Semantics of Plan Revision in Intelligent Agents

M. Birna van Riemsdijk[1] John-Jules Ch. Meyer[1] Frank S. de Boer[1,2,3]

[1] ICS, Utrecht University, The Netherlands
[2] CWI, Amsterdam, The Netherlands
[3] LIACS, Leiden University, The Netherlands

Abstract. In this paper, we give an operational and denotational semantics for a 3APL meta-language, with which various 3APL interpreters can be programmed. We moreover prove equivalence of these two semantics. Furthermore, we relate this 3APL meta-language to object-level 3APL by providing a specific interpreter, the semantics of which will prove to be equivalent to object-level 3APL.

1 Introduction

An agent is commonly seen as an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives [19]. Autonomy means that an agent encapsulates its state and makes decisions about what to do based on this state, without the direct intervention of humans or others. Agents are situated in some environment which can change during the execution of the agent. This requires *flexible* problem solving behaviour, i.e. the agent should be able to respond adequately to changes in its environment. Programming flexible computing entities is not a trivial task. Consider for example a standard procedural language. The assumption in these languages is, that the environment does not change while some procedure is executing. If problems do occur during the execution of a procedure, the program might throw an exception and terminate (see also [20]). This works well for many applications, but we need something more if change is the norm and not the exception.

A philosophical view that is well recognized in the AI literature is, that rational behaviour can be explained in terms of the concepts of *beliefs*, *goals* and *plans*[4] [1,13,2]. This view has been taken up within the AI community in the sense that it might be possible to *program* flexible, autonomous agents *using* these concepts. The idea is, that an agent tries to fulfill its goals by selecting appropriate plans, depending on its beliefs about the world. Beliefs should thus represent the world or environment of the agent; the goals represent the state of the world the agent wants to realize and plans are the means to achieve these goals. When programming in terms of these concepts, beliefs can be compared

[4] In the literature, also the concepts of desires and intentions are often used, besides or instead of goals and plans, respectively. This is however not important for the current discussion.

to the program state, plans can be compared to statements, i.e. plans constitute the procedural part of the agent, and goals can be viewed as the (desired) post-conditions of executing the statement or plan. Through executing a plan, the world and therefore the beliefs reflecting the world will change and this execution should have the desired result, i.e. achievement of goals.

This view has been adopted by the designers of the agent programming language *3APL*[5] [8]. The dynamic parts of a 3APL agent thus consist of a set of beliefs, a plan[6] and a set of goals[7]. A plan can consist of sequences of so-called basic actions and abstract plans. Basic actions change the beliefs[8] if executed and abstract plans can be compared to procedure names. To provide for the possibility of programming flexible behaviour, so-called *plan revision* rules were added to the language. These rules can be compared to procedures in the sense that they have a head (the procedure name) and a body (a plan or statement). The operational meaning of plan revision rules is similar to that of procedures: if the procedure name or head is encountered in a statement or plan, this name or head is replaced by the body of the procedure or rule, respectively (see [4] for the operational semantics of procedure calls). The difference however is, that the head in a plan revision rule can be *any* plan (or statement) and not just a procedure name. In procedural languages it is furthermore usually assumed that procedure names are distinct. In 3APL however, it is possible that multiple rules are applicable at the same time. This provides for very general and flexible plan revision capabilities, which is a distinguishing feature of 3APL compared to other agent programming languages [12,14,6].

As argued, we consider these general plan revision capabilities to be an essential part of agenthood. The introduction of these capabilities now gives rise to interesting issues concerning the *semantics of plan execution*, the exploration of which is the topic of this paper.

Semantics of plan execution can be considered on two levels. On the one hand, the semantics of *object-level* 3APL can be studied as a function yielding the result of executing a plan on an initial belief base, where the plan can be revised through plan revision rules during execution. An interesting question is, whether a denotational semantic function can be defined that is compositional in its plan argument.

On the other hand, the semantics of a 3APL *interpreter* language or *meta-language* can be studied, where a plan and a belief base are considered the data on which the interpreter or meta-program operates. This meta-language is the main focus of this paper. To be more specific, we define a meta-language and provide an operational and denotational semantics for it. These will be proven equivalent. We furthermore define a very general interpreter in this language, the

[5] 3APL is to be pronounced as "triple-a-p-l".
[6] In the original version this was a set of plans.
[7] The addition of goals was a recent extension [18].
[8] A change in the environment is a possible "side effect" of the execution of a basic action.

semantics of which will prove to be equivalent to the semantics of object-level 3APL.

For regular procedural programming languages, studying a specific interpreter language is in general not very interesting. In the context of agent programming languages it however *is*, for several reasons. First of all, 3APL and agent-oriented programming languages in general are non-deterministic by nature. In the case of 3APL for example, it will often occur that several plan revision rules are applicable at the same time. Choosing a rule for application (or choosing whether to execute an action from the plan or to apply a rule if both are possible), is the task of a 3APL interpreter. The choices made, affect the outcome of the execution of the agent. In the context of agents, it is interesting to study various interpreters, as different interpreters will give rise to different *agent types*. An interpreter that for example always executes a rule if possible, thereby deferring action execution, will yield a thoughtful and passive agent. In a similar way, very bold agents can be constructed or agents with characteristics anywhere on this spectrum. These conceptual ideas about various agent types fit well within the agent metaphor and therefore it is worthwhile to study an interpreter language and the interpreters that can be programmed in it (see also [3]).

Secondly, as pointed out by Hindriks [7], differences between various agent languages often mainly come down to differences in their meta-level reasoning cycle or interpreter. To provide for a *comparison* between languages, it is thus important to separate the semantic specification of object-level and meta-level execution.

Finally, and this was the original motivation for this work, we hope that the specification of a denotational semantics for the meta-language might shed some light onto the issue of specifying a denotational semantics for object-level 3APL. It however seems, contrary to what one might think, that the denotational semantics of the meta-language cannot be used to define a denotational semantics for object-level 3APL. We will elaborate on this issue in section 6.2.

2 Syntax

2.1 Object-level

As stated in the introduction, the latest version of 3APL incorporates beliefs, goals and plans. In this paper, we will however consider a version of 3APL with only beliefs and plans as was defined in [8]. The reason is, that in this paper we focus on the semantics of plan execution, for the treatment of which only beliefs and plans will suffice. The language defined in [8] is a first-order language, a propositional and otherwise slightly simplified version of which we will use in this paper.

In the sequel, a language defined by inclusion shall be the smallest language containing the specified elements.

Definition 1. *(belief bases)* Assume a propositional language $\mathcal{L}$ with typical formula ψ and the connectives $\wedge$ and $\neg$ with the usual meaning. Then the set of possible belief bases Σ with typical element σ is defined to be $\wp(\mathcal{L})$.

Definition 2. *(plans)* Assume that a set BasicAction with typical element a is given, together with a set AbstractPlan. The symbol E denotes the empty plan. Then the set of plans Π with typical element π is defined as follows:

- $\{E\} \cup \mathsf{BasicAction} \cup \mathsf{AbstractPlan} \subseteq \Pi$,
- if $c \in (\{E\} \cup \mathsf{BasicAction} \cup \mathsf{AbstractPlan})$ and $\pi \in \Pi$ then $c\,;\pi \in \Pi$[9].

A plan $E;\pi$ is identified with the plan π.

For reasons of presentation and technical convenience, we exclude non-deterministic choice and test from plans. This is no fundamental restriction as non-determinism is introduced by plan revision rules (to be introduced below). Furthermore, tests can be modelled as basic actions that do not affect the state if executed (for semantics of basic actions see definition 8).

A plan and a belief base can together constitute the so-called mental state of a 3APL agent. A mental state can be compared to what is usually called a configuration in procedural languages, i.e. a statement-state pair.

Definition 3. *(mental states)* Let Σ be the set of belief bases and let Π be the set of plans. Then $\Pi \times \Sigma$ is the set S of possible mental states of a 3APL agent.

Definition 4. *(plan revision (PR) rules)* A PR rule ρ is a triple $\pi_h \mid \psi \rightsquigarrow \pi_b$ such that $\psi \in \mathcal{L}$, $\pi_h, \pi_b \in \Pi$ and $\pi_h \neq E$.

Definition 5. *(3APL agent)* A 3APL agent $\mathcal{A}$ is a tuple $\langle \pi_0, \sigma_0, \mathsf{BasicAction}, \mathsf{AbstractPlan}, \mathsf{Rule}, \mathcal{T} \rangle$ where $\langle \pi_0, \sigma_0 \rangle$ is the initial mental state, BasicAction, AbstractPlan and Rule are sets of basic actions, abstract plans and PR rules respectively and $\mathcal{T} : (\mathsf{BasicAction} \times \Sigma) \rightarrow \Sigma$ is a belief update function.

In the following, when referring to agent $\mathcal{A}$, we will assume this agent to have a set of basic actions BasicAction, a set of abstract plans AbstractPlan, a set of PR rules Rule and a belief update function $\mathcal{T}$.

2.2 Meta-level

In this section, we define the meta-language that can be used to write 3APL interpreters. The programs that can be written in this language will be called *meta-programs*. Like regular imperative programs, these programs are state transformers. The kind of states they transform however do not simply consist of an assignment of values to variables like in regular imperative programming, but the states that are transformed are 3APL mental states. In section 3.1, we will define the operational semantics of our meta-programs. We will do this using

[9] For technical convenience, plans are defined to have a list structure.

the concept of a *meta-configuration*. A meta-configuration consists of a meta-program and a mental state, i.e. the meta-program is the procedural part and the mental state is the "data" on which the meta-program operates.

The basic elements of meta-programs are the *execute* action and the $apply(\rho)$ action (called *meta-actions*). The *execute* action is used to specify that a basic action from the plan of an agent should be executed. The $apply(\rho)$ action is used to specify that a PR rule ρ should be applied to the plan. Composite meta-programs can be constructed in a standard way.

Below, the meta-programs and meta-configurations for agent $\mathcal{A}$ are defined.

Definition 6. *(meta-programs)* We assume a set $Bexp$ of boolean expressions with typical element b. Let $b \in Bexp$ and $\rho \in \mathsf{Rule}$, then the set $Prog$ of meta-programs with typical element P is defined as follows:

$$P ::= execute \mid apply(\rho) \mid \texttt{while}\ b\ \texttt{do}\ P\ \texttt{od} \mid P_1; P_2 \mid P_1 + P_2.$$

Definition 7. *(meta-configurations)* Let $Prog$ be the set of meta-programs and let S be the set of mental states. Then $Prog \times S$ is the set of possible meta-configurations.

3 Operational Semantics

In [8], the operational semantics of 3APL is defined using transition systems [11]. A transition system for a programming language consists of a set of derivation rules for deriving transitions for this language. A transition is a transformation of one configuration into another and it corresponds to a single computation step. In the following section, we will repeat the transition system for 3APL given in [8] (adapted to fit our simplified language) and we will call it the *object-level* transition system. We will furthermore give a transition system for the meta-programs defined in section 2.2 (the *meta-level* transition system). Then in the last section, we will define the operational semantics of the object- and meta-programs using the defined transition systems.

3.1 Transition Systems

The transition systems defined in the following sections assume 3APL agent $\mathcal{A}$.

Object-level The object-level transition system (Trans_o) is defined by the rules given below. The transitions are labeled to denote the kind of transition.

Definition 8. *(action execution)* Let $a \in \mathsf{BasicAction}$.

$$\frac{\mathcal{T}(a, \sigma) = \sigma'}{\langle a; \pi, \sigma \rangle \rightarrow_{execute} \langle \pi, \sigma' \rangle}$$

In the next definition, we use the operator $\bullet$. The statement $\pi_1 \bullet \pi_2$ denotes a plan of which π_1 is the first part and π_2 is the second, i.e. π_1 is the prefix of this plan. We need this operator because plans are defined to have a list structure (see definition 2).

Definition 9. *(rule application)* Let $\rho : \pi_h \mid \psi \rightsquigarrow \pi_b \in \mathsf{Rule}$.

$$\frac{\sigma \models \psi}{\langle \pi_h \bullet \pi, \sigma \rangle \rightarrow_{apply(\rho)} \langle \pi_b \bullet \pi, \sigma \rangle}$$

Meta-level The meta-level transition system (Trans_m) is defined by the rules below, specifying which transitions from one meta-configuration to another are possible. As for the object-level transition system, the transitions are labelled to denote the kind of transition.

An *execute* meta-action is used to execute a basic action. It can thus only be executed in a mental state, if the first element of the plan in that mental state is a basic action. As in the object-level transition system, the basic action a must be executable and the result of executing a on belief base σ is defined using the function $\mathcal{T}$. After executing the meta-action *execute*, the meta-program is empty and the basic action is gone from the plan. Furthermore, the belief base is changed as defined through $\mathcal{T}$.

Definition 10. *(action execution)* Let $a \in \mathsf{BasicAction}$.

$$\frac{\mathcal{T}(a, \sigma) = \sigma'}{\langle execute, (a; \pi, \sigma) \rangle \rightarrow_{execute} \langle E, (\pi, \sigma') \rangle}$$

A meta-action $apply(\rho)$ is used to specify that PR rule ρ should be applied. It can be executed in a mental state if ρ is applicable in that mental state. The execution of the meta-action in a mental state results in the plan of that mental state being changed as specified by the rule.

Definition 11. *(rule application)* Let $\rho : \pi_h \mid \psi \rightsquigarrow \pi_b \in \mathsf{Rule}$.

$$\frac{\sigma \models \psi}{\langle apply(\rho), (\pi_h \bullet \pi, \sigma) \rangle \rightarrow_{apply(\rho)} \langle E, (\pi_b \bullet \pi, \sigma) \rangle}$$

In order to define the transition rule for the `while` construct, we first need to specify the semantics of boolean expressions $Bexp$.

Definition 12. *(semantics of boolean expressions)* We assume a function $\mathcal{W}$ of type $Bexp \rightarrow (S \rightarrow W)$ yielding the semantics of boolean expressions, where W is the set of truth values $\{tt, ff\}$ with typical formula β.

The transition for the `while` construct is then defined in a standard way below. The transition is labeled with *idle*, to denote that this is a transition that does not have a counterpart in the object-level transition system.

Definition 13. *(while)*

$$\frac{\mathcal{W}(b)(s)}{\langle \texttt{while } b \texttt{ do } P \texttt{ od}, s\rangle \rightarrow_{idle} \langle P; \texttt{while } b \texttt{ do } P \texttt{ od}, s\rangle}$$

$$\frac{\neg\mathcal{W}(b)(s)}{\langle \texttt{while } b \texttt{ do } P \texttt{ od}, s\rangle \rightarrow_{idle} \langle E, s\rangle}$$

The transitions for sequential composition and non-deterministic choice are defined as follows in a standard way. The variable x is used to pass on the type of transition through the derivation.

Definition 14. *(sequential composition)* Let $x \in \{execute, apply(\rho), idle \mid \rho \in \mathsf{Rule}\}$.

$$\frac{\langle P_1, s\rangle \rightarrow_x \langle P_1', s'\rangle}{\langle P_1; P_2, s\rangle \rightarrow_x \langle P_1'; P_2, s'\rangle}$$

Definition 15. *(non-deterministic choice)* Let $x \in \{execute, apply(\rho), idle \mid \rho \in \mathsf{Rule}\}$.

$$\frac{\langle P_1, s\rangle \rightarrow_x \langle P_1', s'\rangle}{\langle P_1 + P_2, s\rangle \rightarrow_x \langle P_1', s'\rangle} \qquad \frac{\langle P_2, s\rangle \rightarrow_x \langle P_2', s'\rangle}{\langle P_1 + P_2, s\rangle \rightarrow_x \langle P_2', s'\rangle}$$

3.2 Operational Semantics

Using the transition systems defined in the previous section, transitions can be derived for 3APL and for the meta-programs. Individual transitions can be put in sequel, yielding so called *computation sequences*. In the following definitions, we define computation sequences and we specify the functions yielding these sequences, for the object- and meta-level transition systems. We also define the function κ, yielding the last element of a computation sequence if this sequence is finite and the special state $\perp$ otherwise. These functions will be used to define the operational semantics.

Definition 16. *(computation sequences)* The sets S^+ and S^∞ of respectively finite and infinite computation sequences are defined as follows:

$$\begin{aligned} S^+ &= \{s_1, \ldots, s_i, \ldots, s_n \mid s_i \in S, 1 \leq i \leq n, n \in \mathbb{N}\},\\ S^\infty &= \{s_1, \ldots, s_i, \ldots \quad\ \mid s_i \in S, i \in \mathbb{N}\}. \end{aligned}$$

Let $S_\perp = S \cup \{\perp\}$ and $\delta \in S^+ \cup S^\infty$. The function $\kappa : (S^+ \cup S^\infty) \rightarrow S_\perp$ is defined by:

$$\kappa(\delta) = \begin{cases} \text{last element of } \delta & \text{if } \delta \in S^+,\\ \perp & \text{otherwise.} \end{cases}$$

The function κ is extended to handle sets of computation sequences as follows:

$$\kappa(\{\delta_i \mid i \in I\}) = \{\kappa(\delta_i) \mid i \in I\}.$$

Definition 17. *(functions for calculating computation sequences)* The functions $\mathcal{C}_o$ and $\mathcal{C}_m$ are respectively of type $S \to \wp(S^+ \cup S^\infty)$ and $Prog \to (S \to \wp(S^+ \cup S^\infty))$.

$$\begin{aligned}
\mathcal{C}_o(s) \quad &= \{s_1, \ldots, s_n \quad \in \wp(S^+) \mid s \to_{t_1} s_1 \to_{t_2} \ldots \to_{t_n} \langle E, \sigma_n \rangle \\
&\qquad \text{is a finite sequence of transitions in } \mathsf{Trans}_o\} \cup \\
&\quad \{s_1, \ldots, s_i, \ldots \in \wp(S^\infty) \mid s \to_{t_1} s_1 \to_{t_2} \ldots \to_{t_i} s_i \to_{t_{i+1}} \ldots \\
&\qquad \text{is an infinite sequence of transitions in } \mathsf{Trans}_o\} \\
\mathcal{C}_m(P)(s) = \{s_1, \ldots, s_n \quad &\in \wp(S^+) \mid \langle P, s \rangle \to_{x_1} \langle P_1, s_1 \rangle \to_{x_2} \ldots \to_{x_n} \langle E, s_n \rangle \\
&\qquad \text{is a finite sequence of transitions in } \mathsf{Trans}_m\} \cup \\
&\quad \{s_1, \ldots, s_i, \ldots \in \wp(S^\infty) \mid \langle P, s \rangle \to_{x_1} \langle P_1, s_1 \rangle \to_{x_2} \ldots \\
&\qquad\qquad \to_{x_i} \langle P_i, s_i \rangle \to_{x_{i+1}} \ldots \\
&\qquad \text{is an infinite sequence of transitions in } \mathsf{Trans}_m\}
\end{aligned}$$

Note that both $\mathcal{C}_o$ and $\mathcal{C}_m$ return sequences of mental states. $\mathcal{C}_o$ just returns the mental states comprising the sequences of transitions derived in Trans_o, whereas $\mathcal{C}_m$ removes the meta-program component of the meta-configurations of the transition sequences derived in Trans_m. The reason for defining these functions in this way is, that we want to prove equivalence of the object- and meta-level transition systems: both yield the same transition sequences with respect to the mental states (or that is for a certain meta-program, see section 4). Also note that for $\mathcal{C}_o$ as well as for $\mathcal{C}_m$, we only take into account infinite sequences and successfully terminating sequences, i.e. those sequences ending in a mental state or meta-configuration with an empty plan or meta-program respectively.

The operational semantics of object- and meta-level programs are functions $\mathcal{O}_o$ and $\mathcal{O}_m$, yielding, for each mental state s and possibly meta-program P, a set of mental states corresponding to the final states reachable through executing the plan of s or executing the meta-program P respectively. If there is an infinite execution path, the set of mental states will contain the element $\bot$.

Definition 18. *(operational semantics)* Let $s \in S$. The functions $\mathcal{O}_o$ and $\mathcal{O}_m$ are respectively of type $S_\bot \to \wp(S_\bot)$ and $Prog \to (S_\bot \to \wp(S_\bot))$.

$$\begin{aligned}
\mathcal{O}_o(s) \quad &= \kappa(\mathcal{C}_o(s)) \\
\mathcal{O}_m(P)(s) &= \kappa(\mathcal{C}_m(P)(s)) \\
\mathcal{O}_o(\bot) \quad &= \mathcal{O}_m(P)(\bot) \quad = \{\bot\}
\end{aligned}$$

Note that the operational semantic functions can take any state $s \in S_\bot$, including $\bot$, as input. This will turn out to be necessary for giving the equivalence result of section 6.

4 Equivalence of $\mathcal{O}_o$ and $\mathcal{O}_m$

In the previous section, we have defined the operational semantics for 3APL and for meta-programs. Using the meta-language, one can write various 3APL interpreters. Here we will consider an interpreter of which the operational semantics will prove to be equivalent to the object-level operational semantics of 3APL. This interpreter for agent $\mathcal{A}$ is defined by the following meta-program.

Definition 19. *(interpreter)* Let $\bigcup_{i=1}^{n} \rho_i = \mathsf{Rule}$, $s \in S$ and let $notEmptyPlan \in Bexp$ be a boolean expression such that $\mathcal{W}(notEmptyPlan)(s) = tt$ if the plan component of s is not equal to E and $\mathcal{W}(notEmptyPlan)(s) = ff$ otherwise. Then the interpreter can be defined as follows.

$$\texttt{while}\ notEmptyPlan\ \texttt{do}\ (execute\ +\ apply(\rho_1)\ +\ \ldots\ +\ apply(\rho_n))\ \texttt{od}$$

In the sequel, we will use the keyword interpreter to abbreviate this meta-program.

This interpreter thus iterates the execution of a non-deterministic choice between all basic meta-actions, until the plan component of the mental state is empty. Intuitively, if there is a possibility for the interpreter to execute some meta-action in mental state s, resulting in a changed state s', it is also possible to go from s to s' in an object-level execution through a corresponding object-level transition. At each iteration, an executable meta-action is non-deterministically chosen for execution. The interpreter thus as it were, non-deterministically chooses a path through the object-level transition tree. The possible transitions defined by this interpreter correspond to the possible transitions in the object-level transition system and therefore the object-level operational semantics is equivalent to the meta-level operational semantics of this meta-program[10].

Theorem 1. *($\mathcal{O}_o = \mathcal{O}_m$(interpreter))*

$$\forall s \in S : \mathcal{O}_o(s) = \mathcal{O}_m(\mathsf{interpreter})(s)$$

Proof. We prove a weak bisimulation (see [17]).

Note that it is easy to show that $\mathcal{O}_o = \mathcal{O}_m(P)$ does not hold for all meta-programs P.

5 Denotational Semantics

In this section, we will define the denotational semantics of meta-programs. The method used is the fixed point approach as can be found in Stoy [15]. The semantics greatly resembles the one in De Bakker ([4], Chapter 7) to which we refer for a detailed explanation of the subject.

A denotational semantics for a programming language in general, is, like an operational semantics, a function taking a statement P and a state s and yielding a state (or set of states in case of a non-deterministic language) resulting from executing P in s. The denotational semantics for meta-programs is thus, like the operational semantics of definition 18, a function taking a meta-program P and mental state s and yielding the set of mental states resulting from executing

[10] The result only holds if PR rules of the form $E \mid \psi \rightsquigarrow \pi_b$ are excluded from the set of rules under consideration, as was specified in definition 4. A relaxation of this condition would call for a slightly different interpreter to yield the equivalence result. For reasons of space and clarity, we will however not discuss this possibility here.

P in s, i.e. a function of type $Prog \rightarrow (S_\perp \rightarrow \wp(S_\perp))$[11]. Contrary however to an operational semantic function, a denotational semantic function is not defined using the concept of computation sequences and, in contrast with most operational semantics, it *is* defined compositionally [16,10,4].

5.1 Preliminaries

In order to define the denotational semantics of meta-programs, we need some mathematical machinery. Most importantly, the domains used in defining the semantics of meta-programs are designed as so-called complete partial orders (CPO's). A CPO is a set with an ordering on its elements with certain characteristics. We assume the reader is familiar with the notions of partially ordered sets, least upper bounds and chains, in terms of which the concept of a CPO is defined. For a rigorous treatment of the subject, we refer to De Bakker [4].

Definition 20. *(CPO)* A complete partially ordered set is a set C with a partial order $\sqsubseteq$ which satisfies the following requirements:

1. there is a least element with respect to $\sqsubseteq$, i.e. an element $\perp \in C$ such that $\forall c \in C : \perp \sqsubseteq c$,
2. each chain $\langle c_i \rangle_{i=0}^{\infty}$ in C has a least upper bound $(\bigsqcup_{i=0}^{\infty} c_i) \in C$.

The semantics of meta-programs will be defined using the notion of the least fixed point of a function on a CPO.

Definition 21. *(least fixed point)* Let $(C, \sqsubseteq)$ a CPO, $f : C \rightarrow C$ and let $x \in C$.

- x is a fixed point of f iff $f(x) = x$
- x is a least fixed point of f iff x is a fixed point of f and for each fixed point y of f: $x \sqsubseteq y$

The least fixed point of a function f is denoted by μf.

Finally, we will need the following definition and fact.

Definition 22. *(continuity)* Let $(C_1, \sqsubseteq_1), (C_2, \sqsubseteq_2)$ be CPO's. Then a function $f : C_1 \rightarrow C_2$ is continuous iff for each chain $\langle c_i \rangle_{i=0}^{\infty}$ in C_1, the following holds:

$$f(\bigsqcup_{i=0}^{\infty} c_i) = \bigsqcup_{i=0}^{\infty} f(c_i).$$

Fact 1. *(fixed point theorem)* Let C be a CPO and let $f : C \rightarrow C$. If f is continuous, then the least fixed point μf exists and equals $\bigsqcup_{i=0}^{\infty} f^i(\perp)$, where $f^0(\perp) = \perp$ and $f^{i+1}(\perp) = f(f^i(\perp))$.

For a proof, see for example De Bakker [4].

[11] The type of the denotational semantic function is actually slightly different as will become clear in the sequel, but that is not important for the current discussion.

5.2 Definition

We will now show how the domains used in defining the semantics of meta-programs are designed as CPO's. The reason for designing these as CPO's will become clear in the sequel.

Definition 23. *(domains of interpretation)* Let W be the set of truth values of definition 12 and let S be the set of possible mental states of definition 3. Then the sets $W_\perp$ and $S_\perp$ are defined as CPO's as follows:

$$\begin{array}{ll} W_\perp = W \cup \{\perp_{W_\perp}\} & \text{CPO by } \beta_1 \sqsubseteq \beta_2 \text{ iff } \beta_1 = \perp_{W_\perp} \text{ or } \beta_1 = \beta_2, \\ S_\perp \ = S \ \cup \{\perp\} & \text{CPO analogously.} \end{array}$$

Note that we use $\perp$ to denote the bottom element of $S_\perp$ and that we use $\perp_C$ for the bottom element of any other set C. As the set of mental states is extended with a bottom element, we extend the semantics of boolean expressions of definition 12 to a strict function, i.e. yielding $\perp_{W_\perp}$ for an input state $\perp$.

In the definition of the denotational semantics, we will use an if-then-else function as defined below.

Definition 24. *(if-then-else)* Let C be a CPO, $c_1, c_2, \perp_C \in C$ and $\beta \in W_\perp$. Then the if-then-else function of type $W_\perp \to C$ is defined as follows.

$$\texttt{if } \beta \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ fi} = \begin{cases} c_1 & \text{if } \beta = tt \\ c_2 & \text{if } \beta = \mathit{ff} \\ \perp_C & \text{if } \beta = \perp_{W_\perp} \end{cases}$$

Because our meta-language is non-deterministic, the denotational semantics is not a function from states to states, but a function from states to *sets of states*. These resulting sets of states can be finite or infinite. In case of bounded non-determinism[12], these infinite sets of states have $\perp$ as one of their members. This property may be explained by viewing the execution of a program as a tree of computations and then using König's lemma which tells us that a finitely-branching tree with infinitely many nodes has at least one infinite path (see [4]). The meta-language is indeed bounded non-deterministic[13] and the result of executing a meta-program P in some state, is thus either a finite set of states or an infinite set of states containing $\perp$. We therefore specify the following domain as the result domain of the denotational semantic function instead of $\wp(S_\perp)$.

Definition 25. *(T)* The set T with typical element τ is defined as follows: $T = \{\tau \in \wp(S_\perp) \mid \tau \text{ finite or } \perp \in \tau\}$.

The advantage of using T instead of $\wp(S_\perp)$ as the result domain, is that T can nicely be designed as a CPO with the following ordering [5].

---

[12] Bounded non-determinism means that at any state during computation, the number of possible next states is finite.

[13] Only a finite number of rule applications and action executions are possible in any state.

Definition 26. *(Egli-Milner ordering)* Let $\tau_1, \tau_2 \in T$. $\tau_1 \sqsubseteq \tau_2$ holds iff either $\bot \in \tau_1$ and $\tau_1 \setminus \{\bot\} \subseteq \tau_2$, or $\bot \notin \tau_1$ and $\tau_1 = \tau_2$. Under this ordering, the set $\{\bot\}$ is $\bot_T$.

We are now ready to give the denotational semantics of meta-programs. We will first give the definition and then justify and explain it.

Definition 27. *(denotational semantics of meta-programs)* Let $\phi_1, \phi_2 : S_\bot \to T$. Then we define the following functions.

$$\begin{array}{ll} \hat{\phi} & : T \to T = \lambda\tau \cdot \bigcup_{s \in \tau} \phi(s) \\ \phi_1 \circ \phi_2 & : S_\bot \to T = \lambda s \cdot \hat{\phi_1}(\phi_2(s)) \end{array}$$

Let $(\pi, \sigma) \in S$. The denotational semantics of meta-programs $\mathcal{M} : Prog \to (S_\bot \to T)$ is then defined as follows.

$$\begin{array}{lll} \mathcal{M}[\![execute]\!](\pi,\sigma) & = \begin{cases} \{(\pi', \sigma')\} & \text{if } \pi = a;\pi' \\ & \text{with } a \in \mathsf{BasicAction} \text{ and } \mathcal{T}(a,\sigma) = \sigma' \\ \emptyset & \text{otherwise} \end{cases} \\ \mathcal{M}[\![execute]\!]\ \bot & = \bot_T \\ \mathcal{M}[\![apply(\rho)]\!](\pi,\sigma) & = \begin{cases} \{(\pi_b \circ \pi', \sigma)\} & \text{if } \sigma \models \psi \text{ and } \pi = \pi_h \circ \pi' \\ & \text{with } \rho : \pi_h \mid \psi \rightsquigarrow \pi_b \in \mathsf{Rule} \\ \emptyset & \text{otherwise} \end{cases} \\ \mathcal{M}[\![apply(\rho)]\!]\ \bot & = \bot_T \\ \mathcal{M}[\![\texttt{while}\ b\ \texttt{do}\ P\ \texttt{od}]\!] & = \mu\Phi \\ \mathcal{M}[\![P_1; P_2]\!] & = \mathcal{M}[\![P_2]\!] \circ \mathcal{M}[\![P_1]\!] \\ \mathcal{M}[\![P_1 + P_2]\!] & = \mathcal{M}[\![P_1]\!] \cup \mathcal{M}[\![P_2]\!] \end{array}$$

The function $\Phi : (S_\bot \to T) \to (S_\bot \to T)$ used above is defined as $\lambda\phi \cdot \lambda s \cdot \texttt{if}\ \mathcal{W}(b)(s)\ \texttt{then}\ \hat{\phi}(\mathcal{M}[\![P]\!](s))\ \texttt{else}\ \{s\}\ \texttt{fi}$, using definition 24.

Meta-actions The semantics of meta-actions is straight forward. The result of executing an *execute* meta-action in some mental state s, is a set containing the mental state resulting from executing the basic action of the plan of s. The result is empty if there is no basic action on the plan to execute. The result of executing an $apply(\rho)$ meta-action in state s, is a set containing the mental state resulting from applying ρ in s. If ρ is not applicable, the result is the empty set.

While The semantics of the `while` construct is more involved, but we will only briefly comment on it. For a detailed treatment, we again refer to De Bakker [4].

What we want to do, is define a function specifying the semantics of the `while` construct $\mathcal{M}[\![\texttt{while}\ b\ \texttt{do}\ P\ \texttt{od}]\!]$, the type of which should be $S_\bot \to T$, in accordance with the type of $\mathcal{M}$. The function should be defined compositionally, i.e. it can only use the semantics of the guard and of the body of the `while`. This is required for $\mathcal{M}$ to be well-defined.

The requirement of compositionality is satisfied, as the semantics is defined to be the least fixed point of the operator Φ, which is defined in terms of the semantics of the guard and body of the `while`.

The least fixed point of an operator does not always exist. By the fixed point theorem however (fact 1), we now that if the operator is continuous (definition 22), the least fixed point *does* exist and is obtainable within ω steps. By proving that Φ is continuous, we can thus conclude that $\mu\Phi$ exists and therefore that $\mathcal{M}$ is well-defined.

Theorem 2. *(continuity of Φ)* The function Φ as given in definition 27 is continuous.

Proof. For a proof, we refer to [17]. The proof is analogous to continuity proofs given in [4].

Note that in the definition of Φ, the function ϕ is of type $S_\perp \to T$ and $\mathcal{M}[\![P]\!](s) \in T$. This ϕ can thus not be applied directly to this set of states in T, but it must be extended using the $\hat{}$ operator to be of type $T \to T$.

Sequential Composition and Non-deterministic Choice The semantics of the sequential composition and non-deterministic choice operator is as one would expect.

6 Equivalence of Meta-level Operational and Denotational Semantics

In this section, we will state that the operational semantics for meta-programs is equal to the denotational semantics for meta-programs and we will relate this to the equivalence result of section 4. We will furthermore discuss the issue of defining a denotational semantics for object-level 3APL.

6.1 Equivalence Theorem

Theorem 3. *($\mathcal{O}_m = \mathcal{M}$)* Let $\mathcal{O}_m : Prog \to (S_\perp \to \wp(S_\perp))$ be the operational semantics of meta-programs (definition 18) and let $\mathcal{M} : Prog \to (S_\perp \to T)$ be the denotational semantics of meta-programs (definition 27). Then, the following equivalence holds for all meta-programs $P \in Prog$ and all mental states $s \in S_\perp$.

$$\mathcal{O}_m(P)(s) = \mathcal{M}(P)(s)$$

Proof. For a proof, we refer to [17]. The proof is constructed using techniques from Kuiper [9].

In section 4, we stated that the object-level operational semantics of 3APL is equal to the meta-level operational semantics of the interpreter we specified in definition 19. Above, we then stated that it holds for any meta-program that its operational semantics is equal to its denotational semantics. This holds in particular for the interpreter of definition 19, i.e. we have the following corollary.

Corollary 1. *($\mathcal{O}_o = \mathcal{M}(\mathsf{interpreter})$)* From theorems 1 and 3 we can conclude that the following holds.

$$\mathcal{O}_o = \mathcal{M}(\mathsf{interpreter})$$

6.2 Denotational Semantics of Object-level 3APL

Corollary 1 states an equivalence between a denotational semantics and the object-level operational semantics for 3APL. The question is, whether this denotational semantics can be called a denotational semantics for object-level 3APL.

A denotational semantics for object-level 3APL should be a function taking a plan and a belief base and returning the result of executing the plan on this belief base, i.e. a function of type $\Pi \to (\Sigma_\perp \to \wp(\Sigma_\perp))$ or equivalently[14], of type $(\Pi \times \Sigma) \to \wp(\Sigma_\perp)$. The type of $\mathcal{M}(\mathsf{interpreter})$, i.e. $S_\perp \to \wp(S_\perp)$[15], does not match the desired type. This could however be remedied by defining the following function.

Definition 28. *($\mathcal{N}$)* Let *snd* be a function yielding the second element, i.e. the belief base, of a mental state in S and yielding $\perp_{\Sigma_\perp}$ for input $\perp$. This function is extended to handle sets of mental states through the function $\hat{}$, as was done in definition 27. Then $\mathcal{N} : S_\perp \to \wp(\Sigma_\perp)$ is defined as follows.

$$\mathcal{N} = \lambda s \cdot \widehat{snd}(\mathcal{M}[\![\mathsf{interpreter}]\!](s))$$

Disregarding a $\perp$ input, the function $\mathcal{N}$ is of the desired type $(\Pi \times \Sigma) \to \wp(\Sigma_\perp)$. The question now is, whether it is legitimate to characterize the function $\mathcal{N}$ as being a denotational semantics for 3APL. The answer is no, because a denotational semantic function should be compositional in its program argument, which in this case is Π. This is obviously *not* the case for the function $\mathcal{N}$ and therefore this function is not a denotational semantics for 3APL.

So, it seems that the specification of the denotational semantics for meta-programs cannot be used to define a denotational semantics for object-level 3APL. The difficulty of specifying a compositional semantic function is due to the nature of the PR rules: these rules can transform not just atomic statements, but any sequence of statements. The semantics of an atomic statement can thus depend on the statements around it. We will illustrate the problem using an example.

$$\begin{aligned} a \quad &\rightsquigarrow b \\ b; c &\rightsquigarrow d \\ c \quad &\rightsquigarrow e \end{aligned}$$

Now the question is, how we can define the semantics of $a; c$? Can it be defined in terms of the semantics of a and c? The semantics of a would have to be something involving the semantics of b and the semantics of c something with

[14] For the sake of argument, we for the moment disregard a $\perp_{\Sigma_\perp}$ input.

[15] $\mathcal{M}(\mathsf{interpreter})$ is actually defined to be of type $S_\perp \to T$, but $T \subset \wp(S_\perp)$, so we may extend the result type to $\wp(S_\perp)$.

the semantics of e, taking into account the PR rules given above. The semantics of $a; c$ should however also be defined in terms of the semantics of d, because of the second PR rule: $a; c$ can be rewritten to $b; c$, which can be rewritten to d. Moreover, if b is not a basic action, the third rule cannot be applied and the semantics of e would be irrelevant. So, although we do not have a formal proof, it seems that the semantics of the sequential composition operator[16] of a 3APL plan or program cannot be defined using only the semantics of the parts of which the program is composed.

Another way to look at this issue is the following. In a regular procedural program, computation can be defined using the concept of a program counter. This counter indicates the location in the code, of the statement that is to be executed next or the procedure that is to be called next. If a procedure is called, the program counter jumps to the body of this procedure. Computation of a 3APL program cannot be defined using such a counter. Consider for example the PR rules defined above and assume an initial plan $a; c$. Initially, the program counter would have to be at the start of this initial plan. Then, the first PR rule is "called" and the counter jumps to b, i.e. the body of the first rule. According to the semantics of 3APL, it should be possible to get to the body of the second PR rule, as the statement being executed is $b; c$. There is however no reason for the program counter to jump from the body of the first rule to the body of the second rule.

7 Related Work and Conclusion

The concept of a meta-language for programming 3APL interpreters was first considered by Hindriks [7]. Our meta-language is similar to, but simpler than Hindriks' language. The main difference is that Hindriks includes constructs for explicit selection of a PR rule from a set of applicable ones. These constructs were not needed in this paper. Dastani defines a meta-language for 3APL in [3]. This language is similar to, but more extensive than Hindriks' language. Dastani's main contribution is the definition of constructs for explicit planning. Using these constructs, the possible outcomes of a certain sequence of rule applications and action executions can be calculated in advance, thereby providing the possibility to choose the most beneficial sequence. Contrary to our paper, these papers do not discuss the relation between object-level and meta-level semantics, nor do they give a denotational semantics for the meta-language.

Concluding, we have proven equivalence of an operational and denotational semantics for a 3APL meta-language. We furthermore related this 3APL meta-language to object-level 3APL by proving equivalence between the semantics of a specific interpreter and object-level 3APL. Although these results were obtained for a simplified 3APL language, we conjecture that it will not be fundamentally more difficult to obtain similar results for full first order 3APL[17].

[16] or actually of the plan concatenation operator $\bullet$

[17] The requirement of bounded non-determinism will in particular not be violated.

As argued in the introduction, studying interpreter languages of agent programming languages is important. In the context of 3APL and PR rules, it is especially interesting to investigate the possibility of defining a denotational or compositional semantics, for such a compositional semantics could serve as the basis for a (compositional) proof system. It seems, considering the investigations as described in this paper, that it will however be very difficult if not impossible to define a denotational semantics for object-level 3APL. As it *is* possible to define a denotational semantics for the meta-language, an important issue for future research will be to investigate the possibility and usefulness of defining a proof system for the meta-language, using this to prove properties of 3APL agents.

References

1. M. E. Bratman. *Intention, plans, and practical reason.* Harvard University Press, Massachusetts, 1987.
2. P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.
3. M. Dastani, F. S. de Boer, F. Dignum, and J.-J. Ch. Meyer. Programming agent deliberation – an approach illustrated using the 3apl language. In *Proceedings of the second international joint conference on autonomous agents and multiagent systems (AAMAS'03)*, pages 97–104, Melbourne, 2003.
4. J. de Bakker. *Mathematical Theory of Program Correctness.* Series in Computer Science. Prentice-Hall International, London, 1980.
5. H. Egli. A mathematical model for nondeterministic computations. Technical report, ETH, Zürich, 1975.
6. G. d. Giacomo, Y. Lespérance, and H. Levesque. *ConGolog*, a Concurrent Programming Language Based on the Situation Calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.
7. K. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. C. Meyer. Control structures of rule-based agent languages. In J. Müller, M. P. Singh, and A. S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 381–396. Springer-Verlag: Heidelberg, Germany, 1999.
8. K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming in 3APL. *Int. J. of Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.
9. R. Kuiper. An operational semantics for bounded nondeterminism equivalent to a denotational one. In J. W. de Bakker and J. C. van Vliet, editors, *Proceedings of the International Symposium on Algorithmic Languages*, pages 373–398. North-Holland, 1981.
10. P. D. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 575–631. Elsevier, Amsterdam, 1990.
11. G. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, Computer Science Department, 1981.
12. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. van der Velde and J. Perram, editors, *Agents Breaking Away (LNAI 1038)*, pages 42–55. Springer-Verlag, 1996.

13. A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484. Morgan Kaufmann, 1991.
14. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.
15. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, 1977.
16. R. Tennent. *Semantics of Programming Languages*. Series in Computer Science. Prentice-Hall International, London, 1991.
17. M. B. van Riemsdijk, F. S. de Boer, and J.-J. Ch. Meyer. Semantics of plan revision in intelligent agents. Technical report, Utrecht University, Institute of Information and Computing Sciences, 2003. UU-CS-2004-002.
18. M. B. van Riemsdijk, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming in Dribble: from beliefs to goals using plans. In *Proceedings of the second international joint conference on autonomous agents and multiagent systems (AAMAS'03)*, pages 393–400, Melbourne, 2003.
19. M. Wooldridge. Agent-based software engineering. *IEEE Proceedings Software Engineering*, 144(1):26–37, 1997.
20. M. Wooldridge and P. Ciancarini. Agent-Oriented Software Engineering: The State of the Art. In P. Ciancarini and M. Wooldridge, editors, *First Int. Workshop on Agent-Oriented Software Engineering*, volume 1957, pages 1–28. Springer-Verlag, Berlin, 2000.