ISSN 1392–124X (print), ISSN 2335–884X (online) INFORMATION TECHNOLOGY AND CONTROL, 2014, T. 43, Nr. 1

Monadic Foundations for Promises in Jason

Alex Muscar, Costin Bădică

University of Craiova, Blvd. Decebal, nr. 107,
RO-200440, Craiova, Romania
e-mail: amuscar@software.ucv.ro

crossref http://dx.doi.org/10.5755/j01.itc.43.1.4586

Abstract. Even though the agent-oriented paradigm (AOP) has lost some of its charm in the past couple of years, the agent community is still active and a large variety of real world applications have been developed lately. Ranging from web applications to mobile applications, the paradigm has shown it is a viable choice. From an overview of these applications Jason seems to be the most widely used AOP language. But, while the core foundation of Jason, the Belief-Desire-Intention (BDI) theory, has gotten a lot of attention over the years, the language is still lacking with respect to some practical aspects such as concurrent programming. In this paper we propose a design for an extension to Jason that makes concurrent programming easier with the aid of promises. The proposed extension is based on a monadic characterisation of promises which makes it possible to express concurrent flows in a more natural way. It also avoids the inversion of control problem inherent when programming with callbacks. We also take into account some of the drawbacks of our proposed approach and investigate some possible solutions.

Keywords: agent-oriented programming; concurrent programming; asynchronous programming; promises; monads.

1. Introduction

Despite its relatively long history (when compared to the entire history of computer science), the agent oriented paradigm has not succeeded in gaining wide traction. While the promise it makes is compelling—greater productivity and a gentle learning curve for novices—it still did not spark practitioners' interest. We believe that this is partly due to the agent community's failure to cater to more pragmatic needs.

While there are plenty of formal models of agency and multi agent systems to choose from, things are somewhat bleaker when it comes to tooling, and here we are referring especially to agent oriented programming languages (AOPLs) [2]. Programming languages are tools, and the agent community also needs to develop tools to implement the formal models it conceives.

Recently, the situation has started to change somewhat, and the interest in AOPLs seems to be on an ascending trend with new proposals for AOPLs [12, 25]. Also, real world applications have been developed using already established languages. More specifically the Jason language [4] has been used to develop applications ranging from web applications [19] to mobile applications [26], thus establishing it is a viable choice. But, while the core foundation of Jason, the Belief-Desire-Intention (BDI) theory, has gotten a lot of attention over the years, the language is still lacking with respect to some practical aspects. Concurrent programming is an example of a domain in which improving the existing solution would be more than welcomed. By making concurrent computation easier to express in agent languages, a whole new area of applications would open up for AOP, from scientific applications like massively parallel simulations to financial applications like *high frequency trading* [27]. We believe that by addressing more pragmatic issues, the agent community has a chance to get noticed by a broader audience.

In this paper we propose a dialect of Jason that makes concurrent programming easier by using the concept of *promises* introduced by Friedman and Wise in [9]. Promises are objects that represent the (yet unknown) result of an ongoing computa- tion which is executing concurrently with other computations in the system. Promises can have *callbacks* attached which will be called when the value of the promise becomes available (i.e. the promise gets *resolved*). Later, languages like E[1] and Alice ML[2] adopted the concept and popularised it [18, 1]. The proposed extension makes it possible to express concurrent flows in a more natural way. At the moment of writing this paper,

[1] http://www.erights.org
[2] http://www.ps.uni-saarland.de/alice

the monadic model for promises (see Section 3 and Section 4) is implemented in Java and we are working toward integrating it with the Jason interpreter[3].

Our investigation of concurrency in AOPLs was prompted (and is part of) a larger research project concerning the development of a dynamic negotiation mechanism and an accompanying framework [22]. We decided to use Jason for implementing an initial prototype, but given the distributed nature of our framework we were soon faced with some of Jason's limitations whose nature we will illustrate in Section 2. While this work does not focus on distributed systems, concurrency is inherently present in such scenarios, so tackling this problem, even in the context of single agents or many agents running on the same machine, will benefit them as well.

The solution we propose avoids the transformation of the program into explicit *continuation passing style* (CPS) [29] inherent when using promises. We also take into account some of the drawbacks of our proposed approach and investigate some possible solutions.

This paper is structured as follows: in Section 2 we illustrate the problem we are addressing by means of a simple example which we will come back to in later sections. In Sections 3 and 4 we present the main aspects of our approach. We further discuss related works and the implications of the proposed approach in Sections 5 and 6, and sketch possible solutions. We conclude in Section 7, where we present some possible directions for future iterations of this approach. In Appendix A, we show the formal definition of a simplified monad, and, finally, in Appendix B we prove that the simplified promise monad respects the monadic laws.

2. Background

In this section we are going to briefly introduce the syntax of Jason and a working example which we will use throughout the rest of the paper.

2.1. Brief overview of Jason

A program is made out of three sections: beliefs, rules and plans. *Beliefs* and *rules* are very similar to facts and rules in Prolog with one syntactic difference: `&` replaces `,` in conjunctions. Goals are introduced by the `! operator`[4]. *Plans*, which follow the generic form `triggering_event : context <- body.`, are intended to handle goals. *Triggering events* match goals and message. The only relevant triggering events used in this paper are goal addition and message arrival, denoted by atoms with the `+!` and the `+` prefixes. The belief base can be manipulated with the aid of the `+` and `-` operators, in the *body* of the plan. They add, and delete respectively, a belief from the belief base[5].

One last piece of information needed to understand the examples used in this paper is related to Jason's use of *internal actions*. They allow the extension of the Jason interpreter by using Java. Internal actions are qualified by their package name (like in Java, e.g. `package.internal_action`). Actions pre-defined by the Jason distribution don't have a package name, but they retain the leading period (e.g. `.send`). For further details we direct the interested readers to [5].

2.2. Problem formulation and a running example

We will employ a simple scenario which involves a single agent working with two social networks: Facebook and Twitter. Its job is to correlate wall posts and tweets for the user[6]. Since both operations can have considerable delays it would be desirable to run both of them concurrently, and not to block the agent while doing so.

A straight forward implementation in Jason would involve two additional agents —`facebook_client` and `twitter_client`—which act as clients for the social networks. In this setup, the main agent would send a message to each of the client agents to fetch the relevant data[7]. While this obeys both previous requirements, it does so at an added cost for the programmer: manually synchronising the responses from the clients. Fig. 1 shows a possible implementation of this approach.

Because we need the responses from two concurrently running operations to continue, and because we don't know in which order the agent we will receive them, we use the knowledge base for synchronisation: when the response from one client has arrived, we store it in the knowledge base and test if the other client has already sent its response. Only when both clients are done, the main agent can go on with its computation. In this scheme the synchronisation is scattered in three different places: the event handlers for the client responses (`+wall_posts` and `+tweets`), the plan for processing the data (`+!correlate`), and the belief base. Fig. 2 illustrates the interactions between the agents in this approach.

[3] The source for the monadic model can be found at https://dl.dropboxusercontent.com/u/3240227/Async.zip.

[4] To be precise the `!` introduces *achievement* goals, Jason also has *test* goals introduced by the `?` operator, but they are not relevant for the purpose of this paper.

[5] Since these operations usually come in pairs, a deletion followed by an addition, Jason offers the shortcut operator `-+` for this purpose.

[6] While admittedly synthetic, there is no reason why this example could not be scaled to multiple (eventually distributed) agents. Also, while in this example we use agents as simple reactive entities, this is not a limitation of the proposed approach, but a consequence of our desire to keep the example simple

[7] Note that for the rest of this paper we assume the presence of two user-define internal actions, `facebook.get_wall_posts` and `twitter.get_tweets`, which allow agents to interface with the social networks.

```
!start.
+!start
    <- .send(facebook_client,  tell,
       get_wall_posts);
       .send(twitter_client,  tell,
       get_tweets).

   +!correlate
    : have_wall_posts(Posts)  &
       have_tweets(Tweets)
       <-...

   +wall_posts(Posts)
    <- +have_wall_posts(Posts);
       !correlate.

    +tweets(Tweets)
    <- +have_tweets(Tweets);
       !correlate.
```

```
// Facebook client

+get_wall_posts[source(A)]
   <- facebook.get_wall_posts(WallPosts);
      .send(A, tell, WallPosts).

// Twitter client

+get_tweets[source(A)]
   <- twitter.get_tweets(Tweets);
      .send(A, tell, Tweets).
```

Figure 1. Main agent and clients

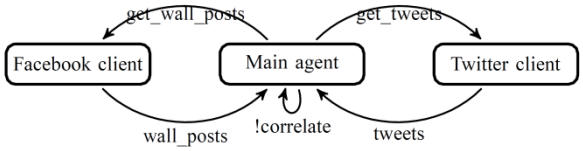

Figure 2. Interactions between the three agents in the straight forward implementation

While this technique leads to the desired behavior, it is not optimal for a couple of reasons:

1. It splits the logic of the program over several execution units – in our case the logic really belongs to the main agent, but it is split over the main agent, and the Facebook and Twitter client agents – which makes it hard to have a global view of the program behavior, especially for more complex scenarios, e.g. where the client agents need access to the belief base of the main agent;
2. It uses one client agent for each such request which leads to inefficient and hard to manage programs. While several strategies are possible – starting all the client agents up-front, dynamically creating and destroying them, or managing a pool of such agents – none of them is optimal, the first two because of the cost of running extra agents, and the third because of the extra complexity involved in managing a pool of client agents; and
3. It does not scale: having to manually store the responses of clients as beliefs for many asynchronous requests and synchronising them by hand is a tedious and error prone task and it leads to mostly duplicate code for handling asynchronous responses (e.g. `+wall_posts, +tweets`).

It is clear that if we want a scalable solution we need to look for an alternative approach.

3. Monadic Foundations for Promises

While most implementations of promises follow some common principles, there is no agreed upon definition of what a promise is. With the recent advent of functional programming there has been a rising interest in using category theoretic concepts, with monads being probably the most prominent example [20]. Using monads to give structure to promises seems promising in the light of projects such the *computation expressions* in F# [30], the C# asynchronous model [17], and Scala promises [13]. Before we go on any further we must first briefly introduce monads.

3.1. Monads

Monads are structures that represent generic computations. They are usually defined as triples composed of a *type constructor* with two associated operations called *unit* (or *return*) and *bind* (often written as the infix operator >>=).

The type constructor defines the monadic type, which is the type of the values that will be used/threaded throughout the computation—e.g. promises. The *unit* operation is used to build simple computations in the given monad. It takes an ordinary value— e.g. an integer, a string—and "lifts" it in the monad. The *bind* operator precisely defines how operations belonging to that computation can be combined.

In order for such a structure to be proper monad the operations must obey three laws, known as, the *monadic laws* [32]. These laws guarantee that by combining two valid computations in the given monad we obtain another valid computation.

3.2. The Promise monad

We will next have another look at promises and try to see how they form a monad. Promises abstract over

```
!start.

+!start <- !social_plan.

+!social_plan[context(promise)]
   <- facebook.get_wall_posts(Posts);
      twitter.get_tweets(Tweets);
      !correlate.
```

```
!start.

+!start
   <- !promise.run(social_plan).

+!social_plan
   <- promise.bind(
       facebook.get_wall_posts, cont1).

+!cont1(WallPosts)
   <- promise.bind(
       twitter.get_tweets, cont2).

   +!cont2(Tweets) <- !correlate.
```

Figure 3. A plan in the promise monad (left) and its hypothetical translation using the operations in the promise monad explicitly (right)

the duration of a computation, denoting a value that will be available at a later time. As we said earlier, the value of a promise can be observed by attaching a callback to the promise. When the promise gets resolved, the callback gets invoked.

The callback is the main component that allows abstracting over time, since it encapsulates the future behaviour of the computation. This hints to a possible shift in our view of promises: instead of looking at promises as containers of (yet unavailable) values we can look at them as computations that when given a callback will (eventually) invoke it in a context where the value of the promise is available.

A more formal definition of the *Promise* monad triple is available in Appendix A, and proofs that it obeys the monadic laws can be found in Appendix B. This is no surprise, since the monad for simplified promises that we have defined is actually the Continuation Passing Style (CPS) monad [8] in disguise. This is indeed consistent with our earlier intuition that promises will continue a computation when the value of the promise becomes available.

Having promises form a monad opens up some interesting options for both the syntax—F#'s computational expressions or Haskell's *do notation* [24]— and semantics—monads can be easily composed in a series of interesting ways—of the language. We will further explore the implications on the design of our solution in Section 4.

4. Implicit Promises

Monads have been called "programmable semicolons" because of the syntactic sugar that they have associated in Haskell, the *do notation* [24]. The notation allows the code to have an imperative feel, while still being declarative in nature.

We will employ a similar scheme in our dialect, by specifying the context in which a plan is going to be executed via annotations. This will allow the programmer to write code in a style that is natural, while allowing the compiler to rewrite the code in order to do the appropriate plumbing. This approach is meant to address the inherent *inversion of control*[8] that arises in concurrent scenarios (see Section 2).

Fig. 3 (left) illustrates our approach. The plan `social_plan` is annotated as executing in the `promise` context. This changes the semantics of the actions inside the plan, making them behave as promise computations instead of regular Jason code.

In order to implement this solution we can employ an approach similar to the one used by the F# language [9] for its *asynchronous computation expressions* [31]. The compiler performs a rewriting transformation that transforms the code in Fig. 3 (left) into code similar to the one in fig. 3 (right). While this technique suffices in the case of F# thanks to its support for anonymous functions that close over variables in their scope (closures), their lack in Jason means that we have to perform an additional transformation, *lambda lifting* [14]. This transformation identifies the variables that a closure would capture and transforms the closure in a top level function with an extra parameter for each captured variable. Using these techniques, the code in Fig. 3 (left) would be rewritten to look like the one in Fig. 3 (right) (modulo some identifiers which would have to be generated by the compiler).

5. Proposal Analysis

The main drawback of our approach is that plans cannot mix computations from different monads. A more accurate description would be that actions from different monads cannot be composed in the same plan, but they are free to *invoke* plans defined in different monads, as long as they define an appropriate invocation mechanism. Because the compiler knows the context in which the plan computations have to run in, it can emit the call to the correct *run* operation. It

[8] Linear control flow is replaced by a scheme where each function call (or predicate/plan) receives an additional functional argument, its "continuation", which is called instead of normally returning a value.

[9] http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/

remains to be seen if this will be flexible enough for real world applications.

Another drawback is that the belief base of an agent represents its global shared state. This is not a problem with regular Jason agents because the interpreter serialises access to the belief base, but, by using external mechanisms to execute actions concurrently this mechanism can be circumvented. This can lead to the typical problems associated with concurrency, e.g. race conditions and deadlocks. This is illustrated by the code in Fig. 4. Suppose this code is part of an agent in charge of running a coffee machine. When the user sends a request, the agent displays a message to the user and starts making the coffee at the same time.

```
+user_request(make_coffee)[context
(promise)]
   <-        lcd.display("Preparing
   coffee");
      -+state(making_coffee);
      !make_coffee.
```

Figure 4. Dangers of global shared state

The key point here is the change in the belief base `-+state(making_coffee)`. This removes any old belief for `state`, and adds a new one with the argument `making_coffee`. If another concurrently running computation were to change the state at the same time, the belief base might be left in an inconsistent state, a classic example of a race condition.

There are a couple of alternatives to address this issue:

- Based on the insight that not all the plans need all the beliefs, we could change the language to introduce local plan beliefs. This is the approach taken by the SimpAL language [25]. While this is an interesting approach for structuring an agent, we feel it would be a disruptive change for the Jason language;
- *Software Transactional Memory* (STM) is a synchronisation mechanism akin to transactions in databases [28]. It is an interesting alternative as it is only a infrastructure change, so it would not surface in the language syntax. But STM has the drawback that the performance of the application may suffer; and
- Represent the internal state of the agent as *synchronised mutable* variables as found in *Concurrent Haskell*[10] [15], which in turn are based on *M-Structures* as presented in [3].

6. Related Works

In [21] we proposed two solutions: the first involving explicit promises, and the second that proposed some changes to the Jason syntax and semantics to accom- modate implicit promises. We clearly analysed in [21] why the former is far from ideal in the context of Jason. Briefly, the lack of anonymous plans in Jason would make the solution cumbersome to use, and it would still lead to inversion of control, because promises' callbacks are defined as separate plans. The latter proposal was an attempt to address this shortcoming by adding support in the language for promises.

Very briefly, it proposed the introduction of a promise composition operator ||, and special treatment of the semicolon operator in the interpreter, in order to capture the continuation of the plan and set it as the promise callback (for an in-depth discussion of the proposed solution cf. [21]). While the proposal was a step in the right direction, it was rather *ad hoc*, and its interaction with the core language semantics was not clear. The solution proposed in this paper is aimed at addressing these ambiguities.

By defining promises as a monad we can take advantage of the monadic prop- erties mentioned earlier. The most important is that composing promises now has a clear semantics. A nice side effect of promises being a monad is that we do not need a special operator for composing promises, we can use the semicolon. This is thanks to the fact that we use the `contest` annotation to specify exactly the type of computations that can be used in a plan. This is a both a blessing and a curse. The downside of running all the computations in a plan inside one monad is that mixing computations is not straightforward. Indeed, this was the exact reason why the semantics proposed in our previous paper was not clear. *Monad transformers* are one possible solution to the monad composition problem [16]. Our language is not expressive enough to allow defining monad transformers, and for now, we have decided that this is an acceptable compromise.

The mechanism proposed for setting plan contexts is not limited to the promise monad. In fact it is similar to the *do notation* from Haskell. As long as appropriate definitions of the *unit* and *bind* operators are available, the context can be used to run plan computations in any monad. This might prove to be an interesting mechanism of experimentation with language features, as many computations can be expressed as monads. We will not explore this direction further in this paper.

Finally, the monad presented in Section 3.2 is a simplified version of a promise monad. It does not handle failure, which is an important and extensive issue in itself. We will address this issue in future research.

A recent effort to address similar issues is the one proposed by Ricci et al. in [25]. While partially targeting the same problems as our own work, the solution proposed by [25] takes a different approach when it comes to concurrency—the simpAL language proposed by the authors uses the Agents & Artifacts

[10] http://www.haskell.org/haskellwiki/GHC/Concurrency

model [23]. While superficially similar, the two approaches are fundamentally different. simpAL uses *artifacts* for coordination, which are passive entities external to the agent. Agents can *subscribe* to artifacts in order to be notified of changes in the artifact's state. Such notifications are handled by using a form of callback with some syntactic sugar, similar to how the E language handles promises. Thus, the flow of the agent depends (at least partially) on outside entities (i.e. artifacts) leading to some degree to the problem we are trying to avoid, inversion of control.

Another early (and interesting) example of an agent oriented programming lan- guage featuring concurrent computation is Go! [6], which has unfortunately mostly gone unnoticed by the community. The model proposed for Go! is closer to our own, with multiple threads of control servicing the same agent. Synchronisation is achieved by using an intra-agent tuple space. This is similar to the approach used by *Concurrent Haskell* (see the end of Section 5).

7. Conclusion and Future Work

In this paper we presented the proposal for a non-intrusive extension for concurrent computations in Jason using a monad for structuring promises.

Being able to easily compose asynchronous computations offers a big advantage for real world scenarios where agents need to use resource that imply latencies, e.g. web services.

The immediate next action is to integrate the implementation of the monadic model in Java with the Jason interpreter.

Once the integration is done, an interesting direction will be to further investigate the semantics of promises in the context of Jason. More specifically, working towards a promise monad that can gracefully handle failure is an important direction.

A related direction is that of finding a synchronisation scheme for the belief base. While orthogonal to the promise model, this is an important aspect nevertheless.

Finally it would be interesting to investigate other asynchronous control constructs based on promises. Some interesting work has already been done for other languages like E and Scala[11].

Acknowledgments

This work was supported by the strategic grant POSDRU/CPP107/DMI1.5/S/78421, Project ID 78421 (2010), co-financed by the European Social Fund "Investing in People, within the Sectoral Operational Programme Human Resources Development 2007" 2013.

[11] http://asyncobjects.sourceforge.net/asyncscala/index.html

References

[1] **D. Aspinall, I. Stark.** Futures and promises in alice ml (2008). URL http://www.inf.ed.ac.uk/teaching/courses/apl/2010-2011/examples/aliceml.pdf.

[2] **C. Badica, Z. Budimac, H. D. Burkhard, M. Ivanovic**. Software agents: Languages, tools, platforms. *Comput. Sci. Inf. Syst.*, 2011, Vol. 8, No. 2, 255–298.

[3] **P. S. Barth, R. S. Nikhil.** Arvind: M-structures: Extending a parallel, non-strict, functional language with state. In: *FPCA*, 1991, pp. 538–568.

[4] **R. H. Bordini, J. F. Hübner, R. Vieira.** Jason and the golden fleece of agent-oriented programming. In: R.H. Bordini, M. Dastani, J. Dix, A.E. Fallah-Seghrouchni (eds.), Multi-Agent Programming, Multiagent Systems, Artificial Societies, and Simulated Organizations, Springer, 2005, Vol. 15, 3–37.

[5] **R. H. Bordini, M. Wooldridge, J. F. Hübner.** Programming Multi-Agent Systems in AgentSpeak using Jason. *Wiley Series in Agent Technology, John Wiley & Sons*. 2007.

[6] **K. L. Clark, F. G. McCabe**. Go! – a multi-paradigm programming language for implementing multi-threaded agents. *Annals of Mathematics and Artificial Intelligence*, 2004, 41 (2-4), 171–206.

[7] **Z. Csörnyei, G. Dévai**. Central European functional programming school. *An Introduction to the Lambda Calculus*. Springer-Verlag, Berlin, Heidelberg. 2008, 87–111. DOI 10.1007/978-3-540-88059-2_3. URL http://dx.doi.org/10.1007/978-3-540-88059-2_3.

[8] **A. Filinski.** Representing monads. In: *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, ACM, POPL '94, pp. 446–457. DOI 10.1145/174675.178047. URL http://doi.acm.org/10.1145/174675.178047.

[9] **D. Friedman, D. Wise.** The Impact of Applicative Programming on Multiprocessing. Tech- nical report. *Indiana University, Bloomington. Computer Science Department.* 1976.

[10] **J. Gibbons**. Calculating functional programs. URL http://www.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/acmmpc-calcfp.pdf, pp. 148–203.

[11] **J. Gibbons, R. Hinze**. Just do it: simple monadic equational reasoning. *SIGPLAN Not.*, 2011, Vol. 46, No. 9, 2–14. DOI 10.1145/2034574.2034777. URL http://doi.acm.org/10.1145/2034574.2034777.

[12] **C. V. Grigore, R. W. Collier**. Af-raf: an agent-oriented programming language with algebraic data types. In: *Proceedings of the compilation of the co-located workshops on DSM'11*, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11, SPLASH'11 Workshops, ACM, New York, USA, 2011, pp. 195–200. DOI 10.1145/2095050.2095081. URL http://doi.acm.org/10.1145/2095050.2095081.

[13] **P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn, V. Jovanovic**. Futures and promises. URL http://docs.scala-lang.org/overviews/core/futures.html.

[14] **T. Johnsson.** Lambda lifting: transforming programs to recursive equations. In: *Proc. of a conference on Functional programming languages and computer architecture*, New York, USA, 1985, pp. 190–203. Springer-Verlag New York.

[15] **S. P. Jones**. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.9622. 2008.

[16] **S. Liang, P. Hudak, M. Jones.** Monad transformers and modular interpreters. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, New York, NY, USA, 1995, pp. 333–343. DOI 10.1145/199448.199528. URL http://doi.acm.org/10.1145/199448.199528.

[17] Microsoft: Asynchronous programming with async and await. URL http://msdn.microsoft.com/en-us/library/vstudio/hh191443.aspx. 2013.

[18] **M. S. Miller, E. D. Tribble, J. Shapiro.** Concurrency among strangers: programming in e as plan coordination. In: *Proceedings of the 1st international conference on Trustworthy global computing*, pp. 195–229. Springer-Verlag, Berlin, Heidelberg. 2005.

[19] **M. Minotti, G. Piancastelli, A. Ricci.** Agent-oriented programming for client-side concurrent web 2.0 applications. In: J. Cordeiro, J. Filipe, W. Aalst, J. Mylopoulos, M. Rosemann, M.J. Shaw, C. Szyperski (eds.), Web Information Systems and Technologies, Lecture Notes in Business Information Processing, Springer Berlin Heidelberg, 2010, Vol. 45, pp. 17–29.

[20] **E. Moggi.** Notions of computation and monads. *Inf. Comput.*, 1991, Vol. 93, No. 1, 55–92.

[21] **A. Muscar**. Extending Jason with promises for concurrent computation. In: G. Fortino, C. Badica, M. Malgeri, R. Unland (eds.), Intelligent Distributed Computing VI, Studies in Computational Intelligence, Springer Berlin Heidelberg, 2013, Vol. 446, pp. 41–50. DOI 10.1007/978-3-642-32524-3_7. URL http://dx.doi.org/10.1007/978-3-642-32524-3_7.

[22] **A. Muscar, C. Badica.** Exploring the design space of a declarative framework for automated negotiation: Initial considerations. In: *L.S. Iliadis, I. Maglogiannis, H. Papadopoulos (eds.)* AIAI (1), IFIP Advances in Information and Communication Technology, Springer, 2012, Vol. 381, pp. 264–273.

[23] **A. Omicini, A. Ricci, M. Viroli**. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 2008, Vol. 17, No. 3, 432–456. DOI 10.1007/s10458-008-9053-x. URL http://dx.doi.org/10.1007/s10458-008-9053-x.

[24] **B. O'Sullivan, J. Goerzen, D. Stewart.** Real World Haskell, 1st edition. *O'Reilly Media, Inc.* 2008.

[25] **A. Ricci, A. Santi.** Designing a general-purpose programming language based on agent- oriented abstractions: the Simpal project. In: *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11, SPLASH '11 Workshops*, ACM, NY, USA, 2011, pp. 159–170. DOI 10.1145/2095050.2095078.

[26] **A. Santi, M. Guidi, A. Ricci.** Jaca-android: An agent-based platform for building smart mobile applications. In: *M. Dastani, A.E. Fallah-Seghrouchni, J. Hübner, J. Leite (eds.) LADS, Lecture Notes in Computer Science, Springer*, 2010, Vol. 6822, pp. 95–114.

[27] International Organization of Securities Commissions, T.C.: Regulatory issues raised by the impact of technological changes on market integrity and efficiency. Tech. rep., International Organization of Securities Commissions. 2011. URL .http://www.iosco.org/library/pubdocs/pdf/IOSCOPD354.pdf.

[28] **N. Shavit, D. Touitou.** Software transactional memory. *Distributed Computing*, 1997, Vol. 10, No. 2, 99–116.

[29] **G. J. Sussman, Jr.** G.L.S.: Scheme: an interpreter for extended lambda calculus. MIT AI Memo 349. *Massachusetts Institute of Technology, Cambridge, Mass.* 1975.

[30] **D. Syme, A. Granicz, A. Cisternino.** Expert F# 2.0, 1st edition. *Apress, Berkely, CA, USA*. 2010.

[31] **D. Syme, T. Petricek, D. Lomov.** The f# asynchronous programming model. In: *R. Rocha, J. Launchbury (eds.) PADL, Lecture Notes in Computer Science, Springer*, 2011, Vol. 6539, 175–189.

[32] **P. Wadler.** Monads for functional programming. In: Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text, Springer-Vergal, London, 1995, 24–52. URL http://dl.acm.org/citation.cfm?id=647698.734146.

Received June 2013.

Appendix A: The Definition of the Promise Monad

A promise for a value of type α is a function which receives a handler that can be called with the value of the promise, and it produces a value of type β[12]. The type constructor for the Promise monad, $M_{promise}$, is defined as:

$$M_{promise} = (\alpha \rightarrow \beta) \rightarrow \beta$$

The unit operation takes a value and returns a promise that will pass the value as an argument to the promise's handler[13]:

$$unit_{promise} = \lambda x.\ \lambda k.\ k\ x \quad (1)$$

[12] We use the standard notation for function types, where $\alpha \rightarrow \beta$ is the type of a function taking an argument of type α and return a value of type β.

[13] We use the standard notation of lambda calculus [7]—functions are introduced by the λ operator which binds a variable in the scope of its body expression, e.g. λvar. body. Function application is denoted by juxtaposition. We use parenthesis for clarity.

The bind operation takes a promise and a continuation of the promise, and returns a promise that will invoke the continuation in a context where the result of the promise is available:

$$\text{bind}_{\text{promise}} = \lambda m.\ \lambda k.\ \lambda c.\ \text{run}\ m\ (\lambda x.\ \text{run}\ (k\ x)\ c) \quad (2)$$

where $\text{run} : M_{\text{promise}}\ \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta$ is a function that executes a promise with the given callback—in our case it is equivalent to function application.

Appendix B: Proofs

In order for any triple made out of a type constructor and the bind and unit operations to form a valid monad, it has to obey three laws: left identity, right identity, and associativity [32]. We will use equational reasoning [10, 11] to prove that this is the case for the Promise monad.

Left identity: $\text{unit}\ x\ =\ f \equiv f\ x$

Proof (Left identity).

$\text{unit}\ x\ = f \equiv (\lambda f_1.\ f_1\ x)\ =\ f$ by (1)

$\equiv \lambda f_2.\ (\lambda f_1.\ f_1\ x)\ (\lambda y.\ (f\ y)\ f_2)$ by (2)

$\equiv \lambda f_2.\ (f\ x)\ f_2$ β-reduction

$\equiv f\ x$ η-conversion

Right identity: $m\ =\ \text{unit} \equiv m$

Proof (Right identity).

$m\ = \text{unit} \equiv m\ = (\lambda x.\ \lambda f.\ f\ x)$ by (1)

$\equiv \lambda f_1.\ m\ (\lambda y.\ ((\lambda x.\ \lambda f.\ f\ x)\ y)\ f_1) m$ by (2)

$\equiv \lambda f_1.\ m\ (\lambda x.\ (\lambda f.\ f\ x)\ f_1)$ β-reduction

$\equiv \lambda f_1.\ m\ (\lambda x.\ f_1\ x)$ β-reduction

$\equiv \lambda f1.\ m\ f1$ η-conversion

$\equiv m$ η-conversion

Associativity: $(m\ =\ f)\ =\ g \equiv m\ =\ (\lambda x.\ (f\ x\ =\ g))$

Proof (Associativity).

$m\ =\ (\lambda x.(\ f\ x\ =\ g)) \equiv m\ = (\lambda x.\ \lambda f_1.\ (f\ x)\ (\lambda y.\ (g\ y)\ f_1))$ by (2)

$\equiv \lambda f_2.\ m\ (\lambda z.\ ((\lambda x.\ \lambda f_1.\ (f\ x)\ (\lambda y.\ (g\ y)\ f_1))\ z)\ f_2)$ by (2)

$\equiv \lambda f_2.\ m\ (\lambda z.\ (\lambda f_1.\ (f\ z)\ (\lambda y.\ (g\ y)\ f_1))\ f_2)$ β-reduction

$\equiv \lambda f_2.\ m\ (\lambda z.\ (f\ z)\ (\lambda y.\ (g\ y)\ f_2))$ β-reduction

$\equiv \lambda f_2.\ (\lambda f_1.\ m\ (\lambda z.\ (f\ z)\ f_1))\ (\lambda y.\ (g\ y)\ f_2)$ η-conversion

$\equiv \lambda f_1.\ m\ (\lambda z.\ (f\ z)\ f_1))\ \ =\ g$ by (2)

$\equiv (m\ =\ f)\ =\ g$ by (2)