesi si concentra l'attenzione sull'individuazione preventiva di deadlock mediante una tecnica statica di analisi di programmi, cioé un'analisi effettuata senza mandare in esecuzione il programma in questione.

Per cominciare, nel primo capitolo viene brevemente presentato il Modello per l'Analisi di Deadlock sviluppato a partire da coreABS$^{--}$, dopodiché si procede la trattazione dettagliando il suddetto linguaggio *Class-based* coreABS$^{--}$ nel capitolo 2.

Il capitolo 3 ha il fine di posare le fondamenta per ulteriori approfondimenti sul tema mediante un'analisi delle differenze tra coreABS$^{--}$ e ASP, un calcolo *Object-based* non tipato, in modo da mostrare come puó essere possibile estendere la suddetta Analisi di Deadlock ai linguaggi *Object-based* in generale.

Vengono poi esplicitate alcune ipotesi a riguardo nel capitolo 4, innanzitutto presentando un possibile, non dimostrato, sistema di tipi per ASP coerente con il Modello per l'Analisi di Deadlock sviluppato per coreABS$^{--}$.
Si conclude poi la presente discussione presentando un'ipotesi piú semplice che mantiene lo stesso fine della precedente e che si pone peró nel contempo l'obiettivo di evitare le difficoltá che inesorabilmente sorgono dalla definizione del sistema di tipi "ad-hoc" discusso precedentemente.

# Introduction

This thesis is entirely built around the notion of *deadlock*; informally, a situation in which the actor X is freezed forever waiting for a resource held by actor Y, which in turn is freezed forever waiting for a resource held by actor X.
Therefore, being both actors completely freezed, none of them will ever be able to release the resource needed by the other in order to continue its computation.

As the reader may have thought, a deadlock can be seen as being analogue to the famous "chicken-egg question", in the sense that it becomes impossible to reach a certain desired outcome - the chicken - because a necessary precondition is not satisfied - the egg -, while to meet that precondition - the egg - in turn requires that the desired outcome - the chicken - has already been realized.

It's worth to say that, in computing, a deadlock may easily involve more than two actors. Infact, a deadlock may arise from a situation in which a number of *processes* or *threads* enters a undefined waiting state, each one waiting to possess a resource held by another, thereby forming a dangerous impasse.
As a potentially clarificant metaphor, a deadlock situation involving exactly three actors can be seen as being very similar to a mexican standoff.

The processes potentially interested by this phenomenon are said to be *competing*, since each one of them uses or may use in the future one or more resources belonging to a set common to all of them.

When at least two competing processes are, in order to take possession of a certain resource, each waiting for the other to finish a computation - and thus neither ever does - the processes are said to be in a *deadlocked state* or in a *deadlock*.

More specifically, a deadlock occurs when each process' computation *needs* a resource held by another in order to complete its computation.

In computer systems, specifically in multithread, parallel and distributed systems, a deadlock is both a very subtle problem - because difficult to prevent during the system coding - and a very dangerous one: a deadlocked system is easily completely stuck, with consequences ranging from simple annoyances to life-threatening circumstances, being also in between the not negligible scenario of economical losses.

Then, how to avoid this problem? A lot of possible solutions has been studied, proposed and implemented. In this thesis we focus on detection of deadlocks with a static program analysis technique, i.e. an analysis performed without actually executing the program.

To begin, we briefly present the static Deadlock Analysis Model developed for coreABS$^{--}$ in chapter 1, then we proceed by detailing the *Class-based* coreABS$^{--}$ language in chapter 2.

Then, in Chapter 3 we lay the foundation for further discussions by analyzing the differences between coreABS$^{--}$ and ASP, an untyped *Object-based* calculi, so as to show how it can be possible to extend the Deadlock Analysis to *Object-based* languages in general.

In this regard, we explicit some hypotheses in chapter 4 first by presenting a possible, unproven type system for ASP, modeled after the Deadlock Analysis Model developed for coreABS$^{--}$.
Then, we conclude our discussion by presenting a simpler hypothesis, which may allow to circumvent the difficulties that arises from the definition of the "ad-hoc" type system discussed in the aforegoing chapter.

# Chapter 1

# A Informal Introduction to the Deadlock Analysis Model

We present here a brief introduction to the Deadlock Analysis Model developed for coreABS$^{--}$[1]. Our goal here is to give the reader a consistent idea of the Model, fundamental to comprehend the rest of the discussion.

The problem we are interested in is to be able to tell with absolute certainty if a given program may or may not exhibit deadlocked behaviour during its execution.
We want to detect this potential threat *before* the actual program execution; we're therefore in the field of Static Program Analysis, that is opposed to the discipline of Dynamic Program Analysis, which aims to detect potential threat or misbehaviours *during* the program execution[2].
For deadlock detection purposes Static analysis offers more guarantees, at the cost of producing in some cases false positives; in this regard Dynamic analysis is more precise, but at the cost of not being exhaustive: Dynamic analysis may "fly over" some deadlocks, thus not detecting them.

We treat programs written in object-oriented languages, where method calls are *asynchronous*; this means that after a method invocation the caller does not wait for the return value to continue its execution, but goes on with its activity.
To realize this behaviour, it's necessary to decouple the method invocation and the returned value by means of *futures*, pointers to values that may not be available yet. A *future* can be thought as a placeholder for a value that will be available when the callee's computation terminates.

The Deadlock Analysis Model[1] can be summarized this way:

1. Given the program source, a *Type Inference* system extracts from it a number of abstract behavioral descriptions pertinent to the deadlock analysis: the *contracts*[3].

2. Because the inference system is constraint-based, it produces a set of generated constraints, which is solved by a standard semi-unification technique.

3. The set of resolved constraints (called Contract Class Table) is given as input to an algorithm that transforms the contracts in a *finite state automata*, whose states are dependencies between entities of the program and whose transitions model how these dependencies may be activated or discarded during the execution of the program.
A *dependency* aims to model the fact that a certain actor may need a resource held by another in order to terminate its execution.

4. The automata is called *lafsa*. Because a potential misbehavior is signaled by the presence of a circularity in some state of it, an algorithm analyze the lafsa searching for these circularities; if none is found, the program is *certainly* lock-free; otherwise, it means that the program *may* reach a deadlocked configuration at run-time (as is the case of deadlocks depending on the scheduler's choices).

To have a more precise idea of what we intend when we talk of circular dependencies, please observe the following simple lafsas[10]:

$$\left\langle \mathbb{I}_{x,y,z}, m1(x,y,z) \right\rangle \rightarrow \left\langle \mathbb{I}_{x,y,z}, (x,y)||m2(y,z) \right\rangle \rightarrow \left\langle \mathbb{I}_{x,y,z}, (x,y)||(y,z)||m3(z) \right\rangle$$

The method $m1$ is defined so as to call the method $m2$ with a dependency between the caller $x$ and the callee $y$, while $m2$ is defined so as to call the method $m3$ with a dependency between the caller, now $y$, and the callee $z$; the method m3, at last, it's defined to do a certain computation on $z$ and only $z$, thus not expressing any dependency.

Therefore, we have the final:

$$\rightarrow \left\langle \mathbb{I}_{x,y,z}, (x,y)||(y,z)||0 \right\rangle$$

which have no circularity.

Informally speaking, the dependencies are modeled as $(x, y)$ and $(y, z)$, which means that $x$ depends on $y$ and $y$ depends on $z$, which terminates by itself; therefore all of them will terminate (at least for the deadlock analyzer's eye), and is thus assured that no deadlock will ever occur.

Instead, consider the following:

$$\left\langle \mathbb{I}_{x,y}, m1(x, y, x) \right\rangle \rightarrow \left\langle \mathbb{I}_{x,y}, (x, y)||m2(y, x) \right\rangle \rightarrow \left\langle \mathbb{I}_{x,y}, (x, y)||(y, x)||m3(x) \right\rangle$$

The method $m1$ is defined so as to call the method $m2$ with a dependency between the caller $x$ and the callee $y$, while $m2$ is in turn defined so as to call the method $m3$ with a dependency between the caller $y$ and the callee, now $x$; the method m3, at last, it's defined to do a certain computation on the callee object $x$. As you can see, there's something tricky going on.

We have infact the final:

$$\rightarrow \left\langle \mathbb{I}_{x,y}, (x, y)||(y, x)||0 \right\rangle$$

which have a circularity.

Still informally, $(x, y)$ i.e. termination of $x$ depends on termination of $y$, which in turn, because $(y, x)$, depends on termination of $x$.
This does *not* mean that this system will surely deadlock; instead it means that a potential deadlock is in the game, thus it can be said with absolute certainty that this system *may* deadlock itself and therefore it *may* never terminate.

We focus our discussion solely on the first step of the Deadlock Analysis: the generation of *contracts*, from which subsequently a lafsa like the ones just presented will be generated, from which in turn circular dependencies - if any - will be found.

Contracts are essentials because they are an abstract representation of the program behavior that allow us to statically collect its *name dependencies*: by being our analysis static - i.e. an analysis performed without actually executing the code - we don't dispose of the usual run-time information; instead, by properly using *names* and *types* we're able to identify every useful entity (objects, threads, ..) in the program; furthermore, we can *statically*

track these entities by "auto-magically" modeling every possible evolution of their names by means of contracts.

We're also interested in contracts because to know if it's possible to perform a deadlock analysys following the aforementioned model on a new language we must determine if they can be properly *remodeled* for it.
A *remodel* is necessary because the original contracts has been built around coreABS$^{--}$, the language introduced in the next chapter, and different object-oriented languages may show significant differences in the theoretical foundations they are built from and in the way they behave, as will be clear after reading the third chapter.

We now continue the dissertation by detailing coreABS$^{--}$.

# Chapter 2

# The coreABS$^{--}$ language

## 2.1 Generalities

First of all, in coreABS$^{--}$[1] the actor of concurrency is the *object*, not the thread or the process.

Objects are dynamically created instances of classes; their attributes are initialized to type-correct default values (e.g., null for object references), but may be redefined in an optional method *init*.
Class-based object-oriented programming languages take object generators as central, while object-based languages emphasize the objects themselves[4]. coreABS$^{--}$ is a class-based language that supports inheritance and uses explicit primitives for synchronization.

A certain object may have multiple tasks associated with it, but only - at most - *one* task per object is active at each point in time.
That task is called the active task, and it's the one that is said to *control* the object.

Tasks are created by method invocations. Each method invocation spawns a new task: the caller object continue its execution, while the callee object spawns a new task, which have within itself the code of the invocation.
In order for the caller object to be able to do that, a *future variable* is temporarily associated to the call's result.
By means of *futures*, i.e. pointers to values that may not be available yet, coreABS$^{--}$ realize the necessary decoupling between method invocation and returned value.

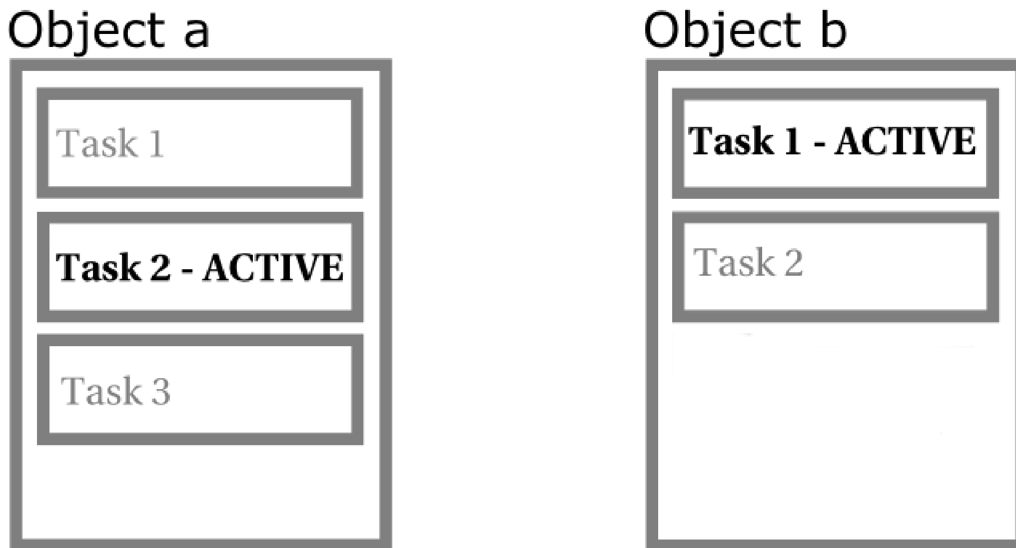


Figure 2.1: coreABS$^{--}$ Concurrency Model

The type inference system takes care of the proper tracking of names and futures by means of *contracts*[3], that retains the necessary information to detect locks.
This is possible because *contracts* introduce object name dependencies in the type system, thus introducing dependencies between actors of concurrency.

A dependency $(a, b)$ specifies that the lock on the caller object $a$ is released as soon as the lock of the callee object $b$ is released. This effectively models the fact that, in order to proceed, $a$ may be needing a result computed by $b$.

As explained in the previous chapter, contracts are subsequently used to create a finite state automata, from which is possible to infer circular object name dependencies, if any. A program whose lafsa does not manifest an object name circularity will never deadlock, thus encompassing the guarantees introduced in chapter 2.

First of all will be presented the syntax and the operational semantics of coreABS$^{--}$, then it'll be presented and thoroughly analyzed its type inference system.

## 2.2 Syntax

Please take a minute to observe the syntax of coreABS$^{--}$:

$$
\begin{aligned}
CL &::= \; class\ C\ extends\ C\ \{\bar{T}\ \bar{f}\ ;\ \bar{M}\} \\
M &::= \; T\ m\ (\bar{T}\ \bar{x})\{return\ e;\} \\
e &::= \; x \mid e.f \mid e!m(\bar{e}) \mid new\ C(\bar{e}) \mid e; e \mid e.get \mid e.await \\
T &::= \; C \mid Fut(T)
\end{aligned}
$$

The notation $\bar{C}$ is a shorthand for $C_1, ..., C_n$ and similarly for the other names, while sequence of pairs $C_1 f_1, ..., C_n f_n$ are abbreviated with $\bar{C}\bar{f}$.

A coreABS$^{--}$ program defines classes, datatypes and functions that may take one or more arguments as input, with a main block that configures the initial state.
Therefore, a coreABS$^{--}$ program is a collection of class definitions plus an expression to evaluate.

After asynchronously calling a method `m` with `x = o!m(e)`, the caller may proceed with its execution without blocking on the call. Here `x` is a future variable, `o` is an object and `e` are expressions.
As specified before, `x` is thus a future variable that refer to a return value which has yet to be computed.

There are two operations on future variables, which explicitly control synchronization in coreABS$^{--}$: `get` and `await`.
The return value is retrieved by the expression `x.get`, which blocks all execution in the caller object until the return value is available.
The expression `x.await`, instead, check if the result is available, releasing the lock.

For completeness, it's worth to specify that the syntax uses four disjoint infinite sets of names: *class names*, *field names*, *method names* and *variables*.

## 2.3 Semantics

The operational semantics of coreABS$^{--}$ uses two additional disjoint infinite sets of names: *object names* and *task names*.

*Values* are terms defined by the following grammar:

$$v ::= t \mid a[\bar{f} : \bar{v}]$$

The operational semantics of coreABS$^{--}$ is presented as a *Reduction Semantic*. A reduction semantic uses the so-called reduction contexts[5] or, in this case, *evaluation contexts* E and S, whose syntax is:

$$\begin{array}{lcl} \mathtt{S} & ::= & [\,] \mid \mathtt{x = E} \mid \mathtt{if\ (E)\ \{s\}\ else\ \{s\}} \mid \mathtt{S\,;\,s} \mid \mathtt{return\ E} \\ \mathtt{E} & ::= & [\,] \mid \mathtt{E!m(\bar{e})} \mid \mathtt{v!m(\bar{v}, E, \bar{e})} \mid \mathtt{new\ C(\bar{v}, E, \bar{e})} \mid \mathtt{E = e} \mid \mathtt{v = E} \end{array}$$

The contexts E and S include a hole [ ] where a term can be plugged in, while the shape of the contexts indicate where reduction can occur, i.e. where a term can be plugged into.

Evaluation contexts models a state transition system, where we have *states* S, S', ... and sets of *tasks* $\mathtt{t} :_a^\ell \mathtt{e}$ where $t$ is a task's name, $a$ is an object name, $\ell$ can be $\top$ if the task owns the control of $a$ (thus if $t$ is the active task, see section 2.1) or $\bot$ if it doesn't, and $e$ is an expression.

The transition relation $\mathtt{t} :_a^\ell \mathtt{e}$ between states that describe the semantics is defined by the rules that follows, where the following notations and shortenings may be encountered:

- the predicate *unlocked*$(\mathcal{S}, a)$ that returns *true* if every $\mathtt{t} :_a^\ell \mathtt{s}$ in $\mathcal{S}$ is such that $\ell = \bot$;
- the function *freshtask*( ) always returns a new task name;
- in $\mathtt{t} :_a^\ell \mathtt{s}$, the superscript $\ell$ is omitted when it is not relevant.

(This)

$$\frac{\mathtt{M} \vdash \bar{\mathtt{f}} : \bar{\mathtt{v}}}{\mathtt{t}, \mathtt{M} :_a^{\top} \mathsf{S}[\mathsf{E}[\mathtt{this}]] \xrightarrow{a} \mathtt{t}, \mathtt{M} :_a^{\top} \mathsf{S}[\mathsf{E}[a[\bar{\mathtt{f}} : \bar{\mathtt{v}}]]]}$$

(Field-Var)

$$\frac{\mathtt{M}(\mathtt{l}) = \mathtt{v}}{\mathtt{t}, \mathtt{M} :_a^{\top} \mathsf{S}[\mathsf{E}[\mathtt{l}]] \xrightarrow{a} \mathtt{t}, \mathtt{M} :_a^{\top} \mathsf{S}[\mathsf{E}[\mathtt{v}]]}$$

(New)

$$\frac{\mathit{fields}(\mathtt{C}) = \bar{\mathtt{T}}\ \bar{\mathtt{f}} \quad b = \mathit{fresh}(\mathtt{C})}{\mathtt{t}, \mathtt{M} :_a^{\top} \mathsf{S}[\mathsf{E}[\mathtt{new}\ \mathtt{C}(\bar{\mathtt{v}})]] \xrightarrow{a} \mathtt{t}, \mathtt{M} :_a^{\top} \mathsf{S}[\mathsf{E}[b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]]]}$$

(Update)

$$\mathtt{t}, \mathtt{M} :_a^{\top} \mathsf{S}[\mathtt{x} = \mathtt{v}] \xrightarrow{a} \mathtt{t}, \mathtt{M}[\mathtt{x} : \mathtt{v}] :_a^{\top} \mathsf{S}[\mathtt{skip}]$$

(Seq)

$$\mathtt{t}, \mathtt{M} :_a^{\top} \mathtt{skip}\ ;\ \mathtt{s} \xrightarrow{a} \mathtt{t}, \mathtt{M} :_a^{\top} \mathtt{s}$$

(Seq-val)

$$\mathtt{t}, \mathtt{M} :_a^{\top} \mathtt{v}\ ;\ \mathtt{s} \xrightarrow{a} \mathtt{t}, \mathtt{M} :_a^{\top} \mathtt{s}$$

The former rules are standard and won't therefore be discussed.

(Invk)

$$\frac{\mathit{mbody}(\mathtt{m}, \mathit{class}(b)) = \bar{\mathtt{x}}.\mathtt{s} \quad \mathtt{t}' = \mathit{freshtask}(\ )}{\mathtt{t}, \mathtt{M} :_a^{\top} \mathsf{S}[\mathsf{E}[b[\bar{\mathtt{f}}' : \bar{\mathtt{v}}]!\mathtt{m}(\bar{\mathtt{v}}')]] \xrightarrow{a} \mathtt{t}, \mathtt{M} :_a^{\top} \mathsf{S}[\mathsf{E}[\mathtt{t}']],\ \ \mathtt{t}', [\bar{\mathtt{f}}' : \bar{\mathtt{v}}, \bar{\mathtt{x}} : \bar{\mathtt{v}}'] :_b^{\perp} \mathtt{s}}$$

This rule (Invk) defines the method invocation. According to this rule, the evaluation of $b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]!\mathtt{m}(\bar{\mathtt{v}}')$ produces a future reference $\mathtt{t}'$ to the value returned by $\mathtt{m}$. The task evaluating the called method is created and the evaluation of the caller can continue – the invocation is asynchronous; however, the evaluation of the called method $\mathtt{m}$ cannot begin until its value of $\ell$ becomes $\top$.

(Get)

$$\frac{\mathtt{M}(\mathtt{x}) = \mathtt{t}'}{\mathtt{t}, \mathtt{M} :_a^{\top} \mathsf{S}[\mathsf{E}[\mathtt{get}\ \mathtt{x}]],\ \ \mathtt{t}', \mathtt{M}' :_b \mathtt{v} \xrightarrow{a} \mathtt{t}, \mathtt{M} :_a^{\top} \mathsf{S}[\mathsf{E}[\mathtt{v}]],\ \ \mathtt{t}', \mathtt{M}' :_b \mathtt{v}}$$

This rule (Get) permits the retrieval of the value returned by a method. The caller object is stuck until the value is actually retrieved.

$$\text{(AwaitT)}\quad \mathtt{M(x) = t't, M} :_a^{\top} \mathtt{E[await\ x?]},\ \mathtt{t'} :_b \mathtt{v} \xrightarrow{a} \mathtt{t} :_a^{\top} \mathtt{E[t']},\ \mathtt{t'} :_b \mathtt{v}$$

$$\text{(AwaitF)}\quad \frac{\mathtt{e} \neq \mathtt{v}}{\mathtt{t} :_a^{\top} \mathtt{E[t'.await]},\ \mathtt{t'} :_b \mathtt{e} \xrightarrow{a} \mathtt{t} :_a^{\bot} \mathtt{E[t'.await]},\ \mathtt{t'} :_b \mathtt{e}}$$

Rules (AwaitT) and (AwaitF) model the `await` operation: if the task $\mathtt{t'}$ is terminated – it is paired to a value – then `await` is unblocking; otherwise the control of the object is released by `t`.

$$\text{(Release)}\quad \mathtt{t} :_a^{\top} \mathtt{v} \xrightarrow{a} \mathtt{t} :_a^{\bot} \mathtt{v}$$

Rule (Release) models task termination, which amounts to store the returned value in the state and releasing the control of the object.

$$\text{(Lock)}\quad \frac{\mathtt{e} \neq \mathtt{v}}{\mathtt{t} :_a^{\bot} \mathtt{e} \xrightarrow{a} \mathtt{t} :_a^{\top} \mathtt{e}} \qquad \text{(State)}\quad \frac{\mathcal{S} \xrightarrow{a} \mathcal{S}' \quad \mathit{unlocked}(\mathcal{S}'', a)}{\mathcal{S}, \mathcal{S}'' \xrightarrow{a} \mathcal{S}', \mathcal{S}''}$$

According to the transition relation, a task $\mathtt{t} :_a^{\ell} \mathtt{e}$ moves provided $\ell = \top$, except for rule (Lock). This rule allows a task with a non-value expression to get the control. The rule must be read in conjunction with rule (State) that lifts transitions $\xrightarrow{a}$ to complex states and enforces the property that there is always at most one task per object owning the control. This means that (Lock) cannot be used if the state has a task $\mathtt{t'} :_a^{\top} \mathtt{e'}$.

## 2.4 Type System and Contracts

As explained in section 2.1, the type inference system takes care of the proper tracking of names and futures by means of contracts, abstract behavioural descriptions fundamental for the deadlock analysis that retains the necessary information to detect locks.

We'll say again that contracts introduce object name dependencies in the type system, thus introducing dependencies between objects in order to track eventual circularities.

The type inference system associates a contract to every method definition and every expression to evaluate.

Instead of immediately presenting the type inference rules, we start by introducing some practical examples of contracts inference, hopefully allowing the reader to form a correct idea of how contracts works and why they are so central to the deadlock analysis.

After this informal presentation of contracts inference, we'll analyze the fundamental formal definitions useful for our discussion. For the complete technical details, please refer to [1].

As a first example of contract, consider first the following coreABS$^{--}$ class:

```
class C extends Object {
    Object f:
    C m() {return new C(this.f);}
    }
```

When the type inference system encounters the class definition, it associates the following contract to the definition of the method `m()`:

$$a[f : X]()\{0\}\ b[f : X]$$

The contract $a[f : X]()\{0\}\ b[f : X]$ is thus the contract of the method `m`.

Now we'll analyze every chunk of the contract, in order to give the reader a practical idea of how contracts works in general; it will be therefore possible

for the reader to more easily understand the type inference system.

$a[f : X]$ specifies that $a$ is the object on which `m()` is called; because `m()` is a method of class `C`, and the class `C` contemplates a field $f$, the contract specifies that `m()` is called on a object named $a$ that have the value $X$ in its field $f$.

The subsequent parenthesis () are in this case empty; this is because the work of this chunk is to specify the argument of the call. Because `m()` is a method that requires no argument, the parenthesis contains none.

The next chunk specifies eventual behaviors of the method that are relevant for the deadlock analysis, which is also none in this case, thus generating 0.

At last, the contract specifies the returned object of the method. With $b[f : X]$ the contract embeds important information: the returned object's name $b \neq a$ indicates that the returned object is different from the one on which the method is called; furthermore, the contract specifies that the value of the new object's field is taken from the value of the original object's field.

Now, condider:

```
class C extends Object {
    Object f:
    C m() {return new C(this.f);}
    }
class E extends C {
    C n(E c) {return c!m().get;}
    }
```

When the type inference system encounters the class definitions, it associates the following contract to the definition of the method `n()`:

$a[f : X](b[f : Y])\{\ E.m\ b[f : Y]().(a, b)\ \}\ c[f : Y].$

As you can see, `n()` is called on the object $a$, with an object $b$ as the argument.

Now the method has a relevant behavior; specifically, it calls the method `m()` on the object passed as argument to the method `n()`, but instead of

allowing the caller object $a$ to continue its execution, the `get` operation constrains $a$ to wait for the termination of $b$ in order to continue its execution. This behaviour is specified by the object dependency (a,b).

The part between brackets is called the *contract body*, it embeds the contract associated with the expression existent in the method's body. Its target is to track information about object dependencies.

If $\mathbb{c}$ is the contract body, then

$\mathbb{c}$ ::=

| | |
|---|---|
| $0$ | no relevant information |
| $C.m\ \mathbb{O}(\overline{arg})$ | C.m invoked on object $\mathbb{O}$ with parameters $\overline{arg}$ |
| $C.m\ \mathbb{O}(\overline{arg}).(a,b)$ | as above with a GET operation on the result |
| $C.m\ \mathbb{O}(\overline{arg}).(a,b)^a$ | as above with an AWAIT operation on the result |
| $(a,b)$ | a GET operation on a field |
| $(a,b)^a$ | an AWAIT operation on a field |
| $\mathbb{c};\mathbb{c}$ | sequential composition of contract bodies |

*Future records* $\mathbb{r}$, $\mathbb{s}$, $\cdots$ are defined by the following grammar:

$$\mathbb{r} \quad ::= \qquad X \ \mid \ a[\bar{\mathtt{f}} : \bar{\mathbb{r}}] \ \mid \ a \rightsquigarrow \mathbb{r}$$

A record name $X$ represents a variable that may be possibly instantiated by substitutions.
The future record $a[\bar{\mathtt{f}} : \bar{\mathbb{r}}]$ defines the object name and the future records of values stored in its fields.
At last, the future record $a \rightsquigarrow \mathbb{r}$ specifies that, in order to access to $\mathbb{r}$ one has to acquire the control of the object with name $a$ (and to release this control once the method has been evaluated).
Future records as $a \rightsquigarrow \mathbb{r}$ are associated to method invocations: the object name $a$ represents the object of the invoked method.

The type inference system is specified by the following rules:

$$\text{(T-VAR)}\quad \Gamma \vdash_a \mathtt{x} \,:\, \Gamma(\mathtt{x})\,,\; \mathtt{0}$$

$$\text{(T-FIELD)}\quad \frac{\Gamma \vdash_a \mathtt{e} \,:\, (\mathtt{C}, a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}])\,,\; \mathbb{c} \qquad \mathit{fields}(\mathtt{C}) = \bar{\mathtt{T}}\,\bar{\mathtt{f}} \quad \mathtt{T}\,\mathtt{f} \in \bar{\mathtt{T}}\,\bar{\mathtt{f}} \quad \mathtt{f} : \mathbb{r} \in \bar{\mathtt{f}} : \bar{\mathbb{r}}}{\Gamma \vdash_a \mathtt{e.f} \,:\, (\mathtt{T}, \mathbb{r})\,,\; \mathbb{c}}$$

Rule (T-FIELD) defines the judgment for accessing to fields of an object produced by e. The rule constraints e to have a class type (not a future) and to have a future record as $a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}]$.

$$\text{(T-INVK)}\quad \frac{\begin{array}{c}\Gamma(\mathtt{C.m}) = a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}](\bar{\mathbb{s}}) \rightarrow \mathbb{r} \\ \Gamma \vdash_a \mathtt{e} \,:\, (\mathtt{C}, \sigma(a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}]))\,,\; \mathbb{c} \qquad \Gamma \vdash_a \bar{\mathtt{e}} \,:\, (\bar{\mathtt{T}}, \sigma(\bar{\mathbb{s}}))\,,\; \bar{\mathbb{c}} \\ \mathit{mtype}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{T}}' \rightarrow \mathtt{T}' \quad \bar{\mathtt{T}} \mathrel{\mathtt{<:}} \bar{\mathtt{T}}'\end{array}}{\Gamma \vdash_a \mathtt{e!m}(\bar{\mathtt{e}}) \,:\, (\mathtt{Fut(T')}, \sigma(a') \rightsquigarrow \sigma(\mathbb{r}))\,,\; \mathbb{c} \,⨟\, \bar{\mathbb{c}} \,⨟\, \mathtt{C.m}\; \sigma(a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}])(\sigma(\bar{\mathbb{s}})) \rightarrow \sigma(\mathbb{r})}$$

Rule (T-INVK) defines the judgments of method invocations $\mathtt{e!m}(\bar{\mathtt{e}})$.
Let $a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}](\bar{\mathbb{s}}) \rightarrow \mathbb{r}$ be the interface of C.m stored in $\Gamma$. Object names and record names in this interface are actually place-holders for actual values. Therefore, in order to type $\mathtt{e!m}(\bar{\mathtt{e}})$, there must exist a substitution $\sigma$ such that $\Gamma \vdash_a \mathtt{e} \,:\, (\mathtt{C}, \sigma(a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}]))\,,\; \mathbb{c}$ and $\Gamma \vdash_a \bar{\mathtt{e}} \,:\, (\bar{\mathtt{T}}, \sigma(\bar{\mathbb{s}}))\,,\; \bar{\mathbb{c}}$. (It is possible to use a unique $\sigma$ since names in $a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}]$ and $\bar{\mathbb{s}}$ are disjoint.) The (standard) type of $\mathtt{e!m}(\bar{\mathtt{e}})$ is a future type $\mathtt{Fut(T')}$, where $\mathtt{T'}$ is determined with standard arguments for object-oriented languages. The future record of $\mathtt{e!m}(\bar{\mathtt{e}})$ is $\sigma(a') \rightsquigarrow \sigma(\mathbb{r})$ indicating that the value may be returned as soon as the control of $\sigma(a')$ is acquired.
The contractual issue of $\mathtt{e!m}(\bar{\mathtt{e}})$ is almost obvious: it composes in sequence the contracts of e, $\bar{\mathtt{e}}$ and the method invocation.

$$\text{(T-NEW)}\quad \frac{\Gamma \vdash_a \bar{\mathtt{e}} \,:\, (\bar{\mathtt{T}}, \bar{\mathbb{r}})\,,\; \bar{\mathbb{c}} \quad \mathit{fields}(\mathtt{C}) = \bar{\mathtt{T}}'\;\bar{\mathtt{f}} \quad \bar{\mathtt{T}} \mathrel{\mathtt{<:}} \bar{\mathtt{T}}' \quad a' \;\mathit{fresh}}{\Gamma \vdash_a \mathtt{new\;C}(\bar{\mathtt{e}}) \,:\, (\mathtt{C}, a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}])\,,\; \bar{\mathbb{c}}}$$

Rule (T-NEW) types object creations that, in the type system, amounts to using a fresh object name – called $a'$ in the rule – for the root of its future record. The remaining part of the judgment is almost standard.

(T-Get)
$$\frac{\Gamma \vdash_a \texttt{e} : (\texttt{Fut(T)}, a' \rightsquigarrow \mathbb{s}),\ \mathbb{c}}{\Gamma \vdash_a \texttt{e.get} : (\texttt{T}, \mathbb{s}),\ \mathbb{c} \between (a, a')}$$

(T-Await)
$$\frac{\Gamma \vdash_a \texttt{e} : (\texttt{Fut(T)}, a' \rightsquigarrow \mathbb{s}),\ \mathbb{c}}{\Gamma \vdash_a \texttt{e.await} : (\texttt{Fut(T)}, a' \rightsquigarrow \mathbb{s}),\ \mathbb{c} \between (a, a')^{\texttt{w}}}$$

Rules (T-Get) and (T-Await) define types for `e.get` and `e.await` expressions. In these cases, the type of `e` has to be `Fut(T)` and, correspondingly, the future record type has the pattern $a' \rightsquigarrow \mathbb{s}$. In case of (T-Get), the type of `e.get` is reduced to $(\texttt{T}, \mathbb{s})$, while, in case of `e.await`, it is not changed.
As regards contracts, (T-Get) and (T-Await) extend the contract of `e` with the pairs $(a, a')$ and $(a, a')^{\texttt{w}}$, respectively, where the index $a$ of the judgment defines the first element of the object name dependency – $a$ stores the the object's root whose method contains the expression `e.get` or `e.await`. The element $a'$ of the object name dependency is the root of the future record of `e`.

(T-Seq)
$$\frac{\Gamma \vdash_a \texttt{e} : (\texttt{T}, \mathbb{r}),\ \mathbb{c} \qquad \Gamma \vdash_a \texttt{e}' : (\texttt{T}', \mathbb{r}'),\ \mathbb{c}'}{\Gamma \vdash_a \texttt{e}\,;\,\texttt{e}' : (\texttt{T}', \mathbb{r}'),\ \mathbb{c} \fatsemi \mathbb{c}'}$$

Rule (T-Seq) simply explicitates the sequential composition of contracts.

The next two rules are rules for method declarations and class declarations, while the already presented ones are rules for coreABS$^{--}$ expressions.

(T-Method)
$$\frac{\begin{array}{c}\Gamma(\texttt{C.m}) = a[\bar{\texttt{f}} : \bar{\mathbb{r}}](\bar{\mathbb{s}}) \to \mathbb{r}' \qquad \Gamma + \bar{\texttt{x}} : (\bar{\texttt{T}}, \bar{\mathbb{s}}) + \texttt{this} : (\texttt{C}, a[\bar{\texttt{f}} : \bar{\mathbb{r}}]) \vdash_a \texttt{e} : (\texttt{T}', \mathbb{r}'),\ \mathbb{c} \\ \texttt{T}' \texttt{<:}\ \texttt{T} \qquad \texttt{C} \texttt{<:}\ \texttt{D} \text{ and } \texttt{m} \in \texttt{D} \quad \text{imply} \quad \mathit{mtype}(\texttt{m}, \texttt{D}) = \bar{\texttt{T}} \to \texttt{T}'\end{array}}{\Gamma \vdash\ \texttt{T m}\ (\bar{\texttt{T}}\ \bar{\texttt{x}})\{\texttt{return e ;}\} : a[\bar{\texttt{f}} : \bar{\mathbb{r}}](\bar{\mathbb{s}})\{\mathbb{c}\}\ \mathbb{r}' \text{ IN } \texttt{C}}$$

Rule (T-Method) defines the type and the contract of a method.

(T-Class)
$$\frac{\Gamma \vdash \bar{\texttt{M}} : \bar{\mathbb{C}} \text{ IN } \texttt{C}}{\Gamma \vdash \texttt{class C extends D}\ \{\bar{\texttt{C}}\ \bar{\texttt{f}};\ \ \bar{\texttt{M}}\} : \{\mathit{mname}(\bar{\texttt{M}}) \mapsto \bar{\mathbb{C}}\}}$$

Rule (T-Class) types a class definition associating to it a mapping from method names to method contracts.

## 2.5 A Deadlock Example in coreABS$^{--}$

In this section we present two simple examples of deadlock in coreABS$^{--}$; the purpose of these is to practically illustrate how deadlock may arise in this language.

A simple circular dependency involves only one task as in the method

```
Int fact(Int n){ return    if (n==0) then 1 ;
                           else n*(this!fact(n-1).get)   ; }
```

This method defines the factorial function (for the sake of the example we include primitive types `Int` and conditional into the coreABS$^{--}$ syntax). In the body of `fact`, the recursive invocation `this!fact(n-1)` is postfixed by a `get` operation that retrieves the value returned by the invocation.

Yet, `get` does not releases the lock of the caller object; therefore the task evaluating `this!fact(n-1)` is fated to be delayed forever because its object is the same of the caller.

Another example involves a trickier situation[6]; consider the classes

```
class C {
    C m() {return new C();}
    C r(C x) {return x!m.get();}
    }
class D extends C {
    Fut(C) q(D y) {
        return y!r(this);
        this!r(y);
        }
    }
```

Suppose we have the following expression to evaluate:

```
new D()!q(new D());
```

First of all, two objects - `o1` and `o2` - of class `D` are created; after a number of method invocations, inside object `o1` it'll be spawned a new task, and the same will happen inside object `o2`.

The problem is that the task spawned inside the object `o1` contains the expression `o2!m().get;`, while the task spawned inside the object `o2` contains the expression `o1!m().get;`.

If, for example, the scheduler decides that `o1` executes the task `o1!m().get;` called by `o2` *before* executing its own call `o2!m().get;`, then no trouble will ever arise. The `get` operation of `o2` will be satisfied by the return value given by `o1`, and subsequently the latter will execute its own call to the former retrieving the result.

On the contrary, if both objects make the two calls as their first operation, a deadlock will arise and they'll be forever hanged.
Infact, imagine that `o2` invokes `m` on `o1`, where a new task is created, and waits for the result returned by that task inside `o1`, keeping in the mean time the lock of `o2` (since it's a `get` operation).
If, analogously, inside `o1` gets to run the invocation of `m` on `o2`, the former will hang waiting for a method in `o2` to return, holding in the mean time its lock because of the `get` operation.
In this case the new tasks created by the method `m`'s invocations will never be able to execute; furthermore - and more importantly - the objects `o1` and `o2` are both indefinitely blocked and hence the program is stuck.

In this last example, which one of the possibilities will materialize during the program's execution will depend from the scheduler's choice.

# Chapter 3

# The ASP Calculus

## 3.1 Generalities

First of all, in ASP[7] the actors of concurrency are not objects as is the case in coreABS$^{--}$ but *activities*.

ASP is an *untyped* object-based language that does not support inheritance and does not use explicit primitives for synchronization; instead, synchronization is implicit and realized by means of the so-called *waits by necessity*, which will be thoroughly discussed later on.

Activities are dinamically created during the program's execution; a certain activity have within itself exactly one master object called the activity's *active object*, and zero or more *passive objects* contained within the activity.

We may say that the model of concurrency of coreABS$^{--}$ is similar to the one of ASP if we pose objects $\approx$ activities and tasks $\approx$ objects (compare Fig. 2.1 and Fig. 3.1). Contrarily to what happens in coreABS$^{--}$ with tasks, in ASP the active object of an activity is always the same: a passive object can never become the activity's active object, and the active object can never become a passive object.

In ASP the creation of objects is a little bit trickier than in coreABS$^{--}$; before explaining how it works, it's necessary to introduce the difference between a *deep copy* and a *shallow copy* (see Fig. 3.2). In both cases, a object $a$ is read and its data is copied in a object $b$.
What diversifies a deep copy from a shallow copy is that the latter duplicates *as they are* all memory references found in $a$, while the former copy in

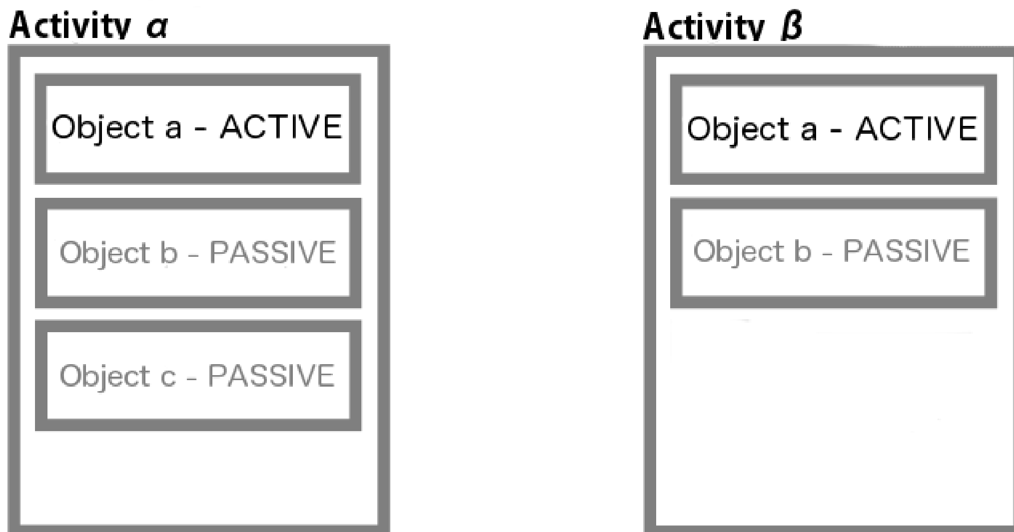


Figure 3.1: ASP Concurrency Model

the new object $b$ all *the data they points to.* With a deep copy, $a$ and $b$ do not depend on each other, while with a shallow copy if one of the aforementioned memory references is modified then the change will affect both objects.

When an activity is created, an active object must be always specified; all the passive objects of the new activity are automatically created by *deep copying* all the dependencies of the new activity's active object: to be more specific, if one of these dependencies encompasses other objects, those are *deep copied* into the new activity as passive objects.

Activities communicate by asynchronous method calls allowing both sender and callee to perform operations between the request sending and its treatment; this decoupling of method invocation and return value is again realized by the use of futures.
Asynchronous replies may occur in any order without observable consequences, a given activity is insensitive to the moment when a result come back.

Of course within each activity the execution is sequential, as is the case of objects in coreABS$^{--}$.

It's necessary to point out that, being ASP an object-based calculus,

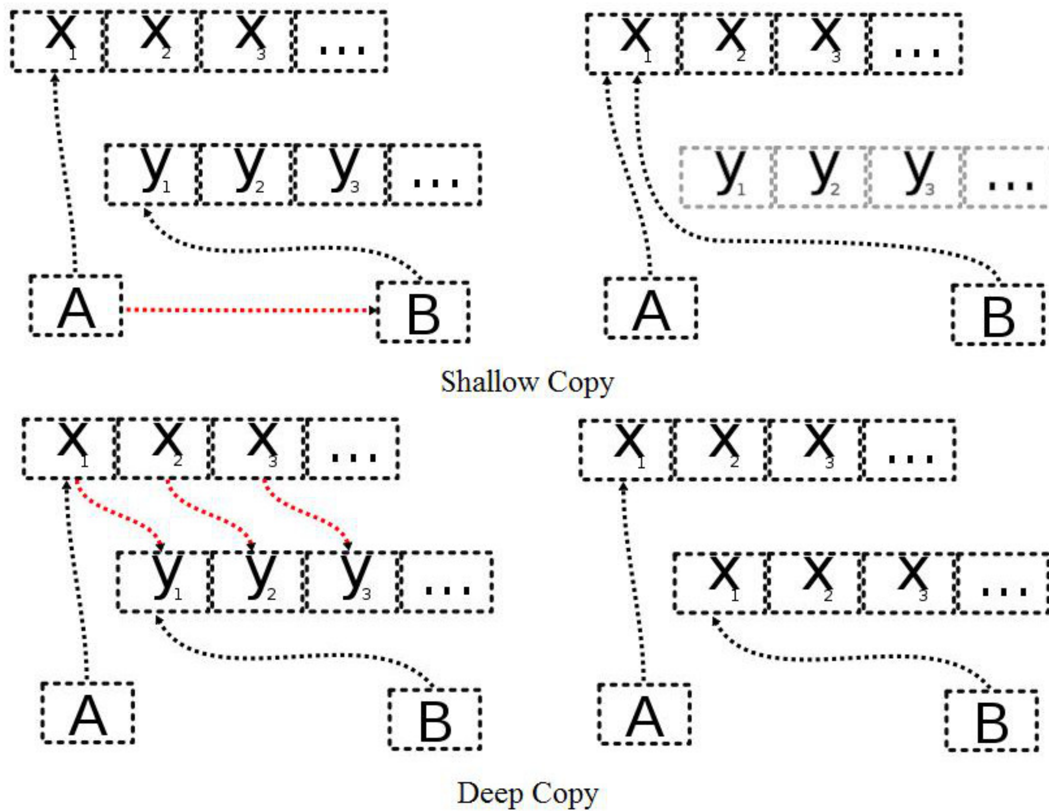


Figure 3.2: Shallow Copy vs Deep Copy

classes does not exist; they are replaced by object constructors, and objects working in the program are generated by its procedures; this means there's no real notion of inheritance; this can only be achieved by properly combining cloning and update operations[8].

You'll see in Section 3.4 that this brings the consequence of a language with a very counter-intuitive syntax, at least with respect to the most common programming languages.

Now will be made a short presentation of the syntax of ASP; subsequently, the semantics of the language will be thoroughly analyzed.
Being ASP untyped, no type inference system for contracts exist. In chapter 4 will be explicitated a not proven type system hypothesis that aims to fill this gap, in order to show how may be possible to extend the deadlock analysis system developed on coreABS$^{--}$ to ASP and object-based languages in general.

## 3.2 Syntax

Please take a minute to observe the syntax of ASP:

$t ::=$

| | |
|---|---|
| $x$ | variable |
| $\left[f_i = b_i;\ m_j = \varsigma(x_j, y_j)a_j\right]^{i\in 1..n}_{j\in 1..m}$ | object definition |
| $a.l_i$ | field access |
| $a.l_i\ := b$ | field update |
| $a.m_j(b)$ | method call |
| $CLONE(a)$ | shallow copy |
| $ACTIVE(a, m_j)$ | activity creation |
| $SERVE(m_1...m_n)^{n>0}$ | method serve |
| $LET\ x\ =\ e\ IN\ e'$ | sequential composition |

where $t$ is a term.

The syntax of ASP is strongly inspired from the imp$\varsigma$-calculus[8]; relatively to imp$\varsigma$, the only characteristics that have been changed in ASP are the following:

- Because arguments passed to active objects methods plays a particular role, a parameter to every method have been added. Therefore, in addition to the *self* argument of methods, noted $x_j$ and representing the object on which the method is invoked, it has been added an argument representing a parameter object to be sent to the method, noted $y_j$.
- Method update it's not included in the calculus but - indirectly - it is still possible to express updatable methods in it[7].
- It's been made a distinction between *user syntax* - i.e. the terms appearing in the programs code - and *runtime syntax* - i.e. terms generated internally during the programs' evaluation -. For example, during the reduction, locations (reference to objects in a store) can be part of terms. Locations do not appear in user syntax, but will appear on section 3.3 in the reduction rules.

Except for the just made punctualizations, the syntax of variables, object definition, field access, field update and method call is quite standard and

won't therefore be discussed.

CLONE($a$) do a *shallow copy* of the object $a$ passed as parameter.

ACTIVE($a$,$m_j$) creates a new activity with $a$ as its active object. $a$ is copied with all its dependencies - *deep copy* - into the new activity. $m_j$ is the name of a method which will be called as soon as the object is activated.

SERVE($m_1 ... m_n$)$^{n>0}$ stops the activity until a request on one of the method specified as parameter is found in the pending requests list; then, after its execution, the activity proceeds from where it stopped.

LET $x = e$ IN $e'$ may be seen as being equivalent to $e; e'$, i.e. the usual sequential composition of expressions.

## 3.3 Semantics

Before going deep into the semantic rules, it's necessary to introduce some concepts being used.

A *store* $\sigma$ is a finite map from locations to reduced objects $\sigma ::= \{\iota_i \mapsto o_i\}$, while a *reduced object* is an object with all field reduced to a location, i.e. $o ::= \left[l_i = \iota_i; m_j = \zeta(x_j, y_j)a_j\right]^{i\in 1..n}_{j\in 1..m}$.

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

The operation $\sigma :: \sigma$ (store append) append two stores with disjoint locations. When the domains are not disjoint, the operation $\sigma + \sigma'$ (store update) updates the values defined in $\sigma'$ by those defined in $\sigma$.

The operational semantics of ASP is divided into sequential semantics (inside each activity) and parallel semantics (between different activities).

As is the case for coreABS$^{--}$, also for ASP the operational semantics is presented as a *Reduction Semantic*[5]. We therefore have reduction contexts $\mathfrak{R}$, whose syntax is:

$$\mathfrak{R} ::= \bullet \mid \mathfrak{R}.l_i \mid \mathfrak{R}.m_j(b) \mid \iota.m_j(\mathfrak{R}) \mid \mathfrak{R}.l_i := b \mid \iota.l := \mathfrak{R} \mid CLONE(\mathfrak{R}) \mid$$
$$\left[l_i = \iota_i; l_k = \mathfrak{R}; l_{k'} = b_{k'}; m_j = \zeta(x_j, y_j)a_j\right]^{i \in 1..k-1, k' \in k+1..n}_{j \in 1..m}$$

The context $\mathfrak{R}$ include a hole $\bullet$ where a term can be plugged in, while the shape of the contexts indicate where reduction can occur, i.e. where a term can be plugged into.

Reduction contexts model a state transition system; in ASP states are the so-called *configurations*, while the transition relation between states that describe the semantics is defined by the set of rules that follows.

To allow a better understanding of ASP's semantics, configurations are differentiates between *sequential configurations* and *parallel configurations*.

The form of a sequential configuration is: $(a, \sigma)$. It can be thought as a state where $a$ is an expression and $\sigma$ is a store.
To evaluate a user term $a$, an initial configuration $(a, \varnothing)$ is created; it contains the user term $a$ and the empty store.

We define as $\rightarrow_s$ the transition relation from a state s to a state s'.
Note that $s_1 \rightarrow_s s_2 \wedge s_1 \rightarrow s_3 \implies s_2 \equiv s_3$, this is a property that assures determinism, and for the sake of clarity we can roughly say that, for what regards static analysis, locations names may be $\alpha$-converted[9].

First, we analyze the sequential reduction rules.

STOREALLOC

$$\frac{\iota \notin DOM(\sigma)}{(\mathcal{R}(o), \sigma) \rightarrow_s (\mathcal{R}[\iota], \{\iota \rightarrow o\} :: \sigma)}$$

This rule simply describes the allocation of a new object in the store with the store append operation. As you can see, the location just associated to the object is given back as return value.

FIELD

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

This rule is self-explanatory. It describes how locations behave when a field's value is requested.

INVOKE

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

This rule specifies the location references in the case of a sequential method invocation, i.e. a method invocation on an object of the same activity of the caller.

UPDATE

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

The field update rule specifies that - in the store - the object is modified according to the new value associated to the relative field; this is accomplished with the store update operation.

CLONE

$$\frac{\iota' \notin DOM(\sigma)}{(\mathcal{R}[CLONE(\iota)], \sigma) \rightarrow_s (\mathcal{R}[\iota'], \{\iota' \rightarrow \sigma(\iota)\} :: \sigma)}$$

With a clone operation - i.e. a shallow copy of the object passed as parameter - a new location is added to the store with the store appending operation. This new location is associated with - i.e. points to - the object passed as

parameter; in this way, the cloned object is - as intended - copied by means of a shallow copy; infact, only a new location is created, not a new object.

Except for the store mechanism, which may seem a little farraginous, the sequential reduction rules are quite straightforward.
On the other hand, in order to understand the parallel reduction rules, it's necessary to introduce other concepts in advance.

First of all, every activity $\alpha$ has its own store $\sigma_\alpha$ which contains one active and many passive objects. It also contain a *pending request queue* which stores the pending method calls and a *future list* which stores the results of finished requests.
As already anticipated, activities run in parallel and interact only through asynchronous method calls. Every remote method call sent to an activity is actually sent to its active object.

Any object in any activity can reference active objects and futures, while passive objects are only referenced by objects belonging to the same activity. It's important to specify that a field access on an active object reference is irreversibly stuck.

Therefore, in a parallel configuration an object name may be associated to:

- $\left[l_i = \iota_i; m_j = \zeta(x_j, y_j)a_j\right]_{j \in 1..m}^{i \in 1..n}$ — an actual object
- $AO(\alpha)$ — an active object reference
- $fut(f_i^{\alpha \to \beta})$ — a future reference

Note that every reference to a future can be replaced by the calculated value at any time. In the mean time, a future is a placeholder for the result of a not-yet-performed method invocation. As a consequence, the calling thread can go on with executing its code, as long as it doesn't need to invoke methods on the returned object.

If this need arises, the caller is automatically blocked; such blocking states are called *wait-by-necessity*.
More specifically, a wait-by-necessity happens when is performed a so-called

*strict operation* on a future. If the method relative to the future finishes its computation and the result is returned, and if an execution was blocked by a call-by-necessity, that execution can now continue.

The *strict operations* are: field access, field update, method access and object cloning.

An asynchronous method call on an active object consists in atomically adding an entry to the *pending requests* of the callee, and associating a future to the response. Arguments of requests and values of futures are deeply copied when they are transmitted between activities.

The runtime syntax $a \Uparrow f, b$ is used to save the state of the computation of the request currently served by an activity when the SERVE primitive is encountered, i.e. when another request - $a$ - have to be immediately served. To be able to continue the old request when the execution of the new is finished and thus the corresponding future is associated with the calculated value, the term $b$ and its future $f$ are saved in the runtime syntax.

The operational semantics of ASP uses - in addition to the name sets for fields, methods and variables identifiers which appear in the user terms - three disjoint name sets: one for activities $(\alpha, \beta, ...)$, one for locations $(\iota, \iota', ...)$ and one for futures $(f, f_i, ...)$.
Activities have unique names, and locations are local to an activity.
A future is characterized by its identifier $f_i$, the source activity $\alpha$ and the destination activity $\beta$ of the corresponding request, thus we'll have futures of this form: $f_i^{\alpha\to\beta}$. Future identifiers must be chosen so as to be unique.

A parallel configuration is a set of activities:

$$P, Q ::= \alpha\Big[a; \sigma; \iota; F; R; f\Big] \Bigg|\Bigg| \beta\Big[\cdots\Big] \Bigg|\Bigg| \cdots$$

where $\alpha \in P$ denotes the fact that an activity named $\alpha$ belongs to the configuration $P$, in which its terms may be defined as follows:

- The first term $a$ is the active object of the activity.
- The second term $\sigma$ is the store containing all the objects of the activity to which it belongs.

- The third term $\iota$ is the active object location. The active object of the activity itself -$a$- may be denoted also by $\sigma(\iota)$.

- The fourth term $F$ is a function mapping for each served request a location $\iota$ to its future $f_i : F = f_i \mapsto \iota$. Infact, the value of the future $f_i$ is the part of the store that has $\iota$ for root. $F :: \{f_i \mapsto \iota\}$ adds a new future association to the set of future values.

- The fifth term $R$ is a list of pending requests $R = \{[m_j; \iota; f_i^{\gamma \to \alpha}]\}$. Each request consists of the name of the target method $m_j$, the location $\iota$ of the argument passed to the request and the future identifier which will be associated to the result: $f_i^{\gamma \to \alpha}$.
  $R :: r$ adds a request $r$ at the end of the request queue $R$, while $R' :: r :: R$ matches a queue containing the request $r$.

- The sixth and last term of a configuration $f$ is the current future, i.e. the future associated with the request currently served; more precisely, if the current term is $a \Uparrow f_i^{\gamma \to \alpha}, b$ then $f$ will be the future associated with the value computed by $a$.

An initial configuration consists of a single activity - we could call it the "main" activity - with the user program $a$ as current term, thus giving $\alpha[a; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset]$. This activity never receives any request, it communicates by sending requests, creating activities or receiving replies.

In order to understand the parallel reduction rules, some additional operators must be explained first:

- The operator $copy(\iota, \sigma)$ creates a store containing the deep copy of $\sigma(\iota)$.

- The operator $Merge(\iota, \sigma, \sigma')$ merges two stores. It creates a new store, merging independently $\sigma$ and $\sigma'$ except for $\iota$ which is taken from $\sigma'$.

- The operator $Copy\&Merge(\sigma, \iota; \sigma', \iota')$ adds the part of $\sigma$ reachable from the location $\iota$ at the location $\iota'$ of $\sigma'$ avoiding collision of locations[7]. Said another way, first it's made a deep copy of the parts of $\sigma$ that are reachable from $\iota$, then this copy is added to the location $\iota'$ of $\sigma'$. Therefore, $Copy\&Merge(\sigma, \iota; \sigma', \iota') \triangleq Merge(\iota', \sigma', copy(\iota, \sigma)\{\{\iota \leftarrow \iota'\}\})$

Now that all the necessary instruments have been introduced, the reduction rules can be analyzed.

Keep in mind that in each rule the terms that are grey coloured are terms not relevant for the relative reduction, and thus they can be safely ignored.

LOCAL

$$\frac{(a,\sigma) \rightarrow_s (a',\sigma')}{\alpha\Big[a;\sigma;\iota;F;R;f\Big] \Big\| P \longrightarrow \alpha\Big[a';\sigma';\iota;F;R;f\Big] \Big\| P}$$

This rule is for local reductions, i.e. reductions that happens inside a certain activity. It somehow comprises all the sequential reduction rules already explained.

NEWACT

$$\frac{\gamma \text{ fresh activity} \quad \iota' \notin DOM(\sigma) \quad \sigma' = \{\iota' \mapsto AO(\gamma)\} :: \sigma \quad \sigma_\gamma = COPY(\iota_2,\sigma)}{\alpha\Big[\mathcal{R}[Active(\iota_2,m_j)];\sigma;\iota;F;R;f\Big] \Big\| P \longrightarrow \alpha\Big[\mathcal{R}[\iota'];\sigma';\iota;F;R;f\Big] \Big\| \gamma\Big[\iota_2.m_j();\sigma_\gamma;\iota_2;\varnothing;\varnothing;\varnothing\Big]}$$

This rule describes the creation of a new activity, possible by means of the operator $ACTIVE(a,m)$.
A new activity $\gamma$ containing the deep copy of the object $\sigma(\iota)$ is created, with empty future values and pending requests list. A reference to the created activity $AO(\gamma)$ is created in the activity $\alpha$ of the caller.
Keep in mind that other references to $\iota$ in $\alpha$ are still pointing to the passive object.
$m_j$ specifies the method run initially by the active object of the new activity.

REQUEST

$$\frac{\sigma_\alpha(\iota) = AO(\beta) \quad \iota'' \notin DOM(\sigma_\beta) \quad f_i^{\alpha\rightarrow\beta} \text{ new future} \quad \iota_f \notin DOM(\sigma_\alpha) \quad \sigma'_\beta = COPY\&MERGE(\sigma_\alpha,\iota';\ \sigma_\beta,\iota'') \quad \sigma'_\alpha = \{\iota_f \mapsto fut(f_i^{\alpha\rightarrow\beta})\} :: \sigma_\alpha}{\alpha\Big[\mathcal{R}[\iota.m_j(\iota')];\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha\Big] \Big\| \beta\Big[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta\Big] \Big\| P \longrightarrow \alpha\Big[\mathcal{R}[\iota_f];\sigma'_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha\Big] \Big\| \beta\Big[a_\beta;\sigma'_\beta;\iota_\beta;F_\beta;R_\beta :: [m_j;\iota'';f_i^{\alpha\rightarrow\beta}];f_\beta\Big] \Big\| P}$$

This rule describes an asynchronous method call between activities. A new request is sent from activity $\alpha$ to activity $\beta$ and a new future $f_i^{\alpha\rightarrow\beta}$ is created

to represent the result of the request. $\alpha$ stores a reference to this future and can continue its execution, while a request containing the name of the method, the location of a deep copy of the argument stored in $\sigma_\beta$ and the associated future is added to the end of the pending requests of the callee with $R_\beta : [m_j; \iota''; f_i^{\alpha \to \beta}]$.

SERVE

$$\frac{m_j \in M \qquad \forall m \in M, m \notin R}{\alpha\left[\mathcal{R}[Serve(M)];\sigma;\iota;F;R::[m_j;\iota_r;f']::R';f\right] \Big\| P \longrightarrow \alpha\left[\iota.m_j(\iota_r)\Uparrow f,\mathcal{R}\left[[]\right];\sigma;\iota;F;R::R';f'\right] \Big\| P}$$

To serve a new request the reduction of the current term is stopped and stored as a continuation. The activity is stuck until a matching request is found in the pending request queue.
Note that this means that if the method associated to the new request is not present in the pending request queue, the activity will be stuck forever!

ENDSERVICE

$$\frac{\sigma' = COPY\&MERGE(\sigma, \iota; \sigma, \iota')}{\alpha\left[\iota' \Uparrow f', a; \sigma; \iota; F; R; f\right] \Big\| P \longrightarrow \alpha\left[a; \sigma'; \iota; F :: \{f \mapsto \iota'\}; R; f'\right] \Big\| P}$$

This rule describes what happens when a treated request terminates its computation: the result of the request is associated to the current future $f$. The result is deep copied to prevent post-service modification of the value.

REPLY

$$\frac{\sigma_\alpha(\iota) = fut(f_i^{\gamma \to \beta}) \qquad F_\beta(f_i^{\gamma \to \beta} = \iota_f) \qquad \sigma'_\alpha = COPY\&MERGE(\sigma_\beta, \iota_f; \sigma_\alpha, \iota)}{\alpha\left[a_\alpha;\boldsymbol{\sigma_\alpha};\iota_\alpha;F_\alpha;R_\alpha;f_\alpha\right] \Big\| \beta\left[a_\beta;\boldsymbol{\sigma_\beta};\iota_\beta;\boldsymbol{F_\beta};R_\beta;f_\beta\right] \Big\| P \longrightarrow \alpha\left[a_\alpha;\boldsymbol{\sigma'_\alpha};\iota_\alpha;F_\alpha;R_\alpha;f_\alpha\right] \Big\| \beta\left[a_\beta;\boldsymbol{\sigma_\beta};\iota_\beta;\boldsymbol{F_\beta};R_\beta;F_\beta\right] \Big\| P}$$

When to the current future is associated the result of the request, the future reference is replaced by the part of the store associated with the result, i.e. by the deep copy of the location associated to $f_i^{\gamma \to \beta}$.

It is deliberately only required that an activity contains a reference to a future, and another one has calculated the corresponding result. Therefore, the moment in which this rule should be applied is purposely not specified. However, please consider that a wait-by-necessity can only be resolved by the update of the future value, which constraints the moment when this rule can be applied.

## 3.4 A Deadlock Example in ASP

In this section we present an example of deadlock in a ASP program; the example's purpose is both to practically illustrate how deadlocks may arise in ASP and at the same time to give the reader an example of a program written in this language.

To somehow connect to what already said in section 2.5, the presented program calculate the factorial function of a number given as input; for the sake of the example we include primitive types `int` and conditional into the ABS syntax; furthermore, to keep things as readable as possible, we suppose that the term `Active(a,m)` returns the value computed by `m`.

The program is defined as follows:

```
let WHAT = [w = 0; what = C(t,b)let w = b.read() in return w] in
let FACT = [p = 1;
            fact = C(t,x)
                if (x==0)
                    then let p = Active(WHAT,what(t)) in return p
                else let p = Active(FACT,fact(x-1)) in return x.p;
            read = C(t,-)return t.p] in
Active(FACT,fact(input))
```

where `C` is the binder $\zeta$.

The program consists mainly in a recursive invocation of the function `fact`; for each of these invocations, a new activity is created, with an object FACT as the active one.
If $n$ is the current value for which to calculate the factorial fucntion, each `fact` invocation represents the computation of the factorial value of $n-1$. The return value, i.e. the return value of the function $fact(n-1)$, is subsequently used to calculate the actual factorial value, i.e. $n * fact(n-1)$.

However, because to the yet-not-calculated return value of the function $fact(n-1)$ is actually associated a future, the calculus of the actual factorial value is a strict operation on a future, thus a *wait-by-necessity* comes into play.

The program is actually fated to be delayed forever, because when $n$ reaches 0 the relative activity instead of simply returning the value 1 as its factorial it creates, with the same pattern as before, another activity to which it depends by making a strict operation on a future; this activity, in turn, depends on the caller activity to be able to terminate, but the latter will never reply because it's already in a wait-by-necessity state.

Therefore, the program is irremediably stuck in a deadlocked state.

# Chapter 4

# Different ways to approach the analysis

In this chapter we briefly present some ideas about how may be possible to extend the deadlock analysis to object-oriented languages and to ASP in particular.

The first hypothesis is to develop from scratch a type system for ASP that includes some sort of contracts, i.e. a type inference system able to generate - if the need arises - dependencies between activities.
In this case a complete subject-reduction proof will be needed to confirm the correctness of the type inference system and to be able to assert that it's consistent with the deadlock analysis.

The second hypothesis, instead, aims to modify coreABS$^{--}$ so as to model the concurrency as it is in ASP.
To do so it's necessary to extend the subject-reduction proof[1] already developed for coreABS$^{--}$ in a way so as to assert that if the proof is valid for a modified coreABS$^{--}$, then it's also valid for ASP.

## 4.1 A Type System for ASP

Here we describe a - not proven - type inference system for ASP that includes the generation of dependencies, when needed.
Being activities the ASP's actors of concurrency, the dependencies will be between them, in opposition to what happens in coreABS$^{--}$, where dependencies are between objects.

The fundamental purpose of a type system is to prevent the occurrence of execution errors during the running of a program[11].
We start by saying that the description of a type system starts with the description of a collection of formal utterances called judgments, where a typical judgment has the form:

$$\Gamma \vdash \Im$$

where $\Gamma$ is an *environment*, i.e. a function that associates names to types, for example:

$$\begin{aligned} &\Gamma : \\ &\quad a : T \\ &\quad b : T1 \\ &\quad z : T \\ &\quad \alpha : T2 \end{aligned}$$

and $\Im$ is an assertion where the free variables of $\Im$ are declared in $\Gamma$.
We say that $\Gamma$ *entails* $\Im$.

A typical *typing judgement* asserts that a term $a$ has type $T$ in the environment $\Gamma$ and it has the form:

$$\Gamma \vdash a : T$$

On the other hand, when we encounter the judgement $\Gamma \vdash T$ it means that the type $T$ is *well-formed* in the environment $\Gamma$, i.e. it's a valid type with respect to the environment.

An often encountered judgement is also $\Gamma \vdash \diamond$, which means that the environment $\Gamma$ is well-formed, i.e. it has been properly constructed.
This is significant because any given judgement can be regarded as *valid* or *invalid.*

The first type rule we introduce is

EMPTY

$$\frac{-}{\varnothing \vdash \diamond}$$

This rule simply states that every empty environment is a well-formed environment.

VAR-1

$$\frac{-}{\Gamma \vdash x : \Gamma(x)}$$

This rule simply states that there's no need of premises in order for the environment to express a valid judgement on one of its variables.

VAR-2

$$\frac{\Gamma \vdash \mathtt{T} \qquad x \notin DOM(\Gamma)}{\Gamma, x : \mathtt{T} \vdash \diamond}$$

This rule models the addition of a type relation to the environment.

FUN

$$\frac{\Gamma \vdash Y \qquad \Gamma \vdash M}{\Gamma \vdash (Y \rightarrow M)}$$

This rule express the logic on which the type function is based.

The four rules just presented represents a basic set of rules on which to build the subsequent ones.

OBJ-1

$$\frac{\Gamma \vdash B_i \qquad \Gamma \vdash (Y_j \rightarrow M_j) \quad \Big|_{j \in 1..m}^{i \in 1..n}}{\Gamma \vdash \Big\{ l_i : B_i ; m_j : Y_j \rightarrow M_j \Big\}_{j \in 1..m}^{i \in 1..n} \equiv \mathtt{T}_{obj}, -}$$

The first type rule for the object type states that the object is well-typed in the environment if all the types of its fields and methods are well-typed. What happens in the same time is that we assert how the type object is composed.
Note that we use $\mathtt{T}_{obj}$ as a shortening for the type object.

OBJ-2

$$\frac{\Gamma \vdash b_i : B_i \qquad \Gamma, x_j : (\mathtt{T}_{obj}, \tau), y_j : Y_j \vdash a_j : M_j \qquad \text{This_Activity} = \tau}{\Gamma \vdash \left[ l_i = b_i ; m_j = \zeta(x_j, y_j) a_j \right] : \mathtt{T}_{obj}, \tau}$$

In this rule we assert that an object has type object and we associate to every type object the activity to which it belongs.
This last information is crucial to ensure our ability in tracking dependencies between activities.

ACT

$$\frac{\Gamma \vdash a : \mathtt{T}_{obj}, - \qquad \tau \text{ fresh}}{\Gamma \vdash Active(a, m) : \mathtt{T}_{obj}, \tau}$$

The type of `Active` describes the fact that when a new activity is created its name becomes associated to the type of the active object that is deep copied within the activity itself.
We need to track the fact that we've created a new activity, and more specifically we've created a new activity name.
Furthermore, to every object we associate the activity to which it belongs to.

AMC

$$\frac{\Gamma \vdash a : \mathtt{T}_{obj}, \tau \qquad \Gamma \vdash y_j : Y_j \qquad \text{This_Activity} = \alpha \qquad AO(\tau) = a}{\Gamma \vdash a.m_j(y_j) : Fut(M_j), \alpha \to \tau}$$

It's here first introduced the type Future, which is associated to a result of a not-yet-computed return value of an asynchronous method call between activities.
To the future is also associated a relation between the caller and the callee activities' names, coherently to what is defined in the ASP semantics in section 3.3.
We associate here a relation, not a dependency, because we don't describe here the typing of a strict operation on a future; instead, we'll use this information to model further dependencies.

FA-0

$$\frac{\Gamma \vdash a : \mathtt{T}_{obj}, -}{\Gamma \vdash a.l_i : B_i}$$

This rule about the typing of a simple field access is self explanatory.

FA-1

$$\frac{\Gamma \vdash f : Fut(\mathtt{T}_{obj}), \alpha \rightarrow \tau \quad \text{This_Activity} = \alpha}{\Gamma \vdash f.l_i : B_i.(\alpha, \tau)}$$

This rule models the typing of a strict field access on a future.
If the activity that have called the asynchronous method call accesses the future variable with a strict operation, then a dependency is added and its information is took from the relation between activities associated to the Future type.

FA-2

$$\frac{\Gamma \vdash f : Fut(\mathtt{T}_{obj}), \alpha \rightarrow \tau \quad \text{This_Activity} = \beta}{\Gamma \vdash f.l_i : B_i.(\beta, \alpha).(\alpha, \tau)}$$

This rule also models the typing of a strict field access on a future, but it does so in the case that the activity that accesses the future it's different from the activity that have called the asynchronous method call.
This is possible in ASP because futures, like active object references, may be referenced from any object, thus from any activity.

In this case and in any other similar case that follows, before the dependency inferred from the information already associated to the type Future we add a further dependency between the activity that do the strict operation and the activity that have created the Future. This because, for a generic activity $\beta \neq \alpha$, $\beta$ is in a wait-by-necessity state waiting for $\alpha$ to retrieve its value from $\tau$.

Note that we intentionally chosen to not accept a direct dependency between $\beta$ and $\tau$ because in the ASP semantics defined in section 3.3 a computed value is replied only to the activity that made the request.

Now, a general consideration: a future may be referenced by any activity, but the same is not true when the future becomes a value; infact, the future may even refer to a field of a passive object, absolutely unaccessible by an external activity.

FU-0

$$\frac{\Gamma \vdash b : B_i \quad \Gamma \vdash a : \mathtt{T}_{obj}, -}{\Gamma \vdash a.l_i := b : \mathtt{T}_{obj}, -}$$

This is the basic rule for the typing of a field update operation.

FU-1

$$\frac{\Gamma \vdash f : Fut(\mathtt{T}_{obj}), \alpha \rightarrow \tau \quad \Gamma \vdash b : B_i \quad \text{This_Activity} = \alpha}{\Gamma \vdash f.l_i := b : \mathtt{T}_{obj}.(\alpha, \tau)}$$

This rule models the typing of a strict field update operation on a future made by the activity that performed the asynchronous method call.

FU-2

$$\frac{\Gamma \vdash f : Fut(\mathtt{T}_{obj}), \alpha \rightarrow \tau \quad \Gamma \vdash b : B_i \quad \text{This_Activity} = \beta}{\Gamma \vdash f.l_i := b : \mathtt{T}_{obj}.(\beta, \alpha).(\alpha, \tau)}$$

This rule models the typing of a strict field update operation on a future made by an activity different by the one that performed the asynchronous method call.

MC-0

$$\frac{\Gamma \vdash a : \mathtt{T}_{obj}, - \quad \Gamma \vdash y_j : Y_j}{\Gamma \vdash a.m_j(y_j) : M_j}$$

This is the basic rule for the typing of a method call operation.

MC-1

$$\frac{\Gamma \vdash f : Fut(\mathtt{T}_{obj}), \alpha \rightarrow \tau \quad \Gamma \vdash y_j : Y_j \quad \text{This_Activity} = \alpha}{\Gamma \vdash f.m_j(y_j) : M_j.(\alpha, \tau)}$$

This rule models the typing of a strict method call operation on a future made by the activity that performed the asynchronous method call.

MC-2

$$\frac{\Gamma \vdash f : Fut(\mathtt{T}_{obj}), \alpha \rightarrow \tau \qquad \Gamma \vdash y_j : Y_j \qquad \text{This_Activity} = \beta}{\Gamma \vdash f.m_j(y_j) : M_j.(\beta, \alpha).(\alpha, \tau)}$$

This rule models the typing of a strict method call operation on a future made by an activity different by the one that performed the asynchronous method call.

CLO-0

$$\frac{\Gamma \vdash a : \mathtt{T}_{obj}, -}{\Gamma \vdash Clone(a) : \mathtt{T}_{obj}, -}$$

This is the basic rule for the typing of a CLONE operation.

CLO-1

$$\frac{\Gamma \vdash f : Fut(\mathtt{T}_{obj}), \alpha \rightarrow \tau \qquad \text{This_Activity} = \alpha}{\Gamma \vdash Clone(f) : \mathtt{T}_{obj}.(\alpha, \tau)}$$

This rule models the typing of a strict CLONE operation on a future made by the activity that performed the asynchronous method call.

CLO-2

$$\frac{\Gamma \vdash f : Fut(\mathtt{T}_{obj}), \alpha \rightarrow \tau \qquad \text{This_Activity} = \beta}{\Gamma \vdash Clone(f) : \mathtt{T}_{obj}.(\beta, \alpha).(\alpha, \tau)}$$

This rule models the typing of a strict CLONE operation on a future made by an activity different by the one that performed the asynchronous method call.

ACF

$$\frac{\Gamma \vdash a : Fut(\mathtt{T}_{obj}), \alpha \rightarrow \beta \qquad \tau \text{ fresh}}{\Gamma \vdash Active(a, m) : \mathtt{T}_{obj}.(\alpha, \beta), \tau}$$

This rule is necessary in order to type an Active operation with a future passed as its active object.

LET-0

$$\frac{\Gamma \vdash a : A \quad \Gamma, x : A \vdash b : B \quad x \notin FV(b)}{\Gamma \vdash let\ x = a\ in\ b : B}$$

This is the basic rule for the typing of a LET concatenation.

LET-1

$$\frac{\Gamma \vdash a : Fut(A), \alpha \rightarrow \beta \quad \Gamma, x : A \vdash b : B \quad x \notin FV(b)}{\Gamma \vdash let\ x = a\ in\ b : B.(\alpha, \beta)}$$

If $a$ is a type Future, then a dependency will be added to the type of the sequential composition because $b$ will use $a$.

LET-2

$$\frac{\Gamma \vdash a : A \quad \Gamma, x : A \vdash b : Fut(B), \alpha \rightarrow \beta \quad x \notin FV(b)}{\Gamma \vdash let\ x = a\ in\ b : B.(\alpha, \beta)}$$

On the contrary, if $b$ is a type Future, then a dependency will be added to the type of the sequential composition because $b$ it's evaluated.

LET-3

$$\frac{\Gamma \vdash a : Fut(A), \alpha \rightarrow \beta \quad \Gamma, x : A \vdash b : Fut(B), \tau \rightarrow \gamma \quad x \notin FV(b)}{\Gamma \vdash let\ x = a\ in\ b : B.(\alpha, \beta).(\tau, \gamma)}$$

From the two former rules it follows that if both $a$ and $b$ have type Future, then two different dependencies must be added.

LET-4

$$\frac{\Gamma \vdash a : A \quad \Gamma, x : Fut(A) \vdash b : B \quad x \notin FV(b)}{\Gamma \vdash let\ x = a\ in\ b : B}$$

Note that the case in which $x$ is a type Future is not relevant; infact, $x$ will be substituted by $a$, and only the type of the latter will matter for the sequential composition type.

SRV

$$\frac{m_k{=}Q(a) \quad k{\in}1..n \quad a.m_k(y_k){:}Y_k{\to}Fut(M_k),\alpha{\to}\beta \quad a{:}\mathtt{T}_{obj} \quad \text{This_Activity}{=}\beta}{\Gamma \vdash Serve(m_1...m_n)^{n\geq 1} : M_k}$$

This rule simply states the resulting type of a SERVE operation, with no dependencies ever necessary.

## 4.2 A Different coreABS$^{--}$

The type system just presented has the not negligible inconvenience that, because for the type system to be *sound*, the absence of execution errors must hold for all of the program runs that can be expressed within the language; therefore, a new proof should be completely built from scratch.

Here we briefly ilustrate a simpler alternative, that may bring to the same conclusions, i.e. the deadlock analysis can be performed on ASP.

In coreABS$^{--}$ is already present a subject-reduction proof[1] that demonstrates the type soundness of the relative type system. The idea is to modify coreABS$^{--}$ so as to make it behave - at the concurrent level - as ASP, thus without explicit synchronizations but with implicit synchronization by means of waits-by-necessity.

We show here only the principles that may allow to completely prove the hypothesis and the differences between coreABS$^{--}$ and its modified counterpart, so as to not bore the reader with pages of repeated rules that are substantially the same of those of coreABS$^{--}$.

The syntax of coreABS$^{--}$ becomes:

$$\begin{array}{rcl} CL & ::= & class\ C\ extends\ C\ \{\bar{T}\ \bar{f}\ ;\ \bar{M}\} \\ M & ::= & T\ m\ (\bar{T}\ \bar{x})\{return\ e;\} \\ e & ::= & x \mid e.f \mid e!m(\bar{e}) \mid new\ C(\bar{e}) \mid e; e \\ T & ::= & C \mid Fut(T) \end{array}$$

As the reader may have noticed, the syntax is exactly the same of the one of coreABS$^{--}$, except for the `get` and `await` operations, which are eliminated. Infact, because we want to model a implicit synchronization pattern, the two

explicit synchronization primitives must be erased from the language.

For what semantics is concerned, by eliminating `get` and `await` it automatically arises the need to differentiate tasks in all the strict operations, i.e. Field Access and Method Invocation.

For illustrating purposes, we simplify the transition relation $\mathtt{t} :_a^\ell \mathtt{e}$ from section 2.3 with $\mathtt{t} : \mathtt{e}$ and we merge the two environment contexts E and S in a unique environment context E.

Thus, we may re-write the rule

$$\text{(Field-Var)}\quad \frac{\mathtt{M}(\mathtt{l}) = \mathtt{v}}{\mathtt{t},\mathtt{M} :_a^\top \mathsf{S}[\mathsf{E}[\mathtt{l}]] \xrightarrow{a} \mathtt{t},\mathtt{M} :_a^\top \mathsf{S}[\mathsf{E}[\mathtt{v}]]}$$

as (FIELD)

$$\frac{f : v \in \bar{f} : \bar{v}}{\mathtt{t} :_a^\top E[b[\bar{f} : \bar{v}].f] \xrightarrow{a} \mathtt{t} :_a^\top E[v]}$$

The logic to follow in order to modify the semantics rules according to the behavior we want to specify is:

$$t : E[t'], t' : v \;\longrightarrow\; t : E[v]$$

where $t$ and $t'$ are tasks, $E$ is the environment and $v$ is a value.

Therefore, it may be written the following rule that models a *strict* Field Access,

(FIELD-MOD)

$$\frac{f : v \in \bar{f} : \bar{v}}{t : E[t'.f], t' : E\Big[b[\bar{f} : \bar{v}]\Big] \;\longrightarrow\; t : E[v]}$$

As you can see now there are two tasks in the bottom left part of the rule, where the task $t$ depends on the evaluation computed by the task $t'$.
The rule models the fact that the evaluation may proceed as long as to the

task $t'$ is associated a computed value; this implies that if $t'$ is still something else - a future for example - no transition will ever occur.

The other strict operation rules, i.e. *strict* Method Invocation, may be written following the same logic.

In the new type system, the (T-GET) rule described in section 2.4 is now applied to the case in which a strict operation occurs on a type future, thus generating the associated contract via an implicit synchronization mechanism.

By modifying in this way syntax, semantics and type system it's possible to prove the adaptability to ASP of the deadlock analysis developed for coreABS$^{--}$ just by slightly modifying the subject-reduction rule[1] of the latter, therefore avoiding the construction from scratch of a proof for a completely new type system as the one previously discussed.

# Conclusions

In this dissertation we confronted two concurrent, object-oriented languages focusing our study on a static program analysys' theory for deadlock detection.
In order to favor the reader's understanding, deadlock examples for both languages have been presented and explained.

The deadlock analysis' theory has been presented in concomitance with the class-based language it has been originally developed for, and the first step of the analysis has been thoroughly explored so as to present the reader with the necessary information for the subsequent discussion.

Then a purely object-based language has been studied in detail and confronted with the aforementioned one, so as to show - explicitly - how subtle differences in the theoretical foundations languages are built from may evolve in significant differences in their behavior and how this may entail even greater hassles when the goal is to build a unique, comprehensive theory for them.

On the other hand, in order to show how it can be possible to extend the already existing deadlock analysis' theory to languages it has not been originally thought for, two significantly different hypotheses has been presented, pointing out at the same time the distinct ways that are necessary to prove them right.

# References


[1] C. Laneve, E. Giachino, T. Lascu, *Deadlock and Livelock Analysis in Concurrent Objects with Futures*, Submitted, December 2011

[2] F. Nielson, H. R. Nielson, C. Hankin, *Principles of Program Analysis*, Springer, 2005

[3] N. Kobayashi, *A new type system for deadlock-free processes*, In Proc. CONCUR 2006, volume 4137 of LNCS, pages 233–247; Springer, 2006

and

C. Laneve, L. Padovani, *The must preorder revisited*, In Proc. CONCUR 2007, volume 4703 of LNCS, pages 212–225; Springer, 2007

[4] L. Cardelli, *Object-based vs. Class-Based Languages*, PDLI'96 Tutorial, Digital Equipment Corporation Systems Research Center, 1996

[5] M. Felleisen, R. Hieb, *The Revised Report on the Syntactic Theories of Sequential Control and State*, Theoretical Computer Science, 1992

[6] E. Giachino, T. Lascu, *Lock Analysis for an Asynchronous Object Calculus*, Presented at ICTCS, 2012

[7] D. Caromel, L. Henrio, B. P. Serpette, *Asynchronous sequential processes*, Information and Computation, Volume 207 Issue 4, Pages 459-495 2009

[8] M. Abadi, L. Cardelli, *A Theory of Objects*, Springer-Verlag, New York, 1996

[9] F. Turbak, D. Gifford, *Design concepts in programming languages*, MIT press, p. 251, 2008

[10] E. Giachino, C. Laneve, *A beginner's guide to the deadLock Analysis Model*, In TGC'12, Lecture Notes in Computer Science, Springer, 2013

[11] L. Cardelli, *Type Systems*, Handbook of Computer Science and Engineering, Chapter 103. CRC Press, Digital Equipment Corporation Systems Research Center, 1997