

An AgentSpeak Meta-Interpreter and its Applications

Michael Winikoff

RMIT University, Melbourne, AUSTRALIA
winikoff@cs.rmit.edu.au

Abstract. A *meta-interpreter* for a language can provide an easy way of experimenting with modifications or extensions to a language. We give a meta-interpreter for the AgentSpeak language, prove its correctness, and show how the meta-interpreter can be used to extend the AgentSpeak language and to add features to the implementation.

1 Introduction

A *meta-interpreter* for a given programming language is an interpreter for that language which is written in the same language. For example, a program written in LISP that interprets LISP programs. A distinguishing feature of meta-interpreters (sometimes described as “meta-circular interpreters”) is that certain details of the implementation are not handled directly by the meta-interpreter, but are delegated to the underlying implementation. For example, the original LISP meta-interpreter [1] defines the meaning of the symbol `CAR` (in code being interpreted by the meta-interpreter) in terms of the function `car` provided by the underlying implementation.

Although meta-interpreters can help in understanding a programming language, they do not give complete formal semantics, because certain aspects are delegated to the underlying language. For example, defining `CAR` in terms of `car` allows the meta-interpreter to correctly interpret programs (assuming that the underlying LISP implementation provides a suitable implementation of `car`), but does not shed any light on the meaning of the symbol `CAR`.

Meta-interpreters are useful as a way of easily prototyping extensions or changes to a language. For example, the Erlang language began life as a Prolog meta-interpreter which was then extended [2], and the interpreter for Concurrent Prolog can be seen as an extended Prolog meta-interpreter [3]. Being able to modify the semantics of an agent platform is often essential to researchers experimenting with extensions to agent platforms (e.g. [4, 5]), and we argue that meta-interpreters can provide a much easier way of doing so than modifying the agent platform itself.

A drawback of meta-interpreters is the efficiency overhead of the additional layer of interpretation. However, this may not be significant in a prototype if the aim is to explore language design, rather than develop software of any significant size. It has also been suggested that *partial evaluation* could be used to “evaluate away” the meta-interpreter given a meta-interpreter and a program that it is to interpret [6].

In this paper we present a meta-interpreter for the *AgentSpeak*¹ agent-oriented programming language [7]. Although meta-interpreters exist for a range of programming languages, to the best of our knowledge this is the first meta-interpreter for an agent-oriented programming language.

Given the meta-interpreter for AgentSpeak, we then show a number of ways in which it can be modified for various purposes such as extending the language or adding functionality. The extensions that we present are very simple to implement: most involve the addition or change of a very small number of lines of code, and the code is written in AgentSpeak itself. By contrast, other approaches for making agent platforms extensible, such as PARADIGMA [8] and the Java Agent Framework², require the user to change languages to the implementation language and to delve into the implementation which includes both high-level control issues (what is the sequence of events), and lower-level representation issues (e.g. how are beliefs represented in terms of Java objects). Another approach, that is closer to the use of meta-interpreters, is that of Dastani et. al. [9] where the essential control cycle of an agent is broken down into primitives such as executing a goal, selecting a rule, etc. and a customised agent deliberation cycle is programmed in terms of these primitives using a meta-language. Compared with our approach, a disadvantage is the need to introduce a distinct meta-language with its own semantics. An interesting direction for future work would be to add similar primitives to the AgentSpeak language: this would extend the range of modifications that could be easily done using a meta-interpreter, and would avoid the need for a distinct meta-language by using AgentSpeak as its own meta-language.

The remainder of this paper is structured as follows. In section 2 we briefly summarise AgentSpeak's syntax, present (another) formal operational semantics for the language, and discuss a number of issues with the language. In section 3 we present a meta-interpreter for AgentSpeak and in section 4 show a number of modifications to the meta-interpreter. The implementation and performance of the meta-interpreter are briefly examined in section 5 and we then conclude in section 6.

2 AgentSpeak

Agent programming languages in the BDI tradition define an agent's behaviour by providing a library of recipes ("plans"), each indicating (i) the goal that it aims to achieve (modelled as a triggering event), (ii) the situations in which the plan should be used (defined using a logical condition, the "context condition"), and (iii) a plan body. Given a collection of plans, the following execution cycle is used:

1. An event is posted (which can be used to model a new goal being adopted by the agent, as well as new information ("percepts"), or a significant change that needs to be responded to).
2. The plans that handle that event are collected and form the *relevant* plan set.

¹ Properly the language is called "AgentSpeak(L)", but in the remainder of the paper we shall refer to it as "AgentSpeak".

² <http://dis.cs.umass.edu/research/jaf/>

3. A plan with a true context condition is *applicable*. An applicable plan is selected and its body is run.
4. If the plan fails, then an alternative applicable plan is found and its body is run. This repeats until either a plan succeeds, or there are no more applicable plans, in which case failure is propagated.

There are a number of agent programming languages in the BDI tradition, such as dMARS [10], JAM [11], PRS [12, 13], UM-PRS [14], and JACK [15]. The language AgentSpeak [7] was proposed by Anand Rao in an attempt to capture the common essence of existing BDI programming languages whilst having precisely defined semantics. Although Rao’s formal semantics are incomplete, the work has inspired a number of implementations of AgentSpeak such as AgentTalk³, an implementation based on SIM_AGENT [16], an implementation in Java that is designed to run on hand-held devices [17], and the Java-based Jason⁴.

2.1 Syntax

An agent program (denoted by Π) consists of a collection of plan clauses of the form⁵ $e : c \leftarrow P$ where e is an event⁶, c is a context condition (a logical formula over the agent’s beliefs) which must be true in order for the plan to be applicable and P is the plan body. A condition, C , is a logical formula over belief terms⁷ (where b is a belief atom). The plan body P is built up from the following constructs. We have primitive actions (*act*), operations to add ($+b$) and delete ($-b$) beliefs, a test for a condition ($?c$), and posting an event ($!e$). These can be sequenced ($P_1; P_2$). These cases are summarised below.

$$C ::= b \mid C \wedge C \mid C \vee C \mid \neg C \mid \exists x.C$$

$$P ::= act \mid +b \mid -b \mid ?c \mid !e \mid P; P$$

In addition to these constructs which can be used by the programmer, we define a number of constructs that are used in the formal semantics. These are:

- *true* which is the empty step which always succeeds.
- *fail* which is a step which always fails.
- $P_1 \triangleright P_2$ is sequential disjunction: P_1 is run, and if it succeeds then P_2 is discarded and $P_1 \triangleright P_2$ has succeeded. Otherwise, P_2 is executed.
- $\langle \Delta \rangle$, where Δ is a set of plan bodies ($P_1 \dots P_n$) which is used to represent a set of possible alternatives. It is executed by selecting a P_i from Δ and executing it (with the remainder of Δ being kept as possible alternatives in case P_i fails).

³ <http://www.cs.rmit.edu.au/~winikoff/agenttalk>

⁴ <http://jason.sourceforge.net/>

⁵ An omitted c is equivalent to *true*, i.e. $e \leftarrow P \equiv e : true \leftarrow P$.

⁶ In Rao’s original formulation, e is one of $+!g, +?g, -!g, -?g, +b, -b$ corresponding respectively to the addition of an achievement or query goal, the deletion of an achievement or query goal, the addition of a belief, and the deletion of a belief. In practice it is rare for any other than the $+!g$ form to be used. Indeed, Rao’s semantics only address this event form, and so in the remainder of this paper we write e in the heads of clauses as shorthand for $+!e$. Similarly, we write e as shorthand for $!e$ in the bodies of clauses.

⁷ In Rao’s formulation only conjunctions of literals were permitted.

2.2 Semantics

In the years since Rao introduced AgentSpeak a number of authors have published (complete) formal semantics for the language. The specification language Z (“Zed”) was used to formally specify the essential execution cycle of AgentSpeak [18], and an operational semantics for AgentSpeak was given by Moreira and Bordini [19]. The operational semantics that we give here is in the style of Plotkin’s Structural Operational Semantics [20], and is based on the semantics of the CAN notation [21], which is a superset of AgentSpeak. Unlike the previous semantics, it includes failure handling: if a plan fails, then alternative plans are tried (step 4 of the execution cycle at the start of this section).

The semantics assume that operations exist that check whether a condition follows from a belief set ($B \models c$), that add a belief to a belief set ($B \cup \{b\}$), and that delete a belief from a belief set ($B \setminus \{b\}$). In the case of beliefs being a set of ground atoms these operations are respectively consequence checking, and set addition/deletion. Traditionally, agent systems have represented beliefs as a set of ground atoms, but there is no reason why more sophisticated representations and reasoning mechanisms (such as belief revision, Bayesian reasoning etc.) could not be used.

We define a basic configuration $S = \langle B, P \rangle$ where B is the beliefs of the agent and P is the plan body being executed (i.e. the intention). A transition $S_0 \longrightarrow S_1$ specifies that executing S_0 a single step yields S_1 . We define $S_0 \xrightarrow{*} S_n$ in the usual way: S_n is the result of zero or more single step transitions. The transition relation is defined using

rules of the form $\frac{S' \longrightarrow S_r}{S \longrightarrow S'}$ or of the form $\frac{S' \longrightarrow S_r}{S \longrightarrow S'_r}$; the latter are conditional with the top (numerator) being the premise and the bottom (denominator) being the conclusion. In order to make the presentation more readable we use the convention that where a component of S isn’t mentioned it is the same in S and S' (and in S_r and S'_r). We also assume that B refers to the agent’s beliefs, and elide angle brackets. Thus each of the following rules on the left is shorthand for the corresponding right rule.

$$\frac{B \models c}{?c \longrightarrow true} ?c \quad \frac{B \models c}{\langle B, ?c \rangle \longrightarrow \langle B, true \rangle} ?c$$

$$\frac{P_1 \longrightarrow P'}{P_1; P_2 \longrightarrow P'; P_2} ; \quad \frac{\langle B, P_1 \rangle \longrightarrow \langle B', P' \rangle}{\langle B, P_1; P_2 \rangle \longrightarrow \langle B', P'; P_2 \rangle} ;$$

The first rule above specifies that the condition test $?c$ transitions to true if the condition c is a consequence of the agent’s beliefs ($B \models c$). The second rule specifies that $P_1; P_2$ transitions to $P'; P_2$ where P' is the result of a single execution step of P_1 . The full set of rules are given in figure 1.

We extend simple configurations (which correspond to a single thread of execution within an agent) to agent configurations $S_A = \langle N, B, Ps \rangle$ which consist of a name, a single (shared) belief set, and a *set* of intentions (executing plans). The following rule defines the operational semantics over agent configurations in terms of the operational

semantics over simple configurations.

$$\frac{P = \mathcal{S}_{\mathcal{I}}(\Gamma) \quad \langle B, P \rangle \longrightarrow \langle B', P' \rangle}{\langle N, B, \Gamma \rangle \longrightarrow \langle N, B', (\Gamma \setminus \{P\}) \cup \{P'\} \rangle} \text{Agent}$$

Note that there is non-determinism in AgentSpeak and in these semantics, e.g. the choice of plan to execute from a set of applicable plans. In addition, the *Agent* rule non-deterministically selects an executing plan. Instead of resolving these non-deterministic choices with an arbitrary policy, AgentSpeak defines selection functions $\mathcal{S}_{\mathcal{I}}$, $\mathcal{S}_{\mathcal{O}}$, and $\mathcal{S}_{\mathcal{E}}$ which respectively select a plan from the set of executing plans, an option from the set of applicable plans, and an event from the set of events. These are assumed to be provided by an AgentSpeak implementation and could also be replaced by the programmer.

Two of these selection functions ($\mathcal{S}_{\mathcal{I}}$ and $\mathcal{S}_{\mathcal{O}}$) are used in our formal semantics. The third selection function ($\mathcal{S}_{\mathcal{E}}$) is not used. The reason is that AgentSpeak splits event processing into two steps: adding the event to a set of events, and then selecting an event from the set and adding an intention corresponding to the applicable plans for that event. Since neither of these steps results in any changes to the agent's beliefs or to its environment, these two steps can be merged into a single atomic step without any loss of generality, i.e. events in AgentSpeak are eliminable (which has been formally proven by Hindriks et. al. [22]), and eliminating events simplifies the formal semantics.

$$\begin{array}{c} \frac{B \models c}{?c \longrightarrow true} ?c_t \quad \frac{B \not\models c}{?c \longrightarrow fail} ?c_f \quad \frac{}{act \longrightarrow true} act \\ \frac{B, +b \longrightarrow B \cup \{b\}, true}{\Delta = \{P_i \theta | (t_i : c_i \leftarrow P_i) \in \Pi \wedge t_i = +!e \wedge B \models c_i \theta\}} +b \quad \frac{B, -b \longrightarrow B \setminus \{b\}, true}{!e \longrightarrow \langle \Delta \rangle} -b \\ \Delta = \{P_i \theta | (t_i : c_i \leftarrow P_i) \in \Pi \wedge t_i = +!e \wedge B \models c_i \theta\} \quad Ev \\ \frac{P_1 \longrightarrow P'}{P_1; P_2 \longrightarrow P'; P_2} ; \quad \frac{}{true; P \longrightarrow P} ;t \quad \frac{}{fail; P \longrightarrow fail} ;f \\ \frac{}{\langle \rangle \longrightarrow fail} Sel_f \quad \frac{P_i = \mathcal{S}_{\mathcal{O}}(\Delta)}{\langle \Delta \rangle \longrightarrow P_i \triangleright \langle \Delta \setminus \{P_i\} \rangle} Sel \\ \frac{P_1 \longrightarrow P'}{P_1 \triangleright P_2 \longrightarrow P' \triangleright P_2} \triangleright \quad \frac{}{true \triangleright P \longrightarrow true} \triangleright_t \quad \frac{}{fail \triangleright P \longrightarrow P} \triangleright_f \\ \frac{P = \mathcal{S}_{\mathcal{I}}(\Gamma) \quad \langle B, P \rangle \longrightarrow \langle B', P' \rangle}{\langle N, B, \Gamma \rangle \longrightarrow \langle N, B', (\Gamma \setminus \{P\}) \cup \{P'\} \rangle} \text{Agent} \\ \frac{P = \mathcal{S}_{\mathcal{I}}(\Gamma) \quad P \in \{true, fail\}}{\langle N, B, \Gamma \rangle \longrightarrow \langle N, B, (\Gamma \setminus \{P\}) \rangle} \text{Agent}_{t_f} \\ \frac{e \text{ is a new external event}}{\langle N, B, \Gamma \rangle \longrightarrow \langle N, B, \Gamma \cup \{!e\} \rangle} \text{Agent}_{ext} \end{array}$$

Fig. 1. Operational Semantics for AgentSpeak

2.3 Issues with AgentSpeak

The semantics of AgentSpeak as presented by Rao [7] are incomplete in a number of ways.

One area of incompleteness is failure recovery. All of the platforms that AgentSpeak was intended to model provide failure handling by trying alternative plans if a plan fails. This form of failure handling is based on the idea that for a given goal the relevant plans offer alternative means of achieving the goal, and that if one way of achieving a goal fails, alternative ways should be considered⁸. However, because AgentSpeak [7] focuses on describing the execution cycle around plan selection, it does not explicitly specify what should be done when a plan fails. This omission has led to certain implementations (such as Jason) not providing this form of failure handling⁹. We regard the omission of failure handling from Rao’s semantics as unfortunate, since it has allowed implementations of AgentSpeak to be consistent with the original AgentSpeak paper, but to be incompatible with each other, and with other BDI-platforms. For example, although the semantics of Jason [19] are consistent with Rao’s semantics [7], Jason’s failure handling is quite different from that of other BDI-platforms such as dMARS [10], JAM [11], PRS [12, 13], UM-PRS [14] and JACK [15].

A more subtle issue concerns the context condition of plans, specifically when are they evaluated? There are two possibilities: one can either evaluate the context conditions of all relevant plans at once giving a set of applicable plans, or one can evaluate relevant plans one at a time. The former – “eager” evaluation – is simpler semantically, but the latter – “lazy” evaluation – has the advantage that when a plan is considered for execution, its context condition is evaluated in the current state of the world, not in the state of the world that held when the event was first posted. The execution cycle at the start of this section is deliberately ambiguous about when context conditions are evaluated because BDI platforms differ in their handling of this issue. For example, JAM is eager whereas JACK is lazy, and in the CAN notation [21] each plan has an eager context condition *and* a lazy context condition. Since Rao’s semantics for AgentSpeak specify eager evaluation [7, Figure 1], this is what our rule for *Ev* specifies.

Another issue concerns multiple solutions to context conditions. Suppose that we have a program clause $e : c \leftarrow P$ and that given the agent’s current beliefs there are two different ways of satisfying c which give different substitutions θ_1 and θ_2 . Should there be a single applicable plan $P\theta_i$ (where i is arbitrarily either 1 or 2), or should there be two applicable plan (instances), $P\theta_1$ and $P\theta_2$? Again, there is no consensus among BDI platforms, for example, JAM doesn’t support multiple solutions to context conditions whereas JACK does. The semantics of AgentSpeak specifies multiple substitutions¹⁰: [7, Figure 1] computes the applicable plans O_e as $O_e =$

⁸ This is not backtracking in the logic programming sense because there is no attempt to undo the actions of a failed plan.

⁹ Jason provides an alternative form of failure handling where failure of a plan posts a failure event of the form $!g$ and this event can be handled by an “exception handling” plan.

¹⁰ But note that AgentSpeak’s semantics are inconsistent as to whether θ is unique: although Figure 1 says that θ is “an” applicable unifier, Definition 10 says that θ is “**the correct answer substitution**” (emphasis added).

$\{p\theta|\theta$ is an applicable unifier for event e and plan $p\}$. Our semantics therefore allows multiple substitutions, providing an applicable plan instance for each substitution.

Finally, a very minor syntactical issue that is nonetheless worth mentioning, is that having to write $!e$ in the bodies of plans is error-prone: it is too easy to write e by mistake.

3 An AgentSpeak Meta Interpreter

Logic programming languages have particularly elegant meta-interpreters. For example, the meta-interpreter for Prolog is only a few lines long [23, section 17.2] and follows the pattern of interpreting connectives and primitives in terms of themselves (lines 1 & 2) and interpreting an atom by non-deterministically selecting a program clause and solving it (line 3).

1. solve(true) \leftarrow true.
2. solve((A,B)) \leftarrow solve(A) , solve(B).
3. solve(A) \leftarrow clause(A,B) , solve(B).

Meta-interpreters for other logic programming languages can be developed along similar lines, for example the logic programming language *Lygon*, which is based on linear logic, has a meta-interpreter along similar lines [24, section 5.6].

A meta-interpreter for AgentSpeak can also be defined similarly:

1. solve(Act) : isAction(Act) \leftarrow do(Act).
2. solve(true) \leftarrow true.
3. solve(fail) \leftarrow fail.
4. solve($-B$) \leftarrow $-B$.
5. solve($+B$) \leftarrow $+B$.
6. solve($?C$) \leftarrow $?C$.
7. solve($P_1 ; P_2$) \leftarrow solve(P_1) ; solve(P_2).
8. solve($!E$) : clause($+!E,G,P$) \wedge isTrue(G) \leftarrow solve(P).

We assume that the agent has a collection of beliefs of the form $clause(H,G,P)$ which represent the program being interpreted.

In order for this meta-interpreter to work the underlying AgentSpeak implementation needs to support multiple solutions for context conditions. This is needed because the meta-interpreter's last clause, where alternative plans are retrieved, needs to have multiple instances corresponding to different solutions to $clause$.

Lemma 1. *If $P \xrightarrow{*} X$ and $X \notin \{true, fail\}$ then there exists Y such that $X \longrightarrow Y$.*

Theorem 1. *The above meta-interpreter is correct. Formally, given an AgentSpeak program Π and its translation into a collection of clause beliefs (denoted by $\hat{\Pi}$), the execution of an intention P with program Π is mirrored¹¹ by the execution of the intention*

¹¹ We don't precisely define this due to lack of space. The formal concept corresponding to this is bisimulation.

$\text{solve}(P)$ with program $\widehat{\Pi} \cup \mathcal{M}$, where \mathcal{M} denotes the above meta-interpreter. By “mirrored” we mean that $\langle B, P \rangle \xrightarrow{*} \langle B', R \rangle$ with $R \in \{\text{true}, \text{fail}\}$ and with a given sequence of actions A , if and only if $\langle B, \text{solve}(P) \rangle \xrightarrow{*} \langle B', R \rangle$ with the same sequence of actions A . We use $S \Rightarrow S'$ where $S' = \langle A, B, R \rangle$ as shorthand for “ $S \xrightarrow{*} \langle B, R \rangle$ with the sequence of actions A ”. In the following proof we use \emptyset to denote the empty sequence and \oplus to denote sequence concatenation.

Proof (sketch): Proof by induction on the length of the derivation, we consider three cases since the other base cases are analogous to the first base case.

- Firstly, consider a base case, $\langle B, +b \rangle \rightarrow \langle B \cup \{b\}, \text{true} \rangle$. Given the meta-interpreter clause $\text{solve}(+B) \leftarrow +B$ we have $\langle B, \text{solve}(+b) \rangle \rightarrow \langle B, \{\!\!\{b\}\!\!\} \rangle \rightarrow \langle B, +b \triangleright \emptyset \rangle \rightarrow \langle B \cup \{b\}, \text{true} \triangleright \emptyset \rangle \rightarrow \langle B \cup \{b\}, \text{true} \rangle$. Since both sequences of transitions involve no actions we have that $\langle B, +b \rangle \Rightarrow \langle \emptyset, B \cup \{b\}, \text{true} \rangle$ and $\langle B, \text{solve}(+b) \rangle \Rightarrow \langle \emptyset, B \cup \{b\}, \text{true} \rangle$. Since both sequences of transitions are deterministic (no other transitions are possible) we have that $\langle B, +b \rangle \Rightarrow S$ iff $\langle B, \text{solve}(+b) \rangle \Rightarrow S$ as required.
- Now consider the case of $P_1; P_2$. We assume by the inductive hypothesis that $\langle B, P_1 \rangle \Rightarrow S_1$ iff $\langle B, \text{solve}(P_1) \rangle \Rightarrow S_1$ (where $S_1 = \langle A_1, B_1, R_1 \rangle$) and similarly for P_2 . There are then two cases: $P_1 \xrightarrow{*} \text{true}$ and $P_1 \xrightarrow{*} \text{fail}$. In the first case we have that $\langle B, P_1; P_2 \rangle \Rightarrow \langle A_1, B_1, \text{true}; P_2 \rangle$ and then $\langle B_1, \text{true}; P_2 \rangle \rightarrow \langle B_1, P_2 \rangle \Rightarrow \langle A_2, B_2, R_2 \rangle$. We also have that $\text{solve}(P_1; P_2) \rightarrow \{\!\!\{\text{solve}(P_1); \text{solve}(P_2)\}\!\!\} \rightarrow \text{solve}(P_1); \text{solve}(P_2) \triangleright \emptyset$, that $\langle B, \text{solve}(P_1); \text{solve}(P_2) \triangleright \emptyset \rangle \Rightarrow \langle A_1, B_1, \text{true}; \text{solve}(P_2) \triangleright \emptyset \rangle$, and that $\langle B_1, \text{true}; \text{solve}(P_2) \triangleright \emptyset \rangle \rightarrow \langle B_1, \text{solve}(P_2) \triangleright \emptyset \rangle \Rightarrow \langle A_2, B_2, R_2 \triangleright \emptyset \rangle$. Now, regardless of whether R_2 is true or fail this transitions to R_2 since $\text{true} \triangleright \emptyset \rightarrow \text{true}$ and $\text{fail} \triangleright \emptyset \rightarrow \emptyset \rightarrow \text{fail}$. Hence, in the first case, where $P_1 \xrightarrow{*} \text{true}$, we have that $\langle B, P_1; P_2 \rangle \Rightarrow \langle A_1 \oplus A_2, B_2, R_2 \rangle$ and that $\langle B, \text{solve}(P_1; P_2) \rangle \Rightarrow \langle A_1 \oplus A_2, B_2, R_2 \rangle$. In the second case, $P_1 \xrightarrow{*} \text{fail}$, we have that $\langle B, P_1; P_2 \rangle \Rightarrow \langle A_1, B_1, \text{fail}; P_2 \rangle$. We then have from the semantics that $\text{fail}; P_2 \rightarrow \text{fail}$ and hence that $\langle B, P_1; P_2 \rangle \Rightarrow \langle A_1, B_1, \text{fail} \rangle$. We also have that $\text{solve}(P_1; P_2) \rightarrow \{\!\!\{\text{solve}(P_1); \text{solve}(P_2)\}\!\!\} \rightarrow \text{solve}(P_1); \text{solve}(P_2) \triangleright \emptyset$, that $\langle B, \text{solve}(P_1); \text{solve}(P_2) \triangleright \emptyset \rangle \Rightarrow \langle A_1, B_1, \text{fail}; \text{solve}(P_2) \triangleright \emptyset \rangle$, and that this then transitions to $\text{fail} \triangleright \emptyset \rightarrow \emptyset \rightarrow \text{fail}$, i.e. that $\langle B, \text{solve}(P_1; P_2) \rangle \Rightarrow \langle A_1, B_1, \text{fail} \rangle$. Thus, in both cases the execution of $P_1; P_2$ and $\text{solve}(P_1; P_2)$ mirror each other.
- We now consider the clause $\text{solve}(!E) : \text{clause}(+!E, G, P) \wedge \text{isTrue}(G) \leftarrow \text{solve}(P)$. We have that if Π contains a program clause $+!e : c \leftarrow p$ then $!e \rightarrow \{\!\!\{\Delta\}\!\!\}$ where $\Delta = \{P_i \theta \mid (t_i : c_i \leftarrow P_i) \in \Pi \wedge t_i = +!e \wedge B \models c_i \theta\}$. We also have $\text{solve}(!e) \rightarrow \{\!\!\{\Omega\}\!\!\}$ where Ω is the instances of the clause in the meta-interpreter (since this is the only clause applicable to solving $!e$), i.e. $\Omega = \{\text{solve}(P) \theta \mid B \models (\text{clause}(+!e, c_i, P) \wedge \text{isTrue}(c_i)) \theta\}$. Since for each clause $t_i : c_i \leftarrow P_i$ in Π there is an equivalent clause (t_i, c_i, P_i) in $\widehat{\Pi}$ we have that $(t_i : c_i \leftarrow P_i) \in \Pi \wedge t_i = +!e$ whenever $B \models \text{clause}(+!e, c_i, P)$. We also have that $B \models c_i \theta$ whenever $B \models \text{isTrue}(c_i) \theta$ (assuming a correct implementation of isTrue), thus Ω has the same alternatives as Δ , more precisely $\Omega = \{\text{solve}(P) \mid P \in \Delta\}$. Hence $!e \rightarrow \{\!\!\{\Delta\}\!\!\}$ and $\text{solve}(!e) \xrightarrow{*} \{\!\!\{\text{solve}(P) \mid P \in \Delta\}\!\!\}$. Once a given P_i is selected from

Δ (respectively Ω) we have by the induction hypothesis that $\langle B, P_i \rangle \Rightarrow R_i$ iff $\langle B, \text{solve}(P_i) \rangle \Rightarrow R_i$ which is easily extended to show that $\langle B, P_i \triangleright (\Delta \setminus \{P_i\}) \rangle \Rightarrow R$ iff $\langle B, \text{solve}(P_i) \triangleright (\Omega \setminus \{\text{solve}(P_i)\}) \rangle \Rightarrow R$, and hence, provided that $\mathcal{S}_O(\Omega) = \text{solve}(\mathcal{S}_O(\Delta))$, that $\langle B, !e \rangle \Rightarrow R$ iff $\langle B, \text{solve}(!e) \rangle \Rightarrow R$ as desired.

■

4 Variations on a Theme

In this section we present a number of variations of the meta-interpreter which extend the AgentSpeak language in various ways or add functionality to the implementation. The key point here is that these modifications are very easy to implement by changing the meta-interpreter. We invite the reader to consider how much work would be involved in making each of these modifications to their favourite agent platform by modifying the underlying implementation . . .

4.1 Debugging

Just as with any form of software, agent systems need to be debugged. Unlike debugging logic programs, multi-agent systems offer additional challenges to debugging due to their concurrency, and due to the use of interaction between agents, i.e. debugging a MAS involves debugging multiple agents, not just a single agent.

One approach to debugging agent interaction is to use interaction protocols that have been produced as part of the design process. An additional “monitoring” agent is added to the system. This agent eavesdrops on conversations in the system and checks that the agents in the system are following the interaction protocols that they are supposed to follow [5, 25].

In order for the monitoring agent to be able to eavesdrop on conversations all agents in the system need to send the monitoring agent copies of all messages that they send. This can be done manually, by changing the code of each agent. However, it is better (and more reliable) to do this by modifying the behaviour of the *send* primitive. Modifying the behaviour of a primitive using a meta-interpreter is quite simple: one merely modifies the existing clause that executes actions to exclude the primitive in question and adds an additional clause that provides the desired behaviour:

- 1a. $\text{solve}(\text{Act}) : \text{Act} \neq \text{send}(\text{R}, \text{M}) \leftarrow \text{do}(\text{Act}).$
- 1b. $\text{solve}(\text{send}(\text{R}, \text{M})) \leftarrow ?\text{myID}(\text{I}) ; \text{send}(\text{monitor}, \text{msg}(\text{I}, \text{R}, \text{M})) ; \text{send}(\text{R}, \text{M}).$

Debugging the internals of agents can be done by enhancing the meta-interpreter in the same ways that one would enhance a Prolog meta-interpreter to aid in debugging [23, Section 17.2 & 17.3]. For example, it is easy to modify a meta-interpreter to trace through the computation. Another possibility is to modify the meta-interpreter to build up a data structure that captures the computation being performed. Once the computation has been completed the resulting data structure can be manipulated in various ways. Finally, another possible modification, suggested by one of the reviewers, is that the interpreter could be modified to send messages to a monitoring agent whenever the agent changes its beliefs:

4. $\text{solve}(-B) \leftarrow -B ; \text{send}(\text{monitor}, \text{delbelief}(B))$.
5. $\text{solve}(+B) \leftarrow +B ; \text{send}(\text{monitor}, \text{addbelief}(B))$.

4.2 Failure handling

The meta-interpreter presented in the previous section delegates the handling of failure to the underlying implementation. However, if we want to change the way in which failure is handled, then we need to “take control” of failure handling. This involves extending the meta-interpreter to handle failure explicitly, which can be done by adding the following clause:

9. $\text{solve}(!E) \leftarrow \text{fail}$.

This additional clause applies the default failure handling rule which simply fails, but it does provide a “hook” where we can insert code to deal with failure. For example, the code could call a planner to generate alternative plans. Note that this clause applies to any intention, and so it must be selected after the other clauses have been tried and failed. For example, if the selection function (\mathcal{S}_O) selects clauses in the order in which they are listed in the program text then this clause should come last.

Another possible response to failure is to consider that perhaps the agent lacks the know-how to achieve the goal in question. One possible source for additional plans that might allow the agent to achieve its goal is other (trusted) agents [26]. Adding additional plans at run-time is difficult to do in compiled implementations, but is very easy to do using a meta-interpreter: since the program is stored as a belief set one simply adds to this belief set. Clause 9 below is intended as a replacement for the failure handling clause above. Once a plan is received it is stored, and then used immediately (*useClause*).

9. $\text{solve}(!E) \leftarrow !\text{getPlan}(E, H, G, B) ; +\text{clause}(H, G, B) ; !\text{useClause}(G, B)$.
10. $\text{getPlan}(E, H, G, B) : \text{trust}(\text{Agent}) \leftarrow \text{send}(\text{Agent}, \text{getPlan}(E)) ; \text{receive}(\text{plan}(H, G, B))$.
11. $\text{useClause}(G, B) : \text{isTrue}(G) \leftarrow \text{solve}(B)$.

4.3 Making selection explicit

AgentSpeak defines a number of selection functions that are used to select which event to process (\mathcal{S}_E), which intention to execute next (\mathcal{S}_I), and which plan (option) to use (\mathcal{S}_O). In some implementations, such as Jason, these selection functions can be replaced with user-provided functions. However, other implementations may not allow easy replacement of the provided default selection functions. Even if the underlying implementation does allow for the selection functions to be replaced, using a meta-interpreter might be easier since it allows the selection functions to be written in AgentSpeak rather than in the underlying implementation language (for example in Jason user-provided selection functions are written in Java). By extending the meta-interpreter to make the selection of plans explicit we can override the provided defaults regardless of whether the implementation provides for this.

Extending the meta-interpreter to do plan selection (i.e. selecting the option, \mathcal{S}_O), explicitly is done as follows. The key idea is that we add an additional argument to *solve*

which holds the alternative options. Then instead of *solve(!E)* having multiple instances corresponding to different options, it collects all of the options into a set of alternatives (using *options*), selects an option (using the user-provided *select*), and solves it. The set of alternatives is ignored by *solve*, except where failure occurs, in which case we explicitly handle it (lines 9 and 10) by selecting an alternative from the set of remaining alternatives and trying it. If there are no alternatives remaining then fail (line 10).

0. $\text{solve}(P) \leftarrow \text{solve}(P, [])$.
1. $\text{solve}(\text{Act}, _) : \text{isAction}(\text{Act}) \leftarrow \text{do}(\text{Act})$.
- ... *Similarly, add an extra argument to solve for the other clauses*
7. $\text{solve}((P_1; P_2), Os) \leftarrow \text{solve}(P_1, Os) ; \text{solve}(P_2, Os)$.
8. $\text{solve}(!E, _) \leftarrow ?\text{options}(E, Os) ; ?\text{select}(I, Is, Os) ; \text{solve}(I, Is)$.
9. $\text{solve}(B, Os) : Os \neq [] \leftarrow ?\text{select}(I, Is, Os) ; \text{solve}(I, Is)$.
10. $\text{solve}(B, Os) : Os = [] \leftarrow \text{fail}$.
11. $\text{options}(E, Os) \leftarrow \text{find all solutions to } \text{clause}(+!E, G, P) \wedge \text{isTrue}(G) \text{ and return the values of } P \text{ in } Os$. In Prolog this could be written as $\text{findall}(P, \text{appClause}(E, P), Os)$ where $\text{appClause}(E, B) \leftarrow \text{clause}(+!E, G, B) \wedge \text{isTrue}(G)$.
12. $\text{select}(I, Is, Os) \leftarrow \text{select an intention } I \text{ from } Os$ (Is is the remaining options, i.e. $Is = Os \setminus \{I\}$)

4.4 A richer plan language

The bodies of plans in AgentSpeak are sequences of primitives (actions, belief manipulation etc.). This is fairly limited, and it can be useful to extend the language with additional constructs such as disjunction and iteration:

9. $\text{solve}(\text{if}(C, P_1, P_2)) : \text{isTrue}(C) \leftarrow \text{solve}(P_1)$.
10. $\text{solve}(\text{if}(C, P_1, P_2)) : \neg \text{isTrue}(C) \leftarrow \text{solve}(P_2)$.
11. $\text{solve}(\text{while}(C, P)) : \text{isTrue}(C) \leftarrow \text{solve}(P) ; \text{solve}(\text{while}(C, P))$.
12. $\text{solve}(\text{while}(C, P)) : \neg \text{isTrue}(C) \leftarrow \text{true}$.

5 Implementation

The meta-interpreter described in section 3 has been implemented and tested. Since Jason doesn't support failure handling and AgentTalk doesn't support multiple solutions to a context condition, we have implemented a simple AgentSpeak interpreter in order to be able to test the meta-interpreter. This simple interpreter runs under Prolog and can be found at <http://www.cs.rmit.edu.au/~winikoff/AS>.

In addition to enabling the meta-interpreter to be tested, not just proven correct¹², the implementation allowed us to quantify the efficiency overhead associated with the additional layer of interpretation introduced by the meta-interpreter.

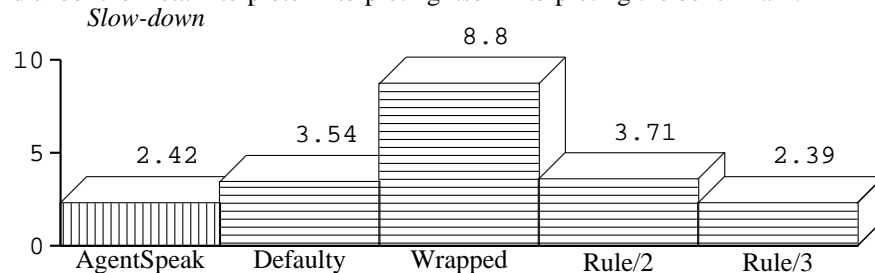
In order to measure the efficiency overhead incurred by the meta-interpreter we use a simple benchmark program. This program controls a hypothetical robot who is

¹² "Beware of bugs in the above code; I have only proved it correct, not tried it" (Donald Knuth, <http://www-cs-faculty.stanford.edu/~knuth/faq.html>)

travelling along a one-dimensional track (perhaps a train track?) containing obstacles which need to be cleared. The robot is given a list of obstacle locations which need to be cleared, and it in turn travels to each obstacle, picks up the obstacle, returns to its starting point and disposes of the obstacle. Since we are interested in the relative efficiency of the program running with and without the meta-interpreter, the details of the program (given below) are not particularly important.

1. `move` \leftarrow `message('moving towards obstacle')`.
2. `moveback` \leftarrow `message('moving back towards base')`.
3. `return(0)` \leftarrow `true`.
4. `return(N) : N > 0 \wedge N1 = N - 1` \leftarrow `!moveback ; !return(N1)`.
5. `get(0)` \leftarrow `true`.
6. `get(N) : N > 0 \wedge N1 = N - 1` \leftarrow `!move ; !get(N1)`.
7. `collect([])` \leftarrow `true`.
8. `collect([X|Xs])` \leftarrow `!get(X) ; message('pickup') ; !return(X) ; message('dispose') ; !collect(Xs)`.
9. `collect30` \leftarrow `!collect([1,2,3,4,5,6,7 . . . ,28,29,30])`.

Handling the event `collect30` with the AgentSpeak interpreter (i.e. without the meta-interpreter) took 169 milliseconds¹³ whereas the same program run with the meta-interpreter took 403 milliseconds. The graph below depicts the slow-down factor ($403/169 = 2.42$) compared with the slow-down factor for various Prolog meta-interpreters reported by O'Keefe [27, Page 273]. The comparison between our slow-down factor and O'Keefe's should be taken only as a rough indication that the overhead incurred by the AgentSpeak meta-interpreter is comparable to that of a carefully-engineered Prolog meta-interpreter. There are too many differences between our measurement and O'Keefe's measurements to allow much significance to be read into the results, e.g. the Prolog implementations are different, the underlying hardware is different, and O'Keefe measured the time to run a naive reverse benchmark under the (Prolog) meta-interpreter and under the meta-interpreter interpreting itself interpreting the benchmark.



6 Conclusion

We presented an AgentSpeak meta-interpreter, proved its correctness, and then showed a number of ways in which it could be used to extend the AgentSpeak language and add facilities, such as debugging, to the AgentSpeak interpreter.

¹³ The AgentSpeak interpreter was run under B-Prolog (<http://www.probp.com>) version 5.0b on a SPARC machine running SunOS 5.9. Timings are the average of ten runs.

Although the extended meta-interpreters that we presented were very simple, not all extensions are easy to do with the meta-interpreter. The meta-interpreter that we presented focuses on the interpretation of individual intentions. Consequently, it is difficult to make changes that cut across intentions, such as changing the mechanism for selecting which intention to work on next (S_I). This doesn't mean that such changes cannot be made using a meta-interpreter, just that a different meta-interpreter is required which explicitly captures the top-level agent processing cycle including intention selection.

Another issue is that although the meta-interpreter has been presented as “an AgentSpeak meta-interpreter”, in fact it won't work with the Jason or with the AgentTalk implementations of AgentSpeak! The reason for this is that due to the incompleteness of the semantics originally presented for AgentSpeak, different implementations of “AgentSpeak” actually implement quite different languages (some of these differences were discussed in section 2.3). There are a number of approaches to addressing this issue. One approach is for the authors of different AgentSpeak implementations to agree on a common semantics for the language. Another, less ambitious, approach to addressing this issue is to develop a more detailed meta-interpreter that explicitly handles areas where there are differences between implementations. For example, the meta-interpreter in section 4.3 explicitly handles alternative plans rather than delegating this to the underlying interpreter, and so should work with the AgentTalk implementation. A third approach is to use a different agent programming language such as CAN [21] or 3APL [9].

Both these areas are left for future work. An additional area for future work is extending the semantics given in section 2.2 to include variables and unification. In AgentSpeak unifying the triggering event with a clause head doesn't bind variables in the triggering event, because the clause could fail. Instead, when the clause succeeds the unification is applied to the triggering event [7]. This is a relatively subtle issue which only affects non-ground events, and if handled incorrectly causes the meta-interpreter clause $solve(?C) \leftarrow ?C$ to work incorrectly. Since this issue is both subtle and causes problems if done incorrectly, we feel that it is valuable to specify it formally.

Acknowledgements

I would like to thank James Harland for comments on a draft of this paper and to acknowledge the support of Agent Oriented Software Pty. Ltd. and of the Australian Research Council (ARC) under grant LP0453486. I would also like to thank the anonymous reviewers for their comments which helped improve this paper.

References

1. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM* (1960) 184–195
2. Armstrong, J., Viriding, S., Williams, M.: Use of prolog for developing a new programming language. In: *The Practical Application of Prolog*. (1992)
3. Shapiro, E.: A Subset of Concurrent Prolog and Its Interpreter. In: *Concurrent Prolog: Collected Papers (Volume 1)*. MIT Press (1987) 27–83

4. Thangarajah, J., Winikoff, M., Padgham, L., Fischer, K.: Avoiding resource conflicts in intelligent agents. In van Harmelen, F., ed.: *Proceedings of the 15th European Conference on Artificial Intelligence*, IOS Press (2002)
5. Poutakidis, D., Padgham, L., Winikoff, M.: Debugging multi-agent systems using design artifacts: The case of interaction protocols. In: *Proceedings of the First International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS'02)*, ACM Press (2002) 960–967
6. Jones, N.D., Gomard, C.K., Sestoft, P.: *Partial Evaluation and Automatic Program Generation*. Prentice Hall International (1993) ISBN 0-13-020249-5.
7. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In de Velde, W.V., Perrame, J., eds.: *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAA-MAW'96)*, Springer Verlag (1996) 42–55 LNAI, Volume 1038.
8. Ashri, R., Luck, M., d'Inverno, M.: Infrastructure support for agent-based development. In: *Foundations and Applications of Multi-Agent Systems*, Springer-Verlag LNAI 2333 (2002) 73–88
9. Dastani, M., de Boer, F., Dignum, F., Meyer, J.J.: Programming agent deliberation: An approach illustrating the 3APL language. In Rosenschein, J.S., Sandholm, T., Wooldridge, M., Yokoo, M., eds.: *Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'03)*, Melbourne, Australia, ACM Press (2003) 97–104
10. d'Inverno, M., Kinny, D., Luck, M., Wooldridge, M.: A formal specification of dMARS. In Singh, M., Rao, A., Wooldridge, M., eds.: *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages*, Springer-Verlag LNAI 1365 (1998) 155–176
11. Huber, M.J.: JAM: A BDI-theoretic mobile agent architecture. In: *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*. (1999) 236–243
12. Georgeff, M.P., Lansky, A.L.: Procedural knowledge. *Proceedings of the IEEE Special Issue on Knowledge Representation* **74** (1986) 1383–1398
13. Ingrand, F.F., Georgeff, M.P., Rao, A.S.: An architecture for real-time reasoning and system control. *IEEE Expert* **7** (1992)
14. Lee, J., Huber, M.J., Kenny, P.G., Durfee, E.H.: UM-PRS: An implementation of the procedural reasoning system for multirobot applications. In: *Proceedings of the Conference on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS'94)*. (1994) 842–849
15. Busetta, P., Rönquist, R., Hodgson, A., Lucas, A.: JACK Intelligent Agents - Components for Intelligent Agents in Java. Technical report, Agent Oriented Software Pty. Ltd, Melbourne, Australia (1998) Available from <http://www.agent-software.com>.
16. Machado, R., Bordini, R.: Running AgentSpeak(L) agents on SIM_AGENT. In Meyer, J.J., Tambe, M., eds.: *Intelligent Agents VIII - Proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001)*, Springer-Verlag LNAI 2333 (2001)
17. Rahwan, T., Rahwan, T., Rahwan, I., Ashri, R.: Agent-based support for mobile users using AgentSpeak(L). In Giorgini, P., Henderson-Sellers, B., Winikoff, M., eds.: *Agent-Oriented Information Systems (AOIS 2003): Revised Selected Papers*, Springer LNAI 3030 (2004) 45–60
18. d'Inverno, M., Luck, M.: *Understanding Agent Systems*. Springer-Verlag (2001)
19. Moreira, A., Bordini, R.: An operational semantics for a BDI agent-oriented programming language. In Meyer, J.J.C., Wooldridge, M.J., eds.: *Proceedings of the Workshop on Logics for Agent-Based Systems (LABS-02)*. (2002) 45–59
20. Plotkin, G.: Structural operational semantics (lecture notes). Technical Report DAIMI FN-19, Aarhus University (1981 (reprinted 1991))

21. Winikoff, M., Padgham, L., Harland, J., Thangarajah, J.: Declarative & procedural goals in intelligent agent systems. In: Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002), Toulouse, France (2002)
22. Hindriks, K.V., Boer, F.S.D., van der Hoek, W., Meyer, J.J.C.: A formal embedding of AgentSpeak(L) in 3APL. In Antoniou, G., Slaney, J., eds.: Advanced Topics in Artificial Intelligence, Springer Verlag LNAI 1502 (1998) 155–166
23. Sterling, L., Shapiro, E.: The Art of Prolog. Second edn. MIT Press (1994)
24. Winikoff, M.: Logic Programming with Linear Logic. PhD thesis, Melbourne University (1997)
25. Padgham, L., Winikoff, M., Poutakidis, D.: Adding debugging support to the prometheus methodology. EAAI special issue on “Agent-oriented software development” **18/2** (2005)
26. Ancona, D., Mascardi, V., Hübner, J.F., Bordini, R.H.: Coo-agentspeak: Cooperation in agentspeak through plan exchange. In Jennings, N.R., Sierra, C., Sonenberg, L., Tambe, M., eds.: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems, ACM Press (2004) 698–705
27. O’Keefe, R.A.: The Craft of Prolog. MIT Press (1990)

A A Failed Attempt

In order to show that the meta-interpreter isn't obvious, here we briefly present an alternative meta-interpreter that was considered and explain why it doesn't work. The intention of this meta-interpreter was that by evaluating the context condition as part of the body, rather than the context condition, we could obtain lazy context conditions regardless of the implementation's semantics.

1. $\text{solve}(\text{Act}) : \text{isAction}(\text{Act}) \leftarrow \text{do}(\text{Act}).$
2. $\text{solve}(\text{true}) \leftarrow \text{true}.$
3. $\text{solve}(\text{fail}) \leftarrow \text{fail}.$
4. $\text{solve}(-B) \leftarrow -B.$
5. $\text{solve}(+B) \leftarrow +B.$
6. $\text{solve}(?C) \leftarrow ?C.$
7. $\text{solve}(P_1 ; P_2) \leftarrow \text{solve}(P_1) ; \text{solve}(P_2).$
8. **$\text{solve}(!E) : \text{clause}(+!E, G, P) \leftarrow ?\text{isTrue}(G) ; \text{solve}(P).$**

Unfortunately this meta-interpreter does not give correct semantics. The reason is that in order to determine that a clause is not applicable it must be selected and tried, after which it is discarded. This means that if another plan is tried and fails, preceding clauses are no longer available. To see this, consider the following program¹⁴:

1. $g : p \leftarrow \text{print}(\text{'lazy.'}).$
2. $g : \text{true} \leftarrow +p ; \text{fail}.$
3. $g : p \leftarrow \text{print}(\text{'clause 3'}).$
4. $g : \text{true} \leftarrow \text{print}(\text{'eager.'}).$

If this program is run with a lazy implementation then the following occurs:

1. Clause 2 is selected (since clause 1 isn't applicable)
2. The belief p is added and clause 2 then fails
3. Clause 1 is now applicable and is selected, printing `lazy` before succeeding.

If the program is run with an eager implementation then the following occurs:

1. Clauses 2 and 4 are applicable, whereas clauses 1 and 3 are discarded.
2. Clause 2 is selected
3. The belief p is added and clause 2 then fails
4. Clause 4 now runs printing `eager` before succeeding.

However, if the program is run with the incorrect meta-interpreter above then the following occurs:

1. Clause 1 is selected and its guard evaluated. Since the guard is false, the clause instance fails, and it is discarded.
2. Clause 2 is selected, its guard succeeds and it runs, adding p and then failing.
3. Clause 3 is now considered, its guard succeeds and it runs, printing `clause 3` before succeeding.

It should be noted that although this interpreter doesn't work, it is certainly possible to write an interpreter that gives lazy context conditions even when run under an eager implementation. This can be done by making selection explicit (see section 4.3).

¹⁴ We assume that $S_{\mathcal{O}}$ selects clause in the order in which they are written.