Available online at www.sciencedirect.com

SCIENCE DIRECT®

Theoretical Computer Science 351 (2006) 240–257

Theoretical Computer Science

www.elsevier.com/locate/tcs

Semantics of plan revision in intelligent agents

M. Birna van Riemsdijk[a,*], John-Jules Ch. Meyer[a], Frank S. de Boer[a,b,c]

[a]ICS, Utrecht University, The Netherlands
[b]CWI, Amsterdam, The Netherlands
[c]LIACS, Leiden University, The Netherlands

Abstract

In this paper, we give an operational and denotational semantics for a meta-language of the 3APL agent programming language. With this meta-language, various 3APL interpreters can be programmed. We prove equivalence of the operational and denotational semantics. Furthermore, we give an operational semantics for object-level 3APL. Using this semantics, we relate the 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.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Agent programming language; Structural operational semantics; Denotational semantics

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*[1] [2,13,3]. 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

*Corresponding author.
E-mail addresses: birna@cs.uu.nl (M.B. van Riemsdijk), jj@cs.uu.nl (J.-J.Ch. Meyer), frb@cwi.nl (F.S. de Boer).

[1] 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.

0304-3975/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2005.09.070

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 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) postconditions 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 3*APL*[2] [7]. The dynamic parts of a 3APL agent thus consist of a set of beliefs, a plan[3] and a set of goals.[4] A plan can consist of sequences of so-called basic actions and abstract plans. Basic actions change the beliefs[5] 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 [1] 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 [12,16,6].

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 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 [4]).

Secondly, as pointed out by Hindriks [8], 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

[2] 3APL is to be pronounced as "triple-a-p-l".

[3] In the original version this was a set of plans.

[4] The addition of goals was a recent extension [14].

[5] A change in the environment is a possible "side effect" of the execution of a basic action.

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 [7]. 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 [7] 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 $\mathsf{BasicAction}$ with typical element a is given, together with a set $\mathsf{AbstractPlan}$. 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 $c\,;\pi \in \Pi$.[6]

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, $\mathsf{BasicAction}$, $\mathsf{AbstractPlan}$ and Rule are finite sets of basic actions, abstract plans and PR rules, respectively, and $\mathcal{T} : (\mathsf{BasicAction} \times \Sigma) \to \Sigma$ is a partial function, defining belief update through action execution.

In the following, when referring to agent $\mathcal{A}$, we will assume this agent to have a set of basic actions $\mathsf{BasicAction}$, a set of abstract plans $\mathsf{AbstractPlan}$, 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 3.2, we

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

will define the transition system with which we will define the operational semantics of our meta-programs. We will do this using 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 [7], 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 [7] (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. Object-level transition system

The transition systems defined in this and the following section assume 3APL agent $\mathcal{A}$. The object-level transition system ($\mathsf{Trans_o}$) is defined by the rules given below. The transitions are labelled 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 $\rho : \pi_h \mid \psi \rightsquigarrow \pi_b \in \mathsf{Rule}$

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

3.2. Meta-level transition system

The meta-level transition system ($\mathsf{Trans_m}$) 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 $a \in \mathsf{BasicAction}$

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

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 $\rho : \pi_h \mid \psi \rightsquigarrow \pi_b \in \mathsf{Rule}$.

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

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 $Bexp \to (S \to W)$ 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 labelled with *idle*, to denote that this is a transition that does not have a counterpart in the object-level transition system.

Definition 13 (*while*).

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

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 $x \in \{execute, apply(\rho), idle \mid \rho \in \mathsf{Rule}\}$.

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

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

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

3.3. 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:

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

Let $S_\perp = S \cup \{\perp\}$ and $\delta \in S^+ \cup S^\infty$. The function $\kappa : (S^+ \cup S^\infty) \to S_\perp$ 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 $\mathcal{C}_\text{o}$ and $\mathcal{C}_\text{m}$ are, respectively, of type $S \to \wp(S^+ \cup S^\infty)$ and $Prog \to (S \to \wp(S^+ \cup S^\infty))$.

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

Note that both $\mathcal{C}_\text{o}$ and $\mathcal{C}_\text{m}$ return sequences of mental states. $\mathcal{C}_\text{o}$ just returns the mental states comprising the sequences of transitions derived in Trans_o, whereas $\mathcal{C}_\text{m}$ 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 $\mathcal{C}_\text{o}$ as well as for $\mathcal{C}_\text{m}$, 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 $\mathcal{O}_\text{o}$ and $\mathcal{O}_\text{m}$, 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 $\perp$.

Definition 18 (*operational semantics*). Let $s \in S$. The functions $\mathcal{O}_\text{o}$ and $\mathcal{O}_\text{m}$ are, respectively, of type $S_\perp \to \wp(S_\perp)$ and $Prog \to (S_\perp \to \wp(S_\perp))$.

$$\begin{aligned} \mathcal{O}_\text{o}(s) &= \kappa(\mathcal{C}_\text{o}(s)), \\ \mathcal{O}_\text{m}(P)(s) &= \kappa(\mathcal{C}_\text{m}(P)(s)), \\ \mathcal{O}_\text{o}(\perp) &= \mathcal{O}_\text{m}(P)(\perp) = \{\perp\}. \end{aligned}$$

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

4. Equivalence of $\mathcal{O}_\text{o}$ and $\mathcal{O}_\text{m}$

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:

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

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.[7]

We prove the equivalence result by proving a weak bisimulation between $\mathsf{Trans_o}$ and $\mathsf{Trans_m}(\mathsf{interpreter})$, which are defined assuming agent $\mathcal{A}$ (see the text below Definition 5 and Section 3.1). From this, we can then prove that $\mathcal{O}_o$ and $\mathcal{O}_m(\mathsf{interpreter})$ are equivalent. In order to do this, we first state the following proposition. It follows immediately from the transition systems.

Proposition 1 (*object-level versus meta-level transitions*).

$$\begin{array}{rl} s \rightarrow_{execute} s' & \textit{is a transition in } \mathsf{Trans_o} \Leftrightarrow \\ \langle execute, s\rangle \rightarrow_{execute} \langle E, s'\rangle & \textit{is a transition in } \mathsf{Trans_m} \end{array}$$

$$\begin{array}{rl} s \rightarrow_{apply(\rho)} s' & \textit{is a transition in } \mathsf{Trans_o} \Leftrightarrow \\ \langle apply(\rho), s\rangle \rightarrow_{apply(\rho)} \langle E, s'\rangle & \textit{is a transition in } \mathsf{Trans_m} \end{array}$$

A weak bisimulation between two transition systems in general, is a relation between the systems such that the following holds: if a transition step can be derived in system one, it should be possible to derive a "similar" (sequence of) transition(s) in system two and if a transition step can be derived in system two, it should be possible to derive a "similar" (sequence of) transition(s) in system one. To explain what we mean by "similar" transitions, we need the notion of an idle transition. In a transition system, certain kinds of transitions can be labelled as an idle transition, for example transitions derived using the while rule (Definition 13). These transitions can be considered "implementation details" of a certain transition system and we do not want to take these into account when studying the relation between this and another transition system. A non-idle transition in system one now is similar to a sequence of transitions in system two if the following holds: this sequence of transitions in system two should consist of one non-idle transition and otherwise idle transitions, and the non-idle transition in this sequence should be similar to the transition in system one, i.e., the relevant elements of the configurations involved, should match.

In the context of our transition systems $\mathsf{Trans_o}$ and $\mathsf{Trans_m}$, we can now phrase the following bisimulation lemma.

Lemma 1 (*weak bisimulation*). *Let* $+^*$ *abbreviate* $(execute + apply(\rho_1) + \cdots + apply(\rho_n))$. *Let* $\mathsf{Trans_m}(P)$ *be the restriction of* $\mathsf{Trans_m}$ *to those transitions that are part of some sequence of transitions starting in initial meta-configuration* $\langle P, s_0\rangle$, *with* $s_0 \in S$ *an arbitrary mental state and let* $t \in \{execute, apply(\rho) \mid \rho \in \mathsf{Rule}\}$. *Then a weak bisimulation exists between* $\mathsf{Trans_o}$ *and* $\mathsf{Trans_m}(\mathsf{interpreter})$, *i.e., the following properties hold.*

$$\begin{array}{l} s \rightarrow_t s' \textit{ is a transition in } \mathsf{Trans_o} \Rightarrow_1 \\ \qquad \langle \mathsf{interpreter}, s\rangle \rightarrow_{idle} \langle +^*; \mathsf{interpreter}, s\rangle \rightarrow_t \langle \mathsf{interpreter}, s'\rangle \\ \qquad\qquad\qquad\qquad \textit{is a transition in } \mathsf{Trans_m}(\mathsf{interpreter}) \end{array}$$

[7] 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.

$$\langle +^*; \mathsf{interpreter}, s\rangle \rightarrow_t \langle \mathsf{interpreter}, s'\rangle \text{ is a transition in } \mathsf{Trans_m}(\mathsf{interpreter}) \Rightarrow_2 s \rightarrow_t s' \text{is a transition in } \mathsf{Trans_o}$$

Proof. ($\Rightarrow_1$) Assume $s \rightarrow_t s'$ is a transition in $\mathsf{Trans_o}$ for $t \in \{execute, apply(\rho) \mid \rho \in \mathsf{Rule}\}$. Using Proposition 1, the following then is a transition in $\mathsf{Trans_m}$:

$$\langle (execute + apply(\rho_1) + \cdots + apply(\rho_n)); \mathsf{interpreter}, s\rangle \rightarrow_t \langle \mathsf{interpreter}, s'\rangle. \quad (1)$$

Furthermore, by the assumption that $s \rightarrow_t s'$ is a transition in $\mathsf{Trans_o}$ and by the assumption that no reactive rules are contained in Rule (see introduction of Section 4), we know that the plan of s is not empty as both rule application and basic action execution require a non-empty plan. Now, using the fact that the plan of s is not empty, the following transition can be derived in $\mathsf{Trans_m}(\mathsf{interpreter})$:

$$\langle \mathsf{interpreter}, s\rangle \rightarrow_{idle} \langle (execute + apply(\rho_1) + \cdots + apply(\rho_n)); \mathsf{interpreter}, s\rangle. \quad (2)$$

The transitions (2) and (1) can be concatenated, yielding the desired result.

($\Rightarrow_2$) Assume $\langle +^*; \mathsf{interpreter}, s\rangle \rightarrow_t \langle \mathsf{interpreter}, s'\rangle$ is a transition in $\mathsf{Trans_m}(\mathsf{interpreter})$. Then, $\langle +^*, s\rangle \rightarrow_t \langle E, s'\rangle$ must be a transition in $\mathsf{Trans_m}$ (Definition 14). Therefore, by Proposition 1, we can conclude that $s \rightarrow_t s'$ is a transition in $\mathsf{Trans_o}$. □

We are now in a position to give the equivalence theorem of this section.

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

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

Proof. As equivalence of object-level and meta-level operational semantics holds for input state $\perp$ by Definition 18, we will only need to prove equivalence for input states $s \in S$. Proving this theorem amounts to showing the following: $s \in \mathcal{O}_o \Leftrightarrow s \in \mathcal{O}_m(\mathsf{interpreter})$.

($\Rightarrow$) Assume $s \in \mathcal{O}_o$. This means that a sequence of transitions $s_0 \rightarrow_{t_1} \cdots \rightarrow_{t_n} s$ must be derivable in $\mathsf{Trans_o}$. By repeated application of Lemma 1, we know that then there must also be a sequence of transitions in $\mathsf{Trans_m}(\mathsf{interpreter})$ of the following form:

$$\langle \mathsf{interpreter}, s_0\rangle \rightarrow_{idle} \cdots \rightarrow_{t_{n-1}} \langle \mathsf{interpreter}, s'\rangle \rightarrow_{idle} \langle +^*; \mathsf{interpreter}, s'\rangle \rightarrow_{t_n} \langle \mathsf{interpreter}, s\rangle. \quad (3)$$

As $s \in \mathcal{O}_o$, we know that there cannot be a transition $s \rightarrow_{t_{n+1}} s''$ for some mental state s'', i.e., it is not possible to execute an *execute* or *apply* meta-action in s. Therefore, we know that the only possible transition from $\langle \mathsf{interpreter}, s\rangle$ in (3) above, is $\cdots \rightarrow_{idle} \langle E, s\rangle$. From this, we have that $s \in \mathcal{O}_m(\mathsf{interpreter})$.

($\Leftarrow$) Assume that $s \in \mathcal{O}_m(\mathsf{interpreter})$. Then there must be a sequence of transitions in $\mathsf{Trans_m}(\mathsf{interpreter})$ of the form:

$$\langle \mathsf{interpreter}, s_0\rangle \rightarrow_{idle} \langle +^*; \mathsf{interpreter}, s_0\rangle \rightarrow_{t_1} \cdots \rightarrow_{t_{n-1}} \langle \mathsf{interpreter}, s'\rangle \rightarrow_{idle} \langle +^*; \mathsf{interpreter}, s'\rangle \rightarrow_{t_n} \langle \mathsf{interpreter}, s\rangle \rightarrow_{idle} \langle E, s\rangle.$$

From this, we can conclude by Lemma 1 that $s_0 \rightarrow_{t_1} \cdots \rightarrow_{t_{n-1}} s' \rightarrow_{t_n} s \not\rightarrow$ must be a sequence of transitions in $\mathsf{Trans_o}$. Therefore, it must be the case that $s \in \mathcal{O}_o$. □

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 [17]. The semantics greatly resembles the one in de Bakker [1, 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 P in s, i.e., a function of type $Prog \to (S_\perp \to \wp(S_\perp))$.[8] 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 [18,10,1].

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 (CPOs). A CPO is a set with an ordering on its elements with certain characteristics. This concept is defined in terms of the notions of partially ordered sets, least upper bounds and chains (see also de Bakker [1] for a rigorous treatment of the subject).

Definition 20 (*partially ordered set*). Let C be an arbitrary set. A partial order $\sqsubseteq$ on C is a subset of $C \times C$ which satisfies:
(1) $c \sqsubseteq c$ (reflexivity),
(2) if $c_1 \sqsubseteq c_2$ and $c_2 \sqsubseteq c_1$ then $c_1 = c_2$ (antisymmetry),
(3) if $c_1 \sqsubseteq c_2$ and $c_2 \sqsubseteq c_3$ then $c_1 \sqsubseteq c_3$ (transitivity).

In the sequel, we will be concerned not only with arbitrary sets with partial orderings, but also with sets of functions with an ordering. A partial ordering on a set of functions of type $C_1 \to C_2$ can be derived from the orderings on C_1 and C_2 as defined below.

Definition 21 (*partial ordering on functions*). Let $(C_1, \sqsubseteq_1)$ and $(C_2, \sqsubseteq_2)$ be two partially ordered sets. An ordering $\sqsubseteq$ on $C_1 \to C_2$ is defined as follows, where $f, g \in C_1 \to C_2$:

$$f \sqsubseteq g \Leftrightarrow \forall c \in C_1 : f(c) \sqsubseteq_2 g(c).$$

Definition 22 (*least upper bound*). Let $C' \subseteq C$. $z \in C$ is called the least upper bound of C' if:
(1) z is an upper bound: $\forall x \in C' : x \sqsubseteq z$,
(2) z is the *least* upper bound: $\forall y \in C : ((\forall x \in C' : x \sqsubseteq y) \Rightarrow z \sqsubseteq y)$.

The least upper bound of a set C' will be denoted by $\bigsqcup C'$.

Definition 23 (*least upper bound of a sequence*). The least upper bound of a sequence $\langle c_0, c_1, \ldots \rangle$ is denoted by $\bigsqcup_{i=0}^{\infty} c_i$ or by $\bigsqcup \langle c_i \rangle_{i=0}^{\infty}$ and is defined as follows, where "c in $\langle c_i \rangle_{i=0}^{\infty}$" means that c is an element of the sequence $\langle c_i \rangle_{i=0}^{\infty}$:

$$\bigsqcup \langle c_i \rangle_{i=0}^{\infty} = \bigsqcup \{c \mid c \text{ in } \langle c_i \rangle_{i=0}^{\infty}\}.$$

Definition 24 (*chains*). A chain on $(C, \sqsubseteq)$ is an infinite sequence $\langle c_i \rangle_{i=0}^{\infty}$ such that for $i \in \mathbb{N} : c_i \sqsubseteq c_{i+1}$.

Having defined partially ordered sets, least upper bounds and chains, we are now in a position to define complete partially ordered sets.

[8] 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.

Definition 25 (*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 $\bot \in C$ such that $\forall c \in C : \bot \sqsubseteq c$,
(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 26 (*least fixed point*). Let $(C, \sqsubseteq)$ a CPO, $f : C \to C$ 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 27 (*continuity*). Let $(C_1, \sqsubseteq_1)$, $(C_2, \sqsubseteq_2)$ be CPOs. Then a function $f : C_1 \to C_2$ is continuous iff for each chain $\langle c_i \rangle_{i=0}^{\infty}$ in C_1, the following holds:

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

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

For a proof, see, for example de Bakker [1].

5.2. Definition

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

Definition 28 (*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 $W_\bot$ and $S_\bot$ are defined as CPOs as follows:

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

Note that we use $\bot$ to denote the bottom element of $S_\bot$ and that we use $\bot_C$ 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 $\bot_{W_\bot}$ for an input state $\bot$.

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

Definition 29 (*if-then-else*). Let C be a CPO, $c_1, c_2, \bot_C \in C$ and $\beta \in W_\bot$. Then the if-then-else function of type $W_\bot \to C$ is defined as follows:

$$\texttt{if } \beta \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ fi} = \begin{cases} c_1 & \text{if } \beta = tt, \\ c_2 & \text{if } \beta = ff, \\ \bot_C & \text{if } \beta = \bot_{W_\bot}. \end{cases}$$

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,[9] these infinite sets of states have $\bot$ 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 [1]). The meta-language is indeed

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

bounded non-deterministic, [10] 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 $\bot$. We therefore specify the following domain as the result domain of the denotational semantic function instead of $\wp(S_\bot)$.

Definition 30 (T). The set T with typical element τ is defined as follows: $T = \{\tau \in \wp(S_\bot) \mid \tau \text{ finite or } \bot \in \tau\}$.

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

Definition 31 (*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 32 (*denotational semantics of meta-programs*). Let $\phi_1, \phi_2 : S_\bot \to T$. Then we define the following functions.

$$\begin{aligned} \hat{\phi} : T \;&\to T = \lambda\tau \cdot \bigcup_{s\in\tau} \phi(s), \\ \phi_1 \circ \phi_2 : S_\bot &\to T = \lambda s \cdot \hat{\phi}_1(\phi_2(s)). \end{aligned}$$

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

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

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 29.

5.2.1. 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 $\mathit{apply}(\rho)$ 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.

5.2.2. 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 [1].

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.

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

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 know that if the operator is continuous (Definition 27), 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* 32 *is continuous.*

Proof. See Appendix A. □

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 ˆ operator to be of type $T \to T$.

5.2.3. 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 ($\mathcal{O}_\mathrm{m} = \mathcal{M}$). *Let $\mathcal{O}_\mathrm{m} : Prog \to (S_\perp \to \wp(S_\perp))$ 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* 32). *Then, the following equivalence holds for all meta-programs $P \in Prog$ and all mental states $s \in S_\perp$.*

$$\mathcal{O}_\mathrm{m}(P)(s) = \mathcal{M}(P)(s).$$

Proof. See Appendix B. □

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 ($\mathcal{O}_\mathrm{o} = \mathcal{M}(\mathsf{interpreter})$). *From Theorems* 1 *and* 3 *we can conclude that the following holds.*

$$\mathcal{O}_\mathrm{o} = \mathcal{M}(\mathsf{interpreter}).$$

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[11], of type $(\Pi \times \Sigma) \to \wp(\Sigma_\perp)$. The type of $\mathcal{M}(\mathsf{interpreter})$, i.e., $S_\perp \to \wp(S_\perp)$,[12] does not match the desired type. This could however be remedied by defining the following function.

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

[12] $\mathcal{M}(\mathsf{interpreter})$ 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 $\wp(S_\perp)$.

Definition 33 ($\mathcal{N}$). 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 32. 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.

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

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 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 [13] 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 [8]. 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 [4]. 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

[13] Or actually of the plan concatenation operator $\bullet$.

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. [14]

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.

Appendix A. Continuity of Φ

The function $\Phi : (S_\perp \to T) \to (S_\perp \to T)$ from Definition 32 is defined as follows:

$$\lambda\phi \cdot \lambda s \cdot \texttt{if } \mathcal{W}(b)(s) \texttt{ then } \hat{\phi}(\mathcal{M}[\![P]\!](s)) \texttt{ else } \{s\} \texttt{ fi}.$$

In this section, we prove continuity of this function. The proof is analogous to continuity proofs given in [1]. In Definition 27, the concept of continuity was defined. As we will state below in fact 2, an equivalent definition can be given using the concept of monotonicity of a function.

Definition 34 (*monotonicity*). Let $(C, \sqsubseteq)$, $(C', \sqsubseteq)$ be CPOs and $c_1, c_2 \in C$. Then a function $f : C \to C'$ is monotone iff the following holds:

$$c_1 \sqsubseteq c_2 \Leftrightarrow f(c_1) \sqsubseteq f(c_2).$$

Fact 2 (*continuity*): Let $(C, \sqsubseteq)$, $(C', \sqsubseteq)$ be CPOs and let $f : C \to C'$ be a function. Then

$$\text{for all chains } \langle c_i \rangle_{i=0}^{\infty} \text{ in } C : f\left(\bigsqcup_{i=0}^{\infty} c_i\right) = \bigsqcup_{i=0}^{\infty} f(c_i)$$
$$\Leftrightarrow$$
$$f \text{ is monotone and for all chains } \langle c_i \rangle_{i=0}^{\infty} \text{ in } C : f\left(\bigsqcup_{i=0}^{\infty} c_i\right) \sqsubseteq \bigsqcup_{i=0}^{\infty} f(c_i).$$

Proof. For a proof, we refer to [15]. □

Below, we will prove continuity of Φ by proving that Φ is monotone and that for all chains $\langle \phi_i \rangle_{i=0}^{\infty}$ in $S_\perp \to T$, the following holds: $\Phi(\bigsqcup_{i=0}^{\infty} \phi_i) \sqsubseteq \bigsqcup_{i=0}^{\infty} \Phi(\phi_i)$.

Lemma 2 (*monotonicity of* Φ). *The function* Φ *as given in Definition* 32 *is monotone*, *i.e.*, *the following holds for all* $\phi_i, \phi_j \in S_\perp \to T$:

$$\phi_i \sqsubseteq \phi_j \Rightarrow \Phi(\phi_i) \sqsubseteq \Phi(\phi_j).$$

Proof. Take arbitrary $\phi_i, \phi_j \in S_\perp \to T$. Let $\phi_i \sqsubseteq \phi_j$. Then we need to prove that $\forall s \in S_\perp : \Phi(\phi_i)(s) \sqsubseteq \Phi(\phi_j)(s)$. Take an arbitrary $s \in S_\perp$. We need to prove that $\Phi(\phi_i)(s) \sqsubseteq \Phi(\phi_j)(s)$, i.e., that

$$\texttt{if } \mathcal{W}(b)(s) \texttt{ then } \hat{\phi}_i(\mathcal{M}[\![P]\!](s)) \texttt{ else } \{s\} \texttt{ fi} \sqsubseteq \texttt{if } \mathcal{W}(b)(s) \texttt{ then } \hat{\phi}_j(\mathcal{M}[\![P]\!](s)) \texttt{ else } \{s\} \texttt{ fi}.$$

We distinguish three cases.

(1) Let $\mathcal{W}(b)(s) = \perp_{W_\perp}$, then to prove: $\{\perp\} \sqsubseteq \{\perp\}$. This is true by Definition 31.
(2) Let $\mathcal{W}(b)(s) = ff$, then to prove: $\{s\} \sqsubseteq \{s\}$. This is true by Definition 31.

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

(3) Let $\mathcal{W}(b)(s) = tt$, then to prove: $\hat{\phi}_i(\mathcal{M}[\![P]\!](s)) \sqsubseteq \hat{\phi}_j(\mathcal{M}[\![P]\!](s))$. Let $\tau' = \mathcal{M}[\![P]\!](s)$. Using the definition of $\hat{\phi}$, we rewrite what needs to be proven into $\bigcup_{s' \in \tau'} \phi_i(s') \sqsubseteq \bigcup_{s' \in \tau'} \phi_j(s')$. Now we can distinguish two cases.
  (a) Let $\perp \notin \bigcup_{s' \in \tau'} \phi_i(s')$. Then to prove: $\bigcup_{s' \in \tau'} \phi_i(s') = \bigcup_{s' \in \tau'} \phi_j(s')$. From the assumption that $\perp \notin \bigcup_{s' \in \tau'} \phi_i(s')$, we can conclude that $\perp \notin \phi_i(s')$ for all $s' \in \tau'$. Using the assumption that $\phi_i(s) \sqsubseteq \phi_j(s)$ for all $s \in S_\perp$, we have that $\phi_i(s') = \phi_j(s')$ for all $s' \in \tau'$ and therefore $\bigcup_{s' \in \tau'} \phi_i(s') = \bigcup_{s' \in \tau'} \phi_j(s')$.
  (b) Let $\perp \in \bigcup_{s' \in \tau'} \phi_i(s')$. Then to prove: $(\bigcup_{s' \in \tau'} \phi_i(s')) \setminus \{\perp\} \subseteq \bigcup_{s' \in \tau'} \phi_j(s')$, i.e., $\bigcup_{s' \in \tau'}(\phi_i(s') \setminus \{\perp\}) \subseteq \bigcup_{s' \in \tau'} \phi_j(s')$. Using the assumption that $\phi_i(s) \sqsubseteq \phi_j(s)$ for all $s \in S_\perp$, we have that for all $s' \in \tau'$, either $\phi_i(s') \setminus \{\perp\} \subseteq \phi_j(s')$ or $\phi_i(s') = \phi_j(s')$, depending on whether $\perp \in \phi_i(s)$. From this we can conclude that $\bigcup_{s' \in \tau'}(\phi_i(s') \setminus \{\perp\}) \subseteq \bigcup_{s' \in \tau'} \phi_j(s')$.

As we now have that Φ is monotone, proving continuity comes down to proving the following lemma. □

Lemma 3. *For all chains* $\langle \phi_i \rangle_{i=0}^{\infty}$ *in* $S_\perp \to T$, *the following holds*:

$$\Phi\left(\bigsqcup_{i=0}^{\infty} \phi_i\right) \sqsubseteq \bigsqcup_{i=0}^{\infty} \Phi(\phi_i).$$

Proof. We have to prove that for all chains $\langle \phi_i \rangle_{i=0}^{\infty}$ in $S_\perp \to T$ and for all $s \in S_\perp$, the following holds: $(\Phi(\bigsqcup_{i=0}^{\infty} \phi_i))(s) \sqsubseteq (\bigsqcup_{i=0}^{\infty} \Phi(\phi_i))(s)$. Take an arbitrary chain $\langle \phi_i \rangle_{i=0}^{\infty}$ in $S_\perp \to T$ and an arbitrary state $s \in S_\perp$. Then to prove:

$$\texttt{if } \mathcal{W}(b)(s) \texttt{ then}\hat{\ }\left(\bigsqcup_{i=0}^{\infty} \phi_i\right)(\tau) \texttt{ else } \{s\} \texttt{ fi} \sqsubseteq \bigsqcup_{i=0}^{\infty} \texttt{if } \mathcal{W}(b)(s) \texttt{ then } \hat{\phi}_i(\tau) \texttt{ else } \{s\} \texttt{ fi},$$

where $\tau = \mathcal{M}[\![P]\!](s)$. We distinguish three cases.

(1) Let $\mathcal{W}(b)(s) = \perp_{W_\perp}$, then to prove: $\{\perp\} \sqsubseteq \bigsqcup_{i=0}^{\infty} \{\perp\}$, i.e., $\{\perp\} \sqsubseteq \{\perp\}$. This is true by Definition 31.
(2) Let $\mathcal{W}(b)(s) = ff$, then to prove: $\{s\} \sqsubseteq \bigsqcup_{i=0}^{\infty} \{s\}$, i.e., $\{s\} \sqsubseteq \{s\}$. This is true by Definition 31.
(3) Let $\mathcal{W}(b)(s) = tt$, then to prove:$\hat{\ }(\bigsqcup_{i=0}^{\infty} \phi_i)(\tau) \sqsubseteq \bigsqcup_{i=0}^{\infty} \hat{\phi}_i(\tau)$. If we can prove that $\forall \tau \in T : \hat{\ }(\bigsqcup_{i=0}^{\infty} \phi_i)(\tau) \sqsubseteq \bigsqcup_{i=0}^{\infty} \hat{\phi}_i(\tau)$, i.e.,$\hat{\ }(\bigsqcup_{i=0}^{\infty} \phi_i) \sqsubseteq \bigsqcup_{i=0}^{\infty} \hat{\phi}_i$, we are finished. A proof of the continuity of $\hat{\ }$ is given in de Bakker [1], from which we can conclude what needs to be proven. □

Proof of Theorem 2. Immediate from Lemmas 2 and 3 and fact 2. □

Appendix B. Theorem 3

We prove the theorem using techniques from Kuiper [9]. Kuiper proves equivalence of the operational and denotational semantics of a non-deterministic language with procedures but without a `while` construct. The proof involves structural induction on programs. As the cases of sequential composition and non-deterministic choice have been proven by Kuiper (and as they can easily be adapted to fit our language of meta-programs), we will only provide a proof for the atomic meta-actions and for the `while` construct. For a detailed explanation of the general ideas of the proof, we refer to Kuiper [9].

In our proof, we will use a number of lemmas from Kuiper or slight variations thereof. We restate those results here.

Lemma 4. *Let* $\mathcal{W}(b)(s) = tt$. *Then the following holds.*

$$\begin{aligned} \mathcal{O}(\texttt{while } b \texttt{ do } P' \texttt{ od})(s) &= \mathcal{O}(P'; \texttt{while } b \texttt{ do } P' \texttt{ od})(s) \\ \mathcal{M}(\texttt{while } b \texttt{ do } P' \texttt{ od})(s) &= \mathcal{M}(P'; \texttt{while } b \texttt{ do } P' \texttt{ od})(s) \end{aligned}$$

Lemma 5.

$$\mathcal{O}(P'; \texttt{while } b \texttt{ do } P' \texttt{ od})(s) = \mathcal{O}(\texttt{while } b \texttt{ do } P' \texttt{ od}) \circ \mathcal{O}(P')(s)$$

Lemma 6. *For all $s \in S$ and $P \in Prog$ for which $\mathcal{C}(P)(s) \in \wp(S^+)$, $\mathcal{C}(P)(s)$ is a finite set.*

We will now explain how the equivalence will be proven. The way to prove the equivalence result as was done by Kuiper, is the following. In case $\mathcal{C}_m(P)(s) \in \wp(S^+)$, induction on the *sum* of the lengths of the computation sequences in $\mathcal{C}_m(P)(s)$ is applied, thus proving $\mathcal{O}_m(P)(s) = \mathcal{M}(P)(s)$ in this case[15]. In case there is an infinite computation sequence in $\mathcal{C}(P)(s)$ and so $\perp \in \mathcal{O}(P)(s)$, we prove $\mathcal{O}(P)(s) \setminus \{\perp\} \subseteq \mathcal{M}(P)(s)$ by induction on the length of *individual* computation sequences. This yields $\mathcal{O}(P) \sqsubseteq \mathcal{M}(P)$. Proving $\mathcal{M}(P) \sqsubseteq \mathcal{O}(P)$ by standard techniques then completes the proof.

Proof of Theorem 3. $\mathcal{O}(P)(s) = \mathcal{M}(P)(s)$ holds trivially for $s = \perp$, so in the sequel we will assume $s \in S$.

1. $\mathcal{O}(P)(s) \sqsubseteq \mathcal{M}(P)(s)$.

Case A: $\perp \notin \mathcal{O}(P)(s)$ i.e., $\mathcal{C}(P)(s) \in \wp(S^+)$.

If $\mathcal{C}(P)(s) \in \wp(S^+)$, then we prove $\mathcal{O}(P)(s) = \mathcal{M}(P)(s)$ by cases, applying induction on the sum of the lengths of the computation sequences.

(1) $P \equiv$ *execute*.
Let $(\pi, \sigma) \in S$ and $\pi = a; \pi'$, with $\pi' \in \Pi$. If $\mathcal{T}(a, \sigma) = \sigma'$ (which implies that $a \in \mathsf{BasicAction}$), then the following can be derived directly from Definitions 18, 16 and 32: $\mathcal{O}(execute)(\pi, \sigma) = \kappa(\mathcal{C}(execute)(\pi, \sigma)) = \{(\pi', \sigma')\} = \mathcal{M}(execute)(\pi, \sigma)$. If $\mathcal{T}(a, \sigma)$ is undefined—meaning that either $a \in \mathsf{BasicAction}$ and $\mathcal{T}(a, \sigma)$ is undefined for this input, or $a \notin \mathsf{BasicAction}$—we have $\mathcal{O}(execute)(\pi, \sigma) = \emptyset = \mathcal{M}(execute)(\pi, \sigma)$.

(2) $P \equiv apply(\rho)$.
The proof is similar to the proof for *execute*.

(3) $P \equiv \texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od}$.
In case $\mathcal{W}(b)(s) = ff$, we have that $\mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) = \{s\} = \mathcal{M}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s)$ by definition. In the sequel, we will show that the equivalence also holds in case $\mathcal{W}(b)(s) = tt$.
The function "length" yields the sum of the lengths of the computation sequences in a set. From the assumption that $\mathcal{C}(P)(s) \in \wp(S^+)$, we can conclude that $\mathcal{C}(P)(s)$ is a finite set (Lemma 6). From Definition 17, we can then conclude the following.

$$\text{length}(\mathcal{C}(P')(s)) < \text{length}(\mathcal{C}(P'; \texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s)) < \infty$$

$$\text{length}(\mathcal{C}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(\kappa(\mathcal{C}(P')(s)))) < \text{length}(\mathcal{C}(P'; \texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s)) < \infty$$

So, by induction we have:

$$\begin{aligned} \mathcal{O}(P')(s) &= \mathcal{M}(P')(s), \\ \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(\kappa(\mathcal{C}(P')(s))) &= \mathcal{M}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(\kappa(\mathcal{C}(P')(s))). \end{aligned}$$

The proof is then as follows.

$$\begin{aligned} &\mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) \\ &\quad = \mathcal{O}(P'; \texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) && \text{(Lemma 4)} \\ &\quad = \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od}) \circ \mathcal{O}(P')(s) && \text{(Lemma 5)} \\ &\quad = \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(\kappa(\mathcal{C}(P')(s))) && \text{(Definition 18)} \\ &\quad = \mathcal{M}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(\kappa(\mathcal{C}(P')(s))) && \text{(Induction hypothesis)} \\ &\quad = \mathcal{M}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(\mathcal{O}(P')(s)) && \text{(Definition 18)} \\ &\quad = \mathcal{M}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od}) \circ \mathcal{M}(P')(s) && \text{(Induction hypothesis)} \\ &\quad = \mathcal{M}(P'; \texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) && \text{(Definition 32)} \\ &\quad = \mathcal{M}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) && \text{(Lemma 4)} \end{aligned}$$

[15] In the sequel, we will omit the subscript "m" to $\mathcal{C}$ and $\mathcal{O}$ which is used to denote that we are dealing with the meta-language.

Case B: $\bot \in \mathcal{O}(P)(s)$.

If P and s are such that $\bot \in \mathcal{O}(P)(s)$ then we prove by cases that $\mathcal{O}(P)(s)\backslash\{\bot\} \subseteq \mathcal{M}(P)(s)$, applying induction on the length of the computation sequence corresponding to that outcome, i.e., we prove that for $s' \neq \bot$: $s' \in \mathcal{O}(P)(s) \Rightarrow s' \in \mathcal{M}(P)(s)$).

(1) $P \equiv \mathit{execute}$ and $P \equiv \mathit{apply}(\rho)$.
Equivalence was proven in case A.

(2) $P \equiv \texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od}$.
Consider a computation sequence $\delta = \langle s_1, \ldots, s_n(= s')\rangle \in \mathcal{C}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s)$. From Definition 17 of the function $\mathcal{C}$, we can conclude that there are intermediate states $s_j, s_{j+1} \neq \bot$ in this sequence δ, i.e., $\delta = \langle s_1, \ldots, s_j, s_{j+1}, \ldots, s_n(= s')\rangle$ (where s_1 can coincide with s_j), with $s_j = s_{j+1}$ and moreover: $\langle s_1, \ldots, s_j\rangle \in \mathcal{C}(P')(s)$ and $\langle s_{j+1}, \ldots, s_n\rangle \in \mathcal{C}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s_j)$. The following can be derived immediately from the above.

$$\begin{aligned}\text{length}(\langle s_1, \ldots, s_j\rangle) &< \text{length}(\langle s_1, \ldots, s_n\rangle)\\ \text{length}(\langle s_{j+1}, \ldots, s_n\rangle) &< \text{length}(\langle s_1, \ldots, s_n\rangle)\end{aligned}$$

We thus have the following induction hypothesis.

$$\begin{gathered}s_j \in \mathcal{O}(P')(s) \Rightarrow s_j \in \mathcal{M}(P')(s)\\ s' \in \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s_j) \Rightarrow s' \in \mathcal{M}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s_j)\end{gathered}$$

As $\langle s_1, \ldots, s_j\rangle \in \mathcal{C}(P')(s)$, we know that $s_j \in \mathcal{O}(P')(s)$ (Definition 18) and similarly we can conclude that $s' \in \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s_j)$. We thus have, using the induction hypothesis that: $s_j \in \mathcal{M}(P')(s)$ and $s' \in \mathcal{M}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s_j)$. From this we can conclude the following, deriving what was to be proven.

$$\begin{aligned}&s' \in \mathcal{M}(P';\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) \quad \text{(Definition 32)}\\ &s' \in \mathcal{M}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) \quad \text{(Lemma 4)}\end{aligned}$$

2. $\mathcal{M}(P)(s) \sqsubseteq \mathcal{O}(P)(s)$

(1) $P \equiv \mathit{execute}$.
Equivalence was proven in case (1).

(2) $P \equiv \mathit{apply}(\rho)$.
Equivalence was proven in case (1).

(3) $P \equiv \texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od}$.
We will use induction on the entity $(i, \mathit{length}(P))$ where $\mathit{length}(P)$ denotes the length of the statement P and we use a lexicographic ordering on these entities, i.e., $(i_1, l_1) < (i_2, l_2)$ iff either $i_1 < i_2$ or $i_1 = i_2$ and $l_1 < l_2$. Clearly, $\mathit{length}(P') < \mathit{length}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})$ holds. Therefore $(i, \mathit{length}(P')) < (i, \mathit{length}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od}))$ holds.

We know that $\mathcal{M}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od}) = \mu\Phi = \bigsqcup_{i=0}^{\infty} \Phi^i(\bot_{S_\bot \to T})$ by continuity of Φ (Theorem 2). Let $\phi_i = \Phi^i(\bot_{S_\bot \to T})$. We thus need to prove that $\bigsqcup_{i=0}^{\infty} \phi_i \sqsubseteq \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})$. So, if we can prove that $\phi_i \sqsubseteq \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})$ holds for all i, we will have the desired result. We will prove this by induction on the entity $(i, \mathit{length}(P))$. As $(i, \mathit{length}(P')) < (i, \mathit{length}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od}))$ and $(i, l) < (i + 1, l)$, our induction hypothesis will be:

$$\mathcal{M}(P') \sqsubseteq \mathcal{O}(P') \quad \text{and} \quad \phi_i \sqsubseteq \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od}).$$

The induction basis is provided as $\phi_0 = \bot_{S_\bot \to T} \sqsubseteq \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})$ holds. From this we have to prove that for all $s \in S : \phi_{i+1}(s) \sqsubseteq \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s)$. Take an arbitrary $s \in S$. We have to prove that:

$$\begin{aligned}\Phi(\phi_i)(s) &\sqsubseteq \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) \quad \text{i.e., by Definition 32}\\ \hat{\phi}_i(\mathcal{M}(P')(s)) &\sqsubseteq \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) \quad \text{i.e., by Lemmas 4 and 5}\\ \hat{\phi}_i(\mathcal{M}(P')(s)) &\sqsubseteq \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(\mathcal{O}(P')(s)) \quad (*).\end{aligned}$$

We know that $\mathcal{M}(P')(s) = \mathcal{O}(P')(s)$ by the induction hypothesis and the fact that we have already proven $\mathcal{M}(P')(s) \sqsupseteq \mathcal{O}(P')(s)$. Let $\tau' = \mathcal{M}(P')(s) = \mathcal{O}(P')(s)$ and let $s' \in \tau'$. By the induction hypothesis, we have that $\phi_i(s') \sqsubseteq \mathcal{O}(\texttt{while } b \texttt{ do } P' \texttt{ od})(s')$ for all $s' \in S_\perp$. From this, we can conclude that $\bigsqcup_{s' \in \tau} \phi_i(s') \sqsubseteq \bigsqcup_{s' \in \tau} \mathcal{O}(\texttt{while } b \texttt{ do } P' \texttt{ od})(s')$ (see, the proof of Lemma 2), which can be rewritten into what was to be proven $(*)$ using the definitions of $\hat{\phi}_i$ and function composition. □

References

[1] J. de Bakker, Mathematical Theory of Program Correctness, Series in Computer Science, Prentice-Hall, London, 1980.

[2] M.E. Bratman, Intention, Plans, and Practical Reason, Harvard University Press, Cambridge, MA, 1987.

[3] P.R. Cohen, H.J. Levesque, Intention is choice with commitment, Artificial Intelligence 42 (1990) 213–261.

[4] M. Dastani, F.S. de Boer, F. Dignum, J.-J.Ch. Meyer, Programming agent deliberation—an approach illustrated using the 3APL language, in: Proc. Second Internat. Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS'03), Melbourne, 2003, pp. 97–104.

[5] H. Egli, A mathematical model for nondeterministic computations, Technical Report, ETH, Zürich, 1975.

[6] G.d. Giacomo, Y. Lespérance, H. Levesque, ConGolog, A concurrent programming language based on the situation calculus, Artificial Intelligence 121 (1–2) (2000) 109–169.

[7] K.V. Hindriks, F.S. de Boer, W. van der Hoek, J.-J.Ch. Meyer, Agent programming in 3APL, Internat. J. of Autonomous Agents and Multi-Agent Systems 2 (4) (1999) 357–401.

[8] K. Hindriks, F.S. de Boer, W. van der Hoek, J.-J.Ch. Meyer, Control structures of rule-based agent languages, in: J. Müller, M.P. Singh, A.S. Rao (Eds.), Proc. Fifth Internat. Workshop on Intelligent Agents V: Agent Theories, Architectures, and Languages (ATAL-98), Vol. 1555, Springer, Heidelberg, Germany, 1999, pp. 381–396.

[9] R. Kuiper, An operational semantics for bounded nondeterminism equivalent to a denotational one, in: J.W. de Bakker, J.C. van Vliet (Eds.), Proc. Internat. Symp. on Algorithmic Languages, North-Holland, 1981, pp. 373–398.

[10] P.D. Mosses, Denotational semantics, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics, Elsevier, Amsterdam, 1990, pp. 575–631.

[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, J. Perram (Eds.), Agents Breaking Away, Lecture Notes in Artificial Intelligence, Vol. 1038, Springer, Berlin, 1996, pp. 42–55.

[13] A.S. Rao, M.P. Georgeff, Modeling rational agents within a BDI-architecture, in: J. Allen, R. Fikes, E. Sandewall (Eds.), Proc. Second Internat. Conf. on Principles of Knowledge Representation and Reasoning (KR'91), Morgan Kaufmann, Los Altos, CA, 1991, pp. 473–484.

[14] M.B. van Riemsdijk, W. van der Hoek, J.-J.Ch. Meyer, Agent programming in Dribble: from beliefs to goals using plans, in: Proc. Second Internat. joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS'03), Melbourne, 2003, pp. 393–400.

[15] M.B. van Riemsdijk, J.-J.Ch. Meyer, F.S. de Boer, Semantics of plan revision in intelligent agents, Technical Report, Utrecht University, Institute of Information and Computing Sciences, UU-CS-2004-002, 2003.

[16] Y. Shoham, Agent-oriented programming, Artificial Intelligence 60 (1993) 51–92.

[17] J.E. Stoy, Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press, Cambridge, MA, 1977.

[18] R. Tennent, Semantics of Programming Languages, Series in Computer Science, Prentice-Hall, London, 1991.

[19] M. Wooldridge, Agent-based software engineering, IEEE Proc. Software Eng. 144 (1) (1997) 26–37.

[20] M. Wooldridge, P. Ciancarini, Agent-oriented software engineering: the state of the art, in: P. Ciancarini, M. Wooldridge (Eds.), First Internat. Workshop on Agent-Oriented Software Engineering, Vol. 1957, Springer, Berlin, 2001, pp. 1–28.