Designing and Implementing a Framework for BDI-Style Communicating Agents in Haskell (Position Paper)

Alessandro Solimando* and Riccardo Traverso*

Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi,
Università di Genova, Italy
{alessandro.solimando,riccardo.traverso}@unige.it

Abstract. In this position paper we present the design and prototypical implementation of a framework for BDI-style agents defined as Haskell functions, supporting both the explicit representation of beliefs and backtracking (at the level of individual agents), and asynchronous communication via message passing. The communication layer is separated from the layers implementing the features of individual agents through different stacked monads, while beliefs are represented through atomic or structured values depending on the user's needs. Our long-term goal is to develop a framework for purely functional BDI agents, which is currently missing, in order to take advantage of the features of the functional paradigm, combined with the flexibility of an agent-oriented approach.

1 Introduction

The Belief-Desire-Intention (BDI) model is a well-known software model for programming intelligent rational agents [11]. Only a few frameworks that implement the BDI approach are developed directly on top of logical languages [7], while most of them are built using imperative or object oriented languages. For example, Jason [2] is developed in Java and exploits inheritance and overriding to define selection functions and the environment in a convenient and flexible way. The drawback is that many features natively available in the logic programming paradigm have to be re-implemented from scratch, resulting in a more onerous mixed-paradigm code. For instance, in Jason, unification is needed to find plans relevant to a triggering event, and to resolve logical goals in order to verify that the plan context is a logical consequence of the belief base. BDI-style agents are usually described in a declarative way, no matter how the language interpreter is implemented. The functional paradigm supports pattern matching for free and gives all the advantages of declarativeness; moreover, the use of types for typing communication channels may provide great benefits to guarantee correctness properties both a priori, and during the execution. Nevertheless, to the best of our knowledge no functional frameworks for BDI-style communicating agents have been proposed so far.

* Both authors of this paper are Ph. D. students at the University of Genova, Italy.

In order to fill this gap, we propose a framework for functional agents taking inspiration from the BDI model (although not implementing all of its features), and supporting communication and backtracking. A generic and easily composable architecture should partition the agents' functionalities into several well-separated layers, and in functional programming monads are a powerful abstraction to satisfy these needs. Intuitively, in our solution agents are monadic actions provided with local backtracking features and point-to-point message passing primitives. Their local belief base is stored within variables that are passed down through execution steps. Goals are defined with functions from beliefs to booleans. When it comes to monadic computations, Haskell [3], being strongly based on them, is the best fit. However, even though we focus on a specific architecture, our purpose is not to propose a definitive implementation, but rather to show that this kind of integration is indeed possible without sacrificing or reimplementing fundamental features of different programming paradigms.

Our work is a generalization of [12], where the authors describe a single-agent monadic BDI implementation relying on CHR [6]; we share with [12] the idea of a BDI architecture based on monads, but instead of relying on CHR to represent beliefs and their evolution, the aim of our work is to provide a better integration with the language by handling them directly as Haskell values and expressions.

In [14] agents executing abstract actions relative to deontic specifications (prohibition, permission, and obligation) are simulated in Haskell. Although close to our approach up to some extent, that work does not take the BDI model into account. We are not aware of other proposals using functional languages to represent BDI-style agents.

2 Preliminaries: Haskell

In this section we provide a very brief overview of Haskell's syntax [9], to allow the reader to understand our design choices.

The keyword `data` is used to declare new, possibly polymorphic, data types. A new generic type may be, e.g., `data MyType a b = MyType a a b`: `a` and `b` are two type variables, and the constructor for new values takes (in the order) two `a` arguments and one `b`. A concrete type for `MyType` could be, e.g., `MyType Int String`. A type signature for `f` is written `f :: a`, where `a` is a type expression; an arrow $\rightarrow$ is a right-associative infix operator for defining domain and codomain of functions. A type class is a sort of interface or abstract class that data types may support by declaring an `instance` for it. A special type `()`, called unit, acts as a tuple with arity 0; its only value is also written `()`.

Further information on Haskell and monads can be found in [4,8,3] and in the freely available book [9].

3 Our Framework

In our framework, we split the definition of the capabilities of the agents in different layers by means of monads. The innermost one, `Agent`, provides support for the reasoning that an agent may accomplish in isolation from the rest of the system, that is without any need to communicate. On top of it we build another monad `CAgent` for communicating agents that provides basic message-passing features.

```
data Agent s a = Agent (s → (s,a))
instance Monad (Agent s) where {- omitted -}
```

The declaration of `Agent` follows the definition of the well-known state monad [4]. It is parameterized on two types: the state `s` of the agent, containing its current beliefs, and the return type `a` of the action in the monad. Each action is a function from the current state to the (possibly modified) new one, together with the return value.

At this layer it is safe to introduce goal-directed backtracking support, because computations are local to the agent and no interaction is involved. In Haskell, one could provide a basic backtracking mechanism for a monad `m` by defining an instance of the `MonadPlus` type class. `MonadPlus m` instances must define two methods, `mzero :: m a` and `mplus :: m a → m a → m a`, that respectively represent failure and choice. Infinite computations, i.e. with an infinite number of solutions, can not be safely combined within `MonadPlus` because the program could diverge. In order to address this problem the authors of [5] propose a similar type class – along with a comparison between different implementations – where its operators behave fairly, e.g. solutions from different choices are selected with a round robin policy. In our work we plan to exploit their solutions to give `Agent` the possibility to handle backtracking even in such scenarios. Goals can be defined as predicates `pred :: Agent s Bool` to be used in guards that may stop the computation returning `mzero` whenever the current state does not satisfy `pred`. It is worth noting how this concept of goals fits well into Haskell: such guards are the standard, natural way to use `MonadPlus`.

```
type AgentId = String
data Message a = Message AgentId AgentId a
data AgentChan a = {- omitted -}
```

Another building block for our MAS architecture is the FIFO channel `AgentChan`. We omit the full definition for the sake of brevity: it is sufficient to know that messages have headers identifying sender and receiver agents and a payload of arbitrary type `a`.

```
data CAgentState a = CAgentState AgentId (AgentChan a)
data CAgent s a b = CAgent (CAgentState a → Agent s (CAgentState a, b))
instance Monad (CAgent s a) where {- omitted -}
```

A `CAgent` is, just like before, defined by means of a state monad. It only needs to know its unique identifier and the communication channel to be used for interacting with other agents. This is why, unlike before, the type that holds the state is fixed as `CAgentState`. The function wrapped by `CAgent`, thanks to its codomain `Agent s (CAgentState a, b)`, is able to merge an agent computation within a communicating agent. Intuitively, a `CAgent` can be executed by taking in input the initial `CAgentState` and beliefs base `s`, producing at each intermediate step a value `b` and the new `CAgent` and `Agent` states. The execution flow of a `CAgent` may use functionalities from `Agent`; once the computation moves to the inner monad we gain access to the beliefs base, goals, and backtracking, but all the interaction capabilities are lost until the execution reaches `CAgent` again. Both monads may be concisely defined through the use of the Monad Transformer Library [4], thus many type class instances and utility functions are already given.

A `CAgent` may interact using point-to-point message exchange. The communication interface is summarized below; all functions are blocking and asynchronous, with the exception of `tryRecvMsg` that is non-blocking.

```
myId       :: CAgent s a AgentId
sendMsg    :: AgentId → a → CAgent s a ()
recvMsg    :: CAgent s a (Message a)
tryRecvMsg :: CAgent s a (Maybe (Message a))
```

Given a set of communicating agents, it is straightforward to define a simple module that manages the threads and the synchronization between them.

4 Conclusion and Future Work

We presented a basic architecture based on monads for MAS composed of Haskell agents. Similarly to other solutions, our system provides backtracking capabilities, even if they are limited to the decisions taken between two communication acts.

We have been able to show how the concepts behind MAS can be naturally instantiated in a purely functional language without any particular influence from other paradigms or solutions that may undermine the integration of the framework with the Haskell standard library.

This is still a preliminary work, as the architecture may change to better address the objectives and the prototype of this framework needs to be developed further in order to provide full support for all the described features. Some ideas for future extensions are (1) integrating the backtracking capabilities described in [5], (2) supporting event-based selection of plans, (3) adding communication primitives (e.g. broadcast, multicast), and (4) enriching the communication model with session types [13] in order to check the correctness of ongoing communication along the lines of [1] and [10].

References

1. Ancona, D., Drossopoulou, S., Mascardi, V.: Automatic Generation of Self-Monitoring MASs from Multiparty Global Session Types in Jason. In: Baldoni, M., Dennis, L., Mascardi, V., Vasconcelos, W. (eds.) DALT 2012. LNCS (LNAI), vol. 7784, pp. 76–95. Springer, Heidelberg (2013)
2. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming multi-agent systems in AgentSpeak using Jason, vol. 8. Wiley-Interscience (2008)

3. Hudak, P., Hughes, J., Jones, S.P., Wadler, P.: A history of Haskell: being lazy with class. In: HOPL III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, pp. 12-1–12-55 (2007)
4. Jones, M.: Functional Programming with Overloading and Higher-Order Polymorphism. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 97–136. Springer, Heidelberg (1995)
5. Kiselyov, O., Shan, C., Friedman, D.P., Sabry, A.: Backtracking, interleaving, and terminating monad transformers (functional pearl). In: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, pp. 192–203. ACM, New York (2005)
6. Lam, E.S.L., Sulzmann, M.: Towards agent programming in CHR. CHR 6, 17–31 (2006)
7. Mascardi, V., Demergasso, D., Ancona, D.: Languages for programming BDI-style agents: an overview. In: Proceedings of WOA 2005, pp. 9–15. Pitagora Editrice Bologna (2005)
8. Moggi, E.: Notions of computation and monads. Inf. Comput. 93(1), 55–92 (1991)
9. O'Sullivan, B., Stewart, D.B., Goerzen, J.: Real World Haskell. O'Reilly Media (2009)
10. Pucella, R., Tov, J.A.: Haskell session types with (almost) no class. In: Haskell, pp. 25–36 (2008)
11. Rao, A.S.: AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In: Perram, J., Van de Velde, W. (eds.) MAAMAW 1996. LNCS, vol. 1038, pp. 42–55. Springer, Heidelberg (1996)
12. Sulzmann, M., Lam, E.S.L.: Specifying and Controlling Agents in Haskell
13. Takeuchi, K., Honda, K., Kubo, M.: An Interaction-based Language and its Typing System. In: Halatsis, C., Philokyprou, G., Maritsas, D., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817, pp. 398–413. Springer, Heidelberg (1994)
14. Wyner, A.Z.: A Functional Program for Agents, Actions, and Deontic Specifications. In: Baldoni, M., Endriss, U. (eds.) DALT 2006. LNCS (LNAI), vol. 4327, pp. 239–256. Springer, Heidelberg (2006)