ASP_{fun}: A Typed Functional Active Object Calculus

Ludovic Henrio

CNRS – I3S – Univ. Nice Sophia-Antipolis – INRIA, Sophia-Antipolis, France

Florian Kammüller

Middlesex University, London UK and Technische Universität Berlin, Germany

Bianca Lutz

Technische Universität Berlin, Germany

Abstract

This paper provides a sound foundation for autonomous objects communicating by remote method invocations and futures. As a distributed extension of ς-calculus we define ASP_{fun}, a calculus of functional objects, behaving autonomously and communicating by a request-reply mechanism: requests are method calls handled asynchronously and futures represent awaited results for requests. This results in an object language enabling a concise representation of a set of active objects interacting by asynchronous method invocations. This paper first presents the ASP_{fun} calculus and its semantics. Then, we provide a type system for ASP_{fun} which guarantees the "progress" property. Most importantly, ASP_{fun} has been formalised; its properties have been formalised and proved using the Isabelle theorem prover and we consider this as an important step in the formalization of distributed languages. This work was also an opportunity to study different binder representations and experiment with two of them in the Isabelle/HOL theorem prover.

Keywords: Theorem proving, object calculus, futures, distribution, typing, binders.

1. Introduction

This paper presents a functional active object language featuring asynchronous method calls and futures; it has been formalised in the Isabelle/HOL theorem prover. ASP_{fun} (asynchronous sequential processes) is an extension of the ς-calculus (Abadi and Cardelli, 1996) where objects are distributed into several activities, and activities are the units of distribution. Communications toward activities are asynchronous (remote) method calls; and futures are identifiers for the result of such asynchronous invocations. A future represents

Email addresses: Ludovic.Henrio@inria.fr (Ludovic Henrio), f.kammueller@mdx.ac.uk, flokam@cs.tu-berlin.de (Florian Kammüller), sowilo@cs.tu-berlin.de (Bianca Lutz)

an evaluation-in-progress in a remote activity. Futures can be transmitted between activities as any object: several activities may refer to the same future. The calculus is said to be functional because method update is realised on a copy of the object: there is no side-effect. The paper also studies a type system for active objects. Typing is a well studied technique (Pierce, 2002); we prove here a classical typing property, progress, in unusual settings, distributed active objects.

We mechanically proved properties about ASP_{fun} and, since the calculus is abstract, our semantics and mechanisation can be a basis for the analysis of related languages. Distributed active objects represent an abstract notion of concurrently computing and communicating activities. Clearly, finding a combination of objects and concurrency is not new as a notion – related notions are summarized in the following paragaraph – but providing a fully formalized and mechanized calculus including typing for this combination is. Mechanical proofs, though more difficult to perform, are more reliable because they should contain no error. This article shows that theorem proving techniques can handle distributed features of programming languages. Our work is an important step toward the mechanisation of calculi for distributed computing. The calculus is a model for distributed frameworks relying on active objects or on actors as explained below.

Object and Distribution: The Active Object Model

The underlying principle for distribution considered in this paper originates from Actors (Agha, 1986; Agha et al., 1997). Our calculus provides a model of computations that are distributed in the same way as the actor or the active object paradigm. In those paradigms, distributed computation relies on absence of sharing between processes allowing them to be placed on different machines. Those models feature asynchronous RMI-like communications. We detail below some characteristic distributed languages adhering to those principles.

Principles of actors are the following. Each actor is an independent functional process, i.e., an object together with its own thread. Actors interact by asynchronous message passing. They receive messages in their inbox and process them asynchronously. Instead of having an internal state, actors can change their behaviour, i.e., their reaction to received messages. Actors are some form of active objects. Our approach is to take distribution and parallelism notions similar to actors but fit them into a calculus of classical objects. This article introduces a formalisation, both on paper and in a theorem prover, of actor paradigms in the context of ς-calculus.

From the original actor paradigm (Hewitt et al., 1973; Agha, 1986; Agha et al., 1997), several languages have been designed. Some languages directly feature actors, distributed active objects (like the ProActive (Caromel et al., 2006) library), or other derived paradigms. The calculus ASP_{fun} provides a simple model for such languages.

The ASP calculus (Caromel et al., 2004; Caromel and Henrio, 2005) provides understanding and proofs of confluence for asynchronous distributed systems; it is a formalisation of the active object model. In ASP, active objects communicate in an actor-like manner. Additionally, ASP uses *future* objects, i.e., objects for which the real value is being calculated. Syntactically, the ASP calculus is an extension of the $\mathbf{imp}\varsigma$-calculus (Abadi and Cardelli,

1996; Gordon et al., 1997) with two primitives (*Serve* and *Active*) to deal with distributed objects.

An active object is similar to an actor in the sense that it has a request queue (corresponding to the actor's mailbox), it does not share memory with other active objects, and active objects communicate by messages. For active objects, communications take the form of a remote method invocation that will be treated asynchronously. We call *activity* the set consisting of an active object, its request queue, the set of normal (also called *passive*) objects known by the active objects, and the set of results the active object has computed. Each active object has a *single* thread; only this thread is allowed to access the active object and the passive ones.

Proactive (Caromel et al., 2006) is a Java middleware for distributed computing. It is based on the notion of active objects and is considered as an implementation of the ASP calculus. It is particularly designed for large scale distributed computations (clusters, Grids, or cloud computing). Deployment is based on the notion of virtual nodes and deployment descriptors: when an activity is created, it is associated with a virtual node, and a deployment descriptor file associates virtual nodes to real machines. As active objects do not share memory they provide a good abstraction of location. Finally, an active object is uniquely associated to a location and an application thread (even if several active objects can be placed on the same machine in practise). Active objects act as the unit of both concurrency and distribution. In ProActive, the programmer only cares about splitting its computation into independent active objects that will run in parallel; then the localisation aspect is delegated to a different role: the deployer. It is a key feature of the programming language and the middleware to guarantee that the program behaves the same whatever physical locations are chosen to deploy the active objects.

Also, the Creol (Johnsen et al., 2006) language features futures with (multi)-active objects; distribution principles in Creol are quite similar to ASP_{fun} except that Creol is an imperative language with a more complex semantics. Johnsen et al. (2006) also advocate the active object paradigm as a model of distributed computation: "The Creol model targets distributed objects by a looser coupling of method calls and synchronization." The mechanised formalisation of an active object language is a major contribution of this paper. Such a formalisation will increase the confidence in the properties of this programming model and our understanding of distributed computation.

Contribution

We define in this paper ASP_{fun}, a calculus of functional active objects with futures. It formalises the notion of active objects presented in the previous paragraph. For example, the behaviour of ProActive active objects follows quite faithfully the semantics of ASP_{fun}, and thus properties proved here can be transferred to this context. Compared to imperative ASP, ASP_{fun} investigates the typing of active objects and ensures progress properties in a functional context.

The language, its type system, and all properties have been completely formalised (`http://gforge.inria.fr/scm/viewvc.php/ASPfun/?root=tods-isabelle`) and proved in Isabelle/HOL (Nipkow et al., 2002). This formalisation is approximatively 14000 lines,

only 10% dealing with the language definition, and the rest dealing with the proof of ASP_{fun} properties. We also believe that the formalisation of a calculus like ASP_{fun} in a theorem prover will be helpful in the future design of distributed languages and can provide a reliable basis for proofs using paradigms such as distributed objects, futures, remote method invocations, actors, or active objects. Our main contributions are:

- A functional active object calculus with futures and its properties. We illustrate the expressiveness of the calculus on a couple of examples.
- A type system for active object languages.
- An investigation on how to provide a type-safe calculus featuring active objects and futures, where typing ensures progress.
- A formalisation of those features in a theorem prover, that will allow further investigations on futures, typing, and active objects paradigms.
- A comparison of different techniques for representing binders together with two implementations of our framework using two different techniques.

ASP_{fun} is the first calculus to our knowledge to feature those characteristics, however each of those characteristics exists in some distributed programming language, and sometimes in other calculi. In this context, the main contribution of this paper is the formalisation of these features as a single calculus, but mainly the mechanised formalisation of this calculus in a theorem prover. This paper will provide a complete description of our formalisations, an analysis of the technical decisions that we have taken to represent distributed objects in Isabelle/HOL, and an overall conclusion on the techniques we used and the tools we provide.

This article is organised as follows. Section 2 presents ASP_{fun} and its semantics. Two examples illustrate the calculus in Section 3. Section 4 gives first properties of the calculus focusing on well-formed configurations and on the impossibility to create cyclic dependencies. Section 5 provides a type system for ASP_{fun} ensuring both subject-reduction and progress. Some details on the formalisation in Isabelle/HOL and on the major proofs are given in Section 6; this section particularly details binder representation. Section 7 discusses alternative semantics we could have chosen. Finally, Section 8 details our position relatively to existing distributed languages and calculi and Section 9 concludes by a summary of our achievements and a discussion of the properties of ASP_{fun} as presented in this paper.

2. Syntax and Semantics

This section presents the ASP_{fun} calculus. We first define its syntax and explain its principles. Then, we give a small-step operational semantics for the calculus.

2.1. Syntax

We use three sets of identifiers: the labels of ς-calculus methods (l_i), the activities (α,β,...), and the futures (f_i). Like in ς-calculus in ASP_{fun} every term is an object either given by its definition or returned by a term evaluation. The syntax of ASP_{fun} includes *object definition*, *method invocation*, and *method override* inherited from ς-calculus. An object consists of a set of labelled methods. A method is a function with two formal parameters: one represents *self*, i.e., the object in which the method is contained, the other, which is new in ASP_{fun}, is an actual parameter given at invocation time. Object fields are considered as degenerate methods not using the parameters. A method call is addressed to an object and receives an object as parameter. A method update acts on an object providing a new value for one method possibly defining it. ς-calculus terms are identified modulo *renaming of bound variables*.

$$
\begin{array}{rll}
s,t ::= & \underline{x} & \text{variable} \\
| & \underline{[l_j = \varsigma(x_j, y_j)t_j]^{j\in 1..n}} & (\forall j,\ x_j \neq y_j)\ \text{object definition} \\
| & \underline{s.l_i(t)} & (i \in 1..n)\ \text{method call} \\
| & \underline{s.l_i := \varsigma(x,y)t} & (i \in 1..n, x \neq y)\ \text{update} \\
| & \underline{\mathit{Active}(s)} & \text{Active object creation} \\
| & \alpha & \text{active object reference} \\
| & f_i & \text{future reference}
\end{array}
$$

Table 1: ASP_{fun} syntax

One of the basic principles of ASP_{fun} is to perform a minimal extension of the syntax of ς-calculus. ASP_{fun} programs only use one additional primitive, *Active*, for creating an active object. The syntax of ASP_{fun} is shown in Table 1; the static syntax (the programs) consists of only underlined constructs; future and active object references are created at runtime.

While the syntactic extension of ς-calculus is minimal, the semantics, that we will define in the following, is (almost) entirely new. For example, in Table 2, only the two first rules are an adaptation of ς-calculus' semantics; all the others are specific to ASP_{fun}.

2.2. Informal Semantics of ASP_{fun}

The semantics of the local object calculus is similar to the one of Abadi and Cardelli (1996). A method invocation reduces to the method body where formal parameters are replaced by actual ones: $[l = \varsigma(x,y)a].l(b)$ reduces to a where x is replaced by $[l = \varsigma(x,y)a]$ and y is replaced by b. A method update returns a new object replacing the original method by the one on the right side of ':='. We focus now on the distributed features of ASP_{fun}.

A *configuration* is a set of activities. Each activity possesses a single active object, which is a ς-calculus term. Activating an object, *Active*(s), means creating a new activity with the object s to be activated becoming an *active object*. It is immutable. The activity is the unit of distribution. A request sent to an activity is an invocation to the active object; it is processed by the activity. The set of requests processed by an activity is called *request queue* by similarity with the active object model but, here, as the calculus is functional, requests

can be treated in an unordered fashion. Indeed, as we do not have any side effect, the order of execution of request has no influence on the result.

Figure 1 illustrates the basic concepts of ASP_{fun}. It shows a configuration consisting of two activities. In each activity an ellipse represents the active object, and each rectangle is a request (i.e., maps a future identifier to a term being evaluated). In ASP_{fun}, all the requests can be evaluated in parallel.

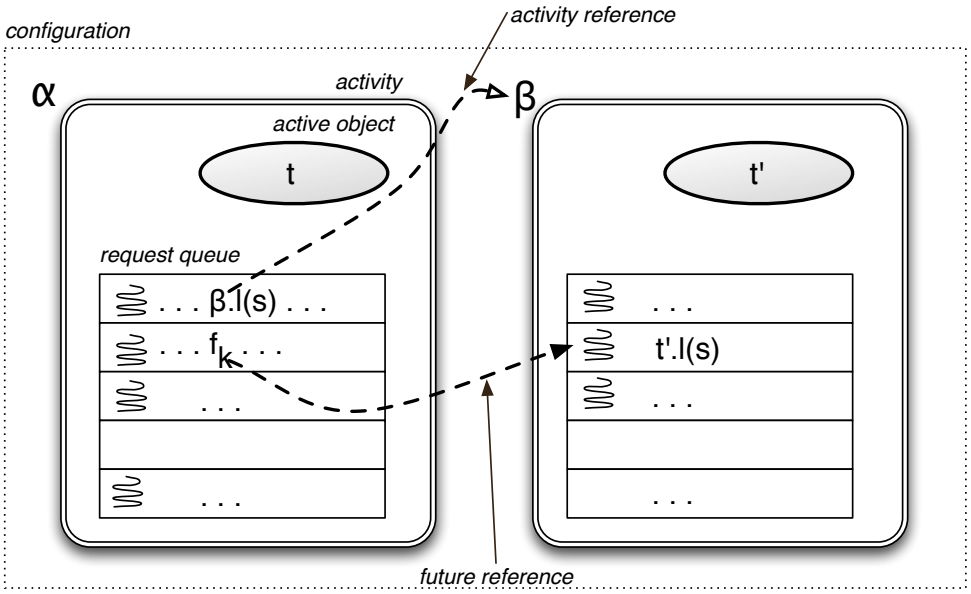

Figure 1: Example configuration in ASP_{fun} with two activities.

Every message sent toward an activity is a method call to the activate object. Such a *remote method invocation* (also called *request*) is asynchronous: the effect of this method call is – both – to create a new request in the request queue of the destination and to replace the original method invocation by a reference to the result of the created request. A reference to a (promised) result is called a *future*. In ASP_{fun}, futures are entities that can be passed to other activities, e.g., as arguments or results of requests; several activities may use the same future. Trying to access the result referenced by a future (e.g., invoking a method on it) is not possible until the future has been received. The current term of any request (even partially evaluated) can be returned at any moment: the current term for the request replaces the corresponding future. This operation is called a *reply*. We chose to allow replies with a partially evaluated term because it fits well with the functional nature of the calculus; but we will see in Section 7 that a more classical semantics returning only requests entirely evaluated also guarantees progress. Future values must be stored forever because future references can spread over the activities and, without a mechanism for counting the future references, it is impossible to know if a future reference still exists in the system. A garbage collection mechanism for future would detect whether a future is still referenced; garbage collection of futures is not studied in this paper.

Figure 1 can also be considered as an illustration of a method call: the configuration consisting of the first line of the request queues is transformed into the configuration consisting of the second lines of the request queues, the reference to the activity β is lost, and the reference to the future f_k is created together with a request computing f_k in β.

Reduction can occur in any request of any activity. The only restriction is that an object cannot be sent to another activity (e.g., as a request parameter) if this object has free variables, else such variables would escape their scope and the moved object would be meaningless. To better understand this restriction, suppose one tries to evaluate the subterm $\varsigma(x,y)\mathit{remoteObject.send}(y)$, which is the body of a method. Sending y first would be meaningless as this variable is bound by $\varsigma(x,y)$ and it would mean nothing in the remote object. We force to perform evaluation until the sent terms have no more free variables; in the example we would wait until the method is invoked with a parameter. Fortunately, the type system ensures that a term typed in an empty environment has no free variable, which is sufficient to guarantee that remote method invocations can be performed at some point of the reduction.

It is difficult to give a natural semantics to the update of an active object. Indeed the usual field update that directly modifies the value of an object field would create an additional way of communicating with an active object by changing its status without performing a method invocation. The functional nature of the calculus (updating an object creates a copy) oriented us toward the following solution: a method update on an active object creates a new activity with the method updated.

Proving confluence for $\mathrm{ASP}_{\mathrm{fun}}$ would lead to numerous technical difficulties and is out of the scope of this paper. Informally, depending on the execution, the set of created activities and the number of requests may vary, but the result of the computation is always the same. For example, depending on the order of execution an activity creation may precede or succeed a term duplication thus creating one or two activities. But if two activities are created, they are equivalent, and, as no side effect exists in $\mathrm{ASP}_{\mathrm{fun}}$, the two activities will always behave the same. A similar reasoning can be applied to the possibly duplicated requests. This explains why the calculus is confluent in the sense that it always produces equivalent results.

As a tiny example of the semantics $\mathit{Active}([l = \varsigma(x,y)[]]).l([])$ first creates an activity with the object $[l = \varsigma(x,y)[]]$, then performs a remote invocation on the method l of this activity (which creates a future), and finally replies replacing the future by the result of the invocation, $[]$. More formally, assuming $\mathit{Active}([l = \varsigma(x,y)[]]).l([])$ is being evaluating inside activity α for calculating the value for a future f_0 (notations will be detailed in the next section):

$$\begin{aligned}\alpha\,[(f_0 \mapsto \mathit{Active}([l = \varsigma(x,y)[]]).l([])), \ldots] &\to_{\parallel} \alpha\,[(f_0 \mapsto \beta.l([])) \ldots] \;\parallel\; \beta\,[\varnothing, [l = \varsigma(x,y)[]]] \\ &\to_{\parallel} \alpha\,[(f_0 \mapsto f_1) \ldots] \;\parallel\; \beta\,[(f_1 \mapsto []), [l = \varsigma(x,y)[]]] \\ &\to_{\parallel} \alpha\,[(f_0 \mapsto []) \ldots] \;\parallel\; \beta\,[(f_1 \mapsto []), [l = \varsigma(x,y)[]]]\end{aligned}$$

We consider this work as a reliable basis for further studies on stateless objects, giving a semantics for autonomous services, which in case they are stateless can be implemented such

that they never dead-lock (i.e., they always progress). ASP_{fun} can also represent component-like distributed systems interacting by invocation of services: an active object exposes its methods to the external world and holds references to required external services provided by other active object.

2.3. Small-Step Operational Semantics

The semantics of ASP_{fun} necessitates the definition of some structures that are used for the dynamic reduction. First, we define a configuration C as an unordered set of activities: a configuration is a mapping from activity identifiers to activities. Each activity is composed of a request queue (mapping from future identifiers to terms) and an active object (term). Configurations are identified modulo reordering of activities and of requests inside an activity.

$$C ::= \alpha_i[(f_j \mapsto s_j)^{j\in I_i}\ ,\ t_i]^{i\in 1..p} \quad \text{where } \{I_i\} \text{ are disjoint subsets of } \mathbb{N}$$

As futures are referenced from anywhere, two requests must correspond to two different futures; uniqueness is ensured in this paper by indexing futures over disjoint families. We use the term *local semantics* to refer to the semantics expressing the execution local to each *activity*, where an activity is the unit of distribution. Abadi and Cardelli (1996) present various ς-calculi that only consider objects and their manipulation as primitive; local semantics of ASP_{fun} (two first rules of Table 2) is just an adaptation of this work. More precisely, local semantics of ASP_{fun} extends ς-calculus with a second parameter for methods.

Classically we define contexts as expressions with a single hole ($\bullet$). $E[s]$ denotes the term obtained by replacing the single hole by s.

$$\begin{aligned} E ::= \ & \bullet \mid [l_i = \varsigma(x,y)E, l_j = \varsigma(x_j,y_j)t_j^{j\in(1..n)-\{i\}}] \mid E.l_i(t) \mid \\ & s.l_i(E) \mid E.l_i := \varsigma(x,y)s \mid s.l_i := \varsigma(x,y)E \mid \mathit{Active}(E) \end{aligned}$$

For a better integration with the distributed calculus, we choose a small-step semantics ($\rightarrow_\varsigma$) for the ς-calculus. It is composed of the two first rules of Table 2; one invokes a method (using the invoked object as first parameter), the other updates a method, i.e., it creates a new object where one method is replaced by a new one.

To simplify the reduction rules, we let $Q, R ::= (f_{ij} \mapsto s_{ij})^{j\in 1..n_p}$ range over request queues and identify mappings modulo reordering: $\alpha[f_i \mapsto s_i :: Q, b] :: C$ is a configuration containing the activity α which contains a request $f_i \mapsto s_i$, where C is the remainder of the configuration that cannot contain an activity α. Now, $\alpha[Q, s] \in C$ means: α is an activity of C with request queue Q and active object s: $\alpha[Q, s] \in C \Leftrightarrow \exists C'.\, C = \alpha[Q, s] :: C'$. Similarly, $(f_i \mapsto s) \in Q$ stands for: a request of Q associates s to the future f_i. The empty mapping is $\varnothing$; the domain of a mapping is dom; e.g., $\text{dom}(C)$ is the set of activities defined by C. Predicate $\mathit{noFV}(s)$ is true if s has no free variables (the only binder being ς this definition is classical). The parallel reduction $\rightarrow_\parallel$ on configurations is defined in Table 2.

Classically, the substitution $s\{x \leftarrow t\}$ is capture avoiding (renaming is performed to avoid free variables in t to be captured by binders in s), whereas the replacement of $\bullet$ by a term in a context is not.

CALL

$$\frac{l_i \in \{l_j\}^{j\in 1..n}}{E\left[[l_j = \varsigma(x_j, y_j)s_j]^{j\in 1..n}.l_i(t)\right] \to_\varsigma E\left[s_i\{x_i \leftarrow [l_j = \varsigma(x_j, y_j)s_j]^{j\in 1..n}, y_i \leftarrow t\}\right]}$$

UPDATE

$$\frac{l_i \in \{l_j\}^{j\in 1..n}}{E\left[[l_j = \varsigma(x_j, y_j)s_j]^{j\in 1..n}.l_i := \varsigma(x, y)t\right] \to_\varsigma E\left[[l_i = \varsigma(x, y)t, l_j = \varsigma(x_j, y_j)s_j^{j\in(1..n)-\{i\}}]\right]}$$

LOCAL

$$\frac{s \to_\varsigma s'}{\alpha[f_i \mapsto s :: Q, t] :: C \to_\| \alpha[f_i \mapsto s' :: Q, t] :: C}$$

ACTIVE

$$\frac{\gamma \notin (\mathrm{dom}(C) \cup \{\alpha\}) \qquad noFV(s)}{\alpha[f_i \mapsto E[Active(s)] :: Q, t] :: C \to_\| \alpha[f_i \mapsto E[\gamma] :: Q, t] :: \gamma[\varnothing, s] :: C}$$

REQUEST

$$\frac{f_k \text{ fresh} \qquad noFV(s) \qquad \alpha \neq \beta}{\alpha\,[f_i \mapsto E[\beta.l(s)] :: Q, t] :: \beta[R, t'] :: C \to_\| \alpha\,[f_i \mapsto E[f_k] :: Q, t] :: \beta\,[f_k \mapsto t'.l(s) :: R, t'] :: C}$$

SELF-REQUEST

$$\frac{f_k \text{ fresh} \qquad noFV(s)}{\alpha\,[f_i \mapsto E[\alpha.l(s)] :: Q, t] :: C \to_\| \alpha\,[f_k \mapsto t.l(s) :: f_i \mapsto E[f_k] :: Q, t] :: C}$$

REPLY

$$\frac{\beta[f_k \mapsto s :: R, t'] \in \alpha[f_i \mapsto E[f_k] :: Q, t] :: C}{\alpha[f_i \mapsto E[f_k] :: Q, t] :: C \to_\| \alpha[f_i \mapsto E[s] :: Q, t] :: C}$$

UPDATE-AO

$$\frac{\gamma \notin \mathrm{dom}(C) \cup \{\alpha\} \qquad noFV(\varsigma(x, y)s) \qquad \beta[R, t'] \in \alpha[f_i \mapsto E[\beta.l := \varsigma(x, y)s] :: Q, t] :: C}{\alpha[f_i \mapsto E[\beta.l := \varsigma(x, y)s] :: Q, t] :: C \to_\| \alpha[f_i \mapsto E[\gamma] :: Q, t] :: \gamma[\varnothing, t'.l := \varsigma(x, y)s] :: C}$$

Table 2: $\mathrm{ASP}_{\mathrm{fun}}$ semantics

- LOCAL performs a local reduction inside an activity: one step of the reduction $\to_\varsigma$ is performed on one request.
- ACTIVE creates an activity; the term s passed as argument to the Active primitive becomes the active object. The newly created activity receives a fresh activity identifier γ. Initially, the new activity has an empty request queue, and γ replaces the activation instruction $Active(s)$ thus allowing future invocations to this activity.
- REQUEST sends a request from the activity α to the activity β with $\alpha \neq \beta$. A new

request is created at the destination invoking the method l on the active object (t'); a fresh future f_k is associated to this request, and replaces the invocation on the sender side. Freshness is defined classically: f_k is fresh in C if $\forall\, \alpha[Q,t] \in C,\ f_k \notin \mathrm{dom}(Q)$.

- SELF-REQUEST is the REQUEST rule when the destination is the sender, $\alpha = \beta$. The semantics of this rule is similar to the preceding one but, as the request queue is modified on both the sender and the receiver side, it would be difficult to express a single simple rule for the two cases.

- REPLY updates a future: it picks the request calculating a value for the future f_k and sends the current value of this request (s) to an activity that refers to the future. The request may be only partially evaluated meaning a reply to a request is enabled as soon as the method invocation is performed. Returning partial replies can have the effect to duplicate computation and will be further discussed in Section 7. Necessarily, $noFV(s)$ holds because, as an active object and a transmitted parameter have no free variables, a request value never has free variables. This time, the structure of the rule avoids introducing a separate rule for $\alpha = \beta$.

- UPDATE-AO updates a method of an activity $\beta[R, t']$. It creates a new activity whose active object performs a (local) update on t': $t'.l := \varsigma(x,y)s$. It requires that the new method definition for l has no free variable.

The requirement $noFV(s)$ for the communicated terms is necessary. Indeed, communicating a term with free variables would cause variables to escape the scope of their binder as explained in Section 2.2. In Section 7, we will discuss the choices that have been made in the $\mathrm{ASP}_{\mathrm{fun}}$ semantics and the alternative possibilities.

2.4. Basic ς-calculus Datatypes

For reasons of completeness of this paper, we introduce here the definitions of standard datatypes in the ς-calculus (Abadi and Cardelli, 1996) that are used in this paper. They give a good illustration of encoding of basic datatypes in ς-calculus.

Booleans and conditional.

$$\begin{aligned} \text{true} &= [\, \mathit{if} = \varsigma(x,y)x.\mathit{then}(y), \mathit{then} = \varsigma(x,y)[], \mathit{else} = \varsigma(x,y)[]\,] \\ \text{false} &= [\, \mathit{if} = \varsigma(x,y)x.\mathit{else}(y), \mathit{then} = \varsigma(x,y)[], \mathit{else} = \varsigma(x,y)[]\,] \\ \text{if } b \text{ then } c \text{ else } d &= ((b.\mathit{then} := \varsigma(x,y)c).\mathit{else} := \varsigma(x,y)d).\mathit{if}([]) \end{aligned}$$

In the third line above, $x, y \notin FV(c) \cup FV(d)$; [] denotes the empty object. The definition shows how – similar to λ-calculus – the functionality of the constructor is encoded in the elements of the datatype: when b is true its method *if* delegates to the method *then*, filled with term c, when false, *if* delegates to *else*, executing term d.

Lists.

$$\begin{aligned} c :: l &= [hd = \varsigma(x,y)c, tl = \varsigma(x,y)l, mty = \text{false}\,] \\ hd\ l &= l.hd \\ tl\ l &= l.tl \\ \langle\rangle_{\text{list}} &= [hd = \varsigma(x,y)[], tl = \varsigma(x,y)[], mty = \text{true}\,] \\ l = \langle\rangle_{\text{list}} &= l.mty \end{aligned}$$

In the first line above, $x, y \notin FV(c) \cup FV(l)$; [] denotes again the empty object. Lists are encoded as accumulation of first elements in the head field *hd*; the predicate judging emptiness of a list is an abbreviation for the third field *mty* that always tracks whether a list is empty.

3. Examples

This section illustrates the ASP_{fun} calculus with two examples, one focusing on futures, and the other showing a few less conventional features of the calculus.

3.1. A Broker

The following example illustrates some of the advantages of futures for the implementation of services. The three activities hotel α, broker β, and customer γ are composed by $\|$ into a configuration (to improve readability, $\|$ separates the different activities in the examples). Here, the customer γ wants to make a hotel reservation in hotel α. He uses a broker β for this service by calling a method *book* provided in the active object of the broker. We omit the actual search of the broker β in his database and instead hardwire the solution to always contact some hotel α. That is, the method *book* is implemented as a call $\varsigma(x, date)\alpha.room(date)$ to a function *room* in the hotel α. Also the internal administration of hotel α is omitted; its method *room* just returns a constant bookingreference. Initially, only the future list of the customer γ contains a request for a booking to broker β; the future lists of α and β are empty. The following steps of the semantic reduction relation $\rightarrow_{\|}$ illustrate how iterated application of reduction rules evaluates the program.

$$\begin{aligned} &\gamma[f_0 \mapsto \beta.book(date), t] \\ \| \ &\beta[\varnothing, [book = \varsigma(x, date)\alpha.room(date), \ldots]] \\ \| \ &\alpha[\varnothing, [room = \varsigma(x,date)\text{bookingreference}, \ldots]] \end{aligned}$$

The following step of the semantic reduction relation $\rightarrow_{\|}^{*}$ creates the new future f_1 in β by rule REQUEST, this call is reduced according to LOCAL, and the original call in the client γ is replaced by f_1.

$$\begin{aligned} &\gamma[f_0 \mapsto f_1, t] \\ \| \ &\beta[f_1 \mapsto \alpha.room(date), \ldots] \\ \| \ &\alpha[\varnothing, [room = \varsigma(x,date)\text{bookingreference}, \ldots]] \end{aligned}$$

The parameter x representing the *self* is not used but the call to α's method *room* with parameter *date* creates again by rule REQUEST a new future in the request queue of the hotel activity α that is immediately reduced due to LOCAL to bookingreference.

$$\begin{array}{l} \gamma[f_0 \mapsto f_1, t]] \\ \| \ \beta[f_1 \mapsto f_2, \ldots] \\ \| \ \alpha[f_2 \mapsto \text{bookingreference}, \ldots] \end{array}$$

Finally, the result bookingreference is returned to the client by two REPLY-steps: first the future f_2 is returned from the broker to the client γ and then this client receives the bookingreference via f_2 directly from the hotel α.

$$\begin{array}{l} \gamma[f_0 \mapsto \text{bookingreference}, t] \\ \| \ \beta[f_1 \mapsto f_2, \ldots] \\ \| \ \alpha[f_2 \mapsto \text{bookingreference}, \ldots] \end{array}$$

The example is intentionally simplified to focus on the flow of control given by the requests, replies, and the passing on of the futures: the booking reference can flow directly to the customer γ possibly without passing by the broker β. This shows that futures allow the implementation of efficient communication flows. The example further illustrates how futures can be employed to provide some confidentiality. The broker β does not need to give away his data base of hotel references: he can instead return just a reference to the result of his negotiations; the booking reference.

3.2. A Service Provider

We illustrate how ASP_{fun} can be used to implement a (generic) service detailing on the control structure and the service administration while abstracting the actual service content. Eventually, we use the informal description "some function on *client_data*" to denote the final function representing the service. What we are interested in is the global service architecture. We want to show how the active object update – generating a new object on update – can be employed efficiently to support creation and delegation of service objects. The service scenario uses three active objects: a client, a server, and a service.

A client object can be any object having some *data* that is passed to the service by a request. Furthermore, each client can be started by supplying a corresponding server s. The method *start* generates a service request with the client's *data* on its request queue. The server object is defined below. Note that the method invocation $s.serve(x.data)$ accepts as parameter $x.data$ due to our extension of the parameter-less ς-calculus.[1]

$$\begin{array}{ll} \mathsf{client} \equiv & [data = \text{"some data"}, \\ & \ start = \varsigma(x, s)(s.serve(x.data)), \\ & \ \ldots \ ] \end{array}$$

[1] In the ς-calculus the parameter has to be simulated by updating a separate field in the object.

On the other side, the service is an object for which a new instance will be generated for the client's use. Such a new instance is created by server objects below by updating the field *client_data* of a service object which automatically creates a new active object representing the service for the client.

$$\begin{array}{ll} \mathsf{service} \equiv & [\mathit{client\_data} = \text{“some data”}, \\ & \mathit{actual\_service} = \text{“some function on } \mathit{client\_data}\text{”}, \\ & \ldots\,] \end{array}$$

A server object generates an individual service personalised by the client's data by instantiating an active object representing the basic service. Initially, the field *base_service* contains the empty object but during initialisation this will be updated.

$$\begin{array}{ll} \mathsf{server} \equiv & [\mathit{base\_service} = [], \\ & \mathit{serve} = \varsigma(x, d)\ (x.\text{base\_service}).\mathit{client\_data} := d] \end{array}$$

3.2.1. Initialisation

We first describe how the service is initialised. The ASP_{fun} program initialising the system is a base object that has a method *init*. The initial configuration will be defined in Section 4.3. It contains a single activity with a unique request. In our case, this request is the activation of the init object and the corresponding call $\mathit{Active}([\mathit{init} = \ldots]).\mathit{init}$.

Now, *init* needs to start clients that know this server. We can use the following ASP_{fun} object to start one client,

$$\mathit{Active}(\mathsf{client}).\mathit{start}(\mathit{Active}(\mathsf{server}.\mathit{base\_service} := \mathit{Active}(\mathsf{service})))$$

where “client”, “server”, and “service” are the abbreviations given before. For several clients being started in the *init* method, we need some iterator construct. We define a *map* function for a method name f as follows. This function applies the method f on each object of a list of objects l while using s as a second parameter to all these calls. It returns a list of objects (which is itself an object). The operator :: is the list constructor, $\langle\rangle_{\text{list}}$ the empty list, *hd* and *tl* give first element and rest of a list, and $l = \langle\rangle_{\text{list}}$ is the empty-list predicate. We, furthermore, use the let and if-then-else construct presented in Section 2.4.

$$\begin{array}{lll} \mathit{map}_f = & \varsigma(x, (s, l)) & \text{if } l = \langle\rangle_{\text{list}} \text{ then } \langle\rangle_{\text{list}} \\ & & \text{else } (\mathit{hd}\ l).f(s) :: (x.\mathit{map}_f(s, \mathit{tl}\ l)) \text{ end} \end{array}$$

Now, we use the following list of $n + 1$ occurrences of client activations,

$$\Lambda = \langle \mathit{Active}(\mathsf{client}.\mathit{data} := d_0), \ldots, \mathit{Active}(\mathsf{client}.\mathit{data} := d_n) \rangle$$

where the d_i denote the different data items of the clients. Summarising, the definition of

the user program is as follows.

$$
\begin{array}{rcl}
\mathit{Active}([\ \mathit{init} & = & \varsigma(x,y) \\
 & & \quad \text{let } \Lambda = \ldots \\
 & & \qquad S = \mathit{Active}(\mathsf{server}.\mathit{base\_service} := \mathit{Active}(\mathsf{service})) \\
 & & \quad \text{in } x.\mathit{map}_{\mathit{start}}((S,\Lambda)), \\
\mathit{map}_{\mathit{start}} & = & \ldots\ ]).\mathit{init}
\end{array}
$$

This user program sent as the only request in the initial activity α sets the server into action.

3.2.2. Server in Action

In this section, we show how the server works. Let us first show the configuration after initialisation; the *Active* commands have all been evaluated; the evaluation of the activation list Λ has created client instances γ_i, $i \in \{0,\ldots,n\}$; the evaluation of S in *init* has

- created a service object σ by evaluating $\mathit{Active}(\mathsf{service})$,
- created a server object Σ by evaluation of $\mathit{Active}(\mathsf{server}.\mathit{base\_service} := \sigma)$,
- sent *start* requests to all client objects γ_i by evaluating the $\mathit{map}_{\mathit{start}}$ invocation putting $\Sigma.\mathit{serve}(\gamma_i.\mathit{data})$ on their request queues.

Note that we choose to evaluate first the innermost service activation, then S itself, before we pass it to $\mathit{map}_{\mathit{start}}$. This leads to the particular service architecture we have in mind; a different order creates several servers ultimately producing the same results (see remark on confluence in Section 2.2) but it would be less economical. The obtained configuration additionally contains an activity ι for the initialiser object with a single served request (*init*) and another α for the initial configuration.

$$
\begin{array}{ll}
[& \gamma_0[f_0 \mapsto \Sigma.\mathit{serve}(\gamma_0.\mathit{data}), [\ldots]], \\
 & \ldots \\
 & \gamma_n[f_n \mapsto \Sigma.\mathit{serve}(\gamma_n.\mathit{data}), [\ldots]], \\
 & \Sigma[\varnothing, [\mathit{base\_service} = \sigma, \\
 & \qquad \mathit{serve} = \varsigma(x,d)\ (x.\text{base\_service}).\mathit{client\_data} := d]], \\
 & \sigma[\varnothing, \mathsf{service}], \\
 & \iota[f' \mapsto [\langle f_0,\ldots,f_n\rangle], [\mathit{init} = \ldots, \mathit{map}_{\mathit{start}} = \ldots]], \\
 & \alpha[f \mapsto f', [\,]] \qquad]
\end{array}
$$

Evaluating the request of the first client γ_0, the above configuration reduces to one in which the server Σ holds one request for creating a service for γ_0.

$$
\begin{array}{ll}
[& \gamma_0[f_0 \mapsto f_{n+1}, [\ldots]], \\
 & \ldots \\
 & \Sigma[f_{n+1} \mapsto (\sigma.\mathit{client\_data}) := \gamma_0.\mathit{data}, [\ldots]], \\
 & \sigma[\varnothing, \mathsf{service}], \ldots \qquad]
\end{array}
$$

Next, evaluation of future f_{n+1} creates a new service object σ' for this service call with the first client's data $\gamma_0.data$ injected as *client_data* in σ' while Σ's request queue holds now the activity reference σ' in future f_{n+1}.

$$\begin{array}{l}\Big[\ \gamma_0[f_0 \mapsto f_{n+1}, [\ldots]], \\ \quad \ldots \\ \quad \Sigma[f_{n+1} \mapsto \sigma', [\ldots], \\ \quad \ldots \\ \quad \sigma'[\varnothing, [client\_data = \gamma_0.data, \\ \qquad\quad actual\_service = \text{“some function on } client\_data\text{”}]], \ldots \qquad \Big]\end{array}$$

The rule REPLY returns the activity reference σ' as a result to the client γ_0 by future f_{n+1}.

$$\begin{array}{l}\Big[\ \gamma_0[f_0 \mapsto \sigma', [\ldots]], \\ \quad \ldots \\ \quad \Sigma[f_{n+1} \mapsto \sigma', [\ldots], \\ \quad \ldots \\ \quad \sigma'[\varnothing, [client\_data = \gamma_0.data, \\ \qquad\quad actual\_service = \text{“some function on } client\_data\text{”}]], \ldots \qquad \Big]\end{array}$$

Now, the client γ_0 has access to its service σ' in a personal instantiation. The client may call at leisure the services of σ' – using this reference to his “personalised” service. The following calls of the other clients $\gamma_2, \ldots, \gamma_n$ all have a similar effect. For each of them a new instance of the first service object σ is automatically created by the semantics of update. All clients finally receive an activity reference and all have been served by the same server Σ. The server in action is illustrated in Figure 2 depicting the moment just when the second client's service call is launched to the server.

As a further extension to this example, we could consider programming a central registry for the service objects. To this end, we just change the *init* method of the base object to make a final update to a local field registry to store the result of the activation map to all clients.

$$\begin{array}{lcl}[init & = & \varsigma(x, y) \\ & & \quad \text{let } \Lambda = \ldots \\ & & \qquad S = Active(\mathsf{server}.base\_service := Active(\mathsf{service})) \\ & & \quad \text{in } x.registry := (x.map_{start}(S, \Lambda)), \\ registry & = & \langle\rangle_{\text{list}}, \\ map_{start} & = & \ldots]\end{array}$$

With this changed base object, the call to *init* has exactly the same effect as before. Only as a final step, the base object is updated to keep the results list of all the created client objects (and thereby also of the services).

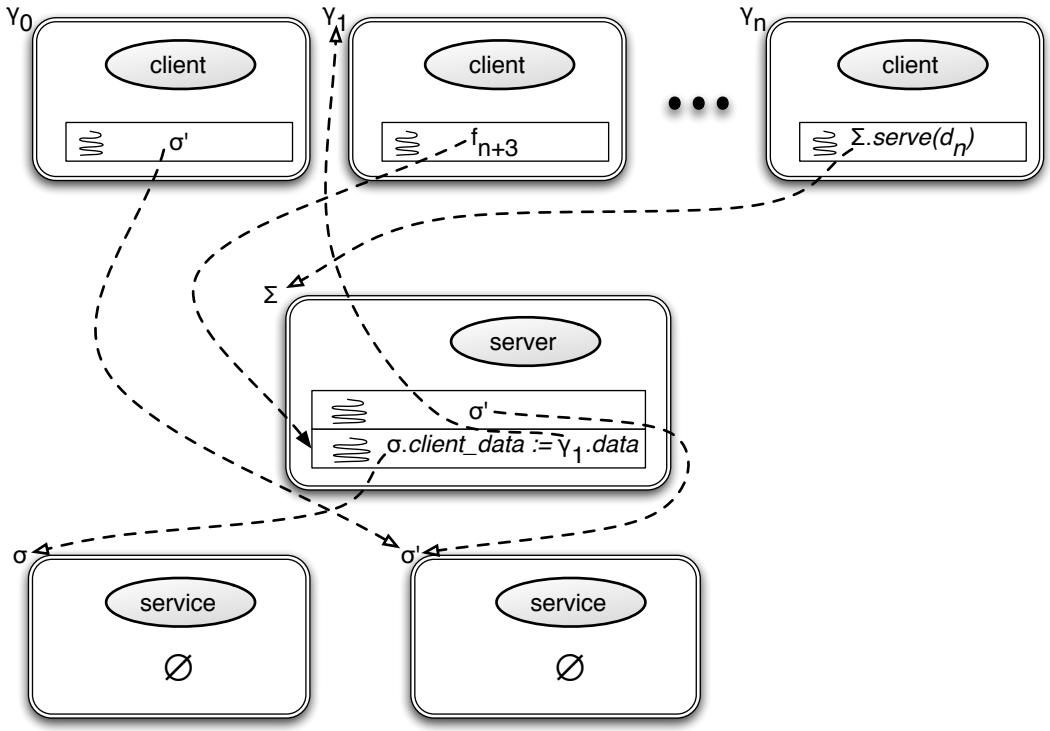

Figure 2: Server in action.

4. Properties of ASP_{fun}

This section presents two major properties of ASP_{fun}: the semantics is well-formed; and reduction does not create cycles of futures and activity references.

4.1. Well-formed Configuration

To show correctness of the semantics, we define a *well-formed configuration* as referencing only existing activities and futures; then we prove that reduction preserves well-formedness.

Definition 4.1 (Well-formed configuration). *A configuration C is well-formed, denoted $wf(C)$, if and only if for all α, f_i, s, Q, and t each of the following holds:*

$$\alpha[Q, E[\beta]] \in C \ \vee \ \alpha[f_i \mapsto E[\beta] :: Q, t] \in C \ \Rightarrow \ \beta \in \text{dom}(C)$$

$$\alpha[Q, E[f_k]] \in C \ \vee \ \alpha[f_i \mapsto E[f_k] :: Q, t] \in C \ \Rightarrow \ \exists\, \gamma,\, R,\, t'.\, \gamma[R, t'] \in C \wedge f_k \in \text{dom}(R)$$

We have shown that, starting from a well-formed configuration, the reduction shown in Table 2 always reaches a well-formed configuration.

Property 1 (Reduction preserves well-formedness).

$$(s \rightarrow_{\|} t \ \wedge \ wf(s)) \ \Rightarrow \ wf(t)$$

This can be considered as a correctness property for the semantics of ASP_{fun}: no ill-formed configuration can be created by the reduction.

4.2. Absence of Cycles

Informally, ASP_{fun} avoids blocking method invocations because a not fully evaluated future can be returned to the caller at any time. The natural question arises whether there is the possibility for live-locks: a cycle of communications (here, a cycle of replies in fact) in which no real progress is made apart from the actual exchange of communication. However, we can show that, given a configuration with no cycle, any possible configuration that may be derived from there has no cycle either. The cycles we consider are formed of activity references and futures.

We say that an activity or a future knows another one if it holds a reference to it. An activity holds a reference if it has this reference inside its active object. A future holds a reference if the request computing this future contains this reference. Table 3 shows the rules defining the $knows_C$ relationship for a configuration C together with the *nocycle* property where $knows_C^+$ is the transitive closure of $knows_C$ ($r \ knows_C^+ \ r' \Leftrightarrow r \ knows_C \ r' \vee \exists r''.(r \ knows_C \ r' \wedge r'' \ knows_C^+ \ r')$). It is necessary to interleave references to futures and activities in the definition of $knows_C$ because, for example, a reference from an active object becomes a reference from a future when a REQUEST rule is evaluated.

$$\frac{\alpha[Q, E[\beta]] \in C}{\alpha \ knows_C \ \beta} \qquad \frac{\alpha[f_i \mapsto E[\beta] :: Q, t] \in C}{f_i \ knows_C \ \beta} \qquad \frac{\alpha[Q, E[f_k]] \in C}{\alpha \ knows_C \ f_k} \qquad \frac{\alpha[f_i \mapsto E[f_k] :: Q, t] \in C}{f_i \ knows_C \ f_k}$$

$$nocycle(c) \Leftrightarrow \neg \exists r. \, r \ knows_C^+ \ r$$

Table 3: The *nocycle* property

We proved that the reduction defined in Table 2 maintains the absence of cycles for a well-formed configuration.

Theorem 1. *Reduction does not create cycles:*

$$nocycle(C) \ \wedge \ wf(C) \ \wedge \ C \rightarrow_{\|} C' \quad \Rightarrow \quad nocycle(C')$$

The theorem relies on the fact that domains of request queues are disjoint, which is enforced by the definition of a configuration in ASP_{fun}. Absence of cycles ensures that there are no live-locks related to the distributed aspects of ASP_{fun}, i.e., no infinite cycle of replies. Live-locks that can exist in ASP_{fun} are inherited from ς-calculus: they are either infinite loops inside a ς-calculus term or infinite sequences of method calls (distributed or not).

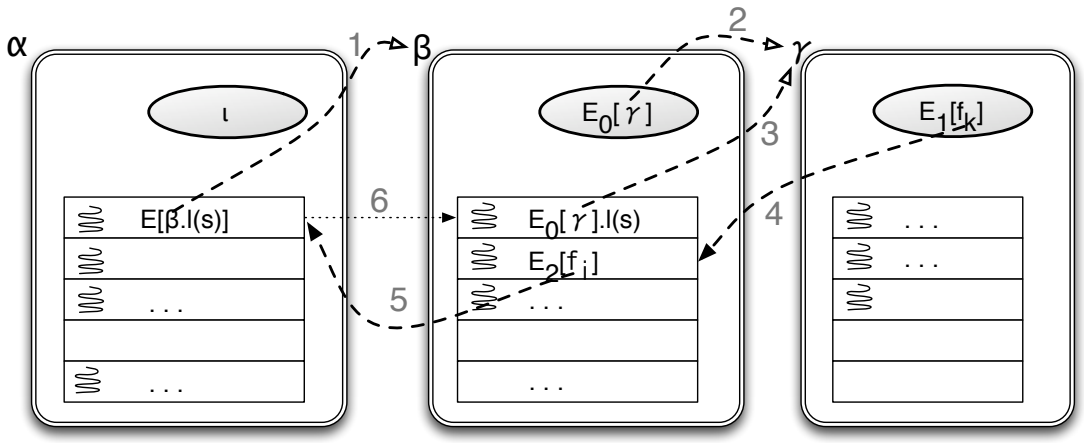

Figure 3: A cycle of future and activity references.

Figure 3 shows cycles of futures and activity references. We have two cycles, one consisting of the arrows numbered $\{1, 2, 4, 5\}$, another one is formed by the arrows $\{3, 4, 5, 6\}$.

Absence of cycle limits the expressiveness of the language (no cross-references), but this restriction is inherited from the functional nature of the language. Indeed, functional languages have no references, whereas active objects and futures create some kind of references; preventing cycles and modification is necessary to keep the functional nature of ASP_{fun}.

4.3. Initial Configuration

This section shows how a reasonable initial configuration can be built from a program. In a usual programming language, a programmer does not write configurations but usual programs invoking some distribution or concurrency primitives (in ASP_{fun} *Active* is the only such primitive). This is reflected by the ASP_{fun} syntax given in Section 2.1. A "program" is a term s_0 given by this static syntax (it has no future or active object reference and no free variable). In order to be evaluated, this program must be placed in an initial configuration. The initial configuration has a single activity with a single request consisting of the user program:

$$\mathit{initConf}(s_0) = \alpha[f_0 \mapsto s_0, []]$$

This configuration is well-formed, and the activity α will never be accessible. Consequently, any reachable configuration is well-formed. We also see that the initial configuration has no cycles, and Theorem 1 ensures that any reachable configuration has no cycles.

Property 2. *Any configuration reachable from an initial configuration is well-formed and has no cycles ($\rightarrow_{\|}^{*}$ is the reflexive transitive closure of $\rightarrow_{\|}$).*

$$\mathit{initConf}(s_0) \rightarrow_{\|}^{*} C \quad \Rightarrow \quad \mathit{wf}(C) \ \wedge \ \mathit{nocycle}(C)$$

5. Typing Active Objects

This section provides a type system for ASP_{fun}. Starting from ς-calculus basic type system, we first define typing for the *Active* primitive; then we define type-checking rules for an ASP_{fun} configuration. After the classical subject-reduction property, we show that the type system ensures *type uniqueness*, *well-formedness* of configurations, and more importantly *progress*. We will see that typing ensures that no method can be invoked on a term that is unable to handle it; the semantics ensures that no invocation or update on a future or an activity can be indefinitely blocked.

5.1. A Local Type system

We first adapt the simple type system that Abadi and Cardelli (1996) devised as $\mathbf{Ob_1}$. Object types are of the form $[l_i : B_i{\to}D_i]^{i\in 1..n}$. The syntax of ASP_{fun} is extended by adding type information on both variables under the binder ($\varsigma(x, y)$ becomes $\varsigma(x : A, y : B)$). As highlighted in (Abadi and Cardelli, 1996), adding type information on the binders ensures type uniqueness.

$$\textsc{Val}\ x \qquad x : A :: T \vdash x : A$$

$$\textsc{Type Object}\quad \frac{A = [l_i : B_i{\to}D_i]^{i\in 1..n} \qquad \forall i \in 1..n,\ x_i : A :: y_i : B_i :: T \vdash t_i : D_i}{T \vdash [l_i = \varsigma(x_i : A, y_i : B_i)t_i]^{i\in 1..n} : A}$$

$$\textsc{Type Call}\quad \frac{T \vdash s : [l_i : B_i{\to}D_i]^{\ i\in 1..n} \qquad j \in 1..n \qquad T \vdash t : B_j}{T \vdash s.l_j(t) : D_j}$$

$$\textsc{Type Update}\quad \frac{A = [l_i : B_i{\to}D_i]^{i\in 1..n} \qquad T \vdash s : A \qquad j \in 1..n \qquad x : A :: y : B :: T \vdash t : D_j}{T \vdash s.l_j := \varsigma(x : A, y : B)t : A}$$

Table 4: Typing the local calculus

Table 4 defines the typing of local ASP_{fun} terms as presented in 2.1. It is an adaptation of the typing of $\mathbf{Ob_1}$ in (Abadi and Cardelli, 1996). A, B, and D range over types. The variable T represents a *type environment* containing type assumptions for variables and is identified modulo reordering. Its extension by a new assumption stating that the variable x has type A is denoted by $x : A :: T$. We now authorise :: to update a mapping entry: $(x : A) :: T$ associates the type A to x even if an entry for x existed in T. The first rule of Table 4 accesses the type environment. Type Object describes how an object's type is checked from its constituents: an object of type $[l_i : B_i{\to}D_i]^{i\in i..n}$ is formed from bodies t_i of types B_i using self parameter x_i of type A and additional parameter y_i of type B_i. When a method l_j is invoked on an object s of type $[l_i : B_i{\to}D_i]^{i\in i..n}$ the result $s.l_j(b)$ has type D_j provided s has type B_j (Type Call). A method update requires that the updated object has the same type as self in the new method (Type Update).

In (Abadi and Cardelli, 1996), additional rules ensure that the typing environment is well-formed. We simplified it here by defining environment as a mapping. Also, a rule for

$$\textsc{Type Active}\quad \frac{\langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash a : A}{\langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash \mathit{Active}(a) : A} \qquad \textsc{Type Activity Reference}\quad \frac{\beta \in \mathrm{dom}(\Gamma_{act})}{\langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash \beta : \Gamma_{act}(\beta)}$$

$$\textsc{Type Future Reference}\quad \frac{f_k \in \mathrm{dom}(\Gamma_{fut})}{\langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash f_k : \Gamma_{fut}(f_k)}$$

$$\textsc{Type Configuration}\quad \frac{\begin{array}{c}\mathrm{dom}(\Gamma_{act}) = \mathrm{dom}(C) \qquad \mathrm{dom}(\Gamma_{fut}) = \bigcup\{\mathrm{dom}(Q) \mid \exists\, \alpha, a.\ \alpha[Q,a] \in C\} \\ \forall\, \alpha[Q,a] \in C.\ \begin{cases} \langle \Gamma_{act}, \Gamma_{fut}\rangle, \varnothing \vdash a : \Gamma_{act}(\alpha) & \wedge \\ \forall\, f_i \in \mathrm{dom}(Q).\ \langle \Gamma_{act}, \Gamma_{fut}\rangle, \varnothing \vdash Q(f_i) : \Gamma_{fut}(f_i) \end{cases}\end{array}}{\vdash C : \langle \Gamma_{act}, \Gamma_{fut}\rangle}$$

Table 5: Typing configurations

correct formation of object types is introduced in (Abadi and Cardelli, 1996) mainly ensuring that there is no infinitely nested object type. This last assumption has been omitted here as it did not seem necessary and, indeed, the properties shown below have been mechanically proved without any additional assumptions on type formation.

5.2. A Type System for ASP_{fun}

The type system for $\mathrm{ASP}_{\mathrm{fun}}$ is based on an inductive typing relation on $\mathrm{ASP}_{\mathrm{fun}}$ terms; it is defined in Table 5. From local typing (Table 4), in addition to types of variables, we need to refer to types for futures and activities. Thus, we add a pair of parameters $\langle \Gamma_{act}, \Gamma_{fut}\rangle$ in the assumptions of a typing statement: we write $\langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash x : A$ instead of $T \vdash x : A$. These parameters consist of a mapping Γ_{act} from activities to the type of their active object and another one Γ_{fut} from future identifiers to the type of the corresponding request value. Thus, we first adorn each rule of Table 4 with those two additional parameters.

Then, we add to these rules the three first rules of Table 5 that define the local typing of $\mathrm{ASP}_{\mathrm{fun}}$. These rules allow the typing of references to activities and futures and define typing of the *Active* primitive: the type of an activated object is the type of the object.

The last rule of Table 5 incorporates into a configuration the local typing assertions. This rule states that a configuration C has the configuration type $\langle \Gamma_{act}, \Gamma_{fut}\rangle$ if the following conditions hold.

- The same activity names are defined in C and in Γ_{act};
- the same future references are defined in the activities of C and in Γ_{fut};
- for each activity of C, its active object has the type defined in Γ_{act};

- and each request has the type defined in Γ_{fut} for the corresponding future.

Similarities can be found between typing of activity or future references and reference types (Pierce, 2002). A closer work seems to be the typing rules for futures (Niehren et al., 2006).

5.3. Basic Properties of the Type System

Let us start by a couple of simple properties of the typing system. First, type-uniqueness existing for $\mathbf{Ob_1}$ is also verified by our type system.

Property 3 (Unique Type). *Each expression in ASP_{fun} has a unique type.*

$$\langle \Gamma_{act}, \Gamma_{fut} \rangle, T \vdash a : A \wedge \langle \Gamma_{act}, \Gamma_{fut} \rangle, T \vdash a : A' \Longrightarrow A = A'$$

Well-typed configurations are well-formed. Indeed, if an activity or a future is referenced in the configuration, it must have a type and thus be defined in Γ_{act} or Γ_{fut}, and also in the configuration.

Property 4 (Typing ensures well-formedness). $\vdash C : \langle \Gamma_{act}, \Gamma_{fut} \rangle \Rightarrow wf(C)$

5.4. Subject Reduction

Subject reduction ensures that reduction preserves the typing relation. Therefore, it is often also called *preservation*. We prove subject reduction of $\mathrm{ASP_{fun}}$ with respect to the type system given in the previous section.

We prove first the subject reduction property for the local reduction:

Property 5 (Local Subject Reduction).

$$\langle \Gamma_{act}, \Gamma_{fut} \rangle, T \vdash t : A \ \wedge \ t \rightarrow_{\varsigma} t' \Rightarrow \langle \Gamma_{act}, \Gamma_{fut} \rangle, T \vdash t' : A$$

Then, we prove subject reduction for the full typing relation of configurations.

Theorem 2 (Subject Reduction).

$$\vdash C : \langle \Gamma_{act}, \Gamma_{fut} \rangle \ \wedge \ C \rightarrow_{\parallel} C' \Rightarrow \exists\ \Gamma'_{act}, \Gamma'_{fut}.\ \vdash C' : \langle \Gamma'_{act}, \Gamma'_{fut} \rangle$$

where $\Gamma_{act} \subseteq \Gamma'_{act}$, *and* $\Gamma_{fut} \subseteq \Gamma'_{fut}$.

Note that activities and futures may be created by the reduction and thus the typing environment may have to be extended.

5.5. Progress and Absence of Dead-locks

Finally, we can prove progress for well-typed configurations. Progress states that any expression of the language is either a result or can be reduced. In ASP_{fun}, we prove progress for each request of a configuration. A term is a result, i.e., a totally evaluated term, if it is either an object (like in (Abadi and Cardelli, 1996)) or an activity reference.

$$\mathit{isresult}(s) \Leftrightarrow \exists l_i,\, t_i,\, A.\, s = [l_i = \varsigma(x_i : A, y_i : B)t_i]^{i\in 1..n} \vee \exists \alpha,\, s = \alpha$$

The type system is useful for ensuring that every accessed method exists on the invoked object. In fact, local typing ensures progress of local reduction. Typing for configurations extends the typing relation to distributed objects ensuring for example that a method invocation on a future will be possible once the result is returned. Absence of dead-locks for the distributed semantics is only ensured by the functional nature of ASP_{fun}, by the absence of loops, and by the particular semantics of the calculus. A first notion of progress can be proved: for a correctly typed configuration, either all requests are reduced to a future, or the configuration can be reduced.

Property 6. $\vdash C : \langle \Gamma_{act}, \Gamma_{fut} \rangle \ \wedge \ \alpha[f_i \mapsto s :: Q, t] \in C \Rightarrow \mathit{isresult}(s) \vee \exists C' \,.\, C \rightarrow_{\|} C'$

More precisely, we can prove that the request that is not yet reduced to a result, i.e., the term s in the theorem above, can be reduced. Unfortunately, as already shown in (Abadi and Cardelli, 1996), ς-calculus does not ensure that a reduced term is different from the source one, but this is an issue related to the local reduction which is not the concern of this paper. We proved that, on the distributed side, the term really always progresses and that no reduction loop is induced by the distributed features of ASP_{fun}. We can reformulate the preceding theorem:

Theorem 3 (Progress).

$$\mathit{nocycle}(C) \wedge \vdash C : \langle \Gamma_{act}, \Gamma_{fut} \rangle \ \wedge \ \alpha[f_i \mapsto s :: Q, t] \in C \ \Rightarrow \mathit{isresult}(s) \vee \exists\, C' \,.\, C \rightarrow_{\|} C'$$

where C' can be chosen to verify: $\alpha[f_i \mapsto s' :: Q, t] \in C' \wedge (s' \neq s \vee s \rightarrow_{\varsigma} s)$ *.*

By proving progress, we also show that ASP_{fun} is dead-lock free: as any term that is not already a result must progress, this ensures the absence of dead-lock.

As configurations reachable from the initial configurations have no cycle, a variant of the progress theorem can be stated by replacing the *nocycle* hypothesis by the reachability from a well-typed initial configuration:

Property 7 (Progress from initial configuration). *Let s_0 be a* static term*; if it is correctly typed (in an empty environment), then each request of any configuration C obtained from s_0 is either reduced to a value or can be further reduced; more formally:*

$$\mathit{initConf}(s_0) \rightarrow_{\|}^{*} C \wedge \langle \varnothing, \varnothing \rangle, \varnothing \vdash s_0 : A \wedge \alpha[f_i \mapsto s :: Q, t] \in C \ \Rightarrow \ \mathit{isresult}(s) \vee \exists C' . C \rightarrow_{\|} C'$$

where C' can be chosen to verify: $\alpha[f_i \mapsto s' :: Q, t] \in C' \wedge (s' \neq s \vee s \rightarrow_{\varsigma} s)$ *.*

6. Formalisation in Isabelle/HOL

The interactive theorem prover Isabelle/HOL (Nipkow et al., 2002) offers a classical higher order logic (HOL) as a basis for the modelling of application logics. Inductive definitions and datatype definitions can be written in a way close to programming language syntax and semantics. Semantic properties over datatypes can be expressed in a clear manner using primitive recursion which is supported by powerful proof automation using rewriting techniques. Nevertheless – unlike model checking or other fully automated proof techniques – the expressivity of HOL comes at a price: the user has to find the gist of proofs concerning his application logics himself even if simple simplification steps are handled automatically.

In this section we will give an outline of the mechanisation of ASP_{fun}, its syntax, semantics, type system, and proofs in Isabelle/HOL. To this end, we begin Section 6.1 by introducing finite maps, a useful extension of Isabelle/HOL we created for representing objects. We also discuss in some detail different techniques for representing *binders* when formalising language meta-theory – necessary for the subsequent experience report. We then describe in Section 6.2 important aspects of our proofs in a manner independent of the actual Isabelle/HOL representation. We give details on the Isabelle/HOL formalisation using de Bruijn indices in Section 6.3. For defining the operational semantics of the local object calculus, we adapted the semantics for the ς-calculus defined in (Henrio and Kammüller, 2007) in order to use reduction contexts. We also proved in Isabelle/HOL that both models are equivalent or, more precisely, that both small step semantics express exactly the same reduction.

In a constant attempt to improve the Isabelle/HOL mechanisation, we have updated the ASP_{fun} mechanization with a different binder technique: we replaced the classical de Bruijn indices by a locally nameless representation that provides a more natural representation of formulae by variable names (Aydemir et al., 2008a). The experience of having thus performed the entire formalisation of ASP_{fun} twice enables us to provide a profound comparison of the two representation techniques in Section 6.4.

6.1. Tools for Programming Languages and Semantics

6.1.1. Finite maps: deep versus shallow

The embedding of the language ASP_{fun} into Isabelle/HOL needs to be deep enough to reason about the language and its semantics while also being shallow enough, i.e., using enough basic concepts of HOL to facilitate reasoning and simulation of examples. Finite maps are a primitive feature we needed to formalise; this feature is defined closely to the HOL type system to reduce the depth of our embedding.

An object in the ς-calculus is a finite unordered list of named elements that is recursive in its self-parameter: objects *are* finite maps. To enable primitive recursive definitions of functions on terms we need a recursive datatype for objects. The inbred recursion of objects forces us to use a primitive function type to represent these object maps. Thus, we use HOL's primitive map type to define finite maps $\alpha \Rightarrow_f \beta$ by coercing their domain α to be in the type class `finite` of all finite types.

We derive the following induction scheme from the induction rule for finite sets using a domain isomorphism between finite maps and finite functional relations. If a property `P` is valid for the empty finite map and it is, furthermore, preserved when an element is added to the finite map by updating the map, then the property is true for all finite maps. Note, that for the general function type $\Rightarrow$ such an induction does not hold.

$$\begin{array}{l}\llbracket\ \texttt{P empty;}\\ \quad \bigwedge \texttt{x (F::}\ \alpha \Rightarrow_f \beta\texttt{) y .}\ \llbracket\ \ \texttt{P F; x} \notin \texttt{dom F}\ \rrbracket \Longrightarrow \texttt{P (F(x} \mapsto \texttt{y))}\\ \rrbracket \quad \Longrightarrow \texttt{P F}\end{array}$$

The brackets $\llbracket \ldots \rrbracket$ indicate the conjunction of meta-level hypotheses of a rule. The additional type judgement $\alpha \Rightarrow_f \beta$ coerces `F` to be an `fmap`.

6.1.2. Binder representation

The formalisation of programming languages in rigorous frameworks, like theorem provers, has revealed some crucial issues summarised in the POPL-mark challenge (Aydemir et al., 2008b) a set of benchmarks for the mechanisation of language meta-theory. The problem of the representation of binders is there identified as a central problem to the challenges. We discuss in this section the main techniques for representing variable binders laying the ground for the following formalisations.

Problem Statement. The representation of binders has already been recognised by Bruijn (1972) in the Automath project as a major problem when mechanising languages. Intuitively, a language that has local scopes and parameterisation – for example functions $\lambda x.fx$ – needs to refer to the formal parameters – here x – when they occur inside these scopes – here x occurs in the context f. The natural, human understandable way is to use variables, like x, to define and denote formal parameters by name but variables are neither well suited for mechanisations nor proofs. For example, variable capture may occur, that is, a variable occurring free in a term t may accidentally be "captured" when substituting t inside a scope where x is the name of a bound variable. For example, in $(\lambda x.xy)[\lambda z.x/y]$, the free variable x in $\lambda z.x$ could be captured by the substitution. To avoid this, we use a consistent renaming. Formally, α-equivalence justifies such renamings. However, α-equivalence creates classes of equivalent terms with equal denotation which complicates the semantics. In particular, when fresh variables are a prerequisite inside semantic rules, the choice of α-conversions inside a term predisposes the choice of fresh variables creating an interference that obstructs compositional reasoning.

De Bruijn indices. The classical solution, proposed by N. G. de Bruijn, is to replace each occurrence of a variable by an integer equal to the number of binders that have to be crossed to reach the binder for the considered variable. In other words, a variable is replaced by the distance from its binding scope. Note, the same "variable" may be represented by different integers. For example, the lambda term $\lambda x.x(\lambda y.x\,y)$ in de Bruijn notation is $\lambda(0(\lambda 1\,0))$; x is once represented as 0, once as 1. The "nominality" of terms is abstracted – semantic denotation becomes unique but substitution becomes very technical because of the "lifting"

of indices when entering a binder or replacing a term under binders. Then a term that has to be substituted at nesting depth n into another term needs to add n to all its indices representing free variables. To this end, one first defines a "lift" operation that performs this addition and the substitution then uses lift.

Locally nameless representation. Already at the time of first devising his concept of indices, Bruijn (1972) suggested an alternative where indices represent bound variables (written *bvar i*) and classical named variables represent free (unbound) variables (written *fvar x*); *open* and *close* operations translate between those representations. This technique, known as *locally nameless* representation, has since recently attracted wide attention (Aydemir et al., 2008a). It seems very attractive as it combines unique representation provided by de Bruijn indices with human understandable expression of specification of theorems using names – avoiding manipulation of explicit indices, in terms, semantics, and lemmata.

The *open* operation, written t^u, substitutes a term u for the outermost bound variable in the term t. For example $\lambda(bvar\,0\,\lambda((bvar\,1)(bvar\,0)))^n$ is equal to $n\,\lambda(n\,(bvar\,0))$. The opposite operation *closes* a term: given a name, the closing replaces the occurrence of variables of this name with an index for a bound variable, such that the variable is bound at the outermost level of the term.

A drawback of the locally nameless approach is that we need to take explicit care that we do include only well-formed terms, i.e., only *bound* variables are represented by indices. The notion of locally closed terms ensures this. E.g., $\lambda(bvar\ 2)$ is not locally closed. Ensuring that we manipulate only locally closed terms will have to be added as prerequisite to our propositions when dealing with locally nameless representation. Another problem arises when reducing a term under a binder. Here, we should close the term under a *fresh variable* (to keep the term locally closed). Formally, we need: $\forall x \notin FV(t).t^x \to (t')^x \Longrightarrow \lambda(t) \to \lambda(t')$. The drawback of this approach is that it is sensitive to the set of free variables, that may vary in an unexpected way. Here, the approach of *cofinite quantification* (Aydemir et al., 2008a) is an important step forward. The basic idea is to abstract over the set of free variables $FV(t)$ and to let fresh variable be taken among the complementary of an *existentially quantified* finite set L, the proposition above becomes: $\exists L$ finite.$\forall x \notin L.t^x \to (t')^x \Longrightarrow \lambda(t) \to \lambda(t')$. This set L can then be instantiated appropriately when handling proofs.

Nominal techniques. Another approach, proposed by Urban and et al. (2006) based on work on nominal logic by Pitts (2003), is called nominal techniques. Here, terms are identified as a set bijective to all terms factorised by α-equivalence. Instead of using substitution, nominal logic uses permutations of atomic names. Permutations are built from elementary *name swaps*: e.g., $(a, b) \cdot t$ replaces all occurrences of a by b and *vice versa* in t. Permutations are only applicable if there are fresh atoms available. This is expressed by keeping track of the support set (fresh atoms). The classical hypothesis, "there is a fresh variable" for a term t is replaced by, "there is a finite *support* for x", i.e., the set of atoms used in t is finite, and infinitely many "fresh" atoms are available. Unfortunately, the Isabelle/HOL package implementing nominal techniques cannot be used as it is – in our case – because we use finite maps in our implementation; consecutively the recursive datatype defining ASP_{fun} syntax

is a bit more complex than the usual simple recursive construction. While it is trivial that a finite map containing terms of finite support has a finite support, such a reasoning is not yet supported by Urban's package.

Higher order abstract syntax. Another technique for formalising binders is Higher Order Abstract Syntax (HOAS) in which binders of applications are directly represented by binders of the meta-level, e.g., (Roeckl and Hirschkoff, 2003; Ciaffaglione et al., 2007). Therefore, by contrast to the above sketched approaches, HOAS is often also called the *direct* encoding. For example, in Isabelle/HOL, we would use the HOL λ-abstraction to encode object-level binders. This approach has advantages in terms of mechanisations: reductions are usually performed automatically but it is restricted when it comes to meta-level reasoning. Sometimes, "meta-theoretic properties involving substitution and freshness of names inside proofs and processes, cannot be proved inside the framework and instead have to be postulated" (Honsell et al., 2001).

6.2. Crucial Aspects of the Proofs

This section details some of the parts of the formalisation that seem the most important to us, it gives proof sketches, and is not much coupled with Isabelle/HOL.

6.2.1. Finiteness

When considering language semantics we often implicitly assume finiteness of programs and configurations. In fact, the implicit assumption is worth mentioning: for programs it grants induction over the recursive datatype of ς-terms, and for configurations, it permits the assumption that there are always fresh activity and future names available. Our formalisation relies on this assumption. We particularly highlight the fact that it becomes necessary to show progress. For example, to create a new activity one must find a fresh identifier. We have shown that initial configurations and configurations reduced from them are all finite: they have a finite number of activities and futures.

6.2.2. Absence of Cycles

Proving the absence of cycles (Theorem 1) required us several steps. We first defined a datatype for future or activity reference and then specified the $knows_C$ and $knows_C^+$ relations defined in Section 4.2. In order to handle the proofs, we refine the $knows_C^+$ relation by remembering the list of intermediate activities: $r\ knows_{C(L)}^+\ r'$ iff $r\ knows_C^+\ r'$ passing by the references in L.

We first prove lemmata relating cycles, $knows_C^+$, and paths. E.g., if $r\ knows_{C(L)}^+\ r'$ and C' is obtained from C by just modifying the request corresponding to f_k, then $r\ knows_{C'(L)}^+\ r'$ provided $f_k \notin L$ and $f_k \neq r$. A similar lemma exists for activity references. Consequently, it is sufficient to prove that no cycle is created by the activities and requests modified by the considered reduction. We also show that, when $r\ knows_{C(L)}^+\ r'$, L can be chosen to include neither r nor r'.

The main proof of absence of cycles is a long case analysis on the reduction rules that uses lemmata presented above, well-formedness of the initial configuration, and shows that

if there is a cycle in the obtained configuration, there was necessarily one in the original configuration. As an example, we detail the main argument for the REQUEST rule referring to the rule of Table 2 with C_1 being the source configuration and C_2 the obtained one. One can first note that, as the source configuration is well-formed by hypothesis, only f_i may refer to f_k in C_2. Secondly, if f_k $knows_{C_2}$ r then either β $knows_{C_1}$ r or f_i $knows_{C_1}$ r. We only have to show that $\neg\exists L.f_k$ $knows^+_{C_2(L)}$ f_k. By contradiction and induction on the length of L, length 0 is impossible because β or f_i would know f_k in C_1 which would not be well-formed. For greater lengths, $L = L'\#r$, and necessarily $r = f_i$ as shown above. Thus, f_k $knows^+_{C_2(L')}$ f_i, where $f_k \notin L$ and $f_i \notin L$. Consequently, f_i $knows^+_{C_2(L')}$ f_i or β $knows^+_{C_2(L')}$ f_i as the request for f_k is only built from the request for f_i and the active object of β ($t'.l(s)$ in Table 2). Since $f_k \notin L$ and $f_i \notin L$, and only f_i and f_k have been modified between C_1 and C_2: f_i $knows^+_{C_1(L')}$ f_i or β $knows^+_{C_1(L')}$ f_i. As f_i $knows_{C_1}$ β, in either case there would be a cycle from f_i in C_1, which is contradictory.

Figure 3, page 17 illustrates this case of the proof. It considers the case of a REQUEST from α to β creating the future f_l and the reference depicted by the arrow 6. Additionally, suppose a cycle is created: arrows 3, 4, 5, 6 in the figure, we consider the sub-case where this cycle was created because of a reference in the active object in β. We decompose the cycle into f_l $knows^+_{C_2(L')}$ f_i, with L' consisting of the arrows 3, 4, 5 on the figure, plus arrow 6 ($f_i knows_{C_2} f_l$). Then, necessarily, before the reduction f_i was involved in a cycle passing by β and by the path consisting of the arrows 1, 2, 4, 5, which shows the contradiction.

6.2.3. Typing and Subject Reduction

Subject reduction is handled in two phases, each proved by case analysis: one for local and one for distributed reduction. We detail below a few useful lemmata. A first lemma states that any term that has a type in an empty environment has no free variable:

$$\langle \Gamma_{act}, \Gamma_{fut} \rangle, \varnothing \vdash a : A \;\Rightarrow\; noFV(a)$$

Conversely, a term without free variable can be typed in an empty environment (in fact, below we could prove $A = A'$ but this was not useful):

$$\langle \Gamma_{act}, \Gamma_{fut} \rangle, T \vdash a : A \;\wedge\; noFV(a) \;\Rightarrow\; \exists A'.\langle \Gamma_{act}, \Gamma_{fut} \rangle, \varnothing \vdash a : A'$$

Both preceding properties are necessary to show that for an activated object or a new request a type can be found.

$$\begin{array}{l} \langle \Gamma_{act}, \Gamma_{fut} \rangle, \varnothing \vdash E[a] : A \\ \wedge\ \langle \Gamma_{act}, \Gamma_{fut} \rangle, \varnothing \vdash a : B \;\wedge\; \langle \Gamma_{act}, \Gamma_{fut} \rangle, \varnothing \vdash b : B \end{array} \Rightarrow \langle \Gamma_{act}, \Gamma_{fut} \rangle, \varnothing \vdash E[b] : A$$

This lemma is both crucial and interesting because it relates contexts and typing. As our reduction relies on the use of contexts, this lemma is decisive for the proof of subject reduction, Theorem 2.

6.2.4. Proving Progress

Proving progress relies on a long case analysis on the reduction rules. We focus first on one crucial argument: how can the absence of free variable be ensured in order to communicate an object between two activities. Each request can be typed in an empty environment (for variables); thus it does not have any free variable, and thus each sub-term of a request that is not under a binder has no free variable. We prove that one can reduce at least the part of the request under the evaluation context F, where $F ::= \bullet \mid F.l_i(t) \mid F.l_i := \varsigma(x,y)\, s \mid Active(F)$. If one replaces E by F in the semantics, this prevents reduction to occur inside the binders. Indeed, in F the term in the position of the hole has no free variables: $\langle \Gamma_{act}, \Gamma_{fut} \rangle, \varnothing \vdash F[a] : A \;\Rightarrow\; noFV(a)$.

Considering the other arguments of the proof, the absence of cycles ensures that an application of a REPLY rule cannot return a future value which is the future itself, in which case the configuration would be reducible but to itself. This ensures that no live-lock exists in the distributed semantics even if the local one can create live-locks. Of course, the proof also massively uses the fact that well-typed configurations are well-formed.

6.3. The Formal Model in Isabelle/HOL with de Bruijn Indices

This section presents a first version of the formalisation of ASP_{fun}, its syntax, and a few theorems in Isabelle/HOL; this version relies on de Bruijn indices. The main objective of this section is to give a real feel for the Isabelle/HOL formalisation and outline the main steps of the formalisation process. We use here the de Bruijn representation for the syntax of ASP_{fun} but the major part of the formalisation process is similar for the locally nameless representation presented in the subsequent section.

6.3.1. Syntax

The formalisation of functional ASP is constructed as an extension of the base Isabelle/HOL theory for the ς-calculus (Henrio and Kammüller, 2007). The term type of the ς-calculus is represented by an Isabelle/HOL datatype definition called `dB`. In this datatype definition, objects are represented as finite maps `Obj (Label` $\Rightarrow_f$`dB) type`. We formalised finite maps in the first argument of `Obj` using the abstract concept of axiomatic type classes. As discussed in Section 6.1.1, it is crucial to have finite maps as a basic Isabelle/HOL type to be able to employ the recursive datatype construction here. The second argument of the constructor `Obj` is a type annotation. The resulting datatype for basic terms of ASP_{fun} is then as follows. Variables are represented by de Bruijn indices. A given index has two entries: one for self, and the other for the parameter as defined by the datatype `Variable`.

```
datatype Variable = Self nat | Param nat

datatype dB =                (*The typed ASPfun datatype*)
    Var Variable             (*Variable - deBruijn index*)
  | Obj "Label ⇒f dB" type(*Objects map labels to terms, and have a type*)
  | Call dB Label dB         (*Call a l b calls meth l on a with param b*)
  | Upd dB Label dB          (*Upd a l b updates meth l of a  with body b*)
  | Active dB                (*Creates an active object*)
  | ActRef ActivityRef       (*References an active object - dynamic syntax*)
  | FutRef FutureRef         (*References a future - dynamic syntax*)
```

The type of configurations relies on partial functions expressed by the constructor $\Rightarrow\!\!\!\!\!\!+$.

```
futmap = FutureRef ⇛ dB
configuration = ActivityRef ⇛ (futmap × dB)
```

6.3.2. Reduction Contexts in Isabelle

In our model we developed a simple mechanisation of a reduction context using again the datatype construct as follows:

```
datatype general_context = (*a general context is a term with a hole*)
    cHole
  | cObj FmapLabel type general_context
  | cCallL general_context Label dB
  | cCallR dB Label general_context
  | cUpdL general_context Label dB
  | cUpdR dB Label general_context
  | cActive general_context;
```

Isabelle/HOL internally generates rules for a datatype specification most notably induction rules for recursive types and injectivity rules for the constructors. Pattern matching facilitates case analysis proofs crucial for reasoning with complex languages.

This representation of contexts by a specific datatype constructor exploits the power of the efficient datatype feature of Isabelle while at the same time finding a first class representation of the syntactical concept of "context". For the use of contexts we define an operator to "fill" the "hole" enabling a fairly natural notation of `E↑t` for $E[t]$ (remember this substitution is not "capture avoiding" contrarily to the variable substitution).

```
consts Fill :: [general_context, dB]  ⇒  dB ("↑")
```

We use this simple function to illustrate the definition of functions in Isabelle/HOL. Functions over datatypes may be defined in a particularly efficient way in Isabelle/HOL using primitive recursion. Efficient means, in this context, that proofs involving these operators may be mostly solved automatically using automatic rewriting techniques provided in Isabelle. The semantics of the `Fill` operator is described by the following set of equations – again, this substitution is, unlike variable substitution, not "capture avoiding".

```
primrec
  Fill cHole x = x
| Fill  (cObj f T E ) x =  Obj ((FLmap f)((FLlabel f) ↦ (Fill E x))) T
| Fill  (cCall E l) x =  Call (Fill E x) l
| Fill (cUpdL E l (y::dB)) x = Upd (Fill E x) l y
| Fill (cUpdR (y::dB) l E) x = Upd y l (Fill E x)
| Fill (cActive E) x = Active (Fill E x)
```

The rest of this section intensively use this operator and thus illustrates its usefulness.

6.3.3. Semantics

The parallel semantics of ASP_{fun} is given as an inductive relation over this type of configurations encoding the reduction relation $\rightarrow_{\parallel}$ (see Table 2). To give some flavor of the expression of the semantic, we depict only the rule REQUEST; this rule is a crucial one for the calculus, and it gives a representative idea of the other semantic rules. This rule is part of an inductive definition for the reduction relation $\rightarrow_{\parallel}$ on configurations. An inductive definition in Isabelle/HOL defines a set, here the relation $\rightarrow_{\parallel}$, by a set of simple rules. The set defined by an inductive definition is the least set that is closed under those rules.

```
request:
 ⟦ ∀ D ∈ dom C. fk ∉ dom(fst(the (C  D))); C A = Some(m',a');
    m'(fi) = Some(E↑(Call(ActRef B) l s)); C B = Some(mb, t'); noFV s; A≠ B ⟧
⟹ C →∥ C(A ↦ (m'(fi ↦ E↑(FutRef(fk))), a'))(B ↦ (mb(fk ↦ (Call t' l s)), t'))
```

Assumptions are enclosed in Isabelle/HOL's meta-logic brackets ⟦⟧, and conclusion is placed after ⟹. Additionally, a partial function admits a `dom` operator defining the domain of the function, and a partial function returns either `None`, if the function is not defined for this value, or `Some(x)` if the function is defined and returns `x`. `C(A ↦ x)` represents the partial function `C` where `A` is now given the value `x`.

The above code for the `request` rule in Isabelle clearly corresponds to the following rule of the semantics of ASP_{fun}. As one can notice, the main differences in Isabelle are that, first the definition "fresh" has been directly encoded in the rule, and second a few assumptions were used to decompose the source configuration, e.g., `C A' = Some(m',a')` states that the activity `A` of configuration `C` is defined by the couple `m` (the request queue) and `a` (the active object). Even with those minor differences, it is easy to see that both rules express the same behaviour.

REQUEST

$$\frac{f_k \text{ fresh} \qquad noFV(s) \qquad \alpha \neq \beta}{\alpha\,[f_i \mapsto E[\beta.l(s)] :: Q, t] :: \beta[R, t'] :: C \rightarrow_{\parallel} \alpha\,[f_i \mapsto E[f_k] :: Q, t] :: \beta\,[f_k \mapsto t'.l(s) :: R, t'] :: C}$$

6.3.4. Typing and Progress

We skip the description of the proofs related to well-formedness and decide to focus on typing. We first define the following datatypes for object type and configuration type and a constant `typing` for typing judgements. The syntactic sugar: `CT, E ⊢ a : A` abbreviates `(CT, E, a, A) ∈ typing`.

```
datatype type = Object (Label ⇒f (type × type))
datatype Ctype =  TConfig (ActivityRef ⇸ type)(FutureRef ⇸ type)
typing :: [Ctype, ((type× type) list), dB, type] ⇒ bool
```

The most remarkable point in the signature above is the use of `(type×type) list` instead of finite maps from variables to types (cf. Section 6.4). A list is sufficient because of the use of de Bruijn indices: the depth in the list represents the de Bruijn index; and a couple of types is necessary because one represents the type of self, and the other represents the parameter type.

Then this relation `typing` is defined using an inductive definition. The rules of the inductive definition are exactly the typing rules for ASP_{fun} introduced in Section 5. For comparison we show just the rule Type Call.

```
⟦Tconf, env ⊢ a : A; l ∈ dom A; A!l=(B,T);Tconf, env ⊢ b : B⟧
⟹ Tconf, env ⊢ (Call a l b) :  T
```

The operator `!` selects a type field `l` in an object type `A`. Typing for configurations is also defined as presented in Section 5. We completely proved in Isabelle/HOL all the theorems presented in this paper. Theorems are expressed similarly in Isabelle as in the paper version. Below is the subject reduction theorem (Theorem 2). Note that, as ⟹ can only be used at the top-level, ⟶ is used to denote implication inside formulae:

```
theorem Csubject_reduction:    ⊢C: CT ⟹ (∀C'. C →|| C' ⟶ ∃CT'. ⊢C': CT')
```

The theorem `progress_ASP_init_conf` below is a particular instance of the progress theorem employing the previous results that all reachable configurations are finite and have no cycles; it corresponds to Property 7.

```
theorem progress_ASP_init_conf:
 ⟦init_config a →|| C; TConfig empty empty, [] ⊢ a : T; A∈ dom C; fi∈C.RA⟧
⟹ (isresult  C.FA<fi>) ∨
   (∃C'. (C →|| C') ∧(C'.FA<fi>≠C.FA<fi>∨ C.FA<fi> →ς C.FA<fi>))
```

6.4. Locally Nameless Representation

The main advantage of the de Bruijn representation is also its biggest handicap: indices instead of variables get rid of α-conversion problems but are very technical. An unwelcome effect of the lifting and substitution functions, necessary for index handling, is that there are many lemmata that are hard to find and difficult to prove. Their difficulty is not their theoretical depth but that they merely shuffle indices – a facility easy for a machine and hard for a human mind. An illustrative example is the following lemma `subst_subst` proving how two substitutions can be swapped.

```
i < j + 1 ⟹
t[lift v i,lift s' i / Suc j][u[v,s'/j],s[v,s'/j]/i] = t[u,s/i][v,s'/j]
```

The locally nameless representation, on the other hand, is closer to paper style notation due to the use of named free variables in addition to indices. The price to pay for the gained understandability are additional concepts. Consequently, new hypotheses in rules and theorems arise. We believe that both representations have their merits and their weaknesses as we will point out in the following exposition of the locally nameless representation of ASP_{fun}.

6.4.1. Basic Constructs

The only difference of the locally nameless representation to the de Bruijn representation concerning the terms is the addition of named free variables. This new type `fVariables` is conveniently chosen to be the type `string`. The datatype of terms stays the same (it is named `term` now instead of `dB`) – only the constructor `Var` is replaced by two new constructors `Bvar` and `Fvar`, the former taking an index and the latter a free variable. Also at the level of configurations there is not much difference: the type of configurations actually stays the same. In the parallel semantics, the only difference is in the rule LOCAL where local terms need to be *locally closed* in order to be reduced according to the local semantics. The main differences in the locally nameless semantic definition is in the reduction relation for the evaluation of the objects. Here, the new concept of named variables is supported by operations for opening and closing of terms.

Opening and closing

Opening is a form of substitution; it corresponds to an instantiation of a bound variable with a given subterm. While the following definition's core part is the first clause, the others just pass the recursion into the term structure. This first clause replaces a bound variable if `n` matches the index of the parameter. Due to the two parameter types of our terms, we always open with a pair of terms and replace depending on whether the bound is `Self` or `Param`, by the first or second element of the pair, respectively.

```
primrec
open :: [nat, term, term, term] ⇒ term ("{_ → [_,_]} _")
and
open_option :: [nat, term, term, term option] ⇒ term option
where
  open_Bvar: {k→[s,p]}(Bvar b) =
      (case b of (Self i) ⇒ (if (k = i) then s else (Bvar b))
                | (Param i) ⇒ (if (k = i) then p else (Bvar b)))
| open_Fvar: {k→[s,p]}(Fvar x) = Fvar x
| open_Call: {k→[s,p]}(Call t l a) = Call({k→[s,p]}t) l ({k→[s,p]}a)
| open_Upd : {k→[s,p]}(Upd t l u) = Upd({k→[s,p]}t) l ({(Suc k)→[s,p]}u)
| open_Obj : {k→[s,p]}(Obj f T) = Obj(λl.open_option(Suc k) s p (f l)) T
| open_Act : {k→[s,p]}(Active a) = Active ({k→[s,p]} a)
| open_ARef: {k→[s,p]}(ActRef g) = ActRef g
| open_FRef: {k→[s,p]}(FutRef f) = FutRef f
| open_None: open_option k s p None = None
| open_Some: open_option k s p (Some t) = Some ({k→[s,p]}t)
```

Let us only describe the most characteristic of the other clauses: `open_Obj`. Recursive opening inside the object is defined by mapping a function (`λl. ...`) on all its methods (most of them being undefined, `None`). This explains why we use two mutually recursive functions `open` and `open_option`, one of them accepting `Some term` or `None`. The function applied to each member method is the recursive application of open but with `Suc k` as index, because we entered a binder (similarly to what we would do for de Bruijn method).

Open is usually used to replace the outermost binder, i.e., `{0 →[s,p]} t` abbreviated by $\mathtt{t}^{\mathtt{[s,p]}}$. For example, one crucial rule of our semantics of objects is to evaluate calls to an object's method $[l_j \mapsto \varsigma(x,y)t, \ldots].l_j(p)$ to the body with substituted parameters: $t[o/x, p/y]$, where $o = [l_j \mapsto \varsigma(x,y)t, \ldots]$. In locally nameless representation, it is expressed by $\mathtt{t}^{\mathtt{[o,p]}}$.

To abstract a variable, `close` is defined as a primitive recursive function of type `[nat, fVariable, fVariable, term] ⇒ term`. As close corresponds to a method abstraction we chose the syntax `{_ ←[_,_]} _`. Its definition uses identical patterns with open; we thus only show the decisive case for `Fvar`.

```
close_Fvar: {k ← [s,p]}(Fvar x) = (if x = s then (Bvar (Self k))
                    else (if x = p then (Bvar (Param k)) else (Fvar x)))
```

Similarly to open, most of the time we will close the variable indexed by `0`; we thus abbreviate `{0 ←[s,p]} t` by σ`[s,p] t`.

Opening and closing efficiently convert between free and bound variables. Remember, however, that the coexistence of free and bound variables necessitates to restrict propositions to terms without "unbound bound variables": preconditions generally restrict propositions to *locally closed terms*.

The predicate `lc` formalises local closure:

```
inductive lc :: term ⇒ bool
where
  lc_Fvar: lc (Fvar x)
| lc_Call: ⟦ lc t; lc a ⟧ ⟹ lc (Call t l a)
| lc_Upd: ⟦ lc t; finite L; ∀s p. s ∉ L ∧ p ∉ L ∧ s ≠ p ⟶ lc (u[Fvar s,Fvar p])⟧
                                ⟹ lc (Upd t l u)
| lc_Obj: ⟦ finite L; ∀l∈dom f. ∀s p. s ∉ L ∧ p ∉ L ∧ s ≠ p
                                ⟶ lc (the(f l)[Fvar s,Fvar p]) ⟧
                                ⟹ lc (Obj f T)
| lc_Act: lc a ⟶ lc (Active a)
| lc_ARef: lc (ActRef g)
| lc_FRef: lc (FutRef f)
```

An explicit substitution operator with syntax `[x →s] t` replaces a free variable `x` by a term `s` in a term `t`. The structure of its primitive recursive definition is similar to open and close but the decisive `Fvar` case is as follows.

```
subst_Fvar: [z → u](Fvar x) = (if (z = x) then u else (Fvar x))
```

Although we use open for a "substitution" in the semantics, the substitution above is better suited for free variables for example in renaming lemmata.

6.4.2. Cofinite Quantification

One problem when changing from bound to free variable is the need for fresh variables. Whenever we have a rule which uses a newly introduced variable name, we need to find a fresh name. For example, suppose that `t` is a subterm under a binder. To make it locally closed, we need to instantiate the top-level bound variable of `t`: $\texttt{t}^{[\texttt{s},\texttt{p}]}$, but to keep the original term `t` (and close the term later with `s` and `p`), we need `s` and `p` fresh. Technically, we can use a function `FV` collecting the free variables of a term and add the additional premise $\texttt{x} \notin \texttt{FV(t)}$ whenever a fresh variable name x is required. This way of formalising can be described as the "exists-fresh" approach (Aydemir et al., 2008a). Unfortunately, the "exists-fresh" approach leads to very clumsy proofs: intuitively, we need to prove statements for a set of free variables differing from the ones given as hypotheses. In recent work by Aydemir et al. (2008a), a more sophisticated technique called cofinite quantification is introduced that eases the proofs involving such rules. The basic idea (cf. Section 6.1.2) is to abstract from sets of free variables $FV(t)$, but instead consider some arbitrary finite set L, i.e., assuming a "cofinite set" of variable names. Since L is arbitrary, it can be chosen later as a convenient set bigger than the set of free variables. Any naïve way using simply locally nameless representation *without* using cofinite induction in the semantic definition would lead to unsolvable proof obligations for some theorems. Thus the semantics of our calculus in the locally nameless representation is expressed by rules of the form:

$$\textsc{Cofinite-update-LN}\quad \frac{\text{finite } L \qquad \forall x\ y.\ x{\neq}y \wedge x, y{\notin} L \longrightarrow \exists t''.t^{[\texttt{x},\ \texttt{y}]} = t'' \ \wedge\ t' = \varsigma[x,y]t'' \qquad \texttt{lc}\ o}{o.l := t \rightarrow_\varsigma o.l := t'}$$

6.4.3. Semantics and Proofs

When comparing the techniques, two criteria must be considered: how easy it is to write the formalisation, and how easy and convincing it is to read it. Locally nameless terms are definitely easier to read as they use named variables instead of de Bruijn indices. However, in the specification of the syntax and semantics we often encounter some technical overhead due to the new constructors for named free variables. Moreover, we need to establish the well-formedness of terms by adding predicates *lc* to the premises of the reduction rules. Fortunately, the additional *lc* condition mainly states that substituted terms correspond to correct ς-calculus terms. We have seen an example in the previous section when considering the semantic rule Cofinite-update-LN for the local update on objects.

Let us focus on the reduction inside binders. Specifying that any field can be reduced in de Bruijn notation leads to the rule:

```
Obj: ⟦s →ς t; l ∈ dom f⟧⟹ Obj (f (l ↦ s)) T→ςObj (f (l ↦ t)) T
```

This is very similar to the paper version. The locally nameless version is less straightforward: we need cofinite quantification:

```
Obj: ⟦ l ∈ dom f; finite L; lc (Obj f T);
     ∀s p. s∉L ∧ p∉L ∧ s≠p⟶ ∃t''. t^[Fvar s,Fvar p] →ς t'' ∧ t' = σ[s,p] t'') ⟧
     ⟹  Obj (f(l ↦ t)) T →ς Obj (f(l ↦ t')) T
```

Additional requirements refine what is meant by "reduce under the binder"; in fact the difficulty is to make the sub-term under the binder locally closed before reducing it, which somehow refines the intuitive notion of (correct) reduction under binders.

The essential relations of the calculus, reduction and typing, are not more readable in the locally nameless versions compared to the de Bruijn incarnations. In both formalisations, the introduction of syntactic sugar can bring some rules very close to a paper version. However, the more restrictive reduction relation for locally nameless variables is closer to the version found on paper, as it does not apply to terms with dangling indices.

Concerning proofs, the notable benefit comes from the explicit distinction between the variable types, which can improve readability and ease reasoning for many lemmata, especially the basic lemmata and confluence proofs, more cases being proved automatically.

Concerning typing, the locally nameless formalisation improves the understandability of proofs but at the price of rather technical lemmata for renaming. We are not able to observe a major improvement in the complexity of the major proofs but, for the most part, there is no notable burden either. The proof principles are similar for either variable representation.

6.4.4. Overall Comparison with the de Bruijn Approach

The clear advantage of the locally nameless formalisation is the handling of free variables. The de Bruijn version did not allow reasoning about free variables for a very simple reason: it is not possible to express free variables. More precisely, unbound de Bruijn indices could sometimes simulate free variables, but such a solution is unsatisfactory because the intent of a free variable is different from a dangling index. Moreover, the explicit distinction between bound and free variables eases the handling of either kind of variable and enhances the readability of proofs and formalisations.

Cofinite quantification, freshness, and renaming are the major reasons for additional and technical proofs in the locally nameless representation, and all of these items are required for the reasoning about named free variables. The locally nameless rules are more complex than their de Bruijn counterparts because the locally nameless representation introduces new concepts and is precise about well-formedness and closure. This initial formal overhead is paid back by a natural notation in theorems and by improvement for interactive proofs. Overall, the locally nameless technique allows a more precise formalisation, avoids proving obscure lemmata on substitution and lifting, and leads to a more natural notation for terms but at the price of additional non-trivial requirements in semantic and typing rules, and additional non-trivial concepts.

6.5. An Experience in the Formalisation of Calculi and Semantics

The entire development takes around 14000 lines of code for each of the two representations. Among those lines less than 10% are necessary for the formalisation of the languages and the properties, and most of the development concerns the proof of the properties and the intermediate lemmata. The development time is difficult to evaluate but is above one man-year for the two formalisations.

The most difficult and crucial step is certainly the definition of the right model for the calculus, its semantics, but also for the additional constructs used in intermediate lemmata.

Of course, the structure and difficulties of the proofs are highly dependent on the basic structures on which the formalisation relies.

Even if the length and form of the proof is not optimal, the development for formalising such a theory is really consequent; and it becomes difficult to keep a proof minimal and well-structured when it grows to several thousands of lines in length. Handling simplification steps in such a complex and rich theory becomes tricky. Additionally, making modular proofs for subject reduction and equivalent properties is difficult in a theorem prover because useful lemmata are tightly coupled with the numerous and complex hypotheses involved by the case analysis; for example it is difficult to specify a lemma that will be used in case the REQUEST rule has been applied, because such a lemma would have numerous and complex hypotheses.

However, globally, we consider that the formalisation of ASP_{fun} is of reasonable size, and provides a set of constructs relatively easy to use. We think this formalisation can be used efficiently to prove new properties on distributed object languages.

7. Discussion and Alternative Semantics

Reduction contexts. There are different ways of specifying at which point(s) of a term a reduction can occur. A convenient and classical technique for this is to use reduction context (a term with a hole). Reduction occurs at the position of the hole, and the definition of the contexts define the possible reduction points. The most operational semantics generally reduce innermost terms and implement a call by value for method parameters. The most general semantics, like the classical semantics of λ-calculus, allows reduction to occur at any point in the term.

Because it gives the most general results, we chose the general semantics where any part of the terms can be reduced. In particular, we allow the reduction to occur inside binders. This is similar to the general semantics of ς-calculus, as in Definition 6.2.1 of (Abadi and Cardelli, 1996) or even example page 62 showing a reduction inside binders. Then for their "operational semantics", in Section 6.2.4 of (Abadi and Cardelli, 1996), Abadi and Cardelli (1996) use *reduction contexts* that do not allow reduction inside binders: $F ::= \bullet \mid F.l_i(t) \mid F.l_i := \varsigma(x, y)\, s \mid \mathit{Active}(F)$. In ASP_{fun}, those reduction contexts would avoid using noFV requirement in the reductions. We chose to specify the most general semantics – allowing reduction inside binders.

Properties and proofs presented in this paper are also valid for reduction contexts (replacing E by F), and reformulating our results for reduction contexts would be trivial. Indeed all the properties are trivially easier to verify for the reduction with F except progress (all of them are more general for E than for F). But, progress was proved using exactly this reduction context. Consequently progress is also verified by the reduction context semantics.

Communicating non-closed terms. In our semantics we prevented terms with free variables to be communicated in order to avoid variables to escape their binders. Technically, all communication rules require the communicated term s to verify "$\mathit{noFV}(s)$". To avoid this requirement in the semantics an alternative semantics could be provided to communicate

free variables without entailing shared memory; but this is out of the scope of this paper, see (Schmitt, 2002) for example.

Optimising parallel evaluation. A few drawbacks could be found in the semantics given in this paper if a real programming language was to be implemented exactly as specified in Table 2. Indeed a straightforward implementation of our semantics could allow some inefficient execution paths especially because too many communications or computations could occur if no optimisation is done.

First, it seems unreasonable to create in practise as many threads as there are requests in an active object: using a thread pool seems a much better implementation choice.

Additionally, the most critical inefficient point is the possibility to return a future partially evaluated, i.e., the result for a request partially computed. This can result in the computation being done twice which is, in general, not efficient. However, the properties proved here allow enough variation on the semantics to make it usable in practise. In our critical example, it is possible to restrict the REPLY rule to only return completely evaluated futures. Then, if one picks a request, there is no more any guarantee that it can evolve, but the absence of cycle ensures that *some request in the configuration can always be reduced.* Some intermediate reductions have to be added to guarantee the progress property: we first reduce the request(s) calculating the future value before returning the future and progressing. Finally, *returning only completely evaluated futures leads to a more efficient semantics, and still ensures a (weaker) form of progress.*

8. Related Works and Positioning

Distributed Languages: Actors and Objects

Actors (Hewitt et al., 1973) is a widely used paradigm for programming distributed autonomous entities and their interactions by messages. They are rather functional entities but their behaviour can be changed dynamically giving them a state.

Agents and Artifacts with simpA, concentrating on the higher-level of modelling concurrent agent based systems, also feature a calculus (Ricci et al., 2011). Although the formalisation is based on Featherweight Java, the agent concept of Agents and Artifacts resembles ASP_{fun}'s activities but the calculus has no type system and proofs. ASP_{fun} framework may be used to provide formal support to this work.

Obliq (Cardelli, 1995) is based on the ς-calculus; it expresses both parallelism and mobility. It relies on threads communicating with a shared memory. Like in ASP_{fun}, calling a method on a remote object leads to a remote execution but this execution is performed by the original thread. Øjeblik, e.g., (Briais and Nestmann, 2002), a subset of Obliq, equally differs from ASP_{fun} by thread execution. The authors investigate safety of *surrogation* meaning that objects should behave the same independent of migration.

The distributed object calculus by Jeffrey (2000) is based on a concurrent object calculus by Gordon et al. (1997) extended with explicit locations. The main objective is to avoid configurations where one object at one location is being accessed by another. A type system enforces these restrictions. Because migrating objects can carry remote calls, in order to

ensure subject-reduction, Jeffrey introduces serialisable objects, which are non-imperative. Compared to our calculus the most decisive difference is that *activities abstract away the notion of location* and are remotely accessible thanks to a request queue. The concept of futures somehow explicitly supports mobility and serialisation.

Futures

Futures have been studied several times in the programming languages literature originally appearing in Multilisp (Halstead, Jr., 1985) and ABCL (Yonezawa et al., 1987).

$\lambda(fut)$ is a concurrent lambda calculus with futures. It features non determinism primitives (cells and handles). Niehren et al. (2006) define a semantics for this calculus and two type systems. They use futures with explicit creation point in the context of λ-calculus; much in the same spirit as in Multilisp. Alice (Niehren et al., 2007) is an ML-like language that can be considered as an implementation of $\lambda(fut)$.

In (de Boer et al., 2007), the authors provide a language with futures that features "uniform multi-active objects": roughly each method invocation is asynchronous because each object is active. Thus, compared to ASP_{fun}, the calculus has no *Active* primitive. Each object has several current threads, but only one is active at each moment. Each object holding a future may block waiting for the future, or it may use the `await` construct to release the current thread and activate a new one. In this framework, futures are also explicit: a `get` operation retrieves their value. The authors also provide an invariant specification framework for proving properties. This work also formalises the Creol language (Johnsen et al., 2006). Indeed, Creol has exactly the same notion of uniform multi-active objects, and of a single thread active at a time. Johnsen et al. (2006) also provide a type system specifying behavioural interfaces, and a semantics for Creol in Maude. Also note that Abrahám et al. (2009) provide a model of Creol's multi-active objects with futures but they focus on the definition of interfaces and on a safety property on *promises* (a generalisation of futures). To summarize, the main difference between Creol and ASP_{fun} are that future creation and access is explicit in Creol, all Creol objects are active, and the functional nature of ASP_{fun}.

ASP (Caromel and Henrio, 2005) is an imperative distributed object calculus; it is based on the $\varsigma_{\mathbf{imp}}$-calculus (Abadi and Cardelli, 1996). It features asynchronous method calls and transparent futures. No instruction deals directly with futures. Activities in ASP are mono-threaded: one request is served at each moment, and a primitive can be used to select the request to serve. Some confluence properties for ASP have been studied in (Caromel and Henrio, 2005; Caromel et al., 2004). ProActive (Caromel et al., 2006) is an implementation of the ASP calculus.

Dedecker et al. (2006) suggest a communication model, called *AmbientTalk*, based on an actor-like language and adapted to loosely coupled small devices communicating over an ad-hoc network. The communication model is quite similar to the ASP calculus but with queues for message sending, handlers invoked asynchronously, and automatic asynchronous calls on futures. The resulting programming model is slightly different from ASP and ASP_{fun} because there is no blocking synchronisation in AmbientTalk. In AmbientTalk, the flow of control might be difficult to understand for complex applications, because one can never ensure that a future has been returned at a precise point of the program. AmbientTalk

should be dead-lock free but, unfortunately, as no formalisation of the language has been proposed to our knowledge, this has not been formally proved. Our framework could be relatively easily adapted to prove the absence of dead-locks in AmbientTalk by transferring our progress property.

Concerning analysis of programs with futures, Cansado et al. (2008) proposed an automatic way to generate a model of a component application with futures in order to verify its correct behaviour. Note that the objective of our paper is quite different because we aim here at proving generic properties of languages that handle futures whereas Cansado et al. (2008) aim at proving properties of a specific application. However, generic properties proved in ASP_{fun} for the programming model are directly used in the verification approach to know that the specified model fits the reality but also to optimise verification procedures by using generic properties of the language.

Mechanical Proofs For Calculi

One of the greatest contributions of this work is the formalisation of the ASP_{fun} language, its semantics, and type system plus the proof of safety and progress in an interactive theorem prover. We believe that in the discipline of language development the application of mechanical verification is particularly relevant even if it comes at the price of intensive and partly cumbersome work. Related works from the viewpoint of mechanised language verification is the formalisation of the imperative ς-calculus in the theorem prover Coq most prominently using a co-inductive definition and higher order abstract syntax by Ciaffaglione et al. (2007). However, they do not consider concurrency or distribution. With respect to concurrency, the formalisation of the π-calculus in Isabelle/HOL by Roeckl and Hirschkoff (2003) is related. There, higher order abstract syntax is employed. More recent work by Bengtson and Parrow (2007) uses nominal techniques in Isabelle/HOL for the formalisation of the π-calculus. The authors prove many standard results concerning bisimulation and congruence of the calculus. In recent work, they formalised their own generalisation of the π-calculus, the Psi-calculus (Bengtson et al., 2009). Concerning mechanisation of calculi, their solution to model *binding sequences* for nominal datatypes in Isabelle/HOL is interesting because it also shows that generalisations of the nominal package in Isabelle/HOL are necessary and possible (see Section 6.1.2). Unfortunately the design of the π-calculus is too far from ASP_{fun} for this formalisation to be directly useful in our case. Moreover, no objects are introduced neither in the π-calculus nor in its extensions. Ridge (2007) works on a formalisation of concurrent OCaml in Isabelle/HOL. However, he concentrates on concurrency using abstraction techniques to improve automation of concrete algorithm proofs and has not formalised objects at all. The originality of our approach lies in the formalisation of distribution concerns and futures.

Positioning

Futures have been formalised in several settings generally functional-based (Niehren et al., 2006; de Boer et al., 2007; Flanagan and Felleisen, 1999); those developments rely on explicit creation of futures by thread creation primitives in a concurrent setting. They are getting more and more used in real life languages; for example, explicitly created futures

are also featured by the `java.util.concurrency` library. ASP's (Caromel et al., 2004; Caromel and Henrio, 2005) particularities are: distribution, *absence of shared memory*, and *transparent futures*, i.e., futures created *transparently* upon a remote method invocation.

This paper presented a *distributed evaluation* of the *functional* ς-calculus using *transparent* futures and active objects. It can also be seen as a study of the functional fragment of ASP. That is why we consider this calculus as complementary to the preceding ones. Futures can be passed around between different locations in a much transparent way; thanks to its functional nature and its type-system, this calculus ensures progress. Progress for active objects means that evaluation cannot lead to dead-locks. ASP_{fun} is called "functional" because objects are immutable. In ASP_{fun}, activities are organised in an actor-like manner. That is why we consider our language as a form of "functional actors" or "functional active objects". The main novelty of ASP_{fun} is that it is simple enough to allow for a mechanised specification and mechanised proofs of typing properties.

In comparison to the first presentation of the ASP_{fun}-calculus at the FOCLASA-workshop (Henrio and Kammüller, 2009), the current paper better illustrates the semantics and further demonstrates the use of the functional update to personalise services (see Section 3). Moreover, this paper gives a precise description of the Isabelle/HOL formalisation comparing the two different approaches we have implemented for binders (see Section 6). In particular, the second implementation using the concept of locally nameless representation with its recent concept of cofinite induction is an independent contribution. We consider that the major contribution of this paper is the mechanical formalisation, and the precise definition of formalisation tools that can be re-used to mechanically formalise other properties or languages.

Beyond the scope of this paper is a recent prototypical implementation of the ASP_{fun}-calculus in the concurrent language Erlang (Fleck and Kammüller, 2010) intended for the practical exploration of privacy concerns in distributed systems. In a second conceptual paper we show that the functional update of ASP_{fun} can be used to implement a hiding mechanism for private data enabling the enforcement of an information flow property (Kammüller, 2010).

9. Conclusion

We presented a functional calculus for communicating objects and its type system. This work can be seen both as a distributed version of ς-calculus and as an investigation on the functional fragment of ASP. The particular impact of this work relies on the fact that it has been entirely formalised and proved in the Isabelle theorem prover. The functional nature of ASP_{fun} should make it influence directly stateless distributed systems like skeleton programming (Cole, 2004). Our approach could be extended to study frameworks where most of the services are stateless, and the state-full operations can be isolated (access to a database), e.g., workflows and SOA. Our formalisation in a theorem prover should also impact other developments in the domain of semantics for distributed languages.

A calculus of communicating objects

The calculus is an extension of ς-calculus with only the minimal concepts for defining active objects and futures. Syntactically, the extension only requires one new primitive: *Active* creates a new activity from a term. The absence of side-effects and the guarantee of progress make the program easy to reason about and easy to parallelise. ASP_{fun} is distributed in the same sense as ASP: it enables parallel evaluation of activities while being oblivious about the concrete locations in which the execution of these activities takes place. The actual deployment is not part of the programming language and should be provided by an application deployer rather than by the application programmer.

Well-formed terms and absence of cycle

We proved that ASP_{fun} semantics is correct: no reference to non-existing activities or futures can be created by the reduction. Also, no cycle of future or activity references can be created. Thus, starting from an initial configuration, we always reach a well-formed configuration without cycle.

A type system for functional active objects

We extended the simple type system for ς-calculus: *Active* returns an object of the same type as its parameter; activities are typed like their active objects; and futures are typed like the request calculating their value. The type system ensures progress and preservation. Preservation states that the types are not changed during execution. Progress states that a program does not get stuck. In ASP_{fun}, this is due to the following facts:

- The type system plus the subject reduction property ensure that all method calls will access an existing method.
- Well-formedness ensures that all accessed activities and futures exist.
- Absence of cycles prevents cycles of mutually waiting synchronisations and infinite loops of replies.
- As partially evaluated futures can be replied, any chosen request can be reduced.
- All operations are defined for both local and active objects avoiding "syntactical" dead-locks like updating a method of an activity.
- Terms under *evaluation* contexts can be safely communicated between activities.

A Formalisation in Isabelle/HOL. The formalisation adds the necessary quality assurance to a language development where rules and properties are intricate while the need for verification is as worthwhile as imperative. The formalisation is relatively long. It involves the definition of several constructs commonly encountered in the semantics for distributed languages (reduction contexts, references, typing, futures, ...) that we think can be re-used in other developments, at least in the domain of semantics for distributed languages.

In practice we provided two formalisations: one uses de Bruijn indices, and the other uses locally nameless representation for representing variables. Those two approaches have been precisely compared.

The overall framework provides, to our mind, a good basis for the formal study of distributed object languages with futures.

Can we find a better progress property?

Let us analyse the limitations of the progress property.

First, though a reduction is possible, the reduced term can sometimes be identical to the original one. The absence of cycle ensures that such a situation can only occur in the local semantics. This is inherent to the ς-calculus and is out of the scope of this paper.

Second, the reduction can occur in any chosen request but not at any chosen place. Indeed, we can only ensure that points specified in restricted reduction contexts can be reduced. (See the definition of F in Section 6.2). This is a consequence of the fact that objects can only be sent between activities if they do not have free variables that otherwise would escape their binder. This restriction seems both natural and safe.

Future Works. Additional properties could be proved on ASP_{fun}. First of all a proof of confluence for ASP_{fun} could be a good followup to this work. ASP_{fun} is also a good basis to study security or fault-tolerance concerns. More generally, we think that our mechanised formalisation is a good tool to prove properties on communication optimisations and protocols in the context of languages for distributed systems. We also aim at providing a formalisation of an imperative distributed object calculus, like ASP, and further mixing functional and imperative activities.

Bibliography

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

B. C. Pierce, Types and Programming Languages, MIT Press, 2002.

G. Agha, Actors: a model of concurrent computation in distributed systems, MIT Press, Cambridge, MA, USA, 1986.

G. Agha, I. A. Mason, S. F. Smith, C. L. Talcott, A foundation for actor computation, Journal of Functional Programming 7 (1997) 1–72.

C. Hewitt, P. Bishop, R. Steiger, A universal modular actor formalism for artificial intelligence, in: IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1973, pp. 235–245.

G. Agha, An overview of actor languages, ACM SIGPLAN Notices 21 (1986) 58–67.

D. Caromel, C. Delbé, A. di Costanzo, M. Leyton, ProActive: an integrated platform for programming and running applications on grids and P2P systems, Computational Methods in Science and Technology 12 (2006) 69–77.

D. Caromel, L. Henrio, B. Serpette, Asynchronous and deterministic objects, in: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press, 2004, pp. 123–134.

D. Caromel, L. Henrio, A Theory of Distributed Objects, Springer-Verlag, 2005.

A. D. Gordon, P. D. Hankin, S. r. B. Lassen, Compilation and Equivalence of Imperative Objects, in: Proceedings FST+TCS'97, {LNCS}, Springer-Verlag, 1997.

E. B. Johnsen, O. Owe, I. C. Yu, Creol: A type-safe object-oriented model for distributed concurrent systems, Theoretical Computer Science 365 (2006) 23–66.
T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL – A Proof Assistant for Higher-Order Logic, volume 2283 of *LNCS*, Springer-Verlag, 2002.
J. Niehren, J. Schwinghammer, G. Smolka, A concurrent lambda calculus with futures, Theoretical Computer Science 364 (2006) 338–356.
L. Henrio, F. Kammüller, A mechanized model of the theory of objects, in: 9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS), LNCS, Springer, 2007.
B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, S. Weirich, Engineering formal metatheory, in: POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York, NY, USA, 2008a, pp. 3–15.
B. E. Aydemir, A. Bohannon, N. Foster, B. Pierce, J. Vaughan, D. Vytiniotis, G. Washburn, S. Weirich, S. Zdancewic, M. Fairbairn, P. Sewell, The poplmark challenge, Web-site, 2008b.
N. G. D. Bruijn, Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem, Indagationes Mathematicae 34 (1972) 381–392.
C. Urban, et al., Nominal methods group, 2006. Project funded by the German Research Foundation (DFG) within the Emmy-Noether Programme.
A. M. Pitts, Nominal logic, a first order theory of names and binding, Information and Computation 186 (2003) 165–193.
C. Roeckl, D. Hirschkoff, A fully adequate shallow embedding of the π-calculus in isabelle/hol with mechanized syntax analysis, Journal of Functional Programming 13 (2003) 415–451.
A. Ciaffaglione, L. Liquori, M. Miculan, Reasoning about object-based calculi in (co)inductive type theory and the theory of contexts, JAR, Journal of Automated Reasoning 39 (2007) 1–47.
F. Honsell, M. Miculan, I. Scagnetto, pi-calculus in (co)inductive-type theory, Theoretical Computer Science 253 (2001) 239–285.
A. Schmitt, Safe Dynamic Binding in the Join Calculus, in: R. Baeza-Yates, U. Montanari, N. Santoro (Eds.), Proceedings of IFIP TCS 2002, volume 96 of *IFIP*, Kluwer, Montreal, Canada, 2002, pp. 563–575. [This is the original version that was accepted for publication, before the page cut requested for the final version. This version contains additional examples.].
A. Ricci, M. Viroli, G. Piancastelli, simpa: An agent-oriented approach for programming concurrent applications on top of java, Science of Computer Programming 76 (2011) 37 – 62. Selected papers from the 6th International Workshop on the Foundations of Coordination Languages and Software Architectures - FOCLASA'07.
L. Cardelli, A language with distributed scope, in: POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York, NY, USA, 1995, pp. 286–297.
S. Briais, U. Nestmann, Mobile objects "must" move safely, in: B. Jacobs, A. Rensink (Eds.), FMOODS, volume 209 of *IFIP Conference Proceedings*, Kluwer, 2002, pp. 129–146.
A. Jeffrey, A distributed object calculus, in: ACM SIGPLAN Workshop Foundations of Object Oriented Languages.
A. D. Gordon, P. D. Hankin, S. B. Lassen, Compilation and equivalence of imperative objects, in: Proceedings FST+TCS'97, LNCS, Springer-Verlag, 1997.
R. H. Halstead, Jr., Multilisp: A language for concurrent symbolic computation, ACM Transactions on Programming Languages and Systems (TOPLAS) 7 (1985) 501–538.
A. Yonezawa, E. Shibayama, T. Takada, Y. Honda, Modelling and programming in an object-oriented concurrent language ABCL/1, in: A. Yonezawa, M. Tokoro (Eds.), Object-Oriented Concurrent Programming, MIT Press, Cambridge, Massachusets, 1987, pp. 55–89.
J. Niehren, D. Sabel, M. Schmidt-Schauß, J. Schwinghammer, Observational semantics for a concurrent lambda calculus with reference cells and futures, in: 23rd Conference on Mathematical Foundations of Programming Semantics, ENTCS, New Orleans. Accepted.
F. S. de Boer, D. Clarke, E. B. Johnsen, A complete guide to the future., in: R. D. Nicola (Ed.), ESOP,

volume 4421 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 316–330.

E. Abrahám, I. Grabe, A. Grüner, M. Steffen, Behavioral interface description of an object-oriented language with futures and promises, Journal of Logic and Algebraic Programming 78 (2009) 491–518.

J. Dedecker, T. V. Cutsem, S. Mostinckx, T. D'Hondt, W. D. Meuter, Ambient-oriented programming in ambienttalk., in: D. Thomas (Ed.), ECOOP, volume 4067 of *LNCS*, Springer, 2006, pp. 230–254.

A. Cansado, L. Henrio, E. Madelaine, Transparent first-class futures and distributed component, in: International Workshop on Formal Aspects of Component Software (FACS'08), Electronic Notes in Theoretical Computer Science (ENTCS), Malaga, 2008.

J. Bengtson, J. Parrow, Formalising the pi-Calculus using Nominal Logic, in: Proc. of the 10th International Conference on Foundations of Software Science and Computation Structures (FOSSACS), volume 4423 of *LNCS*, pp. 63–77.

J. Bengtson, M. Johansson, J. Parrow, B. Victor, Psi-calculi: Mobile processes, nominal data, and logic, in: LICS '09: Proceedings of the 2009 24th Annual IEEE Symposium on Logic In Computer Science, IEEE Computer Society, Washington, DC, USA, 2009, pp. 39–48.

T. Ridge, Operational reasoning for concurrent caml programs and weak memory models., in: K. Schneider, J. Brandt (Eds.), Theorem Proving for Higher Order Logics, TPHOLs'07, volume 4732 of *LNCS*, Springer, 2007.

C. Flanagan, M. Felleisen, The semantics of future and an application, Journal of Functional Programming 9 (1999) 1–31.

L. Henrio, F. Kammüller, Functional active objects: Typing and formalisation, in: 8th International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA'09, volume 255 of *ENTCS*, Elsevier, 2009, pp. 83–101. Satellite to ICALP'09.

A. Fleck, F. Kammüller, Implementing privacy with erlang active objects, in: 5th International Conference on Internet Monitoring and Protection, ICIMP'10, IEEE, 2010.

F. Kammüller, Using functional active objects to enforce privacy, in: 5th Conf. on Network Architectures and Information Systems Security, SAR-SSI 2010.

M. Cole, Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming, Parallel Comput. 30 (2004) 389–406.