Suitability of Haskell for Multi-Agent Systems
Tanja de Jong
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
t.dejong-4@student.utwente.nl

ABSTRACT
This paper gives a comparison between the functional language Haskell and the agent programming language 3APL, in order to determine the suitability of functional languages for the implementation of multi-agent systems. This was done by implementing a multi-agent system in Haskell, and by performing theoretic research into the functioning of 3APL. After this, the languages were compared in a theoretical manner on the aspects of ease of programming, abstraction, reusability, and potential. The results of this comparison indicate that Haskell could very well be used for implementing a multi-agent system. This shows that the concept of multi-agent systems could be much more widely used, since no knowledge of a specific agent programming language and integration of several programming languages are necessary, if the programmer is experienced in programming with a single functional language.

Keywords
Multi-Agent System, Rational Agent, Logic, Reasoning, Functional Language, Haskell, 3APL, Home Automation System, Java, Prolog

1. INTRODUCTION
This paper compares a multi-agent home automation system entirely programmed in Haskell, which is a general purpose functional language, to multi-agent systems that are programmed in 3APL [9] (pronounced as triple-A-P-L), which is a multi-agent programming language with the specific purpose of developing, implementing and testing a multi-agent system. The system that was programmed in Haskell is from now on referred to as MAHAS
 for Multi-Agent Home Automation System (MAHAS) in HASKell.

As the name implies, a multi-agent system (MAS) is a system that consists of multiple cooperating intelligent agents, that can individually solve problems through reasoning and communicate with each other in order to solve a bigger problem together. An intelligent agent is an autonomous entity that observes the environment through sensors and acts upon the environment using actuators. MAHAS is an implementation of a home automation system in which agents have various roles. There exist agents that represent the residents of the house in a way that each resident is represented by one agent, that is always nearby and is basically a sensor for the location of that specific resident. These agents regularly send an update with its location to the rest of the system. There are other agents in the system that receive those updates and determine whether or not action should be undertaken, based on the received location. If this turns out to be the case, such an agent sends a message to a central agent within the house. This agent will then determine how the house should respond and which devices should update their status, and then sends messages to the associated agents. These associated agents each control a different device in the house and are always aware of the status of that device; they can also control that status.

Agents are often described using logic [5] and can be specified quite well using a logical specification language, because this does not dictate how the implementation should satisfy the specification. It can be useful to analyze and specify a system in terms of concepts incorporated in such logics, such as knowledge, beliefs, desires, intentions, and plans. However, it is not clear how such agent specification languages could relate to a system that is implemented using an arbitrary imperative programming language, because there is no clear correspondence between the concepts used for specification and those used for implementation. Therefore, programming languages that incorporate some of the concepts of agent logics have been introduced, such as 3APL. However, at that time the focus lay on programming multi-agent systems using imperative programming languages, while functional languages seem to have been overlooked. Because of the mathematical nature and certain properties of functional languages, a system programmed entirely in a functional language might be close enough to its logical specification to avoid the issue of a lack of correspondence between the specification and implementation of a system altogether. If functional languages would in fact turn out to be suitable for the implementation of multi-agent systems, this would mean that the concept of multi-agent systems could be much more widely used, since no knowledge of a specific agent programming language would be necessary and instead of integrating several very different programming languages, only one functional language could be used.

For the comparison, 3APL was used because this is a widely known and commonly used multi-agent programming language and there is a lot of suitable literature available about this language.

The goal of this research is to determine the suitability of functional languages for the implementation of multi-agent systems in comparison to the existing agent programming language 3APL, and specifically, determine whether the
mathematical nature of functional languages complies well with the logical reasoning of multi-agent systems and their logical specification. In order to perform this research, a new multi-agent system was implemented in Haskell. For the type of system, the area of home automation was chosen, because home automation is a popular subject nowadays for numerous advantages of automation that can add convenience and safety to people's lives. Therefore, by implementing a new generation of home automation, the system might also turn out to be useful for the research area of home automation, even though that is not the immediate goal of this research.

After this, theoretical research was performed into the functioning of 3APL and Haskell and 3APL were compared in a theoretical manner on the aspects of ease of programming, abstraction, reusability, and potential. The research questions are explained in more detail in section 2, after which the existing work in this research area is discussed in section 3. Some background information is then given in section 4 on the topics of multi-agent systems (4.1), mathematical logic and reasoning (4.2), 3APL (4.3), functional programming (4.4), and home automation (4.5). Next, the functioning of 3APL and MAHAS\(^2\) are explained in more detail in sections 5 and 6 respectively. Subsequently, the comparison between these two is made and discussed in section 7. The conclusion is given in section 8 and, finally, suggestions for future work are given in section 9.

2. RESEARCH QUESTIONS
The purpose of this research is to determine whether functional languages are suitable for the implementation of multi-agent systems. Specifically, the main research question that will be addressed in this paper is: How suitable is Haskell for the definition of a multi-agent system, especially in comparison to a specific-purpose agent programming language?

To determine that suitability, an implementation of a multi-agent system in Haskell will be compared to the general implementation of multi-agent systems in the multi-agent programming language 3APL. This comparison will be made for the following aspects: ease of programming, abstraction, reusability, and potential of the implementation of a multi-agent system in both Haskell and 3APL.

Besides this, it will be investigated whether the mathematical nature of functional languages complies with the logical reasoning of multi-agent systems and their logical specification, because this is the main reason why the suitability of functional languages for multi-agent systems is investigated.

3. RELATED WORK
There is an extensive amount of research on multi-agent systems. Some of this research focuses on the logical aspects and reasoning that goes on within multi-agent systems, such as the research that was performed by Wooldridge [12]. A lot of research focuses on the programming of multi-agent systems, for example on 3APL, such as the studies of Hindriks [8] and Dastani et al. [5]. Conventionally, the agent programming languages discussed in the research were based on imperative programming languages, but increasingly more agent programming languages are designed that are based on functional languages. However, the research mostly focuses on the design and application of languages and frameworks that are specifically designed for the implementation of multi-agent system [4].

In contrast, the research that is discussed in this paper focuses on the question of whether functional languages are generally suitable for this purpose in comparison to these programming languages that are specifically designed for agent programming and, in fact, offer a framework for the implementation of multi-agent systems.

4. BACKGROUND
A background of multi-agent systems and the concepts of agents and the BDI model will be given, along with the role that mathematical logic and reasoning play in multi-agent systems. Besides this, some information will be given about what 3APL is and why it was developed, and what functional programming is and why this might be a better alternative for programming multi-agent systems. Finally, some more information is given about home automation and why this area was chosen for implementation in a multi-agent system for this research.

4.1 Multi-Agent System
A multi-agent system consists of multiple cooperating intelligent agents that can individually solve problems through reasoning and communicate with each other in order to solve a bigger problem together. Multi-agent systems can be used for solving problems that can not - or not as easily - be solved by a single agent or a monolithic system. These types of systems can have several advantages, for example robustness, scalability, and adaptability.

Some of the applications for multi-agent systems are workflow and business process management, distributed sensing, information retrieval and management, electronic commerce, human-computer interfaces, virtual environments, social simulation, and many more [13].

In a multi-agent system, each agent has its own goals and its own set of beliefs about the environment that is gathered by reasoning about sensor input and communication with other agents. The agent will use these beliefs and its goals to reason about which goals it can reach at that moment and to establish plans that consist of actions to reach those goals. After establishing plans, the agent will attempt to achieve its goals by performing actions that are specified in the agent’s plans by using actuators and communication with the other agents, as shown in figure 1.

In this research, a multi-agent system consisting of rational agents, that reason according to the Belief-Desire-Intention (BDI) model, was investigated.

![Figure 1. A BDI multi-agent system](image)

4.1.1 Rational Agents
According to Wooldridge "an agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives." [13]. Thus, it is an entity that acts upon the environment it inhabits. An agent is said to be rational if it chooses to perform actions that are in its own best interests, given the beliefs it has about the world. Therefore, rational agents are expected to have the following properties, according to Wooldridge and Jennings [12]: autonomy, proactiveness, reactivity, and social ability.

4.1.2 The Belief-Desire-Intention Model
The Belief-Desire-Intention (BDI) model is a software model developed for programming intelligent agents. It sees agents’ actions as rational in the sense that they are based on beliefs, desires, and intentions. The beliefs of an agent correspond to the agent’s knowledge about its environment, but may be incomplete or incorrect. An agents desires represent what an agent would want to accomplish and its intentions. A mechanism is provided that separates the activity of selecting plans from the execution of already selected plans. As a result, BDI agents are able to balance the time spent on reasoning about plans and executing those plans by performing actions. This model was originally developed by Michael Bratman [6].

4.2 Mathematical logic and reasoning
Mathematical logic is a subfield of mathematics that is one of the most important techniques available in computer science and artificial intelligence today. Logic is concerned with the field of reasoning and its correctness and can therefore be used in the engineering of rational agents. One of the most important aspects of multi-agent systems is the reasoning that is performed within. Because of reasoning, agents can decide which goals they want to reach and which actions they need to perform to reach them, and the reasoning of agents can be expressed very well in mathematical logic. Therefore, logical models, such as BDI models were developed for programming intelligent agents and specifying a system in terms of concepts incorporated in such logics, such as knowledge, beliefs, desires, intentions, and plans.

4.3 3APL
3APL is a multi-agent programming language for the development, implementation and testing of multiple rational agents using the BDI approach. It provides programming constructs to implement individual agents directly in terms of beliefs, goals, plans, actions, and practical reasoning rules.

4.3.1 Motivation
In research on agents, a distinction is often made between agent theories and agent programming languages. Agent theories are concerned with descriptions of the agents and their behavior. In these theories, agents are often described using logic in which concepts such as knowledge, beliefs, desires, intentions, and plans are often incorporated. For several reasons, it can be useful to analyze and specify a multi-agent system using logic and these incorporated concepts. In order to be able to check a system against its specification, it would make sense to implement that system in a programming language that offers a clear correspondence between the concepts used for specification and those used for implementation. Therefore, several programming languages have been introduced that incorporate some of the concepts from agent logics. One of those languages that were introduced in order to support the practical development of intelligent agents is 3APL.

4.3.2 What is 3APL?
The first version of 3APL was designed by Hindriks et al. The language is relatively simple and implements a 3APL agent by defining formal definitions of agent beliefs, capabilities, goals and plans. 3APL is a combination of imperative and logic programming. The full range of regular programming constructs, like recursive procedures and a notion of state-based computation, are inherited from imperative programming. However, the beliefs, capabilities, goals, and plans must correspond to the logical specification of agents. For this part, logic programming is used, from which computation by querying the belief base of an agent is inherited.

The imperative programming language that is used is Java, an object-oriented imperative programming language, and the logic programming language that is used is Prolog [3]. Prolog is a general purpose logic programming language that is used for solving problems that involve objects and the relationship between objects. It is a declarative language, which means that the program logic is expressed in terms of relations, represented as facts and rules. Because Prolog has its roots in logic, a logical specification of an agent corresponds well to an implementation of the reasoning of that agent in Prolog. Prolog makes use of atomic formulas, which are predicate names that can be parameterized with a number of terms. For agents that are in a block world environment, an atomic formula could, for example, be: on(a,b), which could mean that block a is on top of block b. The parameters of an atomic formula might also be variables. A ground atomic formula is an atomic formula that contains no such variable parameters.

The following is an example of a belief base, implemented in Prolog, that belongs to a 3APL agent in a block world environment:

BeliefBase:
  on(a,fl).
  on(b,fl).
  on(c,a),
  clear(Y) :- not(on(X,Y)).

This belief base indicates that blocks a and b are on the floor, block c is on top of block a, and that any block Y is clear is there is not any block X on top of block Y.

4.4 Functional programming
Functional programming is a programming style that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It puts the emphasis on functions that produce results that depend solely on the input and not on the program state.

4.4.1 Motivation
Because functional programming is of such a mathematical nature, if a multi-agent system would be implemented as an embedded domain specific language in a functional language, this could offer the clear correspondence between the concepts used for specification and those used for implementation. So instead of requiring knowledge of 3APL, Java, and Prolog, a single functional language like Haskell could be suitable for implementing an entire multi-agent system.
4.4.2 What is a functional language?

Functional languages are programming languages that have a mathematical nature and have algebraic data types, which are very suitable for pattern matching. Because of this, agents can reason and communicate using so-called Propositions, which are statements that can either be true or false. This can be modelled by using algebraic datatypes, which can be used in pattern matching. For example, for the home automation system there may be the simplified data type:

\[
\text{data Proposition} = \text{OnLocation String}
\]

\[
| \text{InRange String} \\
| \text{Status Bool Device} \\
| \text{Counter Int} \\
| \text{Not Proposition}
\]

The proposition OnLocation String can be used to let another agent know that a certain agent that models a resident, has updated its location. When another agent receives a message with this particular proposition, it can reason about it and decide if it needs to perform certain actions.

By using a functional language for the implementation of a multi-agent system, the language of the multi-agent system can be embedded in Haskell as a datatype and features like pattern matching can be used.

4.5 Home Automation

Home automation is a popular subject nowadays, because of numerous advantages of automation that can add convenience and safety to people’s lives. Therefore, a new generation of such systems with even more automation is called for.

Home automation is "a way of integrating technology and services in order to gain a higher quality of living", according to Stichting Smart Homes, Nationaal Kenniscentrum Domotica & Slim Wonen [10]. This is currently a very popular area of research as the number of new home installations are expected to increase enormously, according to Telecom research firm Berg Insight [2] and market research and intelligence firm ABI Research [1]. Nowadays, home automation systems can range from simple remote controls to complex networks with many different components that can each have their own form of intelligence and automation. The most intelligent systems that are currently in general use are for example smart TVs and smart thermostats. Automation results in numerous advantages for people, for example:

- added convenience/fun/peace of mind: people have to perform less boring tasks themselves
- saving time: automation does work that people would have to do otherwise
- saving money and better for environment: devices can be turned off automatically when nobody is home
- increased home safety: lights can be turned on automatically when somebody arrives home

Multi-agent systems can be widely used for various applications, among which home automation, and can have many advantages in comparison to single-agent or monolithic systems [7].

5. FUNCTIONING OF 3APL

The implementation of a multi-agent system in 3APL requires two programming languages: a multi-agent programming language that can be used to implement organization and coordination of multi-agent systems directly and explicitly, and a single-agent programming language to implement the individual agents.

5.1 Multi-agent programming language

The 3APL platform was built to support the design, implementation, and execution of a set of 3APL agents that share an external environment. It allows the implementation and execution of a set of 3APL agents and therefore fulfills the function of a 3APL multi-agent programming language. The architecture of the 3APL platform as illustrated by Bordini et al. [5] is given in figure 2.

![Figure 2. The 3APL Platform Architecture](image)

It basically consists of a number of agents and their interaction with the user through the graphical user interface (GUI), the shared environment through an application programming interface (API), and each other through a network. The agent management system (AMS) registers agents that are loaded and executed on the platform and answers a set of questions from agents about other agents that are present on the platform.

5.2 Single-agent programming language

The individual 3APL agents can be implemented by the 3APL programming language that facilitates direct implementation of various aspects of cognitive agents. 3APL separates mental attitudes and the deliberation process that manipulate the mental attitudes. Therefore, the 3APL programming language consists of programming constructs to implement the agent’s mental attitudes, represented as data structures, as well as the agent’s deliberation process, represented as instructions, to manipulate the mental attitudes. The architecture of the individual 3APL agents as illustrated by Bordini et al. [5] is given in figure 3.

A part of the EBNF grammar of 3APL as given by Bordini et al. [5] is given below. It represents the programming constructs that are contained in an individual agent:

\[
\begin{align*}
\text{(Program)} & ::= \text{"Program" (string)} \\
& \quad \text{("Load" (string))} \ ? \\
& \quad \text{"Capabilities :" ( (capabilities) )} \ ? \\
& \quad \text{"BeliefBase :" ( (beliefs) )} \ ? \\
& \quad \text{"GoalBase :" ( (goals) )} \ ? \\
& \quad \text{"PlanBase :" ( (plans) )} \ ? \\
& \quad \text{"PG - rules :" ( (p_rules) )} \ ? \\
& \quad \text{"PR - rules :" ( (r_rules) )} \ ?
\end{align*}
\]
The belief base is implemented by a Prolog program consisting of Prolog facts and rules. Agents can also load a separate Prolog file containing the background knowledge through the optional Load construct. Examples of beliefs are on(a,f1), which means that block a is on the floor, and clear(Y) :- not(on(X,Y)), which means that any block Y is clear if there is no arbitrary block X placed on top of it.

### 5.2.2 Goals
The goals of the agent denote the situation the agent wants to realize, which is implemented by an agent’s goal base. The goal base is implemented by a conjunction of ground Prolog atoms, which are general-purpose names with no inherent meaning that are incorporated in Prolog. An example of a goal is on(a,b) and on(b,c), which means that a is on b and b is on c.

### 5.2.3 Plans
A 3APL agent will adopt plans in order to reach its goals. A plan consists of basic actions that can be composed through program operators. Three 3APL program operators exist: the sequential operator (;), the iteration operator (while-do construct), and the conditional choice operator (if-then-else construct). The plan base of an agent consists of sets of plans. An example of a plan is given below and it means: while there is a block X on the floor that does not have any blocks on top of it, find a block Y that is not directly on the floor and put X on top of Y. So, basically, this plan will move all free blocks on top of blocks that are not free.

```prolog
while (on(X,f1) and not(on(Y,X))) do {
    (on(Y,Z) and not(Z==f1));
    Move(X,f1,Y)
}
```

### 5.2.4 Actions
3APL basically distinguishes five types of actions besides the neutral action c: mental actions, communication actions, external actions, test actions, and abstract plans.

#### Mental actions
A mental action is simply an atom, but the first letter of the predicate name has to be a capital letter. An example of a mental action is: Move(X,Y,Z), which means that X is moved from Y to Z. The execution of a mental action has the effect of updating the belief base of the agent if it is successfully executed. The precondition and post-condition of the execution of a mental action should be specified through the capabilities of the program. In order to realize the effect of the actions as specified in its postcondition, a function is defined in the interpreter that updates the belief base of the agent accordingly. An example of a capability is the following, which means that before X can be moved from Y to Z, X has to be on top of Y, and after X is moved, X should not be on Y and X should be on Z: {on(X,Y)} Move(X,Y,Z) {not(on(X,Y))}, on(X,Z)}

#### Communication actions
Agents can communicate with each other through send actions that send a message from one agent to another. Such a message contains the name of the receiver of the message, the function of the message, and the content. If an agent sends a message to another agent, the belief base of the sender is updated with the belief that the message was sent and the belief base of the receiver is updated with the belief that the message was received. An example of a communication action is Send (Receiver, Performatif, Content), when an agent sends a message to Receiver with the function Performatif and the con-
tent Content.

External actions
External actions can be performed to change the external environment in which the agents operate. The agent performing an external action does not influence the effect of this action, the external environment is responsible for this entirely. When an external action is performed, it references a specific method in the Java class that implements the environment, that specifies the effect of that action in the environment. So, external actions are of the form Java(Classname, Method, List).

Test actions
A test action checks whether a well-formed formula is derivable from the belief base of the agent. The information that can be retrieved from a test action is useful for passing it to other actions for further manipulation. An example of a test action is (on(a,X) and on(X,c))?, which checks whether there is a block X placed on top of block c, with block a placed on top of it. It will execute successfully if such a block exists and will then return the relevant block.

Abstract plans
An abstract plan is an abstract representation of a plan which can be instantiated with a more concrete plan during executing. It cannot be executed directly without rewriting it previously into another plan probably containing executable basic actions. It is represented as an atomic formula.

5.2.5 Reasoning rules
3APL contains goal planning rules and plan revision rules to reason about goals and plans.

Goal planning (PG) rules The goal planning rules are used to generate plans to satisfy specific goals. A goal planning rule states that a certain plan π can be adopted whenever the specified belief condition β is true, after execution of which the goal κ is satisfied. So a goal planning rule is defined as κ <− β | π.

For example the following rule states that if X is on top of Y (the belief condition), X can be moved from Y to Z (the plan) in order to satisfy that X is on top of Z (the goal):

\[ \text{on}(X,Z) \leftarrow \text{on}(X,Y) \lor \text{move}(X,Y,Z) \]

Plan revision (PR) rules The plan revision rules are used to revise plans from the plan base. A plan revision rule is defined as πt <− β | πn. The application of plan revision rules is when a belief condition of an agent prevents the execution of a plan, but the agent has the influence to manipulate that belief condition. For example the following rule states that if an agent wants to move X from Y to Z (initial plan), but it can not perform this plan because X is not clear (belief condition), it will revise the initial plan by first querying which block tt U is on X and then moving C from X to the floor, after which it can perform the initial plan (new plan): Move(X,Y,Z) \leftarrow \text{not(clear(X))} \lor \text{on}(U,X) \lor \text{Move(U,X,Z)} \lor \text{Move(U,X,Y,1)} \lor \text{Move(Y,X,Z)}

A plan revision rule can be applied whenever an agent has a plan that is prefixed by the initial plan of any of its plan revision rules. So, whenever it contains a rule πt <− β | πn and a plan π of the form πi; π′.

5.2.6 Deliberation process
A 3APL agent contains data structures that model its beliefs, goals, plans, and reasoning rules. These structures can be modified by deliberation operations, for example by performing an action. The deliberation process for 3APL was illustrated by Dastani et al. [5] and is given in figure 4.

It shows that a 3APL agent starts its deliberation by searching for applicable goal planning rules, which means that it will try to find goal planning rules that are covenant with any of its goals. If it has found any of these rules, it will select those rules of which the belief condition follows from the agent’s beliefs and apply the first one.

After the agent may or may not have applied a goal planning rule, it will try to find plan revision rules that match any of its plans. If it has found any, it will select the rules of which the belief condition follows from the agent’s beliefs and apply the first one.

Again, the agent may or may not have applied a rule, and it will select a plan after this. If a plan is found, it will be executed. Finally, the agent will conclude whether it has performed any activities in this round of deliberation. If it has, there might be more applicable rules or executable plan available and the agent will start the cycle again. If it has not, nothing has changed and the agent will ascertain that it still can not perform any actions. Therefore, it will wait until it receives a message that may make either a rule applicable or a plan executable.

5.3 Shared environment
The shared environment of the agents in a 3APL multi-agent system can be implemented in Java. This Java class contains instance variables that represent the state of the environment, and methods that corresponds to the ex-
ternal actions that can be performed by agents. When a 3APL agent performs an external action, it has the form Java(Classname, Method, List), where Classname is the name of the Java class that implements the environment, Method is the name of the method that performs the action, and List is a list of returned values.

5.4 To be implemented
When a system is implemented in 3APL, the programmer must implement both the individual agents, and the external environment that is shared by the agents, but 3APL basically offers a framework to do so. A 3APL platform exists that provides a user interface that allows 3APL agents to be programmed, loaded, and executed. Basically, a programmer can use the platform to create a new agent and, within the corresponding tab, specify its initial capabilities, belief base, goal base, plan base, goal planning rules, and plan revision rules as explained previously in this section.

Besides this, the programmer needs to implement the shared environment as a Java program. The external actions of agent need to be implemented by methods of the Java program and the state of the environment must be represented by data structures in the Java program. Specifically, the environment Java program is modelled as a plugin to the 3APL platform. The plugin functions as an interface between the platform and the Java classes and facilitates the interactions between individual agents running on the platform and the instantiation of the Java classes. In order to create a plugin, three interfaces need to be implemented that will be loaded by the platform at startup and that are queried for their Java methods that are provided for individual agents, the Plugin class, the Instance class, and the Method class.

Detailed information about the deployment of this platform can be found in the 3APL user guide [11].

6. FUNCTIONING OF MAHAS²
MAHAS² is a multi-agent home automation system entirely programmed in Haskell. It consists of a language, that contains all the data types, and all functions that are necessary in order to run the system by arranging the communication between the agents and manipulating the agents and the external environment by deciding on plans and letting agents perform the corresponding actions to reach their goals. The goal of the system is to reason about when residents of a household are expected to return home in order to prepare the house by, for example, turning on the heater or the radio.

6.1 Language
Haskell offers algebraic data types which are used for the implementation of the language. The language of MAHAS² is given in listing 1.

<table>
<thead>
<tr>
<th>Listing 1. Language of MAHAS²</th>
</tr>
</thead>
<tbody>
<tr>
<td>data HomeAutomationSystem = System { agents :: [Agent], devices :: [DeviceStatus], counter :: Int }</td>
</tr>
<tr>
<td>data Agent = Agent { identification :: String, goals :: [Goal], beliefs :: [Proposition], plans :: [Action], location :: Location }</td>
</tr>
</tbody>
</table>

The language of MAHAS² contains only what is minimally required for the implementation of the home automation system, but it could of course be extended with for example connectors like and or or.

As can be seen, the system contains information about the agents, the devices in the house, and a counter that counts the communication cycles of the system. An agent consists of an identification name, its goals, beliefs, and plans, and its location.

6.1.1 Beliefs
The beliefs are quite similar to the beliefs of a 3APL agent and are simply facts that can be true or false. A specific belief is a type of proposition and there are five propositions possible. The propositions specify that the agent that contains the proposition in its beliefs knows that:

- **OnLocation String**: the specific agent, whose identification is equal to the String parameter of the proposition, has updated its location.
- **InRange String**: the specific agent, whose identification is equal to the String parameter of the proposition, is within the range of returning home.
- **Status Bool Device**: the device is turned on if the boolean parameter is true, and the device is turned off if the boolean parameter is false.
- **Counter Int**: the system has performed the number of rounds of communication specified in the integer parameter.
- **NotProposition**: the proposition specified in the proposition parameter is not true.

6.1.2 Goals
A goal of a MAHAS²-agent consists of a list of conditions and a list of actions. If all the conditions are satisfied, the agent will perform the corresponding actions, so unlike 3APL a goal determines specifically which actions an agent wants to perform instead of which state it wants to reach. This was simply a straightforward way for implementing the home automation system, but it could also
be implemented similarly to 3APL.

These conditions are basically an extension to the existing propositions that denote the beliefs of an agent, and consist of some of these propositions but without their variables. This is done to avoid the problem of having to initialize the parameters of the propositions when the goals of an agent are initialized. For example, if an agent has the goal: if a message with a location update is received ([Condition], send a message with that location update to a certain other agent ([Action]). If the general propositions would be used, this would mean that OnLocation String would have to initialize the String argument, which denotes the agent that updated its location. However, the goal does not want the condition to be that an update is required from a specific agent, it only wants to receive an update from some agent. For this reason, the data type Condition is used. If an agent later on needs to know to which agent the location update belonged, a method exists to extracts that condition from its beliefs.

6.1.3 Plans

The plans of an agent are simply a list of actions that the agent is going to perform in the current round of the system, in order to reach one or more goals.

6.1.4 Actions

A number of actions exist that an agent can perform to reach one or more of its goals:

- **Set Bool Device**: the agent should turn the device specified by the device parameter on or off, depending on the value of the boolean parameter.
- **Send String Update**: the agent should send a message to the agent, whose identification is equal to the String parameter, containing information about its belief that is specified in the update parameter.
- **DetermineRange**: the agent should determine the range of an agent of which it has received a location update. This action can obviously only be performed if the agent already has knowledge of the location of some agent.

6.1.5 Reasoning rules

MAHAS² does not contain specific rules for reasoning that are part of the agent specification as 3APL does. Instead, methods exist that, for each deliberation round, will add the propositions in each agent’s incoming messages to its beliefs by checking whether the negated version of the proposition already exists in the beliefs. If it does, it will replace the negated version by the new proposition and if does not, it will simply add the proposition to the beliefs. This can be done in such a simple way, because not is the only logical constructor available in MAHAS². After this, the system will check which of the agent’s goals follow from its current beliefs. If it has found reachable goals, it will construct plans from the actions of all the goals and perform them. After this, the agent will start again by checking if any goals can be reached and continue this cycle until no other goals can be reached. It will then wait until it receives a message that might change its current beliefs. So these methods basically correspond to the goal planning rules of 3APL.

MAHAS², however, does not contain any plan revision rules, because they are not needed for the current implementation of the home automation system. It should be possible to implement these rules similarly to the way they are implemented in 3APL, by adding them as constructs to agents. This way, a function could be implemented that compares the non executable plans of an agent to all the plan revision rules to check if a plan matches a plan revision rule and revise that plan.

6.1.6 Deliberation process

All the functions that implement the reasoning of the system and update the system accordingly are implemented in Haskell as well. A user can start running the system by executing a function that expects a Home Automation System and a list of incoming messages for each of the agents of the system. The function will then continually process the messages for each agent. The deliberation process of this system is illustrated in figure 5. This means that the following will happen for each agent in every round:

- For each message, recursively, the proposition will be extracted.
- The system will combine the agent’s existing beliefs with the new proposition and determine how to do so by reasoning about it, as explained in section 6.1.5.
- The system will reason about the agent’s beliefs and all of the agent’s goals to determine which of the goals’ conditions are satisfied by the current knowledge that the agent possesses.
- The actions of each goal, for which the condition is satisfied, will then be added to the agent’s plans.
- A function will be executed that takes the first action of the plans of the agent and, through pattern matching, updates the agent and the system accordingly. If this results in the agent sending new messages, these outgoing messages will be returned.
- The same happens for all other actions until the agent has no plans left.
- The updated system and all outgoing messages will be returned and the steps above will be performed for all the other agents and their incoming messages.
- After this is done, all outgoing messages will be returned and grouped per receiving agent, along with the updated system.
- The system will start the next round by processing all new incoming messages in the updated system.

6.2 Shared environment

The environment consists of a Haskell record that denotes the system and that contains a list of agents, a list of the devices that can be manipulated by the agents, and a counter that counts how many communication cycles the system has completed. The environment is updated every cycle of the deliberation process, and the updated version is retrieved for the next cycle. This entails that every function that might change the state of the environment expects the environment as an argument and returns the
6.3 To be implemented

In order to program a multi-agent system in Haskell without the use of a framework, the entire language and all reasoning needed to be implemented needs to be initialized, for which all general Haskell facilities can be used. This would mean that the programmer would have to construct the language from data types and implement the deliberation process through Haskell functions. However, a programmer could also decide to implement a Haskell framework that could offer the same possibilities as the 3APL framework, in which case the language would already exist, so the agents would only have to be initialized, and a part of the deliberation process would have to be implemented. It would be possible to already implement parts of the deliberation process, such as the communication between agents, and the reasoning about how propositions would be added to an agent’s beliefs by use of logical connectors, such as and, or, and not.

7. COMPARISON BETWEEN 3APL AND HASKELL

One of the goals of this research was to compare programming a multi-agent system in Haskell to programming such a system in 3APL for the following properties: ease of programming, abstraction, reusability, and potential. So far, both implementations have been discussed in this paper and will now be compared one by one for each of those properties.

7.1 Ease of programming

Because no system has actually been implemented in 3APL and only a very basic system has been implemented in Haskell, the comparison for ease of programming is mostly theoretical. It seems that the implementation of a system in 3APL differs in several ways from the implementation of a multi-agent system in Haskell. The most pronounced aspects in which the implementations could differ are the way that agents are implemented, the need for integration of programming languages, and the interface between the programmer and the programming language.

7.1.1 Implementation of agents

In both 3APL and Haskell, rational agents can be modeled using the BDI approach. However, in 3APL, the constructs that an agent consists of (beliefs, goals, plans, etcetera) can be implemented within the platform through Prolog atoms, which can contain variables. In Haskell, an agent is implemented by initializing all of its constructs and their parameters, thus not allowing any variables. However, this could be realized by using functions within the constructs for the initialization, but this would require extensive knowledge of Haskell. The problem could also be avoided by creating additional data types, as was done in MAHAS² with the Condition data type, but this is quite circumstantial.

7.1.2 Integration of programming languages

3APL is a language that integrates both Prolog, a declarative language, and Java, an imperative language. Prolog is used for the logical aspects and the reasoning within the system and is very practical in order to have a clear correspondence between the concepts used for specification and those used for integration. Java is used, however, to model the external environment and perform the actions that influence the agents and the environment. So, a user can use the platform to develop the agents, while writing programs in Prolog that contain the logical information, such as the beliefs and goals of an agent, and using Java to program the environment and the functions that actually decide what happens when an agent performs an action and updates the environment accordingly. It also requires knowledge of plugins, because the environment in Java is implemented within plugin interfaces.

In Haskell, a multi-agent system can be programmed entirely as an embedded domain specific language and it therefore does not need any integration. A user can program the language of a multi-agent system using algebraic data types and also initialize the values of an agent, such as the beliefs, in Haskell. The functions that perform the action and that reason about the agents and the environment, can also be written in Haskell. Therefore, no integration is necessary.

7.1.3 User interface

3APL offers a platform that provides a user interface that allows 3APL agents to be programmed, loaded, and executed. It also contains various facilities to aid programmers for example monitor the exchanges of messages between agents or monitor changes in their mental attitudes. A multi-agent system can simply be programmed as an embedded domain-specific language in Haskell, and can therefore be implemented in a text editor and executed using GHC, which is an interactive environment in which Haskell expressions can be interactively evaluated and programs can be interpreted. This way, no specific user interface is provided, but all general Haskell facilities can be used. However, it would be possible to implement an interface in Haskell in order to provide a framework for the general implementation of multi-agent systems, as is provided in 3APL.

7.2 Abstraction

3APL is a programming language with the specific purpose of programming multi-agent systems, while Haskell is a general-purpose programming language, so in this way, Haskell is a lot more abstract. However, at this moment an implementation of a multi-agent system in 3APL is more abstract than an implementation of a multi-agent system in Haskell, because 3APL offers a general template for agents with abstract plans, while in Haskell, the system must be implemented quite specifically in an embedded domain-specific language. However, if a framework would be implemented in Haskell for the implementation of multi-agent systems, these predefined constructs could also be included, which would result in a framework in Haskell that could offer similar abstraction as the 3APL framework.

7.3 Reusability

If a multi-agent system would be implemented in Haskell the way MAHAS² was implemented, it would require the implementation of an embedded domain-specific language with all its details and functions for reasoning and executing actions, while 3APL offers a framework in which the agents can simply be initialized, and the shared environment can be implemented by implementing the data structures that represent the state of the system and the functions that perform the external actions within the Java files that are given as plugin interfaces. In this case, 3APL seems to be more reusable. However, if a framework in Haskell would be created as mentioned before, this could offer the same means of abstraction as the 3APL framework offers.
7.4 Potential
Because Haskell is a general purpose programming language and a multi-agent system can be implemented in it entirely by using an embedded domain-specific language, a programmer is entirely free in implementing anything. 3APL is a lot more specific, and the only part of the platform that is programmable in the platform is the shared environment [5], so a programmer has less potential for the implementation of a multi-agent system. However, it seems that most multi-agent systems can generally be programmed in 3APL.

8. CONCLUSION
First of all, it must be mentioned that a multi-agent system in Haskell could probably be implemented a lot more efficiently than MAHAS$^2$ was implemented, if enough experience and time would be available. In general, implementing a multi-agent system in Haskell seems to be somewhat more difficult than implementing a multi-agent system in 3APL, because the complexity of such a system is not straightforward in certain aspects when entirely implemented in a functional language. This mostly comes forward when initializing agents with abstract goals, as was mentioned earlier in this paper. On the other hand, if one wants to implement a multi-agent system in 3APL, knowledge of the 3APL platform, Prolog, Java, and plugins is needed. It is difficult to say which option offers more ease of programming, but based solely on this aspect, Haskell could be a suitable choice for the definition of a multi-agent system.

When comparing abstraction within a multi-agent system in particular, Haskell would be the more suitable choice, because it can offer the same abstraction that 3APL offers by implementing a similar framework, but one could also implement an entirely new multi-agent system in which a lot more would be possible.

3APL is more reusable in the way that the platform and its integration with Prolog and Java can be used, but since such a framework would also be possible for Haskell, there is not a big difference in the reusability.

And for the proper potential, Haskell seems to be a better option, because a programmer is entirely free within the general-purpose boundaries of Haskell. In 3APL, a programmer is confined to only initializing agents and implementing the external environment.

So, based on these aspects of comparison, it can be concluded that Haskell shows potential for the implementation of multi-agent systems, and therefore functional languages in general as well. Even though it is not yet evident from this research whether implementing a multi-agent system in a general purpose programming language such as Haskell is as good an option or even a better option than implementing a multi-agent system in a specific multi-agent language such as 3APL, it has shown that more research into this topic could give promising results.

Furthermore, the research shows that it is mostly straightforward to implement the logical reasoning of multi-agent systems in Haskell, because of its mathematical nature. Because of this, it was generally quite simple to implement the entire system in Haskell, without the need for integration of several languages, as is done in 3APL.

9. FUTURE WORK
For this research, a specific implementation of a multi-agent home automation system in Haskell was compared to 3APL in general. However, these two languages can be more specifically compared for implementing a multi-agent system, if a specific system would be implemented in both languages with exactly the same functioning. Therefore, it would be relevant for future work to compare implementations in both languages that are equivalent to each other.

Besides this, there is a newer version of 3APL that is called 2APL, so it would be useful to compare Haskell to 2APL in future research if enough documentation of 2APL is available.

At last, the implementation of MAHAS$^2$ in Haskell could be improved. One of the major issues in MAHAS$^2$ was the initialization of abstract goals, and this could probably be implemented much more efficiently with extensive knowledge of Haskell. If this could be done in future studies, it might significantly improve the position of Haskell regarding 3APL.

10. REFERENCES
[10] Stichting Smart Homes, Nationaal Kenniscentrum Domotica & Slim Wonen. The integration of technology and services in the home environment.