Specifying and Controlling Agents in Haskell

Martin Sulzmann and Edmund S. L. Lam

School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
{sulzmann,lamsoonl}@comp.nus.edu.sg

Abstract. We present a domain-specific language embedded into Haskell to program software agents.We use monads to maintain a clear separation among the different levels of reasonings, e.g. lower-level reasoning operations concerning the agent's beliefs and actions versus higher-level operations dealing with the agent's goals and plans. This serves as a basis for composable high level abstractions supporting complex specification and control regimes in a concise and reusable manner.

1 Introduction

Software agents are entities that carry out complex tasks pro-actively in a dynamically changing environment. They have been successfully applied in numerous areas such as air-traffic controlling, tele-communications, space exploration and business processes. While there exist well-established frameworks to specify agents, there is no general consense how to implement them in todays general-purpose programming languages. Hence, there is often a significant gap between agent's specification and the actual implementation.

In this paper, we aim to close this gap by designing an agent-oriented language embedded into Haskell. The key idea of our approach is to maintain a clear distinction between low-level and high-level reasoning operations. Low-level operations involve the agent's beliefs and actions whereas high-level operations involve the agent's goals and plans. Each level of reasoning (computation) is described by a monad [22, 16]. Thus, we can maintain a clear separation among the different levels of reasoning and support the modular development of more complex reasoning operations.

Typical low-level reasoning operations involve non-monotonic updates. For instance, consider an agent moving a certain object from one to location to another. As we will see, linear logic [8] is the natural choice to model this type of reasoning. In our implementation we employ Constraint Handling Rules (CHRs) [7], which are a executable specification of a fragment of linear logic [3], to specify the agent's lower level reasoning. High-level reasoning operations deal with the agent's goals and plans. For example, consider the goal of moving an object to a designated location. A plan is a series of low-level actions to achieve a goal. Finding and implementing the plan is the programmer's task. However, we can assist the programmer, by introducing sophisticated control-structures such as back-tracking which are implemented on top of the low-level operations.

Specifically, our contributions are:

- We define an agent domain monad to implement the agent's low-level reasoning operations. The basic agent domain operations such as actions and queries are specifiable via CHRs (Section 2).
- We demonstrate how to introduce higher-level reasoning operations via a goal-directed planning monad which is based on a back-tracking mechanism (Section 3).

The implementation including examples is available via [2]. To keep the presentation simple, we have resisted to impose a type system on our agent-specific language extension. This would be possible using Haskell's advanced typing features such as generalized algebraic data types [18] and type classes with extensions [21].

Next, we review the basics of BDI agents on which our approach is based. We conclude in Section 4 where we also discuss related work.

The present paper is based on previous work reported in [12]. Our implementation makes use of Gregory J. Duck's CHR solver written in Haskell [6]. We assume that the reader is familiar with the concept of monads in Haskell [17].

1.1 Background: BDI Agents

The **Belief-Desire-Intention** (BDI) agent model [4, 20] allows to explain human practical reasoning and future directed intentions. Fueled by the increasing demands for complex systems performing high level management and control tasks in dynamic environments, the BDI model has been adopted by many researchers of agent-oriented systems as a way of representing the mental attitudes, motivations and decisions of autonomous agents in such complex dynamic environments.

There exist many different interpretations of the BDI agent model. We will consider a simple example known as the Block World to explain our interpretation of the BDI agent model. The Block world comprises of several blocks and tables. Blocks must be stacked on top of tables and can be moved by a robotic arm (the agent) by means of 2 primitive actions, *get* some block x, and *put* some x *on* some y. The aim of the robot is to solve the problem of building a tower of blocks in a certain configuration, starting from an initial state. To describe the agent's knowledge of its domain, we introduce the following predicates:

Predicates	**Meaning**
$On(x, y)$	x is on y, where $x \in Blocks \quad y \in Blocks \cup Tables$
$Clear(x)$	top of x is clear, where $x \in Blocks$
$Empty$	Robot arm is empty
$Holds(x)$	Robot arm is holding x, where $x \in Blocks$

Predicates are the basic atoms of the agent's belief-base. The Block World agent can be described in terms of the BDI model as follows:

– **Beliefs** represent the agent's understanding of the current state of its world. The agent's beliefs also includes inference rules that represent the laws of the domain, as well as the changes caused by actions (eg. *get* and *putOn*) executed in the domain. Such changes in the agent's belief are necessarily non-monotonic, because of the 'linear' nature of the world. For example, an instance of the agent's belief of block world may include the predicate $On(B1, B2)$, stating that block $B1$ is on top of block $B2$. But this belief will no longer hold after the agent executes *get* block $B1$.

  Hence, we will use linear logic where predicates are treated as expendable resources rather than irrefutable truths. This corresponds more to properties of a dynamically changing physical domain. The Block World actions can be accurately represented by the following *linear implications*:

  $$\begin{array}{ll} \textit{(get)} & !(\forall x, y.\ \mathit{Empty} \otimes \mathit{Clear}(x) \otimes \mathit{On}(x, y) \multimap \mathit{Holds}(x) \otimes \mathit{Clear}(y)) \\ \textit{(putOn)} & !(\forall x, y.\ \mathit{Holds}(x) \otimes \mathit{Clear}(y) \multimap \mathit{Empty} \otimes \mathit{On}(x, y)) \end{array}$$

  Informally speaking, the implication ($\multimap$) says replace the left-hand-side conjuncts ($\otimes$) with the right-hand-side. For example, the application of *get* to $\mathit{Empty} \otimes \mathit{Clear}(B1) \otimes \mathit{On}(B1, B2)$ will result in the removal of *Empty* (among others) before introduction of $Holds(B1)$ (among others), thus avoiding inconsistency.

– **Desires** (more commonly known as goals) represent the state of the world which the agent is attempting to achieve. Desires of a Block World agent are simply the stacking configuration of the blocks which the agent must achieve. For example, from the initial configuration

  $$\mathit{Empty}, \mathit{On}(B1, B2), \mathit{On}(B2, B3), \mathit{On}(B3, T1), \mathit{Clear}(B1), \mathit{Clear}(T2), \mathit{Clear}(T3)$$

  the agent tries to achieve the goal

  $$\mathit{Empty}, \mathit{On}(B1, B2), \mathit{On}(B2, B3), \mathit{On}(B3, T3), \mathit{Clear}(B1), \mathit{Clear}(T1), \mathit{Clear}(T2)$$

  Goals are the motivations behind the agent's intentions.

– **Intentions** represent the actions which the agent has chosen to execute. This sequence of actions is also known as the plan. Plans may include other plans. We may also develop plans on the fly and abandon plans if they turn out to be unsuccessful. Plans in the Block World are essentially a sequence of *get* and *putOn* actions which can be executed to achieve a certain desired stacking configuration of blocks. For example, the sequence of actions

  $$\mathit{get}(B1), \mathit{putOn}(B1, T2), \mathit{get}(B2), \mathit{putOn}(B2, B1), \mathit{get}(B3), \mathit{putOn}(B3, T3),$$
  $$\mathit{get}(B2), \mathit{putOn}(B2, B3), \mathit{get}(B1), \mathit{putOn}(B1, B2)$$

  represents a possibly plan to achieve the above goal.

  We write $\mathit{putOn}(b_1, b_2)$ to indicate that action

  $$!(\forall x, y.\ \mathit{Holds}(x) \otimes \mathit{Clear}(y) \multimap \mathit{Empty} \otimes \mathit{On}(x, y))$$

is applied in a context where x is instantiated by b_1 and y is instantiated by b_2. Similarly, we write $get(b)$ to represent application of an instance of the action

$$!(\forall x, y.\ \mathit{Empty} \otimes \mathit{Clear}(x) \otimes \mathit{On}(x, y) \multimap \mathit{Holds}(x) \otimes \mathit{Clear}(y))$$

where x is instantiated by b. There is no need to explicitly instantiate y because every object x can on at most one object y. That is, y is functionally defined by x.

1.2 Our Approach towards BDI Agent Programming

In our approach, we stratify the different levels of reasoning operations of a BDI agent. At the lowest level, we find operations dealing with beliefs and actions (which influence the state of beliefs). As argued above, we can specify these operations with a linear logic. In Section 2, we introduce the agent domain monad which provides the interface to the agent's actions and belief base. Linear logic (action) rules are implemented by CHRs which are executed within the agent domain monad. Via CHRs we can also program queries, i.e. allowing the agent to extract information from its belief base, and impose agent domain specific laws. The functionality of the agent domain monad is sufficient to write a simple planing algorithm for the Block World. In Section 3, we introduce a goal-directed planning monad built on top of the agent domain monad which provides for a basic control structure to support higher-level reasoning operations involving goals and plans. The monadic encapsulation of each level of reasoning supports for a clean interface and also prevents undesirable interactions among the different levels of reasoning.

2 The Agent Domain Monad

2.1 Belief Base and Agent Actions

Beliefs are represented by the following data structures.

```
-- Belief base
data Term       = Var String | Const String
data Constraint = Predicate String [Term]
-- interface
on :: Term -> Term -> Constraint
on x y = Predicate "On" [x,y]
get :: Term -> Constraint
get x = Action "get" [x]
...
```

Notice that beliefs are first-order. That is, constraints may contain variables.

We represent (linear logic) actions via CHR *simplification* rules whose representation is as follows.

```
-- Actions
data CHRRule    = SimpRule [Constraint] [Constraint]
-- interface
(<==>) :: [Constraint] -> [Constraint] -> CHRRule
```

For example, the linear logic rules from above

$$\textit{(get)} \quad !(\forall x, y.\ \mathit{Empty} \otimes \mathit{Clear}(x) \otimes \mathit{On}(x, y) \multimap \mathit{Holds}(x) \otimes \mathit{Clear}(y))$$
$$\textit{(putOn)}\ !(\forall x, y.\ \mathit{Holds}(x) \otimes \mathit{Clear}(y) \multimap \mathit{Empty} \otimes \mathit{On}(x, y))$$

are represented by the CHRs

```
getRule :: Term -> Term -> CHRRule
getRule x y   = [get x,empty,clear x,on x y] <==> [holds x,clear y]
putOnRule x y :: Term -> Term -> CHRRule
putOnRule x y = [putOn x y,holds x,clear y] <==>
                [empty,clear x,on x y]
```

Notice that we use constraints such as `get x` to trigger actions. We assume that these (action) constraints only appear on the left-hand side of CHRs. CHRs have a simple operational reading in terms of rewritings among multisets of constraints. The first CHR states that if we find matching copies of constraints `get x`, `empty`, `clear x` and `on x y` in the constraint store, we replace (simplify) these constraints by `holds x` and `clear y`. In contrast to Prolog, CHRs are multi-headed and we only instantiate constraints in the head of a CHR constraint but not constraints in the store. Hence, CHRs support forward chaining whereas Prolog supports backward chaining. We take it for granted that CHRs provide for an executable specification of a fragment of linear logic. The interested reader is referred to [3] for more details behind the connection between CHRs and linear logic.

The agent domain monad provides a low-level interface to the agent's beliefs and actions specified via CHRs.

```
-- Agent domain monad (simplified)
type State = ([Constraint],[CHRRule])
type Domain a = Domain (State-> (State,a))
instance Monad Domain where ...
initStore :: [Constraint] -> Domain ()
initRules :: [CHRRule] -> Domain ()
newVar :: Domain Term
const :: String -> Domain Term
applyAction :: Constraint -> Domain Bool
-- Monadic action interface
getM :: Term -> Domain Bool
getM x = applyAction (get x)
putOnM :: Term -> Term -> Domain Bool
putOnM x y = applyAction (putOn x y)
```

The above represents a state monad where each CHR application results into a new state (i.e. constraint store). For presentation purposes, we have simplified the `Domain` monad and also leave out the (obvious) body of the instance declaration. In our actual implementation, we also record exceptional behavior such as an inconsistent constraint store etc. We refer to [2] for the details. Functions `initStore` and `initRules` initialize the belief base and the action rules. Function `newVar` deals with variable names generation and `const` create a constant in the agent domain monad. We will see applications of these functions shortly.

The underlying CHR solver is invoked via `applyAction`. We assume that the input argument is a (action) constraint which is added to the constraint store. Then, we exhaustively apply CHRs to the current constraint store. Recall that we use (action) constraints to trigger (action) rules. We return `True` if the (action) constraint has been removed, i.e. the action has fired. Otherwise, we return `False`.

Depending on the application, we may be interested to verify certain properties. A typical property is action determinism. A deterministic action is one which produces a unique effect (output) from each current state (input). The advantage of our approach is that we can check for action determinism by verifying termination and confluence of CHRs. Termination guarantees that there is no infinite sequence of CHR applications starting from some initial constraint store. Confluence guarantees that different derivations starting from the same point can always be brought together again. Clearly, both conditions imply action determinism. However, termination and confluence are often too strong in general. For example, confluence may only hold for all "reachable" states by the agent. The important insight is that to guarantee action determinism it is sufficient to verify the weaker condition of "reachable" confluence. We refer the interested reader to[12] where we show how to refine the standard test of CHR confluence [1] to verify "reachable" confluence for CHRs describing agent actions. Specifically, we verify action determinism for the Block World.

We conclude this (sub)section by giving a simple program making use of the functionality provided by the agent domain monad.

```
blockWorldInstance :: Domain Bool
blockWorldInstance = do
   -- Initialize belief base
   b1 <- const "B1"; b2 <- const "B2"; b3 <- const "B3";
   t1 <- const "T1"; t2 <- const "T2"; t3 <- const "T3";
   initStore [empty,on b1 b2,on b2 b3,on b3 t1,clear b1,
              clear t2,clear t3]
   -- Initialize rules
   x  <- newVar
   y <- newVar
   initRule [getRule x y,putOnRule x y]
   -- Execute block world actions get(B1) followed by putOn(B1,T3)
   getM b1
   putOnM b1 t3
```

Execution of the above program proceeds roughly as follows. The first action `getM b1` adds `get b1` to the initial belief base which leads to

```
[get b1,empty,on b1 b2,on b2 b3,on b3 t1,clear b1,clear t2,clear t3]
```

CHR `[get x,empty,clear x,on x y] <==> [holds x,clear y]` applies which yields

```
[holds b1,clear b2,on b2 b3,on b3 t1,clear b1,clear t2,clear t3]
```

No further CHRs are applicable. Notice that we have removed the action constraint `get b1`. That is, the action has fired. Hence, we return `True`. Next, we apply the action `putOnM b1 t3` which results in

```
[putOn b1 t3,holds b1,clear b2,on b2 b3,on b3 t1,clear b1,clear t2,clear t3]
```

This time the CHR `[putOn x y,holds x,clear y] <==> [empty,clear x,on x y]` is applicable and we obtain

```
[empty,on b1 t3,on b2 b3,on b3 t1,clear b1,clear b2, clear t2]
```

As before, the action rule has successfully fired. Hence, we return `True`.

2.2 Queries and Domain Laws

We yet need to provide some basic query functions such that the agent is able to observe its environment.

```
test     :: [Constraint] -> Domain Bool
queryAny :: [Term] -> [Constraint] -> Domain (Maybe [Term])
queryAll :: [Term] -> [Constraint] -> Domain [[Term]]
```

Function call `test cs` returns `True` if the constraints `cs` are (syntactically) contained in the current constraint store. Otherwise, we return `False`. Function call `queryAny vs cs` returns any mappings to variables `vs` such that constraints `cs` are contained in the constraint store. Via `queryAll` we can find all mappings. For example, `queryAny [y] [on y b0]` finds any block `y` which is on block `b0` whereas `queryAll [y] [clear y]` finds all objects `y` which are clear. In case of `queryAny` we use the `Maybe` monad to signal failure whereas for `queryAll` we simply return the empty list.

The implementation of queries is rather straightforward by using CHR *propagation* rules which are represented as follows.

```
(==>) :: [Constraint] -> [Constraint] -> CHRRule
data CHRRule = SimpRule [Constraint] [Constraint]
             | PropRule [Constraint] [Constraint]
```

In contrast to simplification rules which perform non-monotonic updates (by replacing instances of the right-hand constraints with the left-hand side constraints), a propagation rules only propagates (i.e. adds) the right-hand side

constraints. Thus, we can implement `queryAll [y] [clear y]` by applying the CHR propagation rule `[clear y ==> answer y]`. For each instance of `clear y` we add `answer y` to the constraint store. What remains is to collect (i.e. remove) all answer constraints to build the requested mapping. We remark that CHR implementations avoid infinite application of CHR propagation rules by assuming that these rules are only applied once on the same set of constraints [1].

CHR propagation rules are also useful to encode domain-specific laws. For example, `[on x y, clear y ==> false]` enforces the law that any belief base with a clear object which has another object on top of it is inconsistent. We assume that `false` represents the always unsatisfiable constraint.

2.3 A Simple Block World Agent

We have now everything in place to implement a simple Block World agent to clear the top of a block `b0`.

```
planClearTopOf :: Term -> Domain ()
planClearTopOf b0 = do
   isClear <- test [clear b0] -- check if block b0 is clear
   if isClear then
      return ()               -- b0 is already clear, so do nothing
   else do
      y <- newVar
      z <- newVar
      (Just [b1]) <- queryAny [y] [on y b0] -- find b1, which is on b0
      planClearTopOf b1                     -- clear top of b1
      (Just [b2,b3]) <- queryAny [y,z] [clear y,clear z]
                             -- find 2 possible relocations b2 and b3
      getM b1
      case (b2 == b0) of
         True  -> do -- if b2 is b0, do not put b1 back on b0
                    putOnM b1 b3
                    return ()
         False -> do
                    putOnM b1 b2
                    return ()
```

The current functionality is sufficient to implement agents in terms of the low-level operations such as actions and queries. Higher-level operations involving the agent's goals, intentions and plans must still be programmed in Haskell. The advantage of our approach is that we can build more sophisticated control structures on top of the agent domain monad to reason about goals and plans. A typical example would be a backtracking monad [10] which allows us to search for a plan to achieve a certain goal. We can also easily build more sophisticated queries in terms of basic queries. In the next section, we will explore these possibilities.

3 The Goal-Directed Planning Monad

On top of the agent domain monad, we introduce a state monad to keep track of the goals we are trying to achieve and a simple back-tracking monad to support reasoning by search. The monad definitions are straightforward and can be found elsewhere [2].

```
type Goals  = [Constraint]
type Plan a = Plan (Goals -> Domain (PlanStatus (Goals,a)))
data PlanStatus a = PlanSucc a | PlanFail
instance Monad Plan where ...
```

The existing functions operating on the agent domain monad `Domain` can be straightforwardly lifted to the goal-directed planning monad `Plan` [13]. We can also reuse all function names by applying type class overloading [11, 23]. For brevity, we omit the straightforward details. In the following, we introduce two high-level reasoning operations, `elseTry` and `subGoal`, to support back-tracking and goal-directed planning. We explore their usefulness via a series of small examples.

3.1 Planning by Search via Back-Tracking

Previously (in the agent domain monad), each action returned a Boolean value to indicate whether the action has fired or not. We explicitly had to react to the outcome of an invocation of an action. In the `Plan` monad we will abort the plan if execution of an action is blocked. That is, we will raise an execption if the action did not fire. Hence, we redefine the action invocation function `applyAction` as follows.

```
applyAction :: Constraint -> Plan ()
applyAction act = do
   -- Execute Domain Monad's 'applyAction'
   isSucc <- doDomain (applyAction act)
   case isSucc of
     True  -> return ()
     False -> abortPlan
```

We assume that the `doDomain` lifts a `Domain` computation into the `Plan` monad and `abortPlan` simply signals failure (by returning `PlanFail` inside the monad).

Abortion of plans is only useful if we can program composite plans, where the failure of a primary plan would result in the activation of a contingency plan. For this, we introduce the `elseTry` operation, which provides exactly this functionality.

```
elseTry :: Plan a -> Plan a -> Plan a
elseTry (Plan p1) (Plan p2) =
      Plan (\s -> do st <- getStore
                     stat <- p1 s
                     case stat of
                       PlanFail        -> do setStore st
                                             p2 s
                       PlanSucc (s',a) -> return (PlanSucc (s',a))
```

Functions `getStore` and `setStore` serve the obvious purpose of getting and (re)setting the current store. They are only available within our library to prevent "unsafe" use of them. The definition of `elseTry` should contain no surprises. In the event that plan `p1` is unsuccessful, the constraint store is restored to its original before we attempt to execute plan `p2`. Below is a Block World planning algorithm which makes use of the `elseTry` operation.

```
planGetTopOf :: Term -> Plan ()
planGetTopOf x = (getM x) `elseTry` (getTop x)
                 where
                   getTop x = do
                     y  <- newVar
                     mb <- queryAny [y] [on y x]
                     case mb of
                       Just [y] -> planGetTopOf y
                       Nothing  -> abortPlan
```

We model a 'brute' force get plan operation that tries to get `x`. If `getM x` fails (i.e. aborts), we attempt to get the top most block on top of `x`. If this operation fails as well, we abort the plan (which essentially means that the robot is already holding `x`).

3.2 Goal-Directed Planning with Back-Tracking

So far, we have not used the "goal" state. Goals are lists of constraints which the agent tries to achieve by executing plans, i.e. sequences of actions. After we have achieved a goal, the goal should be removed. Hence, we need to further adjust the definition of the `applyAction` function and update the goals each time an action has fired.

```
applyAction :: Constraint -> Plan ()
applyAction act = do
   -- Execute Domain Monad's 'applyAction'
   isSucc <- doDomain (applyAction act)
   case isSucc of
     True  -> updateGoals
     False -> abortPlan
```

After firing an action, the `updateGoals` function takes the current state of the constraint store and goals and removes from the goals all constraints which are in the store. The following transition relation describes the `applyAction` operation in more detail.

$$\text{(Goal-Directed Action)}\ \frac{\{a\} \cup C \rightarrowtail_P^* C' \quad G' = G - C'}{(G \mid C) \overset{a}{\rightarrowtail} (G' \mid C')}$$

G refers to the input goal state and $\{a\} \cup C \rightarrowtail_P^* C'$ denotes that firing of CHRs on the input constraint store C where we have inserted the action constraint a leads to the output constraint store C'. We assume here that the action has successfully fired. The output goals are set to be the (set) difference between the input goals and the output constraint store.

Using goals to direct plans becomes interesting once we provide the ability to assert the success of goals. For this, we introduce the `subPlan` operation which takes a goal `g` and two plans `p1` and `p2`. If `p1` achieves goal `g`, we continue with `p2`. Otherwise, we abort.

```
subPlan :: Goals -> Plan a -> Plan b -> Plan b
subPlan gs p1 p2 = do
   currGoals <- getGoals
   setGoals gs
   a  <- p1
   gs <- getGoals
   case (gs == []) of
     True  -> do setGoals (currGoals `diff` gs)
                 p2
     False -> abortPlan
```

Functions `getGoals` and `setGoals` get and (re)set the current goals. Function `diff` takes the difference between two lists (treating lists as multi-sets). The important point to remember is that we only proceed with plan `p2`, if after executing of plan `p1` the goals `gs` have been achieved (i.e. they are not present in the final constraint store after `p1`'s execution).

To illustrate the use of `subPlan`, we implement a plan to move a certain block `x` onto any arbitrary object `y`. First, we define two auxiliary functions `queryAnyNotSelected` and `planDropOn`,

```
-- Specialized query
queryAnyNotSelected :: [Term] -> [Constraint] -> [Term] -> Plan (Maybe [Term])
queryAnyNotSelected ts cs sel = do
   vss  <- queryAll ts cs
   let vss' = (filter (\s2 -> sel `intersect` s2 == []) vss)
   case vss' of
     []    -> return Nothing
     (v:_) -> return (Just v)
```

```
-- A plan to drop the current object held onto y
planDropOn :: Term -> Plan ()
planDropOn y = do x <- newVar
                  mb <- queryAny [x] [holds x]
                  case mb of
                    Nothing -> abortPlan
                    Just x  -> putOnM x y
```

Function `queryAnyNotSelected` accepts an additional argument `sel :: [Term]` and only returns a match of variable values `vs` such that `sel `intersect` vs` is the empty list. The plan `planDropOn` drops the current object held by the robot onto some object `y`.

We are in the position to define the plan to move a certain block `x` onto any arbitrary object `y`.

```
-- A plan to move x away from current location.
planMove :: Term -> Term -> Plan ()
planMove x y = do
   (subPlan [holds x] (planGetTopOf x) (putOnM x y))
   `elseTry` (contPlanMove x y))
   where
      contPlanMove x y = do
           z        <- newVar
           (Just z) <- queryAllNotSelected [z] [clear z] [x,y]
           planDropOn z
           planMove x y
```

The `planMove` operation uses the `planGetTopOf` operation described in the previous section to obtain `x` (if possible). Recall that the `planGetTopOf` operation may result into the robot holding `x`, or the robot holding some `y` which is top most of object on `x`. Only when the robot is holding `x`, indicated by the goal `holds x`, we can proceed to put `x` on `y`. If the robot is not holding `x`, it must get rid of whichever object it is holding by dropping it on a object that is not `x` or `y`, and attempt the plan of moving `x` again.

We would like to point that our goals are "dynamic". There is no guarantee that a goal once satisfied will remain to be satisfied For example, consider a subgoal `holds x` where in some later stages of the plan, the robot will hold a different object. This has to do with the fact that our agent domain is "linear' Depending on the application, we may want to add "static" goals but we leave out the details for future work.

4 Related Work and Conclusion

There are numerous agent-oriented programming languages which are based on the BDI agent model. We refer to [15] for an overview. We believe that our work

has a number of novel aspects. We employ a linear logic constraint solver specifiable via CHRs. Thus, we can concisely model the agent's low-level reasoning which involve the agent's beliefs and actions as well as queries and domain specific laws. On top of these basic agent domain operations we can provide for rich control structures implemented in Haskell to support high-level reasoning operations involving the agent's goals and plans. By using monads we can prevent undesirable interactions among the different levels of reasoning and provide for the modular development of more complex reasoning operations.

Our work can also be viewed as an attempt to achieve multi-paradigm programming in Haskell using monads as the key concept. There are similarities to [14] which aims at integrating linear logic forward chaining with Prolog style backward chaining. We yet have to work out the exact details

Currently, we only support single agents. However, we believe that we can support multi agents by simulating concurrency [5]. In the future, we plan to support true concurrency by Haskell with concurrency extensions [19, 9].

The latest release of our system, including further examples, can be downloaded via

`http://www.comp.nus.edu.sg/~lamsoonl/adom`

References

1. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proc. of CP'97*, LNCS, pages 252–266. Springer-Verlag, 1997.
2. Specifying and controlling agents in haskell. http://www.comp.nus.edu.sg/~ lamsoonl/adom.
3. H. Betz and T. Frühwirth. A Linear-Logic Semantics for Constraint Handling Rules. In *Proc of CP'05*, volume 3709 of *LNCS*, pages 137–151. Springer-Verlag, 2005.
4. M. E. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, 1987.
5. K. Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3):313–323, 1999.
6. G.J. Duck. Haskell chr. http://www.cs.mu.oz.au/ gjd/haskellchr/.
7. T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, LNCS. Springer-Verlag, 1995.
8. J. Y. Girard. Linear Logic: Its Syntax and Semantics. In *Proc. of Workshop on Linear Logic, Cornell University*, number 222. Cambridge University Press, 1995.
9. T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proc. of PPoPP'05*, pages 48–60. ACM Press, 2005.
10. R. Hinze. Deriving backtracking monad transformers. In *Proc. of ICFP'00*, pages 186–197. ACM Press, 2000.
11. S. Kaes. Parametric overloading in polymorphic programming languages. In *In Proc. of ESOP'88*, volume 300 of *LNCS*, pages 131–141. Springer-Verlag, 1988.
12. E.S.L. Lam and M. Sulzmann. Towards agent programming in CHR. Technical Report CW 452, Katholieke Universteit Leuven, 2006. Informal Proc. of CHR 2006, Third Workshop on Constraint Handling Rules.

13. S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proc. of POPL'95*, pages 333–343. ACM Press, 1995.
14. P. López, F. Pfenning, J. Polakow, and K. Watkins. Monadic Concurrent Linear Logic Programming. In *Proc. of PPDP'05*, pages 35–46, 2005.
15. V. Mascardi, D. Demergasso, and D. Ancona. Languages for programming BDI-style agents: an overview. In *Proc. of WOA 2005*, pages 9–15. Pitagora Editrice Bologna, 2005.
16. E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.
17. S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
18. S. Peyton Jones, D.Vytiniotis, G. Washburn, and S. Weirich. Simple unification-based type inference for GADTs. http://research.microsoft.com/ simonpj/papers/gadt/, November 2005.
19. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. of POPL'96*, pages 295–308. ACM Press, 1996.
20. A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proc. of the First Intl. Conference on Multiagent Systems*, 1995.
21. M. Sulzmann, G. J. Duck, S. Peyton Jones, and P. J. Stuckey. Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming*, 2006. To appear.
22. P. Wadler. Comprehending monads. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78. ACM Press, 1990.
23. P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proc. of POPL'89*, pages 60–76. ACM Press, 1989.