1. Introduction

When designing an intelligent agent, flexibility and introspection are critical attributes. That is, an agent’s behavior should be dynamic and capable of change. Moreover, it should also be able to analyze its own behavior, so that it can diagnose failures and possibly even reprogram itself.

We have developed a simple, functional language, Proo, which can describe a wide range of agent behavior. Additionally, it allows the specification of theorems with checkable proofs. These theorems can formally prove that a given agent’s behavior meets a given set of requirements in the context of a given environment.

The particular environments we are studying are deterministic, fully observable, and higher-order. We are motivated by the problem of getting an agent to learn how to operate a human computer interface such as a program for coordinating mobile robots. The assumptions of determinism and full observability are reasonable as a first approximation, since user interfaces strive to be predictable without hidden state. However, the agent’s perception of the environment needs to be higher order, because it needs to model a software program, which is logically complex.

To be precise, the environment consists of a set of states $E$. An agent consists of a set of actions $A$. Each action $A_i$, is a function from $E$ to $E$. It manipulates the state of the world, producing a new state. An agent behavior, $B$, is a mapping from states $E$ to action sequences in $A^*$. A goal state of a behavior, $g$, is a state such that $B(g) = ()$. Proo allows us to define these actions and behaviors and prove properties about them.

Proovy is a first step of a research agenda to develop agents that can learn their own behaviors to act in a virtual world. Because we want our agents to be introspective, we want them to produce proofs that their behaviors are correct. Such behaviors and proofs would be represented internally using Proo constructs. Furthermore, the background knowledge that the agent would need to perform effectively in different domains can also be encoded and verified using Proo and Proovy.

2. Related Work

Proof assistants or interactive theorem provers are a well-established area of computer science used to prove mathematical theorems and verify the logical correctness of critical systems like computer chips (Kaufmann et al., 2000).

We are interested in a proof language where facts and implications can be written down clearly in a logical sequence as they would be in a mathematics text. Because proving theorems is not the final goal of this research, we need a system that can concisely represent proofs produced by an agent problem solver and check them in batch. The most advanced work in this area is the proof language Athena (Arkoudas, 2000). However, unlike Athena and the other denotational proof languages, Proovy is not concerned with the formal semantics of what makes one statement provable and another not. So, Proovy can apply multiple tactics at a time, searching a deep array of possibilities to match a given statement to TRUE. Its proofs can be shorter and more succinct.

Theorem provers have been used as the foundation of classical artificial intelligence planning (Fikes & Nilsson, 1971) from the beginning. However, planning has always been considered as a problem in first order logic. Little work has been done in worlds where an agent’s actions have consequences that cannot be represented in first order logic.

3. The Proo Language

Proo is a simple language for defining functions and writing theorems. Its function notation is derived from Scheme, although the top level organization has its own syntax. It is a pure functional language with no side effects, so there is no notion of time or program execution. Objects consist of primitives, conses, and lambda functions.

Functions do not need to specify the types of their arguments, so any expression is syntactically legal and equality is determined by simple symbol comparison. Because of the unsoundness of the untyped lambda calculus, Proo is also not logically sound. Through paradoxical examples,
one can prove that true is false. However, this is not as serious a problem as it would be for an official mathematical theorem prover. Since the human knowledge encoding and the automated reasoning are under our control, we can socially enforce that they not produce lambda functions that lead to paradoxes.

Proo does not allow recursive functions. To get the same behavior, there is a special operator called suchthat, which allows a prover to refer to an object that uniquely satisfies a particular property. For example, factorial would be defined in the following way:

```plaintext
Factorial := (suchthat (lambda (f)
    (= (f n)
       (if (= n 0) 1 0))))
```

suchthat allows us to define a wider class of logical functions than traditional programming languages.

Theorems are defined with a claim and a proof (see figure 1). The proof consists of a sequence of facts with each fact deducible from previous facts in the proof as well as prior theorems, axioms, and definitions. The proof writer must specifically state the prior facts needed to prove a given fact. This enables the proof checker to constrain its search, so proofs can be shorter with fewer intermediate steps explicitly written down.

### 4. The Proovy Proof Checker

At its core, the proovy proof checker uses a matching engine which takes two expressions and a list of free variables and attempts to find a binding to make the two expressions equal. It uses general rewrite rules that specify how a pair of expressions can be made to equal each other. When proving a fact, the proof checker attempts to match the fact with T, the truth primitive.

Whenever a match fails, the proof checker will use its current implication rules and attempt to match the predicate of each implication to T. When a binding matches successfully, the checker adds the consequence of the implication as a new rule to the matcher.

### 5. Using Proo To Specify Agent Behavior

With suitable lemmas and definitions, we can define an agent’s actions. As a basic example, consider a 1x5 one-dimensional world of bits. All of the bits are set to 0 except for an active bit set to 1. The agent has two actions, LEFT and RIGHT, which move the active bit coordinate down and up respectively. We can define the functions thus:

```plaintext
out := (lambda (n)
    (lambda (i) (if (= i n) 1 0)));
down := (lambda (n) (if (= n 0) 0 (- n 1)));
up := (lambda (n) (if (= n 4) 4 (+ n 1)));
inv := (lambda (f) (lambda (y)
    (suchthat (lambda (x) (= y (f x))))));
LEFT := (lambda (s)
    (out (down ((inv out) s))));
RIGHT := (lambda (s)
    (out (up ((inv out) s))));
```

We can similarly define agent behaviors to produce sequences of actions given a particular goal state. With these definitions, we can prove that the action sequence produced by a behavior will correctly reach any goal state from any input state.

The meta-knowledge provided by Proo lets programmers formally describe and analyze agent behavior. In the future, we hope to be able to build agents that could discover their own behaviors from such action descriptions. Agents could then know why they were acting in a certain way instead of just instinctively reacting to the world.

### References

