A Formal Architecture for the 3APL Agent Programming Language

Mark d'Inverno*, Koen Hindriks†, and Michael Luck‡

* Cavendish School of Computer Science, 115 New Cavendish Street, University of Westminster, London W1M 8JS, UK
dinverm@westminster.ac.uk

† Dept. of Computer Science, Universiteit Utrecht, P.O. Box 80.089; 3508 TB Utrecht, The Netherlands
koenh@cs.uu.nl

‡ Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK
mikeluck@dcs.warwick.ac.uk

Abstract. The notion of *agents* has provided a way of imbuing traditional computing systems with an extra degree of flexibility that allows them to be more resilient and robust in the face of more varied and unpredictable forms of interaction. One class of agents, typically called *intelligent agents*, represent their world symbolically according to their beliefs, have goals which need to be achieved, and adopt plans or intentions to achieve them. Now, one approach to building agents is to design a programming language whose semantics are based on some theory of *rational* or *intentional* agency and to program the desired behaviour of individual agents directly using mental attitudes. Such a technique is referred to as *agent oriented programming*. Arguably, the most innovative of these languages is 3APL (pronounced "triple-a-p-l") which supports the construction of intelligent agents for the development of complex systems through a set of intuitive concepts like beliefs, goals and plans. In this paper, we provide a Z specification of the programming language 3APL which provides a basis for implementation and also adds to a growing library of agent techniques and features.

1 Introduction

Recently, there has been an explosion of interest in agent-based systems and the related subfield of distributed artificial intelligence (DAI). The focus of much agent-based work is on building *architectures* for intelligent agents, providing information about essential data structures, relationships between these data structures, the processes or functions that operate on them and the operation or execution cycle of an agent.

Deliberative Agent Systems symbolically model their environment and manipulate these symbols in order to act. In order to model *rational* or *intentional* agency, an abstraction level is chosen for the symbols such that they represent *mental attitudes*. Most agent systems include a deliberative architecture to support deliberative reasoning at the mental-attitude level.

Mental attitudes used to describe and characterise the behaviour of agents include beliefs, goals, assumptions, desires, knowledge, plans, motivations and intentions, and are commonly grouped into three categories, informative, motivational and deliberative [9]. The first refers to that which a system considers to be true about the world and includes knowledge, beliefs and assumptions, the second to the ‘wants’ of a system including goals, desires and motivations, and the third concerns how an agent’s behaviour is directed and includes plans and intentions. The distinction between the second and third categories is subtle since it is possible that a system may *desire* a certain state without planning for it, or *intending* it to happen.

There are several compelling reasons why agents defined using mental attitudes might be useful. First, if an agent can be described in terms of what it knows, what it wants and what it intends then, since it is modelled on familiar concepts, it becomes possible for users to understand and predict its behaviour. Second, understanding the relationship between these different attitudes and how they affect behaviour could provide the control mechanism for ‘intelligent action’ in general. Third, computational agents designed in this way may be able to interpret the behaviour of others independently of any implementation.

Rather than defining an architecture, *agent oriented programming* is a paradigm for directly programming the behaviour of agents using computational languages whose semantics capture some theory of rational agency [14]. Typically agents have an initial set of beliefs, goals and plans and an interpreter that details how agents should achieve their goals given an environmental context.

1.1 The 3APL Programming Language

One such agent programming language is 3APL which supports the design and construction of intelligent agents for the development of complex systems through a set of intuitive concepts like beliefs, goals and plans. In turn, these can be used to describe and understand the computational system in a natural way. Indeed, applications such as personal assistants [12] are naturally seen as agents that act on behalf of their users and in pursuit of user goals, using these concepts.

3APL supports this style of programming by means of an expressive set of primitives to program agents, which consist of sets of beliefs, goals and practical reasoning rules. Beliefs represent the issues the agent must deal with, while goals allow the agent both to focus on what it must achieve and to represent the way in which it can achieve it. In 3APL, goals are thus used to represent achievement goals and as *plans*. The practical reasoning rules provide the agent with planning capabilities to find an appropriate plan to achieve a goal, capabilities to create new goals to deal with a particular situation, and capabilities to use the rules to revise a plan. The architecture for 3APL [8] is based on the *think-act cycle*, which is divided into two parts. The first part corresponds to a phase of practical reasoning by using practical reasoning rules, and the second corresponds to an execution phase in which the agent performs some action.

Originally, the operational semantics of 3APL was specified by means of Plotkin-style transition semantics [7]. In this work, we provide a re-specification of 3APL in Z, which has a number of benefits. First, it helps to get closer to a good implementation of 3APL, because of the tools available for Z which support type-checking, animation, and so on. By specifying 3APL, we can provide a computational model that includes data structures, operation and architecture, thereby isolating the data-types for an efficient implementation of 3APL. Second, the process of re-specification provides a different perspective, highlighting different aspects of the language and architecture that are not manifested in a similar way in the transition style semantics. In carrying out this work, we aim to provide a clearer analysis and insight into agent languages and architectures, and add to a growing library (written in Z) of desirable and reusable agent features.

The next section introduces the basic types used to build the 3APL model, comprising beliefs, actions, goals and practical reasoning rules. Then we define 3APL agents and finally we describe their operation.

2 3APL types

Beliefs and goals are the basic types of expressions in 3APL from which rule expressions are derived. In this specification, beliefs are a subset of first order formulae (though in principle any knowledge representation language could be used), and this first order language is defined in the usual way. First order terms are defined by means of given sets of constants, first order variables, and function symbols. Since the programming language distinguishes between first order variables and variables that range over goals, we first define a partition of the set of variables, and use $FOVar$ to denote the set of first order variables and $Gvar$ to denote the set of goal variables.

$$[Const, Var, FuncSym]$$

$$\begin{array}{|l}
FOVar : \mathbb{P}\ Var \\
GVar : \mathbb{P}\ Var \\
\hline
FOVar \cap GVar = \varnothing \\
FOVar \cup GVar = Var
\end{array}$$

The sets of all constants and function symbols are respectively denoted as $[Const]$ and $[FuncSym]$. In terms of this specification, the contents of these sets are unimportant, and we use them directly without further elaboration. A first order *term* is either a constant, a first order variable, or a function symbol with a non-empty sequence of terms as a parameter. The auxiliary function $fovars$ returns the set of all first order variables in a (first order) term.

$$\begin{array}{lll}
FOTerm & ::= & const\langle\langle Const\rangle\rangle \\
 & | & var\langle\langle FOVar\rangle\rangle \\
 & | & functor\langle\langle FuncSym \times \mathrm{seq}_1\ FOTerm\rangle\rangle
\end{array}$$

$$\begin{array}{|l}
fovars : FOTerm \rightarrow (\mathbb{P}\ FOVar) \\
\hline
\forall c : Const;\ v : FOVar;\ f : FuncSym;\ ts : \mathrm{seq}\ FOTerm \bullet \\
\quad fovars\ (const\ c) = \varnothing \wedge \\
\quad fovars\ (var\ v) = \{v\} \wedge \\
\quad fovars\ (functor(f, ts)) = \bigcup\{t : FOTerm \mid t \in (\mathrm{ran}\ ts) \bullet fovars\ t\}
\end{array}$$

2.1 Beliefs

Beliefs are defined by building types from the above primitives. The set of all predicate symbols is denoted by $[PredSym]$, and a belief *atom* is a predicate symbol with a (possibly empty) sequence of terms as its argument. Beliefs are then either an atom, the negation of an atom, the conjunction of two beliefs, or the implication of one belief by some other belief.

$$[PredSym]$$

$$\begin{array}{|l}
\hline \textit{Atom} \\
head : PredSym \\
terms : \mathrm{seq}\, FOTerm \\
\hline
\end{array}$$

$$\begin{aligned} Belief ::= {} & pos\langle\!\langle Atom\rangle\!\rangle \mid not\langle\!\langle Atom\rangle\!\rangle \mid and\langle\!\langle Belief \times Belief\rangle\!\rangle \\ \mid {} & imply\langle\!\langle Belief \times Belief\rangle\!\rangle \mid \mathbf{false} \mid \mathbf{true} \end{aligned}$$

The running example in this paper concerns a personal assistant for scheduling meetings and other activities, which acts on behalf of its user, monitors the need to schedule activities, and helps the user to find appropriate slots. The agent might also provide its user with information concerning the appropriate means of transportation to go to the location of a scheduled meeting.

Part of the knowledge representation language of a personal assistant for scheduling (PAS) consists of predicates to represent, for example, the agenda of the user and the means of transportation to get from A to B. In particular, the predicate $agenda$ represents the user's agenda, with five arguments:

$agenda(Activity, Time, Duration, People, Loc)$

Each variable, respectively, represents the activity scheduled (such as meetings, lunch, etc), the date and time, the duration of the activity, the set of people involved, and the location. For example, the PAS may have the following belief concerning the user's agenda, indicating that an hour meeting is scheduled for May 5th at 12:00 with John and Peter in Utrecht: $pos\ \mathtt{agenda}(\mathtt{meeting}, \mathtt{may5th12:00}, \mathtt{60min}, \{\mathtt{john}, \mathtt{peter}\}, \mathtt{utrecht})$

Other predicates include $free(Time, Length)$, indicating that a free slot of length $Length$ at time $Time$, and $transport(Means, FromLoc, ToLoc, Time, DurTrans)$, indicating that a possible means of moving from $FromLoc$ to $ToLoc$ at (date and) time $Time$ is $Means$, and taking $DurTrans$ time. Finally, $location(Loc, Time)$ keeps track of the location of the user at a particular time, which is assumed to be persistent. That is, without information to the contrary, the location of the user at a time T' after time T will be assumed to be the same as that at time T.

2.2 Actions

In order to achieve goals or accomplish tasks, an agent must perform actions, represented by action symbols specified in the same way as atoms.

$[ActionSym]$

$$\begin{array}{|l}
\textit{Action} \\
\hline
name : ActionSym \\
terms : \mathrm{seq}\, FOTerm \\
\hline
\end{array}$$

A basic action used by the PAS agent is the action ins_agenda for inserting items in the belief base. The action has five associated arguments matching the arguments of the predicate $agenda$, and is used to insert a particular item in the agenda. Specifically, $ins\_agenda(Activity, Time, Length, People, Loc)$ inserts the corresponding agenda predicate into the belief base of the PAS. For example,

$\mathtt{ins\_agenda}(\mathtt{meeting}, \mathtt{may5th12}:\mathtt{00}, \mathtt{60min}, [\mathtt{john}, \mathtt{peter}], \mathtt{utrecht})$
inserts an hour-long meeting scheduled for May 5th with John and Peter in Utrecht in the agenda of the agent.

At this point, a number of auxiliary functions may also be defined to return the set of variables in an atom, a belief, or an action, and which are used in the specification of the operation of an agent.

$$\begin{array}{|l}
atomvars : Atom \rightarrow (\mathbb{P}\, FOVar) \\
beliefvars : Belief \rightarrow (\mathbb{P}\, FOVar) \\
actionvars : Action \rightarrow (\mathbb{P}\, FOVar) \\
\hline
\forall\, c : Const;\ v : FOVar;\ f : FuncSym;\ ts : \mathrm{seq}\, FOTerm; \\
\qquad at : Atom;\ b_1, b_2 : Belief;\ a : Action \bullet \\
\quad fovars\ (const\ c) = \varnothing \land \\
\quad fovars\ (var\ v) = \{v\} \land \\
\quad fovars\ (functor(f, ts)) = \bigcup\{t : FOTerm \mid t \in (\mathrm{ran}\, ts) \bullet fovars\ t\} \land \\
\quad atomvars\ at = \bigcup\{t : FOTerm \mid t \in (\mathrm{ran}\, at.terms) \bullet fovars\ t\} \land \\
\quad beliefvars\ (pos\ at) = atomvars\ at \land \\
\quad beliefvars\ (not\ at) = atomvars\ at \land \\
\quad beliefvars\ (and\ (b_1, b_2)) = beliefvars\ b_1 \cup beliefvars\ b_2 \land \\
\quad actionvars\ a = \bigcup\{t : FOTerm \mid t \in (\mathrm{ran}\, a.terms) \bullet fovars\ t\}
\end{array}$$

2.3 Goals, Contexts and Front Contexts

In 3APL, goals are used to represent *both* the goals *and* the plans to achieve these goals of the agent. Goals are program-like structures that are built from basic constructs, such as actions, and regular imperative programming constructs, such as sequential composition and nondeterministic choice. 3APL goals can be characterised as *goals-to-do*, which are mental attitudes corresponding to plans of action to achieve a state of affairs, or *goals-to-be*, which are mental attitude corresponding to the state of affairs desired by an agent. For example, an agent may have adopted the *goal-to-do* of learning to play the piano, and then performing at the ZB conference dinner. This might be done in pursuit of the agent's *goal-to-be* of wanting to be a pop-star rather than an academic.

Contexts Before formally describing goals, we introduce the notion of *contexts*, which are goals with an extra feature called 'holes' that act as placeholders within the structure of goals[1]. Contexts are well-known structures in programming language semantics, closely related to the concept of goals, and are used to describe the operation and architecture of 3APL agents. Use of contexts in this way differs substantially from the transition style semantics presented in [7], which is inappropriate in this Z specification, since it would require a recursive relationship between schemas. While contexts, by contrast, allow us to give an elegant specification of the operation of a 3APL agent, we must stress that their role is in the *presentation* of an architecture for 3APL, rather than in the 3APL language itself.

More precisely, a *context* is either a basic action, a query goal, an achieve goal, the sequential composition of two contexts, the nondeterministic choice of two contexts, a goal variable or "$\square$" which represents a place within a context that might contain another context. In the definition below, we use the set of goal variables, $GVar$, to allow a process called *goal revision* to take place as will be described later.

$$\begin{aligned} Context ::= {} & bac\langle\!\langle Action\rangle\!\rangle \mid query\langle\!\langle Belief\rangle\!\rangle \mid achieve\langle\!\langle Atom\rangle\!\rangle \mid \\ & seqcomp\langle\!\langle Context \times Context\rangle\!\rangle \mid choice\langle\!\langle Context \times Context\rangle\!\rangle \mid \\ & goalvar\langle\!\langle GVar\rangle\!\rangle \mid \square \end{aligned}$$

The $\square$, which denotes the placeholder or *hole* within a context, is distinct from a goal variable. Although both $\square$ and a goal variable are placeholders, $\square$ is a facility used for *specifying* 3APL, whereas goal variables are part of 3APL itself. Five examples of contexts are shown in Figure 1.

Goals We can now define a goal as a context without any occurrences of $\square$. Only the third context in Figure 1 is a goal since it contains no squares. None of the other contexts are goals because they contain at least one occurrence of $\square$. The auxiliary function $squarecount$ counts the occurrences of $\square$ in a context.

$$\begin{array}{|l} squarecount : Context \rightarrow \mathbb{N} \\ \hline \forall\, a : Action;\ b : Belief;\ at : Atom;\ c_1, c_2 : Context;\ gv : GVar \bullet \\ \quad squarecount(bac\ a) = 0 \land squarecount(query\ b) = 0 \land \\ \quad squarecount\ (achieve\ at) = 0 \land \\ \quad squarecount\ (seqcomp(c_1, c_2)) = squarecount\ c_1 + squarecount\ c_2 \land \\ \quad squarecount\ (choice(c_1, c_2)) = squarecount\ c_1 + squarecount\ c_2 \land \\ \quad squarecount\ (goalvar\ gv) = 0 \land squarecount\ \square = 1 \end{array}$$

$$Goal == \{g : Context \mid squarecount\ g = 0\}$$

Front Contexts An important type of context, known as a *front context*, is used to illustrate significant properties of 3APL. Front contexts are contexts with precisely *one*

[1] Note that our use of the term *context* is distinct from the notion of a context in such systems as AgentSpeak(L) [13, 3], which is defined as the pre-condition of a plan.

$1. choice\ (seqcomp\ (\square, bac\ \mathtt{ins\_agenda(meeting, Time, Dur, People, Loc)})), \square)$
$2. seqcomp\ (seqcomp\ (query\ pos\ \mathtt{free(Time, Length)}, \square), goalvar\ \mathtt{X})$
$3. goalvar\ \mathtt{X}$
$4. seqcomp\ (\square, bac\ \mathtt{ins(agenda(meeting, Time, Dur, People, Loc)})$
$5. choice\ (seqcomp\ (\square, bac\ \mathtt{ins(agenda(meeting, Time, Dur, People, Loc)}),$
$\quad query\ (pos\ \mathtt{free(Time, Length)}))$

Fig. 1. Examples of Contexts

occurrence of $\square$ *at the front* of the context. Informally, an element *at the front* of a context means that an agent could choose to perform this element first, so that if a $\square$ at the front of a context was replaced by a goal, then that goal *could be performed first*, before the remainder of the overall goal.

A *front* context is defined formally as either a single square, the sequential composition of a front context with a goal, or the choice (in either order) of a front context and a goal. In Figure 1, neither Context 1, 2, nor 3 are front contexts, but both 4 and 5 are.

$$\begin{array}{|l}
frontcontext\_ : \mathbb{P}(Context) \\
\hline
\forall fc, fc_1 : Context;\ g : Goal \bullet frontcontext\ (fc) \Leftrightarrow \\
\qquad fc = \square \vee \\
\qquad (fc = seqcomp(fc_1, g) \wedge frontcontext\ fc_1) \vee \\
\qquad (fc = choice(fc_1, g) \wedge frontcontext\ fc_1) \vee \\
\qquad (fc = choice(g, fc_1) \wedge frontcontext\ fc_1)
\end{array}$$

The type $FrontContext$ is the set of contexts satisfying $frontcontext$.

$$FrontContext == \{fc : Context \mid frontcontext\ fc\}$$

Clearly any front context has only one occurrence of a 'hole'.

$$\forall fc : FrontContext \bullet squarecount\ fc = 1$$

A goal may contain both goal variables as well as first order variables. We therefore define three functions that return the set of *all* variables, *goal* variables and *first order* variables, respectively, of a goal. A definition of optional and related elements can be found in [3].

$$
\begin{array}{|l}
goalvars : optional[Goal] \rightarrow (\mathbb{P}\ Var) \\
goalgvars : optional[Goal] \rightarrow (\mathbb{P}\ GVar) \\
goalfovars : optional[Goal] \rightarrow (\mathbb{P}\ FOVar) \\
\hline
\forall\, g : optional[Goal];\ a : Action;\ b : Belief \\
\quad at : Atom;\ g_1, g_2 : Goal;\ gv : GVar \bullet \\
\qquad goalvars \varnothing = \varnothing\ \wedge \\
\qquad goalvars \{bac\ a\} = actionvars\ a\ \wedge \\
\qquad goalvars \{query\ b\} = beliefvars\ b\ \wedge \\
\qquad goalvars\ \{achieve\ at\} = atomvars\ at\ \wedge \\
\qquad goalvars\ \{seqcomp(g_1, g_2)\} = goalvars\ \{g_1\} \cup goalvars\ \{g_2\}\ \wedge \\
\qquad goalvars\ \{choice(g_1, g_2)\} = goalvars\ \{g_1\} \cup goalvars\ \{g_2\}\ \wedge \\
\qquad goalvars\ \{goalvar\ gv\} = \{gv\}\ \wedge \\
\qquad goalgvars\ g = (goalvars\ g) \cap GVar\ \wedge \\
\qquad goalfovars\ g = (goalvars\ g) \cap FOVar
\end{array}
$$

2.4 Practical Reasoning Rules

A 3APL agent uses *practical reasoning rules* not only to plan in the more conventional sense, but also to *reflect* on its goals. Whilst the use of rules for planning is a familiar concept from the literature, using rules for reflection on goals or plans is less well known. Reflection allows an agent to re-consider one of its plans in a situation in which the plan will fail with respect to the goal it is trying to achieve or has already failed, or where a more optimal strategy can be pursued.

Practical reasoning rules are divided into four classes: reactive rules, which are used not only to respond to the current situation but also to create new goals; plan-rules, which are used to find plans for achievement goals; failure-rules, which are used to replan when plans fail; and optimisation-rules, which can replace less effective plans with more optimal plans. (This classification of rules was first proposed in [8].)

We introduce a type to correspond to each of these categories.

$$PRType ::= reactive \mid failure \mid plan \mid optimisation$$

A practical reasoning rule consists of an (optional) head, which is a goal, an (optional) body which is a goal, a guard which is a belief and a type to define its purpose. Informally, a practical reasoning rule with head g, body p and guard b, states that if the agent tries to achieve goal g and finds itself in a situation b, then it might consider *replacing* g by a plan p as a means to achieve it. If it is a *plan-rule*, the goal g is of the form $achieve\ s$ where s is a simple formula, and the rule states that to achieve goal g in situation b, consider plan p. If it is a *failure-rule*, then g may be any goal and the rule states that if g fails in situation b, consider dropping g and instead, adopting strategy p to deal with the failure. Finally, if it is an *optimisation-rule*, then g may be any goal and the rule states that if g is not so efficient in situation b, consider dropping g and instead, adopting strategy p.

Formally, we define a practical reasoning rule in the schema below, in which the conditions specify that a *reactive-rule* has an empty head (and it can be applied whenever the guard is true) and that a *plan-rule* has an achieve goal as its head.

$head = \{achieve\ \mathtt{schedule}(\mathtt{Activity}, \mathtt{Time}, \mathtt{DurationAct}, \mathtt{People}, \mathtt{Loc})\},$
$guard = and\ (pos\ \mathtt{free}(\mathtt{Time}, \mathtt{DurationAct}), pos\ \mathtt{location}(\mathtt{FromLoc}, \mathtt{Time})),$
$body = \{seqcomp\ (seqcomp\ ($
$\quad query\ (pos\ \mathtt{transport}(\mathtt{Means}, \mathtt{FromLoc}, \mathtt{Loc}, \mathtt{Time}, \mathtt{DurationTrans})),$
$\quad bac\ \mathtt{ins\_agenda}(\mathtt{Means}, \mathtt{Time} - \mathtt{DurationTrans}, \mathtt{DurationTrans}, \mathtt{agent}, \mathtt{FromLoc}))),$
$\quad bac\ \mathtt{ins\_agenda}(\mathtt{Activity}, \mathtt{Time}, \mathtt{Duration}, \mathtt{People}, \mathtt{Loc})))\}$

$head = \{seqcomp\ (goalvar\ \mathtt{X}, bac\ \mathtt{ins}(\mathtt{agenda}(\mathtt{Act}, \mathtt{Time}, \mathtt{Dur}, \mathtt{People}, \mathtt{Loc})))\},$
$guard = neg\ \mathtt{free}(\mathtt{Time}, \mathtt{Dur}),$
$body = \{achieve\ \mathtt{find\_alternative\_time}(\mathtt{Act}, \mathtt{Time}, \mathtt{Dur}, \mathtt{People}, \mathtt{Loc}, \mathtt{AltTime})\}$

Fig. 2. Two Practical Reasoning Rules

$$\begin{array}{|l}
\multicolumn{1}{l}{PRrule} \\
\hline
head, body : optional[Goal] \\
guard : Belief \\
type : PRType \\
\hline
head = \varnothing \Leftrightarrow type = reactive \wedge \\
thehead \in (\mathrm{ran}\ achieve) \wedge body \neq \varnothing \Leftrightarrow type = plan \\
\hline
\end{array}$$

Whilst there is a correspondence between the syntax of a PRrule and its purpose for reactive and plan rules, there is none for optimisation and failure rules because it is not possible to syntactically distinguish plans that fail or plans that can be optimised.

The first rule in Figure 2 inserts an activity in the agenda. It states that a plan to achieve the scheduling of an activity is to find a means of transportation to the specified location, then reserve the time needed for transport in the agenda, and finally insert the activity itself in the agenda. This plan should only be used if the slot at time `Time` of length `DurationAct` is still free in the agenda. The second conjunct in the guard of the rule is used to retrieve the location of the user at time `Time`. Note that `FromLoc` does not occur in the head of the rule so that the binding for it must be retrieved from the agent's beliefs. This illustrates the two uses of a guard as specifying the situation in which the rule might be considered by the agent and alternatively to retrieve some parameters from the agent's beliefs.

The second rule of Figure 2 is a revision rule that deals with failure. It states that if the agent has a sequential goal of doing anything (denoted by the goal variable `X`) followed by the goal of inserting an activity in the agenda of a user at a slot which is not free (specified by the guard), then the agent should consider revising that goal and replacing it with the goal of finding an alternative time for the activity.

Note that the variables that occur in the head and guard of a rule, are called the *global variables* of the rule, and those that occur in the body but not in the head or guard of the rule, are called the *local variables*. This distinction is made to separate the local data-processing in the body from the global variables that may also be used in other parts of a (complex) goal. In the first rule of Figure 2, the variables `Means` and `DurationTrans` are local variables that can only be used in the body of the goal and cannot transfer information to other parts of a (more complex) goal. Global variables, however, can be used to 'communicate with the rest of the goal' by parameter passing.

3 3APL Agents

3.1 Agents and Mental State

An agent can be characterised by specifying its beliefs, goals, practical reasoning rules and expertise. The main difference between these components of an agent are that the former two sets are dynamically updated while the latter two are fixed and do not change. We define an agent as an entity consisting of the static *expertise* and *rulebase*, i.e. a set of practical reasoning rules.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Agent}} \\
\hline
\textit{expertise} : \mathbb{P}\, \textit{Action} \\
\textit{rulebase} : \mathbb{P}\, \textit{PRrule} \\
\hline
\forall a : \textit{expertise} \bullet \textit{actionvars}\ a = \varnothing \\
\hline
\end{array}$$

This differs slightly from earlier work, in that the expertise of an agent is explicitly included as part of an agent. The predicate part of the schema indicates that the agent is only capable of performing grounded actions, since it is not clear how to specify the semantics of actions that contain variables. The interpretation of the instantiation of free variables in an action as a sensing act which semantically could be specified as a function of the environment of the agent fall outside the scope of this paper. We only deal with the specification of the components of a single agent in this paper.

The beliefs of an agent are recorded in its *beliefbase* and the goals of an agent in its *goalbase*. Together, these comprise the *mental state* of an agent. During the execution of an agent, its mental state is updated; the goals and beliefs of the agent are dynamic.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AgentState}} \\
\hline
\textit{Agent} \\
\textit{beliefbase} : \mathbb{P}\, \textit{Belief} \\
\textit{goalbase} : \mathbb{P}\, \textit{Goal} \\
\hline
\end{array}$$

Only the mental state of the agent may change during the operation of an agent, not the expertise or the rulebase. This is shown in the schema below by the Δ convention, which indicates that some state variables change, and the Ξ convention, which states that the dashed variables are equal to their undashed counterparts (i.e. no state change).

$$\begin{array}{|l}
\multicolumn{1}{l}{\Delta \textit{AgentState}} \\
\hline
\textit{AgentState}' \\
\textit{AgentState} \\
\Xi \textit{Agent} \\
\hline
\end{array}$$

3.2 Initial Agent State

A 3APL agent specifies the expertise (repertoire of basic actions), and a set of practical reasoning rules, but does not specify the initial beliefbase or goalbase of the agent, however. The first step in the operation of an agent, therefore, is to initialise the mental state. In the schema below, $b0?$ and $g0?$ denote input variables.

$$\begin{array}{|l}
\textit{InitAgentState} \\
\hline
\Delta \textit{AgentState} \\
b0? : \mathbb{P}\,\textit{Belief} \\
g0? : \mathbb{P}\,\textit{Goal} \\
\hline
\textit{beliefbase}' = b0? \\
\textit{goalbase}' = g0? \\
\hline
\end{array}$$

The operation of an agent is parameterised by two semantic notions. First, the semantics of basic actions is defined by a global function, $\textit{execute}$, which specifies that a basic action is an *update operator* on the beliefs of the agent. For example, the basic action $\textit{bac}\ \texttt{ins}(\texttt{agenda}(\texttt{Act}, \texttt{Time}, \texttt{Dur}, \texttt{People}, \texttt{Loc}))$ of a scheduling agent updates the beliefs by inserting a new item in the agenda. Since $\textit{execute}$ is a global function, any two agents capable of performing an action are guaranteed to do the same thing when executing that action. This is particularly important to prevent confusion when specifying and programming agents.

$$\textit{execute} : \textit{Action} \times \mathbb{P}\,\textit{Belief} \nrightarrow \mathbb{P}\,\textit{Belief}$$

The second semantic notion needed to specify the semantics of agents is a logical consequence relation. The logical consequence relation determines which implications the agent is allowed to derive from its beliefs. Formally, the consequence relation is a relation between two sets of beliefs such that the first set of beliefs implies all the beliefs in the second set. The logical consequence relation is also global. This makes sure that all agents draw conclusions from their beliefs in the same way, which guarantees a "minimal amount of global consistency". That is, one agent will not derive the negation of a belief b from the same set of beliefs from which another agent derives b.

$$\textit{LogCon}\_ : \mathbb{P}(\mathbb{P}\,\textit{Belief} \times \mathbb{P}\,\textit{Belief})$$

4 3APL Agent Operation

4.1 Applying Practical Reasoning Rules

Practical reasoning rules provide 3APL agents with reflective capabilities. The rules can be used to plan, revise, and create goals. The application of a rule is formally defined in this section. The application of a rule r to a goal g results in the replacement of a subgoal g' which matches with the head of rule r by the body of rule r in case the head of the rule is non-empty. If the body of the rule is empty, the subgoal is simply dropped. The application yields a substitution which is applied to the entire resulting goal. In

Rule: $head = \{seqcomp(goalvar\,\mathtt{X},\ bac\ \mathtt{ins\_agenda}(\mathtt{Act}, \mathtt{Time}, \mathtt{Dur}, \mathtt{People}, \mathtt{Loc})))\}$,
$guard = neg\mathtt{free}(\mathtt{Time}, \mathtt{Dur})$,
$body = \{achieve\mathtt{find\_alternative\_time}(\mathtt{Act}, \mathtt{Time}, \mathtt{Dur}, \mathtt{People}, \mathtt{Loc}, \mathtt{AltTime})\}$

Bel: $neg\ \mathtt{free}(\mathtt{may5th10:00}, \mathtt{60min})$

Goal: $seqcomp\ (seqcomp$
$(bac\ \mathtt{ins}(\mathtt{agenda}(\mathtt{train}, \mathtt{may5th9:30}, \mathtt{30min}, [\mathtt{john}], \mathtt{utrecht}))$,
$bac\ \mathtt{ins}(\mathtt{agenda}(\mathtt{meeting}, \mathtt{may5th10:00}, \mathtt{60min}, [\mathtt{john}, \mathtt{peter}], \mathtt{amsterdam})))$,
$achieve\ (\mathtt{new\_scheduling\_task}))$

Fig. 3. Example Scenario for Practical Reasoning Rules

Substitution: $\vartheta = \{\mathtt{X}/bac\ \mathtt{ins}(\mathtt{agenda}(\mathtt{train}, \mathtt{may5th9:30}, \mathtt{30min}, [\mathtt{john}], \mathtt{utrecht}))$,
$\mathtt{Act}/\mathtt{meeting}, \mathtt{Time}/\mathtt{may5th10:00}, \mathtt{Dur}/\mathtt{60min}$,
$\mathtt{People}/[\mathtt{john}, \mathtt{peter}], \mathtt{Loc}/\mathtt{amsterdam}\}$

Guard: $neg\ \mathtt{free}(\mathtt{may5th10:00}, \mathtt{60min})$

New Plan (Goal): $seqcomp\ ($
$achieve\ \mathtt{find\_alternative\_time}(\mathtt{meeting}, \mathtt{may5th10:00}, \mathtt{60min}$,
$[\mathtt{john}, \mathtt{peter}], \mathtt{amsterdam}, \mathtt{AltTime})$,
$achieve\ (\mathtt{new\_scheduling\_task}))$

Fig. 4. Results of Applying the Rule (and Substitution)

case the head of a rule is empty only the guard of the rule needs to be derivable from the beliefs of the agent, and a new goal (the body of the rule) is added to the goalbase of the agent.

Consider the example in Figure 3, with the practical reasoning rule of dealing with a failure of the scheduling of an activity, and a goal and belief as specified. Since the belief is an instance of the guard of the rule, and the goal can be unified with the head of the rule, the rule is applicable. Unifying the head of a rule with a (subgoal of a) goal of an agent amounts to finding a unifier (or substitution) which, when applied to the head of the rule and the goal makes them identical. In this example, unification yields a *most general unifier*, shown in Figure 4 which, when applied to the guard of the rule, gives the instantiated guard that is implied by the belief. Applying a rule to a subgoal means replacing that subgoal by the body of the rule, so that the new plan that replaces the original goal of the agent by applying the substitution, as also shown in the figure.

The example illustrates that a subgoal at the front of a goal of the agent is replaced, rather than just any subgoal, and this is the case for all rule applications. Consequently, the front contexts introduced earlier are very useful in specifying rule application.

Suppose that g' is a subgoal of some goal g that appears at the front of g, and g' matches with the head of a rule r. The task is to find a front context *fc* such that if the subgoal g' is inserted for $\Box$ (at the front of *fc*), the resulting goal is identical to g. Applying r then amounts to *updating* the *fc* with the body of the rule. There is a crucial difference here between inserting and updating. Inserting a goal in a front context means substituting the goal for the $\Box$ in the front context; while updating a front context with a goal means replacing $\Box$ with that goal *and also committing* to the choices made (pursuing a subgoal in a choice goal means committing to the branch in which the subgoal appears in the choice goal).

To formalise this, we define two functions, one to *insert* a goal into the square of a front context and one to *update* a front context with a goal.

$$
\begin{array}{|l}
\mathit{Insert} : (\mathit{Goal} \times \mathit{FrontContext}) \rightarrow \mathit{Goal} \\
\hline
\forall\, g, g' : \mathit{Goal};\ \mathit{fc} : \mathit{FrontContext} \bullet \\
\qquad \mathit{Insert}\ (g, \Box) = g \wedge \\
\qquad \mathit{Insert}\ (g, \mathit{seqcomp}(\mathit{fc}, g')) = \mathit{seqcomp}(\mathit{Insert}(g, \mathit{fc}), g') \wedge \\
\qquad \mathit{Insert}\ (g, \mathit{choice}(\mathit{fc}, g')) = \mathit{choice}(\mathit{Insert}(g, \mathit{fc}), g') \wedge \\
\qquad \mathit{Insert}\ (g, \mathit{choice}(g', \mathit{fc})) = \mathit{choice}(g', \mathit{Insert}(g, \mathit{fc}))
\end{array}
$$

Now, since a front context may be updated with the empty goal if a rule with empty body is applied or an execution step is performed (see below), the goal types of the function $\mathit{UpdateGoal}$ are optional goals. Note that if the front context is a choice context, this commits to the branch where the $\Box$ occurs. The latter branch is the one the agent has chosen to pursue. This is different from Insert, which left these branches intact.

$$
\begin{array}{|l}
\mathit{UpdateGoal} : (\mathit{optional}[\mathit{Goal}] \times \mathit{FrontContext}) \rightarrow \mathit{optional}[\mathit{Goal}] \\
\hline
\forall\, g : \mathit{optional}[\mathit{Goal}];\ \mathit{fc} : \mathit{FrontContext};\ g' : \mathit{Goal} \bullet \\
\mathit{UpdateGoal}\ (g, \Box) = g \wedge \\
\mathit{non} - \mathit{empty}\ \mathit{UpdateGoal}\ (g, \mathit{fc}) \Rightarrow \\
\qquad (\mathit{UpdateGoal}\ (g, \mathit{seqcomp}(\mathit{fc}, g')) = \\
\qquad\qquad\qquad \{\mathit{seqcomp}(\mathit{the}\ (\mathit{UpdateGoal}\ (g, \mathit{fc})), g')\} \wedge \\
\qquad \mathit{UpdateGoal}\ (g, \mathit{choice}(\mathit{fc}, g')) = \mathit{UpdateGoal}\ (g, \mathit{fc}) \wedge \\
\qquad \mathit{UpdateGoal}\ (g, \mathit{choice}(g', \mathit{fc})) = \mathit{UpdateGoal}\ (g, \mathit{fc})) \wedge \\
\mathit{empty}\ \mathit{UpdateGoal}\ (g, \mathit{fc}) \Rightarrow \\
\qquad (\mathit{UpdateGoal}\ (g, \mathit{seqcomp}(\mathit{fc}, g')) = \{g'\} \wedge \\
\qquad \mathit{UpdateGoal}\ (g, \mathit{choice}(\mathit{fc}, g')) = \varnothing \wedge \\
\qquad \mathit{UpdateGoal}\ (g, \mathit{choice}(g', \mathit{fc})) = \varnothing)
\end{array}
$$

Note that if the front context is of the form $\mathit{choice}\ (\Box, g)$ for some goal g, and the empty goal is inserted for $\Box$, Insert yields $\mathit{Insert}(\mathit{choice}\ (\Box, g)) = \{g\}$. This is a natural definition, but the empty goal could result with a different view on dropping a branch from a choice goal. A rule with empty body *prevents* an agent from attempting to execute a particular goal. If the dropped goal is a branch of a choice goal, Insert simply removes this branch. The definition of Insert above does not remove the alternative branch, however, so that the choice goal thus is still not completed. If there is a reason to remove the choice goal, then this reason should be stated in the guard of a rule which removes the choice goal completely.

A rule is *applicable* if the head unifies with a (sub)goal of the agent *and* the guard of the rule follows from the agent's beliefs. If the rule has no head, it is applicable simply if the guard follows from the beliefbase. For a treatment of substitutions, binding and unification please see Appendix A.

$$\begin{array}{|l}
applicable\_ : \mathbb{P}(PRrule \times Goal \times (\mathbb{P}\, Belief)) \\
\hline
\forall\, g : Goal;\ r : PRrule;\ bb : \mathbb{P}\, Belief \bullet \\
non-empty\ r.head \Rightarrow (applicable(r, g, bb) \Leftrightarrow \\
\quad (\exists\, \vartheta, \gamma : Substitution;\ subg : Goal;\ fc : FrontContext \bullet \\
\qquad Insert(subg, fc) = g \land mgu((the\ r.head), subg) = \vartheta \land \\
\qquad (\mathrm{dom}\, \gamma) \subseteq (beliefvars\ r.guard) \land \\
\qquad LogCon(bb, \{ASBelief\, (\vartheta \ddagger \gamma) r.guard\}))) \land \\
empty\ r.head \Rightarrow (applicable(r, g, bb) \Leftrightarrow \\
\quad (\exists\, \vartheta : Substitution \mid (\mathrm{dom}\, \vartheta) \subseteq (beliefvars\ r.guard) \bullet \\
\qquad LogCon(bb, \{ASBelief\ \vartheta\ r.guard\})))
\end{array}$$

Using this definition, we can now specify the rule application. If the head of the rule is not empty, applying the rule amounts to replacing a subgoal by the body of the rule. Otherwise, we simply add the body of the rule to the goalbase of the agent. Care must be taken here to avoid interference of variables occurring in rules and those variables occurring in goals (cf. [7]. For this reason, all variables in the rule applied are renamed to variables not occurring in the target goal. A function $RuleRename(r, V)$, which is not defined in this paper due to space constraints (but available on request from the authors), renames the variables in the rule r so that no variable from the set V of variables occurs in the renamed rule.

$$\begin{array}{|l}
RuleRename : (PRrule \times (\mathbb{P}\ Var)) \rightarrow PRrule
\end{array}$$

$$\begin{array}{|l}
\hline
\multicolumn{1}{|l}{ApplyRule} \\
\Delta AgentState \\
g? : Goal;\ r? : PRrule;\ rr : PRrule \\
\hline
rr = RuleRename(r?, goalvars\ \{g?\}) \\
r?.type \neq reactive \Rightarrow \\
\quad (\exists\, fc : FrontContext;\ subg : Goal \bullet Insert(subg, fc) = g? \land \\
\qquad (\exists\, \vartheta, \gamma : Substitution \mid (\mathrm{dom}\, \gamma) \subseteq (beliefvars\ rr.guard) \bullet \\
\qquad\quad mgu(the\ rr.head, subg) = \vartheta \land \\
\qquad\quad LogCon(beliefbase, \{ASBelief\ (\vartheta \ddagger \gamma) rr.guard\}) \land \\
\qquad\qquad beliefbase' = beliefbase \land \\
\qquad\qquad goalbase' = goalbase \setminus \{g?\} \cup \\
\qquad\qquad\quad ASGoal\ (\vartheta \ddagger \gamma)\ \{(Insert\ (the(rr.body), fc))\})) \\
r?.type = reactive \Rightarrow \\
\quad (\exists\, \gamma : Substitution \mid (\mathrm{dom}\, \gamma) \subseteq (beliefvars\ rr.guard) \bullet \\
\qquad\quad LogCon(beliefbase, \{ASBelief\ \gamma\ rr.guard\}) \land \\
\qquad\qquad beliefbase' = beliefbase \land \\
\qquad\qquad goalbase' = goalbase \cup ASGoal\ \gamma\ rr.body) \\
\hline
\end{array}$$

4.2 Goal Execution

The execution of a goal is specified through the computation steps an agent can perform on a goal. A computation step corresponds to a simple action of the agent, which is either a basic action or else a query on the beliefs of the agent. Recall that the semantics of basic actions is given by a global function $execute$ and the semantics of beliefs is specified by the $LogCon$ relation.

The agent is only allowed to execute a basic action or query that occurs at the front of a goal, i.e. it is one of the first things the agent should consider doing. The notion of front context is useful to find an action or query which the agent might execute. If there is a front context fc in which a basic action or query can be inserted for $\Box$, and which results in a goal of the agent, the agent might consider executing that basic action or query. After executing the goal, the goal needs to be updated, and this updating is the same as updating the front context by removing $\Box$.

The execution of a basic action amounts to changing the beliefbase of the agent in accordance with the function $execute$. The condition $(a, beliefbase) \in (\mathrm{dom}\, execute)$ expresses that the basic action a is enabled, and thus can be executed.

$$\begin{array}{|l}
\multicolumn{1}{l}{ExecuteBasicAction} \\ \hline
\Delta AgentState \\
\Xi Agent \\
g? : Goal \\ \hline
(\exists fc : FrontContext;\ a : Action \mid a \in expertise \wedge \\
\quad Insert((bac\ a), fc) = g? \wedge \ \ (a, beliefbase) \in (\mathrm{dom}\, execute) \bullet \\
\quad\quad beliefbase' = execute(a, beliefbase) \wedge \\
\quad\quad goalbase' = (goalbase \setminus \{g?\}) \cup UpdateGoal\ (\{\}, fc)) \\ \hline
\end{array}$$

Queries are goals to check if some condition follows from the beliefbase of the agent. Any free variables in the condition of the query can be used to retrieve data from the beliefbase. The values retrieved are recorded in a substitution ϑ. A query can only be executed if it is a consequence of the beliefbase (otherwise, nothing happens).

$$\begin{array}{|l}
\multicolumn{1}{l}{ExecuteQueryGoal} \\ \hline
\Delta AgentState \\
\Xi Agent \\
g? : Goal \\ \hline
(\exists fc : FrontContext;\ b : Belief \bullet \\
\quad Insert(query\ b, fc) = g? \wedge \\
\quad (\exists \vartheta : Substitution \bullet LogCon\ (beliefbase, \{ASBelief\ \vartheta\ b\}) \wedge \\
\quad\quad beliefbase' = beliefbase \wedge \\
\quad\quad goalbase' = (goalbase \setminus \{g?\}) \cup (ASGoal\ \vartheta\ (UpdateGoal\ (\{\}, fc))))) \\ \hline
\end{array}$$

Executing a goal is then defined as the disjunction of these two functions.

$$ExecuteGoal == ExecuteBasicAction \vee ExecuteQueryGoal$$

5 Conclusions

In this paper we have described a specification of the agent-oriented programming language 3APL. The arguments for using Z in agent-based systems are well-rehearsed (eg.[5, 3]), and we will not re-state them here. In particular, however, Z enables a uniform presentation of *both* the 3APL programming language *and* its architecture in a clear and concise way. We are not familiar with any work that specifies both these aspects in this way, and believe that our work moves a step closer to a unified account of agent languages and architectures.

The contribution of this work is threefold. First, we provide an operational specification of 3APL that can be used as the basis of a subsequent implementation, so that the transition from what might be called theory to practice is facilitated. At lower levels, this kind of transition is demonstrated, for example, through the provision of a simple agent simulation environment [11], and a sophisticated Jini-based development environment [1], both based on an extensive agent framework [10]. This work addresses the more detailed aspects involved in dealing with the transition of a fully designed system, rather than an outline structure. Second, we allow an easy and simple comparison of 3APL and its *competitor* systems that have been specified in a similar style such as AgentSpeak(L) [3] and dMARS [2], as illustrated in [6]. Third, we provide an accessible resource in the specification of techniques for the development of agent systems that might not otherwise be available in a form relevant both to agent architects and developers. This work can thus be viewed in a standalone fashion in contributing to the understanding of 3APL on the one hand, and in using it to provide a window on the larger area of generic agent architecture on the other.

Acknowledgements. We would like to thank Jean-Jules Meyer for many constructive and helpful discussions during the preparation of this work, and the Universities of Westminster and Utrecht for hosting and supporting the authors during their collaboration.

References

1. R. Ashri and M. Luck. Agent implementation through jini. In *Proceedings of the Eleventh International Workshop on Database and Expert Systems Applications*. IEEE Computer Society Press, to appear 2000.
2. M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages, Lecture Notes in Artificial Intelligence 1365*, pages 155–176. Springer-Verlag, 1998.
3. M. d'Inverno and M. Luck. Engineering agentspeak(L): A formal computational model. *Journal of Logic and Computation*, 8(3):233–260, 1998.
4. M. R. Genesereth and N. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufman, 1987.
5. R. Goodwin. A formal specification of agent properties. *Journal of Logic and Computation*, 5(6):763–781, 1995.
6. K. Hindriks, M. d'Inverno, and M. Luck. Architecture for agent programming languages. In *ECAI 2000: Proceedings of the Fourteenth European Conference on Artificial Intelligence*, to appear 2000.
7. K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J-J. Ch. Meyer. Formal Semantics for an Abstract Agent Programming Language. In *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages, Lecture Notes in Artificial Intelligence 1365*, pages 215–229. Springer-Verlag, 1998.

8. K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J-J. Ch. Meyer. Control structures of rule-based agent languages. In *Intelligent Agents V, Lecture Notes in Artificial Intelligence 1555*. Springer-Verlag, 1999.
9. G. Kiss. Goal, values, and agent dynamics. In G. M. P. O'Hare and N. R. Jennings (eds), editors, *Foundations of Distributed Artificial Intelligence*, pages 247–268. John Wiley and Sons, 1996.
10. M. Luck and M. d'Inverno. Structuring a Z specification to provide a formal framework for autonomous agent systems. In J. P. Bowen and M. G. Hinchey, editors, *ZUM'95: The Z Formal Specification Notation, 9th International Conference of Z Users, Lecture Notes in Computer Science 967*, pages 48–62. Springer-Verlag, 1995.
11. M. Luck, N. Griffiths, and M. d'Inverno. From agent theory to agent construction: A case study. In *Intelligent Agents III: Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages, Lecture Notes in Artificial Intelligence, 1193*, pages 49–63. Springer Verlag, 1997.
12. P. Maes. Agents that reduce work and information overload. *Communication of the ACM*, 37(7):30–40, 1994.
13. A. S. Rao. Agentspeak(l): BDI agents speak out in a logical computable language. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Lecture Notes in Artificial Intelligence 1038*, pages 42–55. Springer-Verlag, 1996.
14. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

A Substitutions

Here, we provide further details of the standard definitions of binding and unification, described in Z. First we define a substitution. This maps variables to terms, as well as mapping goal variables to goals. We therefore introduce a new type in order to define a substitution which we call $SubTerm$.

A substitution is represented as a partial function between variables and terms since, in general, only some variables will be mapped to a term. (Remember that $fovars$ is a function that returns the variables of a term, as defined earlier, at the end of Section 4.1.)

$$\begin{aligned} SubTerm ::= {} & term\langle\langle FOTerm\rangle\rangle \\ \mid {} & goal\langle\langle Goal\rangle\rangle \end{aligned}$$

$$\begin{array}{|l} allvars : SubTerm \rightarrow (\mathbb{P}\ Var) \\ \hline \forall\, t : FOTerm;\ g : Goal \bullet \\ \quad allvars(term\ t) = fovars\ t \land \\ \quad allvars(goal\ g) = goalvars\ \{g\} \end{array}$$

$$Substitution == \{\vartheta : Var \rightarrow\!\!\!\!\!+ SubTerm\}$$

The standard definition of a substitution is a mapping from variables to terms such that no variable contained in any of the terms is in the domain of the mapping [4].

$$\begin{aligned} &\forall\, \vartheta : Substitution \bullet \\ &\quad (\mathrm{dom}\,\vartheta) \cap \bigcup\{s : SubTerm \mid s \in (\mathrm{ran}\,\vartheta) \bullet allvars\ s\} = \varnothing \end{aligned}$$

We also have the following predicate concerning substitutions.

$$\forall \vartheta : Substitution;\ v : Var;\ s : SubTerm;\ t : FOTerm \mid (v, s) \in \vartheta \bullet \\ \quad (v \in FOVar \Leftrightarrow s \in (\operatorname{ran} term)) \land (v \in GVar \Leftrightarrow s \in (\operatorname{ran} goal))$$

A.1 Application of Substitutions

The function, $ASFOVar$, applies either the identity mapping to a variable if the variable is not in the domain of the substitution, or it applies the substitution if it is in the domain. (Note that this function is only defined for elements of $FOVar$ not of $GVar$.)

$$\begin{array}{|l} ASFOVar : Substitution \rightarrow FOVar \rightarrow FOTerm \\ \hline \forall \vartheta : Substitution;\ v : FOVar \bullet \\ \quad (v \notin (\operatorname{dom} \vartheta)) \Rightarrow ASFOVar\ \vartheta\ v = var\ v \land \\ \quad (v \in (\operatorname{dom} \vartheta)) \Rightarrow ASFOVar\ \vartheta\ v = term^{-1}(\vartheta\ v) \end{array}$$

We can then define what it means for a substitution to be applied to a term.

$$\begin{array}{|l} ASTerm : Substitution \rightarrow FOTerm \rightarrow FOTerm \\ \hline \forall t : FOTerm;\ f : FuncSym;\ ts : \operatorname{seq} FOTerm;\ \vartheta : Substitution \mid \\ \quad t = functor\ (f, ts) \bullet \\ \qquad t \in \operatorname{ran} const \Rightarrow ASTerm\ \vartheta\ t = t \land \\ \qquad t \in \operatorname{ran} var \Rightarrow ASTerm\ \vartheta\ t = ASFOVar\ \vartheta\ (var^{-1} t) \land \\ \qquad t \in \operatorname{ran} functor \Rightarrow ASTerm\ \vartheta\ t = \\ \qquad\quad (\mu\ new : FOTerm \mid first\ (functor^{-1} new) = f \land \\ \qquad\qquad second\ (functor^{-1} new) = map\ (ASTerm\ \vartheta)\ ts\ ) \end{array}$$

A substitution to be applied to an atom and an action.

$$\begin{array}{|l} ASAtom : Substitution \rightarrow Atom \rightarrow Atom \\ ASAction : Substitution \rightarrow Action \rightarrow Action \\ \hline \forall a, a' : Atom;\ act, act' : Action;\ \vartheta : Substitution \\ \quad \bullet ASAtom\ \vartheta\ a = a' \Leftrightarrow a'.head = a.head \land \\ \qquad a'.terms = map(ASTerm\ \vartheta) a.terms \land \\ \quad ASAction\ \vartheta\ act = act' \Leftrightarrow act'.name = act.name \land \\ \qquad act'.terms = map(ASTerm\ \vartheta) act.terms \end{array}$$

$$\begin{array}{|l} ASBelief : Substitution \rightarrow Belief \rightarrow Belief \\ \hline \forall b, c : Belief;\ l, a : Atom;\ s : Substitution \bullet \\ \quad ASBelief\ s\ (pos\ a) = (pos\ (ASAtom\ s\ a)) \land \\ \quad ASBelief\ s\ (not\ a) = (not\ (ASAtom\ s\ a)) \land \\ \quad ASBelief\ s\ (\mathbf{true}) = \mathbf{true} \land \\ \quad ASBelief\ s\ (\mathbf{false}) = \mathbf{false} \end{array}$$

A substitution to be applied to a goal. Remember, goals contain goal variables as well as terms variables so as well as using all our previous definitions for substitution application we must define what it means to apply a substitution to a goal variable. This is considered in the final two predicates.

$$
\begin{array}{|l}
ASGoal : Substitution \rightarrow optional[Goal] \rightarrow optional[Goal] \\
\hline
\forall\, s : Substitution;\ a : Action;\ b : Belief;\ at : Atom; \\
\qquad\qquad\qquad\qquad g_1, g_2 : Goal;\ gv : GVar \bullet \\
\quad ASGoal\ s\ \varnothing = \varnothing \wedge \\
\quad ASGoal\ s\ \{bac\ a\} = \{bac\ (ASAction\ s\ a)\} \wedge \\
\quad ASGoal\ s\ \{query\ b\} = \{query\ (ASBelief\ s\ b)\} \wedge \\
\quad ASGoal\ s\ \{achieve\ at\} = \{achieve(ASAtom\ s\ at)\} \wedge \\
\quad ASGoal\ s\ \{seqcomp(g_1, g_2)\} = \\
\qquad\qquad \{seqcomp(the(ASGoal\ s\ \{g_1\}), the(ASGoal\ s\ \{g_2\}))\} \wedge \\
\quad ASGoal\ s\ \{choice(g_1, g_2)\} = \\
\qquad\qquad \{choice(the(ASGoal\ s\ \{g_1\}), the(ASGoal\ s\ \{g_2\}))\} \wedge \\
\quad gv \notin (\mathrm{dom}\, s) \Rightarrow \\
\qquad ASGoal\ s\ \{goalvar\ gv\} = \{goalvar\ gv\} \wedge \\
\quad gv \in (\mathrm{dom}\, s) \Rightarrow \\
\qquad ASGoal\ s\ \{goalvar\ gv\} = \{goal^{-1}(s\ gv)\}
\end{array}
$$

A.2 Composition of Substitutions

Consider two substitutions τ and σ such that no variable bound in σ appears anywhere in τ. The composition of τ with σ, written $\tau \ddagger \sigma$, is obtained by applying τ to the terms in σ and combining these with the bindings from τ.

For example, if $\tau = \{x/A, y/B, z/C\}$ and $\sigma = \{u/A, v/F(x, y, z)\}$ then, since none of the variables bound in σ (u, v) appear in τ, it is meaningful to compose τ with σ. In this case $\tau \ddagger \sigma = \{u/A, v/F(A, B, C), x/A, y/B, z/C\}$.

The definition is a bit convoluted though because of the typing needed top include goal variables.

$$
\begin{array}{|l}
\_ \ddagger \_ : (Substitution \times Substitution) \rightarrow Substitution \\
\hline
\forall\, \tau, \sigma : Substitution \mid \\
\quad (\mathrm{dom}\, \sigma) \cap ((\mathrm{dom}\, \tau)\ \cup\ \bigcup\{t : FOTerm;\ s : SubTerm \mid \\
\qquad\qquad (s = term\ t) \wedge (s \in (\mathrm{ran}\, \tau)) \bullet fovars\ t\}) = \varnothing \bullet \\
\qquad \tau \ddagger \sigma = (\tau \cup \{x : Var;\ t : FOTerm;\ s : SubTerm \mid \\
\qquad\qquad (s = term\ t) \wedge (x, s) \in \sigma \bullet (x, term\ (ASTerm\ \tau\ t))\})
\end{array}
$$

A.3 Unification

A substitution is a *unifier* for two terms if the substitution, applied to both of them, makes them equal.

$$
\begin{array}{|l}
unifyterms\_ : \mathbb{P}(Substitution \times (Term \times FOTerm)) \\
\hline
\forall\, t_1, t_2 : FOTerm;\ s : Substitution \bullet \\
\quad unifyterms(s, (t_1, t_2)) \Leftrightarrow (ASTerm\ s\ t_1 = ASTerm\ s\ t_2)
\end{array}
$$

A substitution is a *unifier* for two goals if the substitution, applied to both of them, makes them equal.

$$\begin{array}{|l}
\mathit{unifygoals}\_ : \mathbb{P}(\mathit{Substitution} \times (\mathit{Goal} \times \mathit{Goal})) \\
\hline
\forall\, g_1, g_2 : \mathit{Goal};\ \vartheta : \mathit{Substitution} \bullet \\
\qquad \mathit{unifygoals}(\vartheta, (g_1, g_2)) \Leftrightarrow (\mathit{ASGoal}\ \vartheta\ \{g_1\}) = (\mathit{ASGoal}\ \vartheta\ \{g_2\})
\end{array}$$

A substitution is *more general* than another substitution if there exists a third substitution which, when composed with the first, gives the second.

$$\begin{array}{|l}
\_\ \mathit{mg}\ \_ : \mathbb{P}(\mathit{Substitution} \times \mathit{Substitution}) \\
\hline
\forall\, \vartheta, \gamma : \mathit{Substitution} \bullet \gamma\ \mathit{mg}\ \vartheta\ \Leftrightarrow (\exists\, \omega : \mathit{Substitution} \bullet (\gamma \ddagger \omega) = \vartheta)
\end{array}$$

The mgu of two goals is specified as follows.

$$\begin{array}{|l}
\mathit{mgu} : (\mathit{Goal} \times \mathit{Goal}) \rightarrow \mathit{Substitution} \\
\hline
\forall\, g_1, g_2 : \mathit{Goal};\ \gamma : \mathit{Substitution} \bullet \mathit{mgu}\ (g_1, g_2) = \gamma \Leftrightarrow \\
\qquad (\mathit{unifygoals}(\gamma, (g_1, g_2)) \wedge \\
\qquad\qquad \neg\, (\exists\, \omega : \mathit{Substitution} \bullet (\mathit{unifygoals}(\omega, (g_1, g_2)) \wedge (\omega\ \mathit{mg}\ \gamma))))
\end{array}$$