Semantics of Plan Revision in Intelligent Agents

M. Birna van Riemsdijk

John-Jules Ch. Meyer

Frank S. de Boer

institute of information and computing sciences, utrecht university

technical report UU-CS-2004-002

www.cs.uu.nl

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 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 *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

[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.
[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.

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, 15, 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 [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. 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.

3APL interpreter language, what's the role of the interpreter (scheduling rule application and action execution)), operational vs denotational semantics, plan revision, reflection, introspection

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 $\mathsf{BasicAction}$ 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 $c\,;\pi \in \Pi$.

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 so-called 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 $\psi \in \mathcal{L}$ and $\pi_h, \pi_b \in \Pi$ 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 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 $\mathsf{BasicAction}$, 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 3.1, 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. Through the execution of meta-actions, a mental state can thus be transformed. To iterate execution of meta-actions, meta-programs can also contain a `while` construct. Composite meta-programs can be constructed using a sequential composition operator and a non-deterministic choice operator.

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. This is needed in the proofs of section 4.1.

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 transition rule for rule application, we use the operator $\bullet$. $\pi_1 \bullet \pi_2$ denotes a plan of which π_1 is the first part and π_2 is the second: π_1 is the prefix of this plan. The operator is needed, because plans have a list structure (see definition 2). The plan $\pi_1; \pi_2$ is thus not syntactically correct. Plans have a list structure, because we do not want to semantically distinguish two plans with equal leaves, but a differing tree structure. There should for example be no semantic difference between the plans $a; (b; c)$ and $(a; b); c$. A list structure therefore suffices, but the trade-off is that we cannot use the sequential composition operator to denote prefixing of more than one element.

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

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

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 $Prog \to (S \to \wp(S^+ \cup S^\infty))$.

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

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. In the sequel we will provide some lemma's and a corollary from which this equivalence result will prove to follow immediately. As equivalence of object-level and meta-level operational semantics holds for input state $\bot$ by definition 18, we will only need to prove equivalence for input states $s \in S$.

Before moving on to proving the equivalence theorem, we have to make the following remark. The equivalence between object-level 3APL and the interpreter defined above only holds if Rule does not contain reactive rules of the form $E \mid \psi \rightsquigarrow \pi_b$. The reason is, that we chose the empty plan as a termination condition for the interpreter and the interpreter will thus stop applying rules in a mental state once the plan in this state is empty. Rule application is however still a possible transition in the object-level transition system in case of the presence of reactive rules, as these are applicable to empty plans. We thus exclude reactive rules from the set Rule. This restriction could be relaxed, but the interpreter would have to be adapted to yield the equivalence result (the condition of the `while` would have to be *true*). For reasons of space and clarity, we will not discuss this possibility here. Furthermore, having an empty plan or program as a termination condition is in line with the notion of successful termination in procedural programming languages.

4.1 Equivalence Theorem

We prove a weak bisimulation between Trans_o and Trans_m(interpreter). From this, we can then prove that $\mathcal{O}_o$ and $\mathcal{O}_m$(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)*

$$\begin{array}{rll} s \rightarrow_{execute} & s' & \text{is a transition in } \mathsf{Trans}_o \ \Leftrightarrow \\ \langle execute, s\rangle \rightarrow_{execute} & \langle E, s'\rangle & \text{is a transition in } \mathsf{Trans}_m \end{array}$$

$$\begin{array}{rll} s \rightarrow_{apply(\rho)} & s' & \text{is a transition in } \mathsf{Trans}_o \ \Leftrightarrow \\ \langle apply(\rho), s\rangle \rightarrow_{apply(\rho)} & \langle E, s'\rangle & \text{is a transition in } \mathsf{Trans}_m \end{array}$$

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 Trans_o and Trans_m, we can now phrase the following bisimulation lemma.

Lemma 1. *(weak bisimulation)* Let $+^*$ abbreviate $(execute + apply(\rho_1) + \ldots + apply(\rho_n))$. Let $\mathsf{Trans}_m(P)$ be the restriction of 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. Then a weak bisimulation exists between Trans_o and $\mathsf{Trans}_m(\mathsf{interpreter})$, i.e. the following properties hold.

$s \rightarrow_x s'$ is a transition in Trans_o $\Rightarrow_1$ $\langle \mathsf{interpreter}, s\rangle \rightarrow_{idle} \langle +^*; \mathsf{interpreter}, s\rangle \rightarrow_t \langle \mathsf{interpreter}, s'\rangle$ is a transition in $\mathsf{Trans}_m(\mathsf{interpreter})$

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

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

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

Furthermore, by the assumption that $s \rightarrow_t s'$ is a transition in 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 $\mathsf{Trans}_m(\mathsf{interpreter})$.

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

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 Trans_m (definition 14). Therefore, by proposition 1, we can conclude that $s \rightarrow_x s'$ is a transition in 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. 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 $s_0 \rightarrow_{t_1} \ldots \rightarrow_{t_n} s$ must be derivable in 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:

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

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 $\ldots \rightarrow_{idle} \langle E, s \rangle$. 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:

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

From this, we can conclude by lemma 1 that $s_0 \rightarrow_{t_1} \ldots \rightarrow_{t_{n-1}} s' \rightarrow_{t_n} s \not\rightarrow$ must be a sequence of transitions in 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 ([16]). 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 P in s, i.e. a function of type $Prog \rightarrow (S_\perp \rightarrow \wp(S_\perp))$[9]. 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 ([17], [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 (see definition 25). This concept is defined in terms of the notions of partially ordered sets, least upper bounds and chains.

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 $C_1 \rightarrow C_2$ 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 $C_1 \rightarrow C_2$ is defined as follows, where $f, g \in C_1 \rightarrow C_2$:

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

[9] 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 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 $\langle c_i \rangle_{i=0}^{\infty}$:

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

We will now define the notion of a chain. A chain on a set C with some order $\sqsubseteq$ can be defined in two ways. It can first of all be defined as a subset $X \subseteq C$ with a total ordering, i.e. for all $x, y \in X$, either $x \sqsubseteq y$ or $y \sqsubseteq x$. Secondly, it can be defined as a finite or infinite sequence, the elements of which have to be ordered in a certain way. This is specified for infinite sequences in the definition below.

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

The two definitions of chains are related as follows. If $(C, \sqsubseteq)$ is a partial order, the following holds: if $\langle c_i \rangle_{i=0}^{\infty}$ is a chain in C, the set $\{c \mid c \text{ in } \langle c_i \rangle_{i=0}^{\infty}\}$ is also a chain in C. For an ordering that is not partial, this does not have to hold (take for example as C the set of natural numbers with the non-reflexive and non-transitive ordering $x \sqsubseteq y \Leftrightarrow y = x + 1$ and take as a sequence $\langle 0, 1, 2, \ldots \rangle$). Note that the number of distinct elements of an infinite sequence can be finite or infinite. In line with other literature on semantics of programming languages ([4], [9], [14], [17]), we will use the definition of chains as sequences. As we only use it in the context of partial orders, these chains are related to chains defined as sets as explained.

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

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 following facts about CPO's of functions will turn out to be useful. For proofs, see for example De Bakker ([4]).

Fact 1. *(CPO of functions)* Let $(C_1, \sqsubseteq_1)$ and $(C_2, \sqsubseteq_2)$ be CPO's. Then $(C_1 \to C_2, \sqsubseteq)$ with $\sqsubseteq$ as in definition 21 is a CPO.

Fact 2. *(least upper bound of a chain of functions)* Let $(C_1, \sqsubseteq_1)$ and $(C_2, \sqsubseteq_2)$ be CPO's and let $\langle f_i \rangle_{i=0}^{\infty}$ be a chain of functions in $C_1 \to C_2$. Then the function $\lambda c_1 \cdot \bigsqcup_{i=0}^{\infty} f_i(c_1)$ is the least upper bound of this chain and therefore $(\bigsqcup_{i=0}^{\infty} f_i)(c_1) = \bigsqcup_{i=0}^{\infty} f_i(c_1)$ for all $c_1 \in C_1$.

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.

If a function f on a CPO is continuous, the fixed point theorem as specified below in fact 3 tells us that the least fixed point of f exists and that it is equal to the least upper bound of the chain $\langle f^i(\bot)\rangle_{i=0}^{\infty}$. This will turn out to be a useful fact in proving that the denotational semantic function is well-defined and in addition, it will prove to be more intuitive to explain the semantics in terms of chains of functions, rather than in terms of least fixed points of functions.

Definition 27. *(continuity)* Let $(C_1, \sqsubseteq_1), (C_2, \sqsubseteq_2)$ be CPO's. 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:

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

Fact 3. *(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 ([4]).

5.2 Definition

Having explained some basics about CPO's in general, 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 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 CPO's as follows:

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

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 now augment the specification of the semantics of boolean expressions to include a clause for this element.

Definition 29. *(semantics of boolean expressions)* The semantics of boolean expressions is as assumed in definition 12 for $s \in S$ and is augmented with the following clause for the mental state $\bot \in S_\bot$, yielding a function of type $Bexp \to (S_\bot \to W_\bot)$.

$$\mathcal{W}(b)(\bot) = \bot_{W_\bot}$$

In the sequel, it will be useful to have an if-then-else function as defined below.

Definition 30. *(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. An example of a meta-program yielding an infinite set of states is the following: TODO good example. In case of bounded non-determinism[10], 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

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

tree with infinitely many nodes has at least one infinite path (see [4]). The meta-language is indeed bounded non-deterministic[11] 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 31. *(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 32. *(Egli-Milner ordering)* Let $\tau_1, \tau_2 \in T$. $\tau_1 \sqsubseteq \tau_2$ holds iff either $\bot \in \tau_1$ and $\tau_1 \backslash \{\bot\} \subseteq \tau_2$, 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 33. *(denotational semantics of meta-programs)* Let $\phi_1, \phi_2 : S_\bot \to T$. Then we define the following functions.

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

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

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

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 30.

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*(ρ) 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. We will try to explain it by first defining a set of requirements on the semantics and then giving an intuitive understanding of the general ideas behind the semantics. Next, we will go into more detail on the semantics of a `while` construct in deterministic languages, followed by details on the semantics as defined above for our non-deterministic language.

Now, 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 can moreover not be defined circularly, but it should be defined compositionally, i.e.

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

it can only use the semantics of the guard and of the body of the `while`. This ensures that $\mathcal{M}$ is well-defined. The semantics of the `while` can thus not be defined as $\mathcal{M}[\![\texttt{while } b \texttt{ do } P \texttt{ od}]\!] = \mathcal{M}[\![\texttt{if } b \texttt{ then } P; \texttt{while } b \texttt{ do } P \texttt{ od else } \mathit{skip} \texttt{ fi}]\!]$ where *skip* is a statement doing nothing, because this would violate the requirement of compositionality.

Intuitively, the semantics of `while` b `do` P `od` should correspond to repeatedly executing P, until b is false. The semantics could thus be something like:

$$\mathcal{M}[\![\texttt{if } b \texttt{ then } P; \texttt{if } b \texttt{ then } P; \ldots \texttt{ else } \mathit{skip} \texttt{ fi else } \mathit{skip} \texttt{ fi}]\!].$$

The number of nestings of if-then-else constructs should however be infinite, as we cannot determine in advance how many times the body of the while loop will be executed, worse still, it could be the case that it will be executed an infinite number of times in case of non-termination. As programs are however by definition finite syntactic objects (see definition 6), the semantics of the `while` cannot be defined in this way. The idea of the solution now is, *not* to try to specify the semantics *at once* using infinitely many nestings of if-then-else constructs, but instead to specify the semantics using approximating functions, where some approximation is at least as good as another if it contains more nestings of if-then-else constructs. So to be a little more specific, what we do is specify a sequence of approximating functions $\langle \phi_i \rangle_{i=0}^{\infty}$, where ϕ_i roughly speaking corresponds to executing the body of the `while` construct less than i times and the idea now is that the limit of this sequence of approximations is the semantics of the `while` we are looking for.

Deterministic languages Having provided some vague intuitive ideas about the semantics, we will now go into more detail and use the theory on CPO's of section 5.1. To simplify matters, we will first consider a deterministic language. The function defining the semantics of the `while` should then be of type $S_\perp \to S_\perp$. The domain $S_\perp$ is designed as a CPO and therefore the domain of functions $S_\perp \to S_\perp$ is also a CPO (see fact 1). What we are looking for, is a chain of approximating functions $\langle \phi_i \rangle_{i=0}^{\infty}$ in this CPO $S_\perp \to S_\perp$, the least upper bound of which should yield the desired semantics for the `while`. As we are looking for a chain (see definition 24) of functions $\langle \phi_i \rangle_{i=0}^{\infty}$, it should be the case that for all $s \in S_\perp : \phi_i(s) \sqsubseteq \phi_j(s)$ for $i \leq j$. Intuitively, a function ϕ_j is "as least as good" an approximation as ϕ_i.

Approximating functions with the following behaviour will do the trick. A function ϕ_i should be defined such, that the result of ϕ_i applied to an initial state s_0, is the state in which the while loop terminates if less than i runs through the loop are needed for termination. If i or more runs are needed, i.e. if the execution of the `while` has not terminated after $i-1$ runs, the result should be $\perp$. To illustrate how these approximating functions can be defined to yield this behaviour, we will give the first few approximations ϕ_0, ϕ_1 and ϕ_2. In order to do this, we need to introduce a special statement *diverge*, which yields the state $\perp$ if executed. It can be shown that with the functions ϕ_i as defined below, $\langle \phi_i \rangle_{i=0}^{\infty}$ is indeed a chain. Therefore it has a least upper bound by definition, which ensures that the semantics is well-defined.

$$\begin{aligned}
\phi_0 &= \mathcal{M}[\![\mathit{diverge}]\!] \\
\phi_1 &= \mathcal{M}[\![\texttt{if } b \texttt{ then } P; \mathit{diverge} \texttt{ else } \mathit{skip} \texttt{ fi}]\!] \\
\phi_2 &= \mathcal{M}[\![\texttt{if } b \texttt{ then } \\
&\qquad\qquad P; \texttt{if } b \texttt{ then } P; \mathit{diverge} \texttt{ else } \mathit{skip} \texttt{ fi} \\
&\qquad \texttt{else } \mathit{skip} \texttt{ fi}]\!] \\
&\vdots
\end{aligned}$$

These definitions can be generalized, yielding the following definition of the functions ϕ_i in which we use the if-then-else function of definition 30.

$$\begin{aligned}
\phi_0 &= \perp_{S_\perp \to S_\perp} \\
&= \lambda s \cdot \perp \\
\phi_{i+1} &= \lambda s \cdot \texttt{if } \mathcal{W}(b)(s) \texttt{ then } \phi_i(\mathcal{M}[\![P]\!](s)) \texttt{ else } s \texttt{ fi}
\end{aligned}$$

Now, as $\langle \phi_i \rangle_{i=0}^{\infty}$ is a chain, so is $\langle \phi_i(s_0) \rangle_{i=0}^{\infty}$. Using fact 2, we know that $\bigsqcup_{i=0}^{\infty} \phi_i = \lambda s_0 \cdot \bigsqcup_{i=0}^{\infty} \phi_i(s_0)$ or $(\bigsqcup_{i=0}^{\infty} \phi_i)(s_0) = \bigsqcup_{i=0}^{\infty} \phi_i(s_0)$, i.e. the semantics of the execution of the `while` construct in an initial state s_0, is the least upper bound of the chain $\langle \phi_i(s_0) \rangle_{i=0}^{\infty}$.

The semantic function defined in this way, will indeed yield the desired behaviour for the while loop as sketched above, which we will show now. If the while loop is non-terminating, the chain $\langle \phi_i(s_0) \rangle_{i=0}^{\infty}$ will be $\langle \bot, \bot, \bot, \ldots \rangle$. This is because the guard b will remain true and therefore the statement *diverge* will be "reached" in each ϕ_i. Taking the least upper bound of this chain will give us $\bot$, which corresponds to the desired semantics of non-terminating while loops. If the loop terminates in some state s, the sequence $\langle \phi_i(s_0) \rangle_{i=0}^{\infty}$ will be a chain of the form $\langle \bot, \bot, \bot, \ldots, \bot, s, s, s, \ldots \rangle$, as can easily be checked. The state s then is the least upper bound.

Finally, to prepare for the treatment of the semantics as we have defined it for the non-deterministic case, the following must still be explained. Above, we have defined $\mathcal{M}[\![\texttt{while } b \texttt{ do } P \texttt{ od}]\!]$ as $\bigsqcup_{i=0}^{\infty} \phi_i$. Instead of defining the semantics using least upper bounds, we could have given an equivalent least fixed point characterization as follows. Let $\phi = \bigsqcup_{i=0}^{\infty} \phi_i$. Then we can give an operator $\Phi : (S_\bot \to S_\bot) \to (S_\bot \to S_\bot)$, i.e. a function on CPO $S_\bot \to S_\bot$, such that the least fixed point of this operator equals ϕ, i.e. $\mu\Phi = \phi$. This operator Φ is the function $\lambda\phi \cdot \lambda s \cdot \texttt{if } \mathcal{W}(b)(s) \texttt{ then } \phi(\mathcal{M}[\![P]\!](s)) \texttt{ else } s \texttt{ fi}$. We know that if Φ is continuous, $\mu\Phi = \bigsqcup_{i=0}^{\infty} \Phi^i(\bot_{S_\bot \to S_\bot})$ by the least fixed point theorem (fact 3). It can be shown that Φ is indeed continuous and furthermore, that $\Phi^i(\bot_{S_\bot \to S_\bot}) = \phi_i$. Therefore $\bigsqcup_{i=0}^{\infty} \phi_i = \bigsqcup_{i=0}^{\infty} \Phi^i(\bot_{S_\bot \to S_\bot})$ and thus $\phi = \mu\Phi$. The question of whether to specify the semantics of a `while` construct using least upper bounds or least fixed points, is basically a matter of taste. We have chosen to use a least fixed point characterization in the semantics of meta-programs.

Non-deterministic languages Having explained the denotational semantics of a while loop in deterministic languages, we will now move on to the non-deterministic case, as our meta-programming language is non-deterministic. In the non-deterministic case, the execution of a while loop could lead to a set of possible resulting end states, including the state $\bot$ if there is a possibility of non-termination. A certain approximation ϕ_i now is a function yielding a set of end states that can be reached in less than i runs through the loop. It will contain bottom if it is possible that the execution of the loop has not terminated after i runs. The limit of the sequence $\langle \phi_i(s_0) \rangle_{i=0}^{\infty}$ will thus be either a finite set of states possibly containing $\bot$ (if there is a possibility of non-termination) or an infinite set which will always contain $\bot$ because of Königs lemma (see introduction). The semantic function we are looking for, will thus be of type $S_\bot \to T$.

As stated, the semantics of the `while` construct in our meta-language is defined using least fixed points. To be more specific, it is defined as the least fixed point of the operator $\Phi : (S_\bot \to T) \to (S_\bot \to T)$ (see definition 33). Φ is thus a function on the CPO $S_\bot \to T$ (definitions 28, 32 and fact 1) and the semantics of the `while` is defined to be $\mu\Phi$. We must make sure that $\mu\Phi$ actually exists, in order for $\mathcal{M}$ to be well-defined. We do this by showing that Φ is continuous (see section 5.3), in which case $\mu\Phi = \bigsqcup_{i=0}^{\infty} \Phi^i(\bot_{S_\bot \to T})$. The bottom element $\bot_{S_\bot \to T}$ of the CPO $S_\bot \to T$ is $\lambda s \cdot \{\bot\}$, i.e. a function that takes some state and returns a set of states containing only the bottom state. Note that the type of Φ is such, that $\mu\Phi$ yields a semantic function $\phi : S_\bot \to T$, corresponding to the type of the function $\mathcal{M}[\![\texttt{while } b \texttt{ do } P \texttt{ od}]\!]$. The operator Φ is thus of the desired type.

The operator Φ we use, is a non-deterministic version of the $\Phi -$ operator of the previous paragraph. This is established through the function $\hat{} : (S_\bot \to T) \to (T \to T)$. This function takes a function ϕ of type $S_\bot \to T$ and a set $\tau \in T$ and returns the union of ϕ applied to each element $s \in \tau$, i.e. $\bigcup_{s \in \tau} \phi(s)$. We will now give the first few elements of the sequence of approximations $\langle \Phi^i(\bot_{S_\bot \to T}) \rangle_{i=0}^{\infty}$, to illustrate how Φ is defined in the non-deterministic case. For reasons of presentation, we will assume that $\mathcal{M}[\![P]\!](s) \neq \emptyset$ in which case $(\hat{}(\lambda s \cdot \{\bot\}))(\mathcal{M}[\![P]\!](s)) = \{\bot\}$. As it follows from the definition of $\hat{\phi}$ (definition 33) that $\hat{\phi}(\emptyset) = \emptyset$, some equivalences as stated

below would not hold in this case.

$$
\begin{aligned}
\Phi^0(\bot_{S_\bot \to T}) &= \lambda s \cdot \{\bot\} \\
\Phi^1(\bot_{S_\bot \to T}) &= \Phi(\Phi^0(\bot_{S_\bot \to T})) \\
&= \Phi(\lambda s \cdot \{\bot\}) \\
&= \lambda s \cdot \texttt{if } \mathcal{W}(b)(s) \texttt{ then } (\hat{}(\lambda s \cdot \{\bot\}))(\mathcal{M}[\![P]\!](s)) \texttt{ else } \{s\} \texttt{ fi} \\
&= \lambda s \cdot \texttt{if } \mathcal{W}(b)(s) \texttt{ then } \{\bot\} \texttt{ else } \{s\} \texttt{ fi} \\
\Phi^2(\bot_{S_\bot \to T}) &= \Phi(\Phi^1(\bot_{S_\bot \to T})) \\
&= \Phi(\lambda s \cdot \texttt{if } \mathcal{W}(b)(s) \texttt{ then } \{\bot\} \texttt{ else } \{s\} \texttt{ fi}) \\
&= \lambda s \cdot \texttt{if } \mathcal{W}(b)(s) \\
&\qquad \texttt{then} \\
&\qquad\quad (\hat{}(\lambda s' \cdot \texttt{if } \mathcal{W}(b)(s') \\
&\qquad\qquad\qquad \texttt{then} \\
&\qquad\qquad\qquad\quad \{\bot\} \\
&\qquad\qquad\qquad \texttt{else} \\
&\qquad\qquad\qquad\quad \{s'\} \texttt{ fi}))(\mathcal{M}[\![P]\!](s)) \\
&\qquad \texttt{else } \{s\} \texttt{ fi} \\
\Phi^3(\bot_{S_\bot \to T}) &= \Phi(\Phi^2(\bot_{S_\bot \to T})) \\
&= \ldots
\end{aligned}
$$

The zeroth approximation by definition (see fact 3) always yields the bottom element of the CPO $S_\bot \to T$. The first approximation is a function that takes an initial state s and yields either the set $\{\bot\}$ if the guard is true in s, i.e. if there will be one or more runs through the loop, or the set $\{s\}$ if the guard is false in s, i.e. if the while loop is such that it terminates in s without going through the loop. The second approximation is a function that takes an initial state s and yields, like the first approximation, the set $\{s\}$ if the guard is false in s. It thus returns the same result as the first approximation if it takes less than one runs through the loop to terminate. This is exactly what we want, as the first approximation will be as good as it gets in this case. If the guard is true in s, the function $\lambda s' \cdot \texttt{if } \mathcal{W}(b)(s') \texttt{ then } \{\bot\} \texttt{ else } \{s'\} \texttt{ fi}$ is applied to each state in the set of states resulting from executing the body of the while in s, i.e. the set $\mathcal{M}[\![P]\!](s)$ which we will refer to by τ'. The function thus takes some state s' from τ' and either yields $\{s'\}$ if the guard is true in s', i.e. if the while can terminate in s' after one run through the loop, or yields $\{\bot\}$ if the guard is false in s', i.e. if the execution path going through s' has not ended. The function $\hat{}$ now takes the union of the results for each s', yielding a set of states containing the states in which the while loop can end after one run through the loop, and containing $\bot$ if it is possible that the while loop has not terminated after one run. The function Φ is thus defined such that a certain approximation $\Phi^i(\bot_{S_\bot \to T})$ is a function yielding a set of end states that can be reached in less than i runs through the loop. It will contain $\bot$ if it is possible that the execution of the loop has not terminated after i runs.

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

5.3 Continuity of Φ

In the previous section, we have given the denotational semantics of meta-programs. In this definition, the semantics of the `while` construct was defined to be the least fixed point of the operator Φ on the CPO $S_\bot \to T$. As stated, we must make sure that this least fixed point $\mu\Phi$ actually exists, in order for $\mathcal{M}$ to be well-defined. This can be done by showing that Φ is continuous (see fact 3), which is what we will do in this section. We will repeat the definition of Φ here.

Definition 34. *(the operator Φ)* The operator $\Phi : (S_\bot \to T) \to (S_\bot \to T)$ is defined for some meta-program $P \in Prog$ as follows, with $\mathcal{W}$ as in definition 29, the if-then-else function as in

definition 30 and $\mathcal{M}$ and $\hat{\phi}$ as in definition 33.

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

In definition 27, the concept of continuity was defined. As we will state below in fact 4, an equivalent definition can be given using the concept of monotonicity of a function.

Definition 35. *(monotonicity)* Let $(C, \sqsubseteq), (C', \sqsubseteq)$ be CPO's 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 4. *(continuity)* Let $(C, \sqsubseteq), (C', \sqsubseteq)$ be CPO's and let $f : C \to C'$ be a function. Then:

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

Proof. ($\Rightarrow$): Assume for all chains $\langle c_i\rangle_{i=0}^{\infty}$ in $C : f(\bigsqcup\langle c_i\rangle_{i=0}^{\infty}) = \bigsqcup\langle f(c_i)\rangle_{i=0}^{\infty}$. Then $f(\bigsqcup\langle c_i\rangle_{i=0}^{\infty}) \sqsubseteq \bigsqcup\langle f(c_i)\rangle_{i=0}^{\infty}$ trivially holds for all chains $\langle c_i\rangle_{i=0}^{\infty}$ in C. To prove: monotonicity of f.
Let $c, c' \in C$ and assume $c \sqsubseteq c'$. The following holds: $f(c) \sqsubseteq \bigsqcup\{f(c), f(c')\}$. As $\bigsqcup\{f(c), f(c')\} = \bigsqcup\langle f(c_i)\rangle_{i=0}^{\infty}$ with $\langle f(c_i)\rangle_{i=0}^{\infty} \in Chain(\{f(c), f(c')\})$, we can conclude that $f(c) \sqsubseteq f(\bigsqcup\langle c_i\rangle_{i=0}^{\infty})$ with $\langle c_i\rangle_{i=0}^{\infty} \in Chain(\{c, c'\})$ by assumption. As $\bigsqcup\langle c_i\rangle_{i=0}^{\infty} = \bigsqcup\{c, c'\}$, we can conclude that $f(c) \sqsubseteq f(c')$, using that $c' = \bigsqcup\{c, c'\}$.

($\Leftarrow$): Assume that f is monotone and that $f(\bigsqcup\langle c_i\rangle_{i=0}^{\infty}) \sqsubseteq \bigsqcup\langle f(c_i)\rangle_{i=0}^{\infty}$ holds for all chains $\langle c_i\rangle_{i=0}^{\infty}$ in C. Then we need to prove that for all chains $\langle c_i\rangle_{i=0}^{\infty}$ in $C : \bigsqcup\langle f(c_i)\rangle_{i=0}^{\infty} \sqsubseteq f(\bigsqcup\langle c_i\rangle_{i=0}^{\infty})$. Take an arbitrary chain $\langle c_i\rangle_{i=0}^{\infty}$ in C. Let $X = \{c \mid c \text{ in } \langle c_i\rangle_{i=0}^{\infty}\}$ and let $X' = \{f(c) \mid c \text{ in } \langle c_i\rangle_{i=0}^{\infty}\} = \{f(x) \mid x \in X\}$. To prove: $\bigsqcup X' \sqsubseteq f(\bigsqcup X)$.
Take some $x \in X$. Then $x \sqsubseteq \bigsqcup X$ and thus by monotonicity of f: $f(x) \sqsubseteq f(\bigsqcup X)$. With x having been chosen arbitrarily, we may conclude that $f(\bigsqcup X)$ is an upper bound of X'. Hence, $\bigsqcup X' \sqsubseteq f(\bigsqcup X)$.

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

Monotonicity of Φ

Lemma 2. *(monotonicity of Φ)*

The function Φ as given in definition 34 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 $\phi_i, \phi_j \in S_\perp \to T$. Suppose that $\phi_i \sqsubseteq \phi_j$. 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. Suppose $\mathcal{W}(b)(s) = \perp_{W_\perp}$, then to prove: $\{\perp\} \sqsubseteq \{\perp\}$. This is true by definition 32.
2. Suppose $\mathcal{W}(b)(s) = ff$, then to prove: $\{s\} \sqsubseteq \{s\}$. This is true by definition 32.
3. Suppose $\mathcal{W}(b)(s) = tt$, 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) Suppose $\bot \notin \bigcup_{s' \in \tau'} \phi_i(s')$. Then to prove: $\bigcup_{s' \in \tau'} \phi_i(s') = \bigcup_{s' \in \tau'} \phi_j(s')$. From the assumption that $\bot \notin \bigcup_{s' \in \tau'} \phi_i(s')$, we can conclude that $\bot \notin \phi_i(s')$ for all $s' \in \tau'$. Using the assumption that $\phi_i(s) \sqsubseteq \phi_j(s)$ for all $s \in S_\bot$, 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) Suppose $\bot \in \bigcup_{s' \in \tau'} \phi_i(s')$. Then to prove: $(\bigcup_{s' \in \tau'} \phi_i(s')) \setminus \{\bot\} \subseteq \bigcup_{s' \in \tau'} \phi_j(s')$, i.e. $\bigcup_{s' \in \tau'} (\phi_i(s') \setminus \{\bot\}) \subseteq \bigcup_{s' \in \tau'} \phi_j(s')$. Using the assumption that $\phi_i(s) \sqsubseteq \phi_j(s)$ for all $s \in S_\bot$, we have that for all $s' \in \tau'$, either $\phi_i(s') \setminus \{\bot\} \subseteq \phi_j(s')$ or $\phi_i(s') = \phi_j(s')$, depending on whether $\bot \in \phi_i(s)$. From this we can conclude that $\bigcup_{s' \in \tau'} (\phi_i(s') \setminus \{\bot\}) \subseteq \bigcup_{s' \in \tau'} \phi_j(s')$.

Theorem 2. *(continuity of Φ)* The function Φ as given in definition 34 is continuous, i.e. Φ is monotone and for all chains $\langle \phi_i \rangle_{i=0}^{\infty}$ in $S_\bot \to T$, the following holds:

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

Proof. Monotonicity of Φ was proven in lemma 2. We therefore only need to prove that for all chains $\langle \phi_i \rangle_{i=0}^{\infty}$ in $S_\bot \to T$ and for all $s \in S_\bot$, 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 $S_\bot \to T$ and an arbitrary state $s \in S_\bot$. Then to prove:

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

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

1. Suppose $\mathcal{W}(b)(s) = \bot_{W_\bot}$, then to prove: $\{\bot\} \sqsubseteq \bigsqcup_{i=0}^{\infty} \{\bot\}$, i.e. $\{\bot\} \sqsubseteq \{\bot\}$. This is true by definition 32.
2. Suppose $\mathcal{W}(b)(s) = \mathit{ff}$, then to prove: $\{s\} \sqsubseteq \bigsqcup_{i=0}^{\infty} \{s\}$, i.e. $\{s\} \sqsubseteq \{s\}$. This is true by definition 32.
3. Suppose $\mathcal{W}(b)(s) = \mathit{tt}$, 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 [4], from which we can conclude what needs to be proven.

6 Equivalence of Meta-level Operational and Denotational Semantics

In the previous section, we have given a denotational semantic function for meta-programs and we have proven that this function is well-defined. In section 6.1, we will prove that the denotational semantics for meta-programs is equal to the operational semantics for meta-programs. From this we can conclude that the denotational semantics of the interpreter of section 4 is equal to the operational semantics of this interpreter. As the operational semantics of this interpreter is equal to the operational semantics of object-level 3APL (as proven in section 4.1), the denotational semantics of the interpreter is equal to the operational semantics of 3APL. One could thus argue that we give a denotational semantics for 3APL. This will be discussed in section 6.2.

6.1 Equivalence Theorem

Theorem 3. *($\mathcal{O}_m = \mathcal{M}$)* Let $\mathcal{O}_m : \mathit{Prog} \to (S_\bot \to \wp(S_\bot))$ be the operational semantics of meta-programs (definition 18) and let $\mathcal{M} : \mathit{Prog} \to (S_\bot \to T)$ be the denotational semantics of meta-programs (definition 33). Then, the following equivalence holds for all meta-programs $P \in \mathit{Prog}$ and all mental states $s \in S_\bot$.

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

In this section, we will use $\mathcal{O}$ to denote $\mathcal{O}_m$ and $\mathcal{C}$ to denote $\mathcal{C}_m$.

Proof. Kuiper [9] 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.

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 $\mathcal{C}(P)(s) \in \wp(S^+)$, induction on the *sum* of the length of the computation sequences in $\mathcal{C}(P)(s)$ is applied, thus proving $\mathcal{O}_m(P)(s) = \mathcal{M}(P)(s)$ in this case. In case there is an infinite computation sequence in $\mathcal{C}(P)(s)$ and so $\bot \in \mathcal{O}(P)(s)$, we prove $\mathcal{O}(P)(s) \setminus \{\bot\} \subseteq \mathcal{M}(P)(s)$ by induction on the length of *individual* computation sequences. This yields $\mathcal{O}(P) \sqsubseteq \mathcal{M}(P)$. For the reasons behind this way of proving the result, we refer to Kuiper [9]. Proving $\mathcal{M}(P) \sqsubseteq \mathcal{O}(P)$ by standard techniques then completes the proof.

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

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

Case A: $\bot \notin \mathcal{O}(P)(s)$ 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. $P \equiv \mathit{execute}$
   Let $(\pi, \sigma) \in S$. If $\pi = a; \pi'$ with $a \in \mathsf{BasicAction}$, $\pi' \in \Pi$ and $\mathcal{T}(a, \sigma) = \sigma'$, then the following can be derived directly from definitions 18, 16 and 33.

$$\begin{aligned} \mathcal{O}(\mathit{execute})(\pi, \sigma) &= \kappa(\mathcal{C}(\mathit{execute})(\pi, \sigma)) \\ &= \{(\pi', \sigma')\} \\ &= \mathcal{M}(\mathit{execute})(\pi, \sigma) \end{aligned}$$

   Otherwise $\mathcal{O}(\mathit{execute})(\pi, \sigma) = \emptyset = \mathcal{M}(\mathit{execute})(\pi, \sigma)$.
2. $P \equiv \mathit{apply}(\rho)$
   Let $(\pi, \sigma) \in S$. If $\sigma \models \psi$ and $\pi = \pi_h \circ \pi'$ and if $\rho : \pi_h \mid \psi \rightsquigarrow \pi_b \in \mathsf{Rule}$ then the following can be derived directly from definitions 18, 16 and 33.

$$\begin{aligned} \mathcal{O}(\mathit{apply}(\rho))(\pi, \sigma) &= \kappa(\mathcal{C}(\mathit{apply}(\rho))(\pi, \sigma)) \\ &= \{(\pi_b \circ \pi', \sigma)\} \\ &= \mathcal{M}(\mathit{apply}(\rho))(\pi, \sigma) \end{aligned}$$

   Otherwise $\mathcal{O}(\mathit{apply}(\rho))(\pi, \sigma) = \emptyset = \mathcal{M}(\mathit{apply}(\rho))(\pi, \sigma)$.
3. $P \equiv \texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od}$
   In case $\mathcal{W}(b)(s) = \mathit{ff}$, 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 $\mathcal{W}(b)(s) = \mathit{tt}$. For this, we will need the following addition and variation to lemma's given by Kuiper, which only hold in case $\mathcal{W}(b)(s) = \mathit{tt}$.

$$\begin{aligned} \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) &= \mathcal{O}(P'; \texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) \quad \text{(lemma 7, Kuiper)} \\ \mathcal{M}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) &= \mathcal{M}(P'; \texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) \quad \text{(lemma 13, Kuiper)} \end{aligned}$$

   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 16, Kuiper). From definition 17, we can then conclude the following.

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

So, by induction we have:

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

The proof is then as follows:

$$\begin{array}{ll} & \mathcal{O}(\texttt{while } b \texttt{ do } P' \texttt{ od})(s) \\ = & \mathcal{O}(P'; \texttt{while } b \texttt{ do } P' \texttt{ od})(s) & \text{(lemma 7, above)} \\ = & \mathcal{O}(\texttt{while } b \texttt{ do } P' \texttt{ od}) \circ \mathcal{O}(P')(s) & \text{(lemma 7, Kuiper)} \\ = & \mathcal{O}(\texttt{while } b \texttt{ do } P' \texttt{ od})(\kappa(\mathcal{C}(P')(s))) & \text{(definition 18)} \\ = & \mathcal{M}(\texttt{while } b \texttt{ do } P' \texttt{ od})(\kappa(\mathcal{C}(P')(s))) & \text{(induction hypothesis)} \\ = & \mathcal{M}(\texttt{while } b \texttt{ do } P' \texttt{ od})(\mathcal{O}(P')(s)) & \text{(definition 18)} \\ = & \mathcal{M}(\texttt{while } b \texttt{ do } P' \texttt{ od}) \circ \mathcal{M}(P')(s) & \text{(induction hypothesis)} \\ = & \mathcal{M}(P'; \texttt{while } b \texttt{ do } P' \texttt{ od})(s) & \text{(definition 33)} \\ = & \mathcal{M}(\texttt{while } b \texttt{ do } P' \texttt{ od})(s) & \text{(lemma 13, above)} \end{array}$$

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. $P \equiv$ *execute*
   Equivalence was proven in case A.
2. $P \equiv$ *apply*(ρ)
   Equivalence was proven in case A.
3. $P \equiv \texttt{while } b \texttt{ do } P' \texttt{ od}$
   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:

   $$\begin{array}{ll} \langle s_1, \ldots, s_j \rangle & \in \mathcal{C}(P')(s) \quad \text{and} \\ \langle s_{j+1}, \ldots, s_n \rangle & \in \mathcal{C}(\texttt{while } b \texttt{ do } P' \texttt{ od})(s_j). \end{array}$$

   The following can be derived immediately from the above.

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

   We thus have the following induction hypothesis.

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

   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:

   $$\begin{array}{ll} s_j & \in \mathcal{M}(P')(s) \quad \text{and} \\ s' & \in \mathcal{M}(\texttt{while } b \texttt{ do } P' \texttt{ od})(s_j). \end{array}$$

   From this we can conclude the following, deriving what was to be proven.

   $$\begin{array}{ll} s' \in \mathcal{M}(P'; \texttt{while } b \texttt{ do } P' \texttt{ od})(s) & \text{(definition 33)} \\ s' \in \mathcal{M}(\texttt{while } b \texttt{ do } P' \texttt{ od})(s) & \text{(lemma 13, above)} \end{array}$$

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

1. $P \equiv execute$
   Equivalence was proven in case (1).
2. $P \equiv apply(\rho)$
   Equivalence was proven in case (1).
3. $P \equiv \texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od}$
   We will use induction on the entity $(i, length(P))$ where $length(P)$ denotes the length of the statement P and we use ordering $(i_1, l_1) < (i_2, l_2)$ iff either $i_1 < i_2$ or $i_1 = i_2$ and $l_1 < l_2$. Clearly, $length(P') < length(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})$ holds. Therefore $(i, length(P')) < (i, length(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od}))$ 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 $(i, length(P))$. As $(i, length(P')) < (i, length(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od}))$ and $(i, l) < (i+1, l)$, our induction hypothesis will be:

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

   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:

   $$\begin{array}{lll} \Phi(\phi_i)(s) & \sqsubseteq \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) & \text{i.e. by definition 33} \\ \hat{\phi}_i(\mathcal{M}(P')(s)) & \sqsubseteq \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) & \text{i.e. by lemma 7 above and Kuiper} \\ \hat{\phi}_i(\mathcal{M}(P')(s)) & \sqsubseteq \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(\mathcal{O}(P')(s)) & (*). \end{array}$$

   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 $s' \in S_\bot$. From this, we can conclude that $\bigcup_{s' \in \tau} \phi_i(s') \sqsubseteq \bigcup_{s' \in \tau} \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s')$ (see the proof of monotonicity of Φ, lemma 2), which can be rewritten into what was to be proven $(*)$ using the definitions of $\hat{\phi}_i$ and function composition.

In section 4, we showed that the object-level operational semantics of 3APL is equal to the meta-level operational semantics of the interpreter we specified in definition 19. In section 6.1 we then showed 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 (or operational) semantic function in general takes a statement and an initial state and yields the state (or set of states in case of a non-deterministic language) resulting from executing this statement in the initial state. When viewing 3APL on the object-level, plans are the statements which can be executed, resulting in a change to a belief base. Now let $\Sigma_\bot = \Sigma \cup \{\bot_{\Sigma_\bot}\}$.

A denotational (or operational) semantics for 3APL then, should be a function taking a plan $\pi \in \Pi$ and a belief base $\sigma \in \Sigma_\perp$ and yielding a set of belief bases in $\wp(\Sigma_\perp)$, i.e. a function of type:

$$\Pi \to (\Sigma_\perp \to \wp(\Sigma_\perp)). \tag{4}$$

This type can be rewritten into the following equivalent type[12].

$$(\Pi \times (\Sigma \cup \{\perp_{\Sigma_\perp}\})) \to \wp(\Sigma_\perp) \tag{5}$$

The functions $\mathcal{O}_o$ and $\mathcal{M}(\mathsf{interpreter})$ however, are of type $S_\perp \to \wp(S_\perp)$[13]. These functions thus yield a set of mental states as opposed to yielding a set of belief bases. This can be remedied for $\mathcal{O}_o$ in a natural way by adapting the definition of the computation sequence generating function $\mathcal{C}_o$. In case of the denotational semantics, a function of the desired type could for example be defined as follows.

Definition 36. *($\mathcal{N}$)* 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 33. Then $\mathcal{N} : S_\perp \to \wp(\Sigma_\perp)$ is defined as follows.

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

As $S_\perp = (\Pi \times \Sigma) \cup \{\perp\}$ (see definitions 3 and 16), the function $\mathcal{N}$ is of type:

$$((\Pi \times \Sigma) \cup \{\perp\}) \to \wp(\Sigma_\perp). \tag{6}$$

If we for the moment disregard the possibility of a $\perp$ input, we get the type:

$$(\Pi \times \Sigma) \to \wp(\Sigma_\perp). \tag{7}$$

Disregarding a $\perp_{\Sigma_\perp}$ input, this type is equivalent to the desired type 5. So, if we assume an input mental state $s \in S$, the function $\mathcal{N}$ is of the type we were looking for and it is defined using the denotational semantic function $\mathcal{M}$. 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, for the following reason. As stated in the introduction of section 5, a denotational semantic function in general is a function of type $Prog \to (S_\perp \to \wp(S_\perp))$, i.e. taking a statement in $Prog$ and a state in $S_\perp$ and yielding a set of states in $\wp(S_\perp)$ and, most importantly, it should be defined *compositionally in* $Prog$. A denotational semantic function for 3APL should thus be defined compositionally in Π, i.e. the semantics of for example $a; \pi$ should be defined in terms of the semantics of a and 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

[12] Let f be a function of type 4 and define f^* as $f^*(\pi, \sigma) = (f(\pi))(\sigma)$. The function f^* is of type 5 and has the same behaviour as f by definition.

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

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[14] 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[15].

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.

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

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

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. J. C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, Cambridge, 1998.
15. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.
16. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, 1977.
17. R. Tennent. *Semantics of Programming Languages*. Series in Computer Science. Prentice-Hall International, London, 1991.
18. M. B. van Riemsdijk, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming in Dribble: from beliefs to goals with 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. *IEE 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.