Master’s Theses (Excerpt)

YAGI Programming Language

Christopher Maier
Graz, 2015

Institute for Software Technology
Graz University of Technology

Supervisor/First reviewer: Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa
Second reviewer: Ass. Prof. Dipl.-Ing. Dr. techn. Gerald Steinbauer
In this chapter we will define the software architecture of a YAGI-based software system. The goals of the architecture are to decouple the syntax and semantics of the language as cleanly as possible, provide easy extensibility and a clear separation of concerns for easy maintainability. Again, we want to emphasize that Golog is typically implemented as a set of Prolog clauses and Golog programs run inside a Prolog environment. As a consequence, the distinction between what are features of Golog and what are side-effects of Prolog is challenging for novices and sometimes even for experts. Due to this fuzzy separation of semantics writing correct Golog programs can be challenging and error-prone. Moreover, the tight coupling of Golog to Prolog is also problematic from a purely technical point of view since a Prolog interpreter might not always be feasible due to resource constraints and/or performance limitations of a target system and - even if a Prolog implementation exists - integrating a Golog interpreter into a rather complex technical ecosystem might be a non-trivial task.

To overcome these issues, we propose a 3-tier layered architecture for YAGI applications that allows a clear separation of syntax and semantics and can be adopted for a variety of different target systems depending on the practical needs of a certain problem domain. We start with an overview over our architecture in Section 1.1 and proceed with the description of the purpose of each of the three layers, i.e. front-end (Section 1.2), back-end (Section 1.3) and system interface (Section 1.4). We finish this chapter with a description of the inter-layer communication between the formerly specified layers in Section 1.5.

1.1. Architecture Overview

In this section, we give an overview of the YAGI software stack. The YAGI software stack is a layered architecture consisting of the following three layers:

1. **Front-End**: The front-end provides the YAGI user interface as well as the parser for YAGI source code. The front-end handles YAGI source code on a purely syntactical level, i.e. it is responsible for checking syntactical correctness of YAGI code entered by a user. Further, the front-end transforms YAGI code into a suitable intermediate representation the allows the back-end to process the YAGI program efficiently. The user interface can vary depending on the specific needs of a specific implementation, one can imagine a wide range of different user interfaces, from simple console-based interfaces to graphical user interfaces for mobile devices such as mobile phones or tablets.

2. **Back-End**: The back-end consumes data from the front-end, i.e. the front-end passes an abstract representation of a YAGI program (the so-called abstract syntax tree, or AST) to the back-end. The back-end stores the current state of the world (i.e. the YAGI basic action theory $D_{YAGI}$) as well as executable program elements (e.g. procedures, YAGI actions). The back-end handles the YAGI
program on a semantics level, i.e. it modifies the YAGI basic action theory and/or the YAGI program elements according to the semantics of the given YAGI program. Further, the back-end responds to the front-end depending on the type of YAGI statement, which we will discuss in detail in Section 1.5.

3. System Interface: The system interface provides external data (e.g. data generated due to exogenous events) for the back-end and is responsible for executing actions, i.e. responding to YAGI signal-commands.

This 3-tier architecture is illustrated in Figure 1.1.

1.2. Front-End

In this section, the elements of the first layer (front-end) will be discussed.

1.2.1. YAGI Command Shell

The YAGI command shell allows the user to specify a certain YAGI-domain as well as to interact with this domain, e.g. change the YAGI world of fluents or query world information. The YAGI command shell serves as user interface, hence is part of the front-end.

1.2.2. YAGI Parser

The YAGI parser transforms YAGI input source code into an abstract model of the input, i.e. into an abstract syntax tree (AST). This abstract representation ensures syntactical correctness of the input and can be used for further processing.

1.3. Back-End

In this section, the elements of the second layer (back-end) will be discussed.

![YAGI 3-tier architecture](image-url)
1.3.1. YAGI Basic Action Theory

The back-end stores the state of the YAGI world, i.e. the YAGI basic action theory $\mathcal{D}_{YAGI}$. The back-end is also responsible for transforming the data provided by the front-end into a reasonable format that allows efficient storing and updating, hence the exact format depends on the specific implementation of the back-end.

1.3.2. Program

The back-end stores executable structures that can modify the state of the world. Executable structures can be rather complex and their execution happens solely in the back-end, hence there need to be a proper representation for these executable structures. Examples for complex structures are procedures, loops and non-deterministic choice of actions, among others defined in Chapter 3. Moreover, the back-end is responsible for properly executing YAGI statements provided by the front-end. Properly in this context means that the back-end shall implement the YAGI semantics as specified in Chapter 3.

1.4. System Interface

The system interface serves as the lowest level in the YAGI 3-tier architecture. The system interface varies depending on the area of application, one can think of a range of system interfaces from autonomous robots to video game bots. The purpose of the system interface is to serve as low-level communication layer that executes actions triggered by YAGI signal–commands and provides data acquired from external sources for the back-end.

1.5. Inter-Layer Communication

In this section, the communication mechanisms between each of the layers will be discussed.

1.5.1. Front-End $\rightarrow$ Back-End Communication

The front-end passes YAGI lines of code in the form of an abstract syntax tree (AST) to the back-end, i.e. for a YAGI line of code $\alpha$ the parser returns its abstract representation as a function $\text{AST}(\alpha)$, which is passed to the back-end for further processing. The types of messages are as follows:

1. **Fluent Query**: The front-end can query information about the current state of the world, i.e. states of fluents.

2. **Program Specification**: The front-end can pass YAGI programs to the back-end. Such programs have no initial effect until they get executed.

3. **Program Execution**: The front-end can pass statements that initiate program execution to the back-end. There are various consequences depending on the program structure that gets executed, e.g. modifications of the state of the world or testing certain conditions. These various effects are described in detail in Chapter 3.

The exact format of the output of $\text{AST}(\alpha)$ as well as the communication mechanism between front-end and back-end depends on the particular implementation of the system, i.e. may vary depending on the requirements of a specific implementation. We describe our sample implementation in Chapter ??.
1.5.2. Back-End → Front-End Communication

The back-end responds to the front-end depending on the type of message as follows:

1. **Fluent Query**: If the front-end queries the state of a fluent, the reply is a set of tuples representing the state of the fluent or `false` if the fluent is not defined.

2. **Program Specification**: The back-end returns `true` iff the program could be stored properly and `false` in any other case.

3. **Program Execution**: The back-end returns information about the program that is being executed. Such information can be status information, data produced by `signal`-blocks of YAGI actions or diagnostics in case any run-time errors occur.

The data exchange format for this part of the communication may also vary between specific implementations.

1.5.3. Back-End → System Interface Communication

The back-end communicates with the system interface via a string signaling mechanism. The content of the string can either be plain text in a natural language or executable code in an arbitrary programming language. This decision depends on how a specific system interface processes the contents of the string signal. Signals can for example trigger an action that executes a real-world action (e.g. motion of an autonomous robot) or query some information about the world (e.g. a synchronous/blocking request to a specific sensor).

The data exchange format as well as how to distinguish between action- and sensing-signals may vary between different implementations of the system.

1.5.4. System Interface → Back-End Communication

The system interface responds to a signal from the back-end accordingly, i.e. providing some status information about an executed action or returning an actively triggered status query result. Moreover, the system-interface asynchronously provides data from exogenous events via call-backs for the back-end. The back-end buffers exogenous event data in a queue if an action is being executed. These buffered values are then consumed by the back-end and the next action is executed.

The data exchange format may vary between different implementations of the system.
YAGI By Example

In this chapter, we provide an implementation of our object delivery robot running example to illustrate one specific scenario we plan to use YAGI for, including a non-formal description of the intended semantics.

2.1. Running Example

In this section, we provide an implementation of our running example written in YAGI. For the sake of simplicity, we define that every variable and every element in a fluent starting with lowercase r corresponds to a room, lowercase o corresponds to an object and lowercase p corresponds to a person.

We proceed with the definition of fluents, facts, actions and procedures including a non-formal description of the intended semantics. A detailed specification of the syntax and semantics of YAGI follows in Chapter 3. The complete source listing is included in Appendix ??.

2.1.1. Fluents

The following listing specifies the fluents for the object delivery robot:

```yagi
//location of the robot (room1, ..., room3)
fluent at ["r1", "r2", "r3"];
at = ["r1"];

//location of objects (object1 in room1 etc)
fluent is_at ["o1", "o2", "o3"]["r1", "r2", "r3"];
is_at = ["o1", "r1"], ["o2", "r2"], ["o3", "r3"];

//object carried by robot
fluent carry ["o1", "o2", "o3"];

//requests moving an object (param 1) from a sender (param 2) //to a receiver (param 3)
fluent request ["o1", "o2", "o3"]["p1", "p2", "p3"]["p1", "p2", "p3"];

//states what person has been detected in what room
fluent detectedPerson ["p1", "p2", "p3"]["r1", "r2", "r3"];
```

Listing 2.1: Object Delivery Robot Fluents
Chapter 2. YAGI By Example

After the name of every fluent, one or more pairs of brackets follow. The number of pairs define the arity of the fluent. Inside every pair of brackets their need to be the specification of the domain inside a pair of braces, e.g. the fluent at has arity one and the domain is the set of available rooms, i.e. {"r1", "r2", "r3"}. Fluents can subsequently be assigned to their initial values. The fundamental type of a fluent assignment is a set of tuples, e.g. the fluent is_at is assigned with a set of object-room tuples. A tuple is denoted by enclosing angle brackets, whereas a set is denoted by enclosing braces. We decided to use tuples and sets as our basic concept because we believe that their semantics are widely familiar and easy to understand. Further, the concepts of sets and tuples relate closely to the semantics of relational databases, which is rather convenient as we will describe in detail in Section ??.

2.1.2. Facts

The following listing specifies the facts for the object delivery robot:

```plaintext
//one or more rooms are assigned to one person, 
//i.e. the person's offices

fact office [{"p1","p2","p3"}][{"r1","r2","r3"}];

Listing 2.2: Object Delivery Robot Facts
```

Facts are similar to fluents, the only difference is that facts can only be assigned once and remain constant after the first assignment. Semantically, there is no difference between a fact and a fluent, its intended purpose is solely to enable a programmer to express constness of certain properties of the world.

2.1.3. Actions

The following listing specifies the action for moving the object delivery robot to a specific room:

```plaintext`
//move robot to room $r
action move($r)
precondition:
//robot is not in room $r
not (<$r> in at);

Listing 2.3: Object Delivery Robot Move Action
```

The action uses set-operators to describe the effects of moving a robot to a specific room. Furthermore, the precondition is defined using the binary operator in, which evaluates whether or not a concrete object (more specifically, a concrete tuple) is part of a set of tuples. The actions for picking up and putting down an object look similar:

```plaintext`
//pickup object $o
action pickup($o)
precondition:
//robot doesn't carry anything and is in the room where the object is
(not(exists <$x> in carry) and exists <$y> in at such <$o,$y> in is_at);

Listing 2.4: Object Delivery Robot Pickup Action
```
2.1. Running Example

```plaintext
carry += \$o;
signal:
  "Pickup object " + \$o;
end action
```

Listing 2.4: Object Delivery Robot Pickup Action

```plaintext
//putdown object
action putdown(\$o)
precondition:
  //he carries the object stored in \$o
  \$o in carry;
effect:
  //now he's not
  carry -= \$o;
  //where ever it was, its now somewhere else...
  is_at -= \$o,
  //...namely: here!
  foreach \$r in at do
    is_at += \$o, \$r;
  end for
signal:
  "Put down object " + \$o;
end action
```

Listing 2.5: Object Delivery Robot Putdown Action

The following listing specifies the action for detecting a person:

```plaintext
//"setting" action to detect a person, i.e.
//\$p gets its value from an external src
action detectPerson() external (\$p)
effect:
  //remove person
  detectedPerson -= \$p,
  //add the detected person + room tuple to the fluent
  foreach \$r in at do
    detectedPerson += \$p, \$r;
  end for
signal:
  "detect person"
end action
```

Listing 2.6: Object Delivery Robot Detect Person Action

The listing above illustrates an action that uses external information to set the value of a fluent. Consequently, we call these types of actions setting actions, denoted by the external-modifier in the first line of the action declaration. Note that every variable stated after the external keyword gets its value from an external source and can subsequently be used just like any other local variable. In contrast, an action without an external-modifier specifies projection effects without using any external data. Moreover, note the usage of the underscore character in the line that removes the detected person from the fluent. The underscore character serves as wild-card, i.e. can be replaced by any possible value of the domain. This feature resembles the pattern matching functionality from functional programming languages like Scala.
Chapter 2. YAGI By Example

2.1.4. Exogenous Events

Exogenous events differ from setting actions in that they can’t be actively triggered by a YAGI statement. Exogenous events are triggered by an external event, hence they modify the internal representation of the world based on some external input. The following listing specifies the exogenous event for receiving a request to transport an object from a sender to a receiver:

```
//exogenous event to initiate transportation
//of object $o from $sender to $receiver
exogenous-event receiveRequest($o, $sender, $receiver)
//add request
request += <$o, $sender, $receiver>;
end exogenous-event
```

Listing 2.7: Object Delivery Robot Receive Request Exogenous Event

2.1.5. Procedures

The following listing specifies the procedure for serving a request:

```
//serves a request
proc serve($object, $sender, $receiver)

    pick <$sender, $roomSender> from office such
        move($roomSender);
    //search for person in the room
        detectPerson();
    //sender is actually in the room
    if <$sender, $roomSender> in detectedPerson() then
        pickup($object);
    //deliver object to receiver
    pick <$receiver, $roomReceiver> from office such
        move($roomReceiver);
    //search for person in the room
        detectPerson();
    //receiver is actually in the room
    if <$receiver, $roomReceiver> in detectedPerson() then
        putdown($object);
    end if
end pick
end if
end pick
end proc
```

Listing 2.8: Object Delivery Robot Serve Request Procedure

Finally, the following listing specifies the main controller of the object delivery robot, which simply serves a randomly picked request:

```
proc main()

    //serve a random request
    pick <$object, $sender, $receiver> from request such
        serve($object, $sender, $receiver);
end pick
```

10
Listing 2.9: Object Delivery Robot Main Procedure
In this chapter, we specify the syntax and semantics of the YAGI language.

3.1. Notation

To specify the syntax we use statements written in Backus-Naur Form (BNF) of the form \( (a) ::= B \), where non-terminal symbols are denoted by enclosing \( \langle \text{angle brackets} \rangle \) and syntactical elements (i.e., terminals) of the language are \textbf{bold}. Moreover, we use regular expressions to quantify occurrences of elements in BNF-formulas, applying default semantics of regular expression elements, i.e. one-or-more (+), zero-or-more (*), zero-or-one (?), negation (¬). To specify semantics, we use logical connectives of propositional logic and first-order logic with their conventional meanings. Furthermore, we use the notation and semantics of situation calculus as defined by (McCarthy, 1963) and (Reiter, 2001) to model the state of the world and IndiGolog’s programming constructs and their semantics defined by (De Giacomo et al., 2009) to specify program flow.

To specify the semantics of the situation calculus language \( L_{YAGI} \) over the basic action theory \( D_{YAGI} \) we define \( L_{YAGI} \) initially to be empty, i.e. no fluents, actions or constant symbols (except \( S_0 \)) are defined and \( D_{YAGI} \) to only contain domain-independent information, i.e. everything except \( \Sigma \) and \( D_{unc} \) is empty. We consider this initial state as the interpretation of an empty YAGI program \texttt{null}\(^1\). For arbitrary sequences of YAGI lines of code \( \langle l_1, \ldots, l_n \rangle \) the resulting theory \( D_{YAGI}' \) over the language \( L_{YAGI}' \) is obtained by modifying their respective predecessors \( D_{YAGI} \) over \( L_{YAGI} \) obtained by the YAGI lines of code \( \langle l_1, \ldots, l_{n-1} \rangle \), depending on the type of YAGI language construct of line \( l_n \) as specified in the following sections.

3.2. Basic Language Elements

To be able to specify the semantics of the YAGI language (i.e. the mapping to situation calculus sentences and IndiGolog programming constructs) we need to briefly define a set of basic language elements that will be used throughout this chapter.

3.2.1. String

\[ (\text{string}) ::= "(¬ (" | "))" \]

\(^1\)We define \texttt{null} to be the empty YAGI program to avoid confusion with the empty Golog program, which is often denoted as \texttt{nil}. 

13
Chapter 3. YAGI Language Specification

Defines a valid sequence of characters, i.e. every concatenation of characters (except double quote and double slash) surrounded by a leading and trailing double quote character.

### 3.2.2. List of Strings

\[
\text{string_list} ::= \text{string}(\text{string})^*
\]

A sequence of strings.

### 3.2.3. Identifier

\[
\text{id} ::= (a\ldots z|A\ldots Z)(a\ldots z|A\ldots Z|0\ldots 9|\_)^+
\]

An identifier has no standalone semantics, it solely specifies the structure that a valid name of an entity in the YAGI language must fulfill. Under certain conditions, identifiers must be unique, e.g. the names of two different actions must not be equal. The exact conditions of name uniqueness will be discussed in later sections.

### 3.2.4. Variable

\[
\text{var} ::= $\langle \text{id} \rangle$
\]

Defines an identifier to which a value can be assigned to.

### 3.2.5. List of Variables

\[
\text{var_list} ::= \text{var}(\text{var})^*
\]

A sequence of variables.

### 3.2.6. Value

\[
\text{value} ::= \langle \text{string} \rangle | \langle \text{var} \rangle
\]

A value is a shortcut for something that is either a string or a variable.

### 3.2.7. Value-Expression

\[
\text{valexpr} ::= \langle \text{value} \rangle ((\text{value})^*)
\]

Addition of two values. Due to the fact that variables can only hold string values (as specified in Section 3.7.2) each such expression ultimately boils down to an operator + being applied to string elements. Hence, we can define the semantics of a value expression as the concatenation of character sequences.

### 3.2.8. Tuple

\[
\text{tuple} ::= < (\text{tuple_val}) (\text{tuple_val})^* > | \varepsilon
\]

\[
\text{tuple_val} ::= \langle \text{var} \rangle | \langle \text{string} \rangle | * | _
\]

Defines a (possibly empty) mathematical tuple \(x_1,\ldots,x_n\). Possible elements in such a tuple can be variables, strings, the star character (\(*\)), which denotes incomplete information (as discussed in Section 3.7.3) and the underline character (\(_\)), which denotes pattern matching (as discussed in Section 3.8).
3.2.9. Set

\[ \langle\text{set}\rangle ::= \{\langle\text{tuple}\rangle\}^* \mid \epsilon \]

Defines a finite (possibly empty) mathematical set \(\{x_1^1, x_1^2, \ldots, x_1^n\}, \ldots, \{x_m^1, x_m^2, \ldots, x_m^n\}\) of tuples.

3.2.10. Set-Expression

\[ \langle\text{setexpr}\rangle ::= \langle\text{set}\rangle \ (\langle\text{+} \mid \cdot\rangle \ \langle\text{set}\rangle)^* \]

Defines the set-based union and complement, i.e. let \(A\) and \(B\) be sets, then the YAGI expression \(A + B\) denotes the union \(A \cup B\) and \(A - B\) denotes the complement \(A \setminus B\). For the sake of conformance to the majority of well-known general purpose programming languages we define that both operators + and − have the same precedence and are both left-to-right associative.

3.3. Fluent Declaration

3.3.1. Syntax

\[ \langle\text{fluent\_decl}\rangle ::= \text{fluent} \ \langle\text{id}\rangle \ (\ [\langle\text{String}\rangle \ [\langle\text{string\_list}\rangle\ ]\ ]\ )^* ; \]

3.3.2. Semantics

Let \(l_m\) be a YAGI fluent declaration of a fluent \(F\) with arity \(m\), where \(m\) denotes the number of square bracket pairs following the name of the fluent. Each pair of square brackets define the domain of its corresponding dimension, i.e. we say that the \(n\)-th dimension of fluent \(F\) with arity \(m\) and \(0 \leq n \leq m\) has domain \(S^n_F\). Furthermore, we say that a fluent \(F\) has domain \(S^n_F\), i.e. the fluent has domain \(S^n_F\) in dimension one, \(S^2_F\) in dimension two and so on, and \(\vec{S}_F = (S^1_F, S^2_F, \ldots, S^m_F)\). The sort of the \(n\)-th dimension of a fluent is defined as follows, where the term \(\text{sort}\) is used as in many-sorted first order languages and will from now on be used equivalently to the term \(\text{domain}\). The sort string represents the countably infinite set of possible character sequences, i.e. the Kleene closure \(V^*\) over the alphabet \(V = \{A, Z, a, z, 1, \ldots, 9\}_\). The axiomatization is achieved by mapping every string value to a constant with the same name as the value of the string and providing corresponding unique-name axioms for these constants. We call this set of axioms \(D_{\text{mstr}}\). Note that the domain can either be the full range of \(V^*\) (denoted by \([\text{String}]\)) or any finite subset \(\{s_1, \ldots, s_n\} \subset V^*\) denoted by the enumeration of all the valid elements, i.e. \([s_1, s_2, \ldots, s_n]\).

The declaration of a fluent \(F\) with arity \(m\) extends \(L_{\text{YAGI}}\) by adding the corresponding \((m + 1)\)-ary predicate \(F(\vec{x}, s)\) and two \(m\)-ary action symbols \(\text{addF}(\vec{x})\) and \(\text{removeF}(\vec{x})\), leading to \(L_{\text{YAGI}}'\), where \(\vec{x}\) denotes the vector of fluent arguments \(x_1, \ldots, x_m\) and \(s\) denotes the situation term. \(D_{\text{YAGI}}'\) is the same as \(D_{\text{YAGI}}\) except that all sentences that mention \(F, \text{addF}\) or \(\text{removeF}\) in \(D_{\text{S0}}, D_{\text{sat}}\) and \(D_{\text{a0}}\) are removed and the axioms \(\forall \vec{x}. F(\vec{x}, S_0) \equiv \text{false}\) is added to \(D_{\text{S0}}\). This can be considered as some form of initialization of the theory for the fluent \(F\). Moreover, the axiom \(F(\vec{x}, \text{do}(a, s)) \equiv a = \text{addF}(\vec{x}) \vee F(\vec{x}, s) \wedge a \neq \text{removeF}(\vec{x})\) is added to \(D_{\text{sat}}\). The purpose of the situation calculus simple actions \(\text{addF}\) and \(\text{removeF}\) are to make the fluent \(F\) true (or false, respectively) for a given parameter vector \(\vec{x}\). Note that each element in \(\vec{x}\) is an instantiation of an element of the sort of the corresponding dimension in the fluent declaration, i.e. \(x_1 \in \vec{x}\) has domain \(S^1_F\) (the sort of the first dimension of the fluent declaration for the fluent \(F\)), \(x_n \in \vec{x}, n \leq m\) has domain \(S^n_F\) and so on. To enforce this correspondence, we add the necessary preconditions \(\text{Poss(\text{addF}(\vec{x}), s) \equiv \bigwedge_{i=1}^{m} \tau(S^n_F, x_i)\) and \(\text{Poss(\text{removeF}(\vec{x}), s) \equiv \bigwedge_{i=1}^{m} \tau(S^n_F, x_i)\) to \(D_{\text{a0}}\), with \(\tau(S^n_F, x_i)\) being a binary predicate that holds iff \(x_i\) is an element of its corresponding sort \(S^n_F\). Also note that the initial database \(D_{\text{S0}}\) is in closed form, according to the definition from (Reiter, 2001).

\[^2\text{In the special case of a fluent \(F\) having arity 0, the action preconditions are defined as } \text{Poss(\text{addF}, s)} \equiv \text{true and Poss(\text{removeF}, s)} \equiv \text{true.}\]
3.4. Fact Declaration

3.4.1. Syntax

\[ \text{fact_decl} ::= \text{fact} \langle \text{id} \rangle (\langle \text{String} | \{ \langle \text{string_list} \rangle \} \rangle)^* ; \]

3.4.2. Semantics

The semantics of \( \text{fact_decl} \) is identical to the semantics of \( \text{fluent_decl} \), the only difference is that a fact can only be assigned once and becomes immutable after it has been assigned for the first time. According to this definition, we can simplify the underlying theory for facts by omitting the definitions of the situation calculus actions \text{add} and \text{remove} for each declared fact. This leads to a theory that makes the constness of facts more explicit since there exists no mechanism in the theory that is able to modify a fact. Initialization of facts is implemented by directly updating \( D_{YAGI} \), i.e. adding \( F(\vec{x}, S_0) \equiv \vec{x}_1 = \vec{x}_2 \lor \ldots \lor \vec{x}_n \) for the fact \( F \) and the \( n \) parameter vectors which are used to initialize the fact. Further, we define that \( F(\vec{x}_1, S_0) \equiv \text{true} \) and \( \forall \vec{x}.F(\vec{x}, S_0) \equiv \text{false} \) for the special cases for \( n = 1 \) and \( n = 0 \), respectively.

3.4.3. Implementation Remarks

Any implementation shall check that there is no assignment to a fact after its initialization. Any further left-hand side appearance of a fact in an assignment shall lead to an error. Moreover, any implementation shall ensure that a fact is subsequently assigned after its declaration, i.e. let \( l_n \) be a YAGI line of code that declares a fact, then the subsequent line \( l_n+1 \) must be the initialization of the formerly declared fact. Any other type of statement shall lead to an error.

3.5. Action Declaration

3.5.1. Syntax

\[ \langle \text{action_decl} \rangle ::= \text{action} \langle \text{id} \rangle (\langle \text{varlist} \rangle?) (\text{external} (\langle \text{varlist} \rangle))? \langle \precondition: \langle \text{formula} \rangle \rangle? \langle \text{effect:} \langle \text{assignment} \rangle^+ \rangle? \langle \text{signal:} \langle \text{valexpr} \rangle? \rangle? \text{end action} \]

3.5.2. Semantics

Let \( \alpha \) be a YAGI action declaration for an action \( A \) with arity \( m \), where \( m \) denotes the number of parameters for that respective action, i.e. the number of elements in \( \langle \text{varlist} \rangle \). Then \( L_{YAGI} \) and \( D_{YAGI} \) remain unchanged and a Golog procedure of the form \text{proc} \( A(\vec{x}) \delta \text{endProc} \) is added to the set of Golog procedures. We choose the name of the Golog procedure to be the same as the name of the action and \( \vec{x} \) as the vector of the \( m \) parameters passed to the YAGI action. The Golog program \( \delta \) consists of a test-action as first statement that evaluates the formula constructed from the YAGI \text{precondition} as specified in Section 3.6. If no \text{precondition}-block is present, the test-action in the corresponding Golog procedure can be omitted. The YAGI effect-block is mapped to a (possibly empty) sequence of Golog statements constructed from the sequence of \( \langle \text{assignment} \rangle \)-statements as discussed in Section 3.7. The optional signal-block is solely responsible for communication with the system interface as described in Section 1.5.3, i.e. it has no influence on \( L_{YAGI} \) and \( D_{YAGI} \) and can therefore be omitted in the Golog procedure. A schematic representation of the correspondence between a YAGI action and a Golog procedure is sketched in the listing below.
3.5. Action Declaration

Additionally, a YAGI action declaration can be augmented with an optional external modifier, followed by a non-empty list of variables. The semantics of this extension is that the variables listed after the external modifier are set to a value based on some data from external sources. Consequently, we call actions with an external modifier present setting actions. We claim that the semantics (i.e. the mapping to situation calculus and Golog) of ordinary YAGI action declarations and setting action declarations are equivalent. This claim can be justified by the observation that setting actions solely assign values to variables, i.e. bind a value to an identifier. Due to the fact that assignments to variables have no influence on the underlying domain theory (see Section 3.7) variable assignments can be considered transparent from a theoretical point of view. Furthermore, note that the activity of setting values to variables is triggered via the signal-expression of the action declaration. A schematic representation of the correspondence between a YAGI setting action and a Golog procedure is sketched in the listing below.

3.5.3. Relating YAGI Actions, Situation Calculus Actions and Golog Procedures

Considering the fact that one of the basic elements of situation calculus are actions the question might arise why we decided to map YAGI actions to Golog procedures and not directly to situation calculus actions. To answer that question consider how a mapping from a YAGI action to a situation calculus action might look like. For each YAGI action $A_{YAGI}$ we would create a situation calculus action $A_{sitcalc}$ with the same name and the same parameters as the YAGI action. Furthermore, we would construct the action precondition of the form $Poss(A_{sitcalc}(\vec{y}, s)) \equiv \Pi_{\lambda_{sitcalc} (\vec{y}, s)}$ from the YAGI action precondition $\langle formula \rangle$ and successor state axioms of the form $F(\vec{x}, do(A_{sitcalc}, s)) \equiv \Phi_F(\vec{x}, A_{sitcalc}, s)$ for each fluent involved in an assignment in the YAGI action effect block. Constructing the action precondition formula $\Pi_{\lambda_{sitcalc}}$ from the YAGI $\langle formula \rangle$ is straight-forward according to the definition of $\langle formula \rangle$ in Section 3.6 whereas the construction of successor state axiom formulas from a sequence of YAGI assignments requires deeper analysis. First of all, YAGI supports for-loop assignments of the form
Chapter 3. YAGI Language Specification

foreach <$x_1, x_2, ..., x_n> in <setexpr> do <assignment> end for

which, loosely speaking, means that the loop iterates over each tuple in a set and uses those tuples for arbitrary assignments inside the loop body. Note that <$x_1, x_2, ..., x_n> in the example above is not a syntactically valid YAGI tuple. When we use this notation in YAGI code in this chapter we actually mean that instead of "..." all the concrete elements in the tuple are explicitly stated. Furthermore, the exact semantics of such a loop is discussed in Section 3.7.2 and are of minor importance for the further discussion in this section. Note that the syntax of the foreach-loop above closely resembles iteration constructs from general purpose languages like Java, C++ and C#, hence people familiar with such languages might assume similar (i.e. iterational) semantics just based on the syntax. When compiling such a loop directly into a successor state axiom (i.e. a formula) we would lose any sequential/iterational semantics since the evaluation of a formula is inherently "parallel".

We strongly believe that removing iterational semantics from such a loop would lead to a huge level of confusion among people who are not aware of the exact semantics of situation calculus. Moreover, we claim that rewriting arbitrary sequences of YAGI assignments to a single successor state axiom formula is a non-trivial task, even though we want to emphasize that we’re positive that it is possible to prove that one could rewrite YAGI effect blocks to successor state axioms, which is something we plan to show in the near future. Additionally, we can argue that the YAGI basic action theory is always progressable when we map YAGI actions to Golog procedures since the only situation calculus actions involved are add- and remove-actions for each fluent (as defined in Section 3.3.2), which makes the YAGI basic action theory local-effect and for local-effect basic action theories a first-order strong progression always exists according to the work done by (Vassos et al., 2008). Moreover, local-effect basic action theories and progression of basic action theories in general have been discussed in detail by (Vassos and Levesque, 2008), (Liu and Lakemeyer, 2009), (Vassos and Patrizi, 2013) and others, providing a solid theoretical foundation for our intended semantics of YAGI.

Lastly, we want to mention that the decision to map YAGI actions to IndiGolog procedures instead of situation calculus actions may also have an impact from a purely practical point of view. Here, by practical we mean a concrete implementation of a YAGI software system. More precisely, one could argue that the rewriting to IndiGolog procedures induces a performance (i.e. run-time) overhead compared to a direct mapping to situation calculus simple actions since IndiGolog procedures are more complex constructs. For the sake of completeness we want to mention that we also think that this is a valid argument and needs proper discussion, even though we consider it to be of minor importance at this point in time and - hence - delay it to future work.

3.5.4. Implementation Remarks

Any implementation shall ensure that the process of setting values to the variables listed after the external-keyword happens synchronously, i.e. the execution of the YAGI program shall block until the sensing process has finished. Furthermore, any implementation shall provide a timeout mechanism to prevent the application from waiting indefinitely. Moreover, any attempt to put a variable that is passed as parameter to the action after the external-keyword shall result in an error.

3.6. Formulas

3.6.1. Syntax

(formula) ::= (atom) | not ( (formula) )
exists (tuple) in (setexpr) (such (formula))? all (tuple) in (setexpr) (such (formula))? (tuple) in (setexpr)

\[
\langle \text{atom} \rangle ::= \langle \text{value} \rangle \langle \text{comp_op} \rangle \langle \text{value} \rangle \\
\langle \text{setexpr} \rangle ::= \langle \text{comp_op} \rangle \langle \text{setexpr} \rangle \\
\langle \text{true} \rangle \langle \text{false} \rangle
\]

\[
\langle \text{connective} \rangle ::= \text{and} | \text{or} | \text{implies}
\]

### 3.6.2. Semantics

Instances of \(\langle \text{formula} \rangle\) evaluate to a logical truth value. The elements allowed in such a first-order formula have the following semantics:

- **Truth Values:** \textit{true} is \textit{true}, \textit{false} is \textit{false}.

- **Comparisons:** On string values, two elements \(s_1\) and \(s_2\) are considered equal iff they have the same length and each character at the same position in both \(s_1\) and \(s_2\) are equal. If this equality relation holds the operator \(==\) returns \textit{true}, otherwise it returns \textit{false}. Consequently, the operator \(\neg\) is the negation of \(\Rightarrow\). The ordering comparisons \(<,\leq,\geq,\rangle\) and \(\rangle\) are performed lexicographically. On sets, comparisons are element-based, i.e. two sets \(A\) and \(B\) are equal iff every element of \(A\) is in \(B\) and vice versa. Order comparisons are mapped to (proper) subset/superset relations, i.e. let \(X\) and \(Y\) be sets, then \(X < Y\) is \textit{true} iff \(X\) is a proper subset of \(Y\), i.e. \(X \subset Y\). Consequently, the operator \(\rangle\) denotes the proper superset \(\supset\). The operators \(<\leq\rangle\) and \(\geq\rangle\) follow intuitively as subset and superset without the strictness property, i.e. \(\subseteq\) and \(\supseteq\).

- **Logical Connectives:** The logical connectives \textit{and} (\&), \textit{or} (\lor) and \textit{implies} (\rightarrow) have their usual meanings.

- **Negation:** The operator \textit{not} (\neg) negates the truth value.

- **First-Order Quantifiers:** The operators \textit{all} (\forall) and \textit{exists} (\exists) have their usual meanings. Note that they operate on the sorts of the respective \(\langle\text{setexpr}\rangle\), i.e. let \(F\) be a fluent of sort \(\mathcal{S}_f\), then \textit{exists} \(\langle x_1,\ldots,\rangle \in F\) translates to \(\exists_{\mathcal{S}_f} x_1 \exists_{\mathcal{S}_f} x_2 \ldots \exists_{\mathcal{S}_f} x_n F(x_1,\ldots,x_n,s)\), where \(\exists_{\mathcal{S}_f}\) is the existential quantifier over the sort of the \(n\)-th dimension of fluent \(F\). The \textit{all}-quantifier follows similarly, with \(\forall_{\mathcal{S}_f}\) being the universal quantifier over the sort of the \(n\)-th dimension of fluent \(F\). Note that the YAGI variables \(\langle x_1,\ldots,\rangle\) must be \textit{fresh} in a sense that they must not be bound to a value before they are used in an \textit{all} or \textit{exists} statement. The optional \textit{such} (\langle\text{formula}\rangle) is connected to the quantified formula either via a logical conjunction (in case of an existential quantifier, i.e. \textit{exists} \(\langle x_1,\ldots,\rangle \in F\) \textit{such} \(\langle\text{formula}\rangle\) translates to \(\exists_{\mathcal{S}_f} x F(x,s) \land \varphi\)) or a logical implication (in case of an all quantifier, i.e. \textit{all} \(\langle x_1,\ldots,\rangle \in F\) \textit{such} \(\langle\text{formula}\rangle\) translates to \(\forall_{\mathcal{S}_f} x F(x,s) \rightarrow \varphi\)), where \(\varphi\) corresponds to a YAGI \(\langle\text{formula}\rangle\) instance. Note that in case no \textit{such}-block is present the semantics of \textit{all} and \textit{exists} are identical, i.e. \textit{exists} \(\langle x_1,\ldots,\rangle \in F\) and \textit{all} \(\langle x_1,\ldots,\rangle \in F\) hold iff there is at least one element for which the Fluent \(F\) holds.

- **Operator \textit{in}:** The keyword \textit{in} is used to specify the domain of discourse when used with a first-order quantifier as defined above. Moreover, \textit{in} can also be used without a quantifier, which changes its semantics as follows. \(\langle x_1,\ldots,\rangle \in F\) translates to \(F(x_1,\ldots,x_n,s)\), i.e. the truth value of the Fluent \(F\) in situation \(s\) is evaluated for concrete elements \(\langle x_1,\ldots,\rangle\). Note that - contrary to YAGI variables used with first-order quantifiers as discussed above - the variables \(\langle x_1,\ldots,\rangle\) must be bound to a value before being used on the left-hand side of the standalone operator \textit{in}. 

---

3.6. Formulas
3.6.3. Implementation Remarks

Any implementation shall report different errors based on the following scenarios:

- **First-Order Quantification With Bound Variables**: Anything but unbound variables used in a first-order quantified formula shall result in an error, e.g.

  ```
  fact floors[{"0", "1", "2", "3", "4", "5", "6"}];
  floors = {{"0"}, {{"1"}, {{"2"}, {{"3"}, {{"4"}, {{"5"}, {{"6"}}}}}}};
  exists <$x$> in floors such $x < "1"$;  //valid, evaluates to 'true'
  $y = "4";
  exists <$y$> in floors such $x < "1"$;  //invalid, $y$ is already bound
  exists <"0"> in floors such "2" < "1";  //invalid, "0" is a constant
  
  Listing 3.5: First-Order Quantification Examples
  ```

- **Unbound Variables on the Left-Hand Side of the Standalone Operator in**: Any use of an unbound variable on the left-hand side of the operator in shall result in an error, e.g.

  ```
  fact floors[{"0", "1", "2", "3", "4", "5", "6"}];
  floors = {{"0"}, {{"1"}, {{"2"}, {{"3"}, {{"4"}, {{"5"}, {{"6"}}}}}}};
  $y = "6";
  <$y$> in floors;  //valid, $y$ is already bound; evaluates to ‘false’
  <$"0"$> in floors;  //valid, evaluates to ‘true’
  <$"x"$> in floors;  //invalid, $x$ is unbound
  
  Listing 3.6: Operator ‘in’ Examples
  ```

3.7. Assignment

3.7.1. Syntax

\[
\text{⟨assignment⟩} ::= \text{⟨assign⟩}; \\
| \text{⟨for_loop_assign⟩} \\
| \text{⟨conditional_assign⟩}
\]

\[
\text{⟨assign⟩} ::= \text{⟨var⟩} = \text{⟨value⟩} \\
| \text{⟨id⟩} (= | += | -=) (⟨id⟩ | \text{⟨setexpr⟩})
\]

\[
\text{⟨for_loop_assign⟩} ::= \text{foreach} \ (\text{⟨tuple⟩}) \ \text{in} \ (⟨\text{id}⟩ | \text{⟨setexpr⟩}) \ \text{do} \ \text{⟨assignment⟩}^+ \ \text{end for}
\]

\[
\text{⟨conditional_assign⟩} ::= \text{if} \ (\text{⟨formula⟩}) \ \text{then} \ \text{⟨assignment⟩}^+ \ \text{else} \ \text{⟨assignment⟩}^+) \ ? \ \text{end if}
\]

3.7.2. Semantics

The simplest case of \text{⟨assign⟩} is an assignment of a value to variable, which simply binds a single value to the name of the variable. Note that variables can only hold simple values, i.e. instances of sort string.
3.7. Assignment

Assigning more complex structures (i.e. tuples and sets) to variables is not permitted. Since this type of assignment merely maps a value to a name it has no influence on $D_{YAGI}$ or $L_{YAGI}$. The second base case of (assign) is the assignment to a variable $F$. Due to the fact that either another fluent $F_0$ or set of constants $σ_F = \{x_1^1, x_1^2, \ldots, x_1^n\}, \ldots, \{x_n^1, x_n^2, \ldots, x_n^n\}$ can be assigned to a fluent $F$ we need to look at both of these cases separately. In the first case, we already have a situation calculus representation we can use to formalize the assignment since the fluent $F_0$ must have been declared first. In the second case we need to construct a situation calculus representation from the set of constants $σ_F$, as follows. We transform $σ_F$ to what we call a shadow fluent. A shadow fluent is the situation calculus representation of $σ_F$, i.e. a fluent $F_0(\vec{x}, s)$ is created with $\vec{x}$ according to the elements in $σ_F$ and $s$ as situation term, the axiom $F_0(\vec{x}, s_0) \equiv \vec{x} = x_1 \lor \vec{x} = x_2 \lor \ldots \lor \vec{x} = x_n$ is added to $D_{s_0}$ and the successor state axiom $\forall \vec{x}. F_0(\vec{x}, do(a, s)) \equiv F_0(\vec{x}, s)$ is added to $D_{sta}$. Note that each $\vec{x}$ in $\forall \vec{x}$ corresponds to one tuple in $σ_F$ and the assignment is only valid iff the arity of the fluents are equal and each element of the assignment belongs to the same domain. Now, having a situation calculus representation for both of the valid assignment cases, we can proceed with the specification of the assignment semantics. Assignments to fluents expand to YAGI programs as follows. Let $F$ be the fluent at the left-hand side of an assignment and let $F_0$ be the fluent at the right-hand side of the assignment, then we need to distinguish the following scenarios:

- **Add-Assignment**: An add-assignment (assignment operator $+=\$) adds all the tuples from $F_0$ to $F$, leaving all other elements in $F$ unchanged. Consequently, given the YAGI assignment $F += F_\sigma$, we can construct a YAGI program as follows:
  
  ```golang
  foreach $x_1, \ldots, x_n$ in $F_\sigma$
  addF($x_1, \ldots, x_n$);
  end for
  ```

  This YAGI-loop essentially adds all the elements from $F_0$ to the fluent $F$ using the situation calculus simple action $addF$. Recall that the situation calculus simple actions $addF$ and $removeF$ are created for every YAGI fluent $F$ at its declaration, see Section 3.3.2. The exact semantics of $foreach$ (i.e. mapping of a YAGI-foreach to Golog) are discussed in Section 3.18.2.

- **Remove-Assignment**: A remove-assignment (assignment operator $-=\$) removes all tuples in $F_0$ from $F$, leaving all other elements in $F$ unchanged. Similar to the add-assignment, given the YAGI assignment $F -= F_\sigma$, we can construct a YAGI program as follows:
  
  ```golang
  foreach $x_1, \ldots, x_n$ in $F_\sigma$
  removeF($x_1, \ldots, x_n$);
  end for
  ```

  Note that the only difference to the YAGI program for the add-assignment is the different situation calculus action $removeF$.

- **Override Assignment**: An override assignment (assignment operator $=\$) makes the fluent $F$ true for all and only all tuples in $F_0$. In other words, an override assignment removes all elements from $F$ and adds all the tuples from $F_0$ to it. Consequently, we can express an override assignment as a remove-assignment $F -= F$ followed by an add-assignment $F += F_0$. Hence, we can specify the override assignment by applying the construction rules for add- and remove-assignment specified above.

Based on the specification of assignments to fluents we can proceed with complex assignment statements. $⟨for\ loop\ assign⟩$ defines an iteration over all tuples in one $⟨setexpr⟩$ with multiple assignments in the loop body. The intention is to provide a convenient way to assign (potentially multiple) fluents to some value that is determined by iterating over a set of tuples. Note that the semantics of the different types of assignments specified above still apply since $⟨for\ loop\ assign⟩$ is basically just a less verbose way to formulate a list of consecutive assignments. The mapping of a $⟨for\ loop\ assign⟩$ to a Golog program works in the same way as the mapping of a $⟨for\ loop⟩$ discussed in Section 3.18, the only difference is that in a $⟨for\ loop\ assign⟩$ only multiple instances of $⟨assignment⟩$ can be executed in the loop body whereas the loop body in a $⟨for\ loop⟩$ consists of an arbitrary $⟨block⟩$. Due to the fact that we specify that it is not permitted to modify the set the loop iterates over inside the loop body we are not able to perform rewriting for override assignments since the expression $F -= F$ would violate this specification. To avoid
Chapter 3. YAGI Language Specification

this specification violation we remove the modification restriction for loop assignments and specify the execution semantics of assignment for-loops as follows. The \langle setexpr \rangle the assignment loop iterates over is evaluated once (and only once) before the loop gets executed. Using this semantics we can make sure that assignment rewritings for assignments like \texttt{F -= F} work correctly.

\langle conditional_assign \rangle is driven by a similar motivation as \langle for_loop_assign \rangle, i.e. to provide a convenient way to formulate (potentially multiple) assignments based on the evaluation of some \langle formula \rangle. One can think of it as conditional branching like if-then-else constructs known from most of the common programming languages, with the restriction that in each of the branches the only type of statement allowed is \langle assignment \rangle. The mapping to a Golog program works in the same way as the mapping of a \langle conditional \rangle discussed in Section 3.16.

Having defined the semantics and rewriting rules of YAGI assignment statements we want to emphasize that the most complex construct we get from rewriting is a loop that iterates over a finite set of tuples and performs adding and removing elements to/from fluents. We can guarantee that even at worst we always deal with finite sets of tuples since we specified a set to always contain finitely many elements and any operation that adds elements to a set (i.e. add-assignment and operator plus) can only occur finitely many times in a program. Hence, any set produced by these operations can only contain a finite number of tuples. Note that the pattern matching extension discussed in Section 3.8 does not contradict that observation in any way. This conclusion becomes immensely important for work we plan to do in the near future, which is to prove that one can compile arbitrary YAGI action effects to situation calculus successor state axioms.

3.7.3. Incomplete Information

The assignments described above exclusively deal with information that is known. One can imagine that there exist various practical cases where one want to express information that is not yet known, but may (or may not) be sensed to its actual value during the lifetime of the agent. Consider for example a fluent that stores the location (e.g. the room) that a robot resides in. Initially, (i.e. on start-up) the robot might not know in what room he is currently residing, but he might be able do narrow down the possibilities during his lifetime. Various approaches of how to deal with incomplete information in different contexts have been discussed by (Etzioni et al., 1992), (Petrick and Bacchus, 2004), (Vassos and Levesque, 2007) and others. For the time being, we’re not able to express something like incomplete knowledge in YAGI. The ingredients we need are on the one hand a syntactical element to denote incomplete information and on the other hand a mechanism to eliminate possible values due to some (external) information. We discuss the latter issue in Section 3.10 and continue with our proposed solution to the former.

To express incomplete information we use the character star (*) at the right-hand side of an assignment to a fluent. Loosely speaking, an assignment of the form \texttt{F = \{<*>\};} expresses that the value of the fluent \texttt{F} is not yet known. Using our running example, the assignment \texttt{at = \{<*>\};} expresses the fact that we don’t know where the robot is, but we do know all the valid assignments, i.e. the powerset of all tuples that can be generated from the finite domain \texttt{S_{at}} = \{r1, r2, r3\}. Due to the fact that the precise semantics of incomplete information in YAGI is not yet clear, we stick with the syntactical specification for the time being and defer the specification of the semantics to future work.

3.7.4. Implementation Remarks

Any implementation shall check that assignments of a \langle setexpr \rangle are semantically valid, i.e. that every element in every tuple of the \langle setexpr \rangle is an element of the sort of the corresponding dimension of the fluent at the left-hand side of the assignment. If an \langle id \rangle (i.e., another fluent) is assigned to the fluent at the left-hand side the assignment is only valid if both fluents (left-hand side and right-hand side of the assignment) have the same arity and the same domains in each dimension. Any other case shall lead to an error. Furthermore, any implementation shall check that the \langle setexpr \rangle (or fluent) the \texttt{foreach} assignment loop iterates over is not modified inside the loop body. Any modification attempt shall result in an error.
Lastly, any attempt to assign incomplete information to a fluent shall be ignored and shall result in a warning.

3.8. Pattern Matching

For YAGI assignments that contain interactions with sets of any kind we introduce a pattern matching functionality inspired by functional programming languages like Scala. Syntactically, we use underscore `"_"` as wildcard character. The set-theoretic semantics of pattern matching is defined as follows. Let \( F \) be a fluent of sort \( S_F \) and \( \sigma = \{ \{ x_1, \ldots, x_n \} \} \) the set that is assigned to \( F \) using a YAGI assignment operator as specified in Section 3.7. Then, it must hold that each element \( x_i, 1 \leq i \leq n \) of \( \sigma \) is an element of the sort of the \( i \)-th dimension of \( F \), i.e. \( x_i \in S_F^i \) and the number of elements in the \( n \)-tuple of \( \sigma \) must be equal to the number of dimensions of the fluent \( F \). Then, assignment works as specified in the section above. Now let \( \sigma' \) be the same as \( \sigma \) except that the \( i \)-th element in the \( n \)-tuple of \( \sigma \) is the wildcard character, i.e. \( \sigma' = \{ \{ x_1, \ldots, x_{i-1}, _, x_{i+1}, \ldots, x_n \} \} \). Now pattern matching applied to \( \sigma' \) leads to \( \sigma'' = \{ \{ x_1, \ldots, x_{i-1}, X_1, x_{i+1}, \ldots, x_n \}, \{ x_1, \ldots, x_{i-1}, X_2, x_{i+1}, \ldots, x_n \}, \ldots, \{ x_1, \ldots, x_{i-1}, X_m, x_{i+1}, \ldots, x_n \} \} \), i.e. for each of the \( m \) elements in the \( i \)-th domain of the fluent \( F \) a new \( n \)-tuple is added to \( \sigma'' \), with the wildcard character replaced with a concrete element \( X_j \in S_F^j \), \( 1 \leq j \leq m \). In some sense, such a replacement of a wildcard symbol with a set of concrete elements resembles grounding (i.e., replacing a program with variables with an equivalent program without variables) for finite domains in answer set programming (ASP), as discussed by (Gelfond and Lifschitz, 1988) and (Lifschitz, 2008). The general case with an arbitrary number of wildcards in a single assignment statement follows the same principle, the difference being that the expansions are equal to the Cartesian product of their corresponding domains, e.g. let the wildcard character be present at two arbitrary positions \( i \) and \( j \) in a tuple that is assigned to a fluent \( F \). Then, pattern matching generates tuples with all elements of \( S_F^i \times S_F^j \) and all the non-wildcarded elements of the original tuple. Having defined the set-theoretic semantics of pattern matching we want to construct YAGI code that implements the expansion as specified above. To be able to express this in YAGI, we need to introduce a new construct called shadow facts.

3.8.1. Shadow Facts

Essentially, shadow facts are ordinary YAGI facts as specified in Section 3.4, with the additional property that they’re internally created and hence not accessible for the developer of a YAGI program. Note that they’re conceptually similar to shadow fluents as specified in Section 3.7.2. A shadow fact is internally created if a fluent is involved in a pattern matching assignment, as follows. Let \( F \) be a fluent of sort \( S_F \) and \( \sigma = \{ \{ x_1, \ldots, x_{i-1}, _, x_{i+1}, \ldots, x_n \} \} \) a set that should be assigned to \( F \), with the wildcard character at the \( i \)-th position. Then, a YAGI fact \( F^* \) is created according to the semantics of fluent/fact declaration discussed in Section 3.3.2 (and 3.4, respectively) and is subsequently assigned to \( F^* = \{ \{ x_1 \}, \{ x_2 \}, \ldots, \{ x_n \} \} \), \( \forall x \in S_F^i \). Note that this assignment can be expressed in YAGI using the rules specified in Section 3.7.2. Having a definition for a shadow fact, we can continue with the specification of the YAGI pattern matching expansion.

3.8.2. YAGI Pattern Matching Expansion

Let there be a YAGI assignment of the form

```
fluent F["a^n", "b^n"];
F += {< >};  //equal to F = {<"a">, <"b">};
```

Then, we can rewrite the pattern matching expansion as YAGI code of the form
Chapter 3. YAGI Language Specification

```
foreach <$chi_1> in F*_1 //F*_1 is the 'shadow fact' of dimension 1 of the fluent F.
F += <$chi_1>;
end for
```

The expanded YAGI code essentially iterates over the shadow fact of the domain of the fluent mentioned at the left-hand side of the assignment and executes the assignment with each value from the domain. Due to the fact that every shadow fact gets assigned with all the values of its corresponding domain and becomes immutable afterwards (according to the definition of a YAGI fact) we can argue that the YAGI foreach loop from above iterates over the complete domain at any given point in time during program execution. The rewriting for the assignment operators -= and = follow similarly. The general case follows the same principle, adding one nested loop for every wildcard character in the assignment. Consider an assignment with two wildcard characters at arbitrary positions \( i \) and \( j \) of the form

```
F += { <$x_1>,..., _i,..., _j,..., $x_n>};
```

Then, the expansion leads to two nested loops of the form

```
foreach <$chi_i> in F*_i
    foreach <$chi_j> in F*_j
        F += { <$x_1>,..., $chi_i,..., $chi_j,..., $x_n>};
    end for
end for
```

Note that these nested loops express exactly the Cartesian product, which we used to specify the set-theoretic semantics of pattern matching. In the general case (i.e., an arbitrary number of wildcards in a single assignment) for each of the wildcards an additional loop that iterates over the corresponding shadow fact is added to the nesting as outlined above. Moreover, note that any variables from the original assignment (\( x_1 \) and \( x_n \) in the example above) remain untouched by the pattern matching rewriting. For the time being, we restrict pattern matching assignments to fluents that have a user-defined set of strings as domain. This restriction is driven by the fact that in the case that the domain of a fluent is the full (countably infinite) set of strings (as defined in Section 3.3.2) we would induce a loop iterating over countably infinitely many elements, which we are not able to express in YAGI. For the time being, pattern matching over finite domains suffices our needs, even though we plan to loosen that restriction in future work.

3.8.3. Implementation Remarks

The same remarks as for assignments (see Section 3.7) apply, i.e., any implementation shall check that assignments of a \(<setexpr>\) are semantically valid, i.e., that every element in every tuple of the \(<setexpr>\) is an element of the sort of the corresponding dimension of the fluent at the left-hand side of the assignment. Any mismatch shall lead to an error. Furthermore, any use of the wildcard character outside of an assignment statement shall result in an error. Lastly, any attempt to use pattern matching on a dimension of a fluent that has the countably infinite set of strings as domain shall result in an error.

3.9. Exogenous Events

3.9.1. Syntax

\(<exogenous_event_decl> ::= exogenous-event \langle id\rangle (\langle var_list\rangle) (assignment)^+ end exogenous-event\)
3.10. Sensing

3.9.2. Semantics

Semantically, exogenous events are equivalent to YAGI actions with an external-modifier, the only difference of exogenous events is the fact that the point in time where an exogenous event gets executed is non-deterministic, i.e. depends on arbitrary external (real-world) events. To cope with this issue, we define the following mode of execution, similar to IndiGolog’s sense-think-act main cycle described by (De Giacomo et al., 2009):

1. Assimilate all pending data generated by exogenous events.
2. Update the underlying domain theory using the data from exogenous events according to the semantics of (assign) discussed in Section 3.7.2.
3. Progress $D_X$ by executing the next YAGI action in the program.
4. Go back to 1.

3.9.3. Implementation Remarks

Any attempt to actively call an exogenous event via a YAGI statement shall result in an error. Furthermore, any implementation shall guarantee that exogenous events are processed as specified above. Moreover, any implementation shall prevent the loss of data provided by exogenous events, i.e. some kind of buffering mechanism as mentioned in Section 1.5.4 shall be implemented.

3.10. Sensing

3.10.1. Syntax

\[
\text{sensing} ::= \text{sense } \langle \text{id} \rangle (\langle \text{varlist} \rangle )? (\text{external} (\langle \text{varlist} \rangle ))? (\text{formula}) \text{ end sense}
\]

3.10.2. Semantics

Sensing actions are specified by (Scherl and Levesque, 1993), (Levesque, 1996), (De Giacomo and Levesque, 1999) and others as actions that can be taken by the agent or robot to obtain information about the state of certain fluents, rather than to change them. Sensing actions are particularly relevant when the initial state of the world is incompletely specified, which is something YAGI allows us to do, as discussed in Section 3.7.3. Similar to the distinction between YAGI actions (without an external modifier) and YAGI setting actions (with an external modifier, see Section 3.5.2) we distinguish between binary sensing actions (without an external modifier) and n-ary sensing actions (with an external modifier). Loosely speaking, the difference is that binary sensing only provides information about whether or not a certain condition holds, i.e. returns a truth value (hence the term binary) and n-ary sensing returns a list of entities rather than a truth value. The idea is similar to the distinction between relational and functional fluents. This distinction is necessary because we plan to use different formalizations for binary and n-ary sensing actions, namely sensed fluent axioms for binary sensing actions as defined by (Levesque, 1996) and sensing result axioms for n-ary sensing actions as defined by (Scherl and Levesque, 2003).

First, we consider the case of binary sensing. (Levesque, 1996) introduced sensed fluent axioms of the form $SF(a, s) \equiv \phi_d(s)$, where $SF$ is a distinguished predicate like Poss, relating the action to the fluent. For example, (Levesque, 1996) use an airport scenario that shows how the action of checking a departure screen is connected to knowing where a certain plane is parked as $SF(check\_departures, s) \equiv \text{Parked(Flight123, gateA, s)}$. In other words, $\phi_d(s)$ gets asserted to a truth value by its corresponding sensing action. The basic action theory is therefore extended with the set of sensed fluent axioms $D_{SF}$ and the
task is to show that \( D \cup D_{SF} \models \phi[s'] \) for a goal formula \( \phi \) in a situation \( s' \). We can map YAGI sensing actions to sensed fluent axioms as follows. Let \( a_y \) be the name of a YAGI sensing action (i.e. the value of \( \langle id \rangle \)) with arity \( m \), where \( m \) denotes the number of parameters for that respective sensing action, i.e. the number of elements in \( \langle \text{varlist} \rangle \). Then we construct the sensed fluent axiom as \( SF(a_y, s) \equiv \phi(a_y(s)) \), with \( \phi(a_y(s)) \) being the formula constructed from \( \langle \text{formula} \rangle \) as discussed in Section 3.6.

Having defined the binary case, we continue with the n-ary case. (Scherl and Levesque, 2003) specified sensing result axioms of the form \( SR(\alpha(\vec{x}), s) = r \equiv \phi(\alpha(\vec{x}, r, s)) \), with \( \alpha \) being the name of the action, \( \vec{x} \) being the parameter vector, \( r \) being the result and \( s \) being the situation term. For example, (Scherl and Levesque, 2003) show a sensing result axiom to obtain information about the weather as \( SR(\text{sense_weather}, s) = r \equiv (r = \text{"sunny" \lor r = \text{"rainy" \lor r = \text{"snow"}}}) \land \text{weather}(s) = r. \) As of yet, there exists no mapping from a YAGI sensing action with an external modifier to a sensing result axiom due to the fact that - as of today - there exists no precise and theoretically sound description of the intended semantics. Moreover, \( SR \) is strongly coupled to functional fluents and there is no syntactical construct to express a functional fluent in YAGI until today. Based on these issues, we decided to stick with the syntactical specification of n-ary sensing for the time being and defer the specification of the exact semantics to future work.

### 3.11. Setting- and Sensing-Actions Revisited

After having defined both setting-actions (setting values of fluents based on externally generated data) and sensing-actions (obtaining information about the state of a fluent) we explicitly want to outline their difference regarding their semantics. Recall that setting-actions (and also exogenous events, for that matter) change the state of the world, i.e. modify the underlying theory, whereas sensing can be considered as a form of cutting down on possible models generated by incomplete information. To clarify the semantic difference we provide a simple example illustrated in Figure 3.1. Consider two fluents \( f \) and \( g \) both declared over the same domain \( \{a, b\} \). Initially, we assign \( f \) to be unknown and \( g \) to a concrete value of the domain. Subsequently, we execute the assignment \( g = f \), leading to the successor state \( S'_0 \). Note that in \( S'_0 \) we end up with four models \( M_1, \ldots, M_4 \) due to the fact that we assign incomplete information to the fluent \( g \), hence we need to generate models for all the possible sets of tuples of the domain of fluent \( f \). Now we can analyze two different successor states, namely \( S''_0 \) generated by sensing the concrete value \( a \) of fluent \( g \) and \( S'''_0 \) generated by setting the fluent \( g \) to the same concrete value \( a \). In the first case we obtained information about the fluent \( g \), namely we sensed that its value is \( a \), hence we eliminate all models that don’t match the sensed information. Consequently, we end up with the single model \( M_2 \) remaining. In the second case we explicitly set the fluent \( g \) to a new value, resulting in a state where still four different models exist and each of which gets updated with the new value from the setting action.

This simple example illustrates that setting and sensing are fundamentally different things even though they might look similar at first sight. Moreover, we hope that this motivational example emphasizes the importance of having both mechanisms in YAGI.
3.12. Transition Semantics Prerequisites

In the following sections, we specify the syntax and semantics of YAGI language constructs that are responsible for program execution. Recall that in the earlier sections of this chapter we exclusively specified YAGI constructs responsible for modeling the state of the YAGI world. In the following sections, we specify program execution semantics using programming constructs from IndiGolog. The semantics of IndiGolog’s programming constructs has been defined by (De Giacomo et al., 2009), using transition semantic predicates \( \text{Trans} \) and \( \text{Final} \). Hence, we can map YAGI statements to IndiGolog language constructs to specify their intended semantics. We discuss the relation between Golog transition predicates and YAGI program execution semantics in more detail in Chapter ?? as this relation becomes particularly important when we argue about specification conformance of our implementation.

3.13. Test

3.13.1. Syntax

\[
\langle \text{test} \rangle ::= \text{test} \langle \text{formula} \rangle ;
\]

3.13.2. Semantics

Tests whether or not a corresponding formula holds. Semantically, it’s the counterpart of IndiGolog’s test action \( \phi \).
Chapter 3. YAGI Language Specification

3.14. Choose

3.14.1. Syntax

\(\langle \text{choose} \rangle \) \(::=\) choose \(\langle \text{block} \rangle\) \(\text{or}\) \(\langle \text{block} \rangle\)\(^{+}\)

3.14.2. Semantics

Non-deterministically chooses one of the given blocks for execution. Semantically, it’s the counterpart of IndiGolog’s \textit{nondeterministic branch} \(\delta_1 \mid \delta_2\).

3.15. Pick

3.15.1. Syntax

\(\langle \text{pick} \rangle \) \(::=\) pick \(\langle \text{tuple} \rangle\) \textbf{from} \(\langle \text{setexpr} \rangle\) \textbf{such} \(\langle \text{block} \rangle\) \textbf{end pick}

3.15.2. Semantics

Non-deterministically picks a \(\langle \text{tuple} \rangle\) from a given \(\langle \text{setexpr} \rangle\) and executes the subsequent block using the picked tuple as parameter, i.e. non-deterministically take a tuple from \(\langle \text{setexpr} \rangle\), bind its values to fresh variables in the tuple provided by the \(\langle \text{tuple} \rangle\)-expression and execute the \(\langle \text{block} \rangle\) with this variable assignment. Any attempt to state something different than a variable in the \(\langle \text{tuple} \rangle\) of the \textit{pick}-statement is not permitted. Semantically, it’s the counterpart of IndiGolog’s \textit{non-deterministic choice of argument} \(\pi v \delta\).

Note that besides fresh variables that will be bound to a value by the \textit{pick} statement as described above a \(\langle \text{tuple} \rangle\) may also contain variables that are already bound to a value. In this case we simply use the already available value instead of binding the variable via the \textit{pick}-statement. To clarify the semantics, we provide examples for the different cases using the domain of our running example, as follows.

```plaintext
at = { <"r1", "r2", "r3">};
//$x is unbound...
pick <$x> from at;
//...hence its value after the pick is either "r1", "r2", or "r3"
Listing 3.7: Pick With Unbound Variable

//Bind $x to a constant
$x = "r1";
//$x is already bound...
pick <$x> from at;
//...hence its value stays exactly the same after the pick
Listing 3.8: Pick With Bound Variable

is_at = { <"o1","r1">, <"o2","r2">, <"o1","r3">};
//Bind $x to a constant
$x = "o1";
//$x bound, $y unbound
pick <$x,$y> from is_at;
```

28
3.15.3. Implementation Remarks

Any implementation shall check that only variables are stated in the \(\langle\text{tuple}\rangle\)-expression. Any attempt of stating a constant shall result in an error.

3.16. Conditional

3.16.1. Syntax

\[
\langle\text{conditional}\rangle ::= \text{if } \langle\text{formula}\rangle \text{ then } \langle\text{block}\rangle \text{ else } \langle\text{block}\rangle? \text{ end if}
\]

3.16.2. Semantics

Executes one of two given blocks based on the evaluation of \(\langle\text{formula}\rangle\). Semantically, it’s the counterpart of IndiGolog’s \textit{synchronized conditional} \(\text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2 \text{ endIf}\).

3.17. While Loop

3.17.1. Syntax

\[
\langle\text{while_loop}\rangle ::= \text{while } \langle\text{formula}\rangle \text{ do } \langle\text{block}\rangle \text{ end while}
\]

3.17.2. Semantics

Executes a block as often as \(\langle\text{formula}\rangle\) holds. Semantically, it’s the counterpart of IndiGolog’s \textit{synchronized loop} \(\text{while } \phi \text{ do } \delta \text{ endWhile}\).

3.18. For Loop

3.18.1. Syntax

\[
\langle\text{for_loop}\rangle ::= \text{foreach } \langle\text{tuple}\rangle \text{ in } \langle\text{setexpr}\rangle \text{ do } \langle\text{block}\rangle \text{ end for}
\]

3.18.2. Semantics

Executes a \(\langle\text{block}\rangle\) for every tuple in a given \(\langle\text{setexpr}\rangle\). Due to the fact that IndiGolog has no specification for this kind of program flow construct, we rewrite \(\langle\text{for_loop}\rangle\) into \(\langle\text{while_loop}\rangle\) as follows. Let \(F\) be an \(n\)-ary fluent and consider
to be a YAGI for loop over $F$. We transform this loop into the following YAGI code:

```yagi
foreach <$x_1, x_2, \ldots, x_n> in F do
  <block>
end for

F^* = F;
while (exists <$x_1, x_2, \ldots, x_n> in F^*) do
  pick <$x_1, x_2, \ldots, x_n> from F^* such
  <block>
  F^* -= <$x_1, x_2, \ldots, x_n>);
end pick
end while
```

with $F^*$ being a copy of the fluent $F$. Note that all the identifiers ending with a star (*) are no valid names according to the specification of $⟨id⟩$. These names were purposely chosen to contradict the syntactical specification to illustrate that these names are chosen internally by the interpreter, i.e. it’s syntactically impossible for a programmer to access these elements. The transformation checks if a tuple exists in $F^*$ via the while-condition and subsequently binds a tuple from $F^*$ to the variables $x_1, x_2, \ldots, x_n$ via the pick-statement, making it semantically equivalent to an execution of the statement `foreach <$x_1, x_2, \ldots, x_n> in F`. Then, the same ⟨block⟩ as in the for-loop gets executed. Finally, the chosen tuple is removed from the fluent $F^*$. Note that this transformation works correctly if and only if the value of the fluent the `foreach`-loop iterates over is not modified in its loop body, hence we don’t permit any modifications of the fluent the `foreach`-loop iterates over in the loop body.

To justify the claim that the rewritten loop above satisfies the specified semantics we argue inductively, as follows:

- For the base case, let $F$ be a fluent that doesn’t hold for any parameter vector, i.e. $\forall \vec{x}. F(\vec{x}, s) \equiv False$. Since $F^*$ is specified to be a copy of the fluent $F$ it also holds that $\forall \vec{x}. F^*(\vec{x}, s) \equiv False$. Consequently, the YAGI formula `exists <$x_1, x_2, \ldots, x_n> in F^*` evaluates to `False` because it translates to $\exists \vec{x}. F^*(\vec{x}, s)$ according to the specification of YAGI formulas in Section 3.6.2. Thus, the while loop becomes `while (False) do` and the code in the while-block doesn’t get executed.

- For the inductive step we assume that for every fluent with $k$ tuples the transformation is correct. For any arbitrary Fluent $F$ that holds for a set of parameter vectors $S = \{\vec{x}_1, \ldots, \vec{x}_{k+1}\}$ the YAGI formula `exists <$x_1, x_2, \ldots, x_n> in F^*` evaluates to `True` because it holds that $\exists \vec{x}. F^*(\vec{x}, s)$ and `$<$x_1, x_2, \ldots, x_n$>` corresponds to one parameter vector $\vec{x}_i \in S$. Consequently, the ⟨pick⟩ statement in the loop body gets executed, i.e. a tuple is non-deterministically picked from $F^*$. Due to the fact that $\exists \vec{x}. F^*(\vec{x}, s)$ holds ⟨pick⟩ is guaranteed to succeed. Subsequently, an arbitrary ⟨block⟩ gets executed for the picked tuple and the very same tuple is removed from $F^*$ in the last statement of the ⟨pick⟩-block. Due to the fact that we don’t permit any modifications of the fluent the `foreach`-loop iterates over in the loop body the line $F^* -= <$x_1, x_2, \ldots, x_n$>; is guaranteed to be the only line that modifies $F^*$. Hence, after one iteration of the while-loop it is guaranteed that the number of tuples for which $F^*$ holds is decreased by one, thus the claim holds by induction. If no elements remain it holds that $\forall \vec{x}. F^*(\vec{x}, s) \equiv False$, which is exactly the base case described in the section above.

### 3.18.3. Implementation Remarks

Any implementation shall check that the fluent the `foreach`-loop iterates over is not modified inside the loop body. Any modification attempt shall result in an error.

### 3.19. Search

#### 3.19.1. Syntax

⟨search⟩ ::= search ⟨block⟩ end search
3.19.2. Semantics

Like IndiGolog, YAGI uses an online execution semantics as defined by (De Giacomo and Levesque, 1999) and (De Giacomo et al., 2009). To be able to introduce offline execution semantics for certain parts of a YAGI program, we specify the operator `search`. The operator `search` applies offline execution semantics to a YAGI ⟨block⟩ it is applied to. In offline execution mode, YAGI searches for an appropriate sequence of actions before actually executing any of it. Note that `search` can - syntactically - be applied to an arbitrary ⟨block⟩, which imposes several issues. For example, recall that YAGI supports sensing actions that can potentially be called in such a ⟨block⟩. In the context of offline execution this implies that the system must be able to take potential sensing results into account during offline deliberation. Due to the fact that dealing with incomplete knowledge during offline execution is a non-trivial task (potential approaches have already been discussed by (Levesque, 2005), (Vassos and Levesque, 2007) and others) we’re not able to provide a sound solution on how to approach this issue in YAGI for the time being. Other constructs that would further increase the complexity of `search` are setting actions and exogenous events since they both deal with data from external sources. This implies that there need to be a strategy of how to model these external influences when doing offline deliberation. For the time being, we decided to exclude setting actions, sensing and exogenous events from search and defer the specification of their exact offline deliberation semantics to future work. Moreover, note that with these restrictions we stay consistent with IndiGolog’s `search operator` `Σ`, which we consider to be the semantic counterpart of `search` in YAGI. More formally, (De Giacomo et al., 2009) define the semantics of offline execution as
\[
\text{Do}(\delta, s, s') = \exists \delta'. \text{Trans}^*(\delta, s, \delta', s') \land \text{Final}(\delta', s'),
\]
where `Trans*` is the reflexive transitive closure of `Trans`\(^3\). We discuss the semantics of `Trans` and `Final` in more detail in Chapter ??.

3.19.3. Implementation Remarks

Any implementation shall check that no sensing-/setting-action or exogenous event is part of a search-⟨block⟩. Any appearance of any of these constructs in a search-⟨block⟩ shall result in an error.

3.20. Procedure Declaration

3.20.1. Syntax

\[
\langle \text{proc\_decl} \rangle ::= \text{proc} \langle \text{id} \rangle \ (\langle \text{var\_list}\rangle?) \ \langle \text{block} \rangle \ \text{end proc}
\]

3.20.2. Semantics

Declares a procedure with a name and a (possibly empty) list of parameters, leaving \(D_{YAGI}\) and \(L_{YAGI}\) unchanged. Semantically, `procedure declaration` is the counterpart of IndiGolog’s `procedure definition` `proc P(x) δ endProc`, i.e. the same semantics and restrictions as defined by (Levesque et al., 1994) apply.

3.20.3. Implementation Remarks

Any implementation shall ensure that procedures are unique. We define uniqueness for procedures as follows. Given a procedure \(P\) with arity \(m\) we say that the name-arity tuple \(⟨P, m⟩\) must be unique, i.e. two procedures are equal iff they have the same name and the same arity. Any redeclaration of an already declared procedure overrides the former with the latter and shall result in a warning.

\(^3\)`Trans*` can be defined as a situation calculus second-order formula. For the sake of simplicity we omit the details here and refer to (De Giacomo et al., 2009) for more details.
3.21. Procedure Call

3.21.1. Syntax

\[ \langle \text{proc\_call} \rangle ::= \langle \text{id} \rangle (\langle \text{arglist} \rangle) ; \]
\[ \langle \text{arglist} \rangle ::= \langle \text{value} \rangle (, \langle \text{value} \rangle)^* \]

3.21.2. Semantics

The execution of a YAGI procedure is the counterpart of Golog's procedure call \( P(\theta) \). Since we map both YAGI actions and YAGI procedures to Golog procedures the concept of a YAGI action call vanishes, hence we need no additional specification for calling YAGI actions. Arguments (i.e. elements in \( \langle \text{arglist} \rangle \)) are passed in a call-by-value manner. Note that IndiGolog also specifies the semantics of calling a primitive action, i.e. a situation calculus action. Due to the fact that the only primitive actions in YAGI are the actions add and remove (which are automatically generated for each declared fluent, see Section 3.3.2) and neither of these types of actions should be invoked explicitly by a YAGI programmer we don’t need a syntactic construct that maps to IndiGolog’s primitive action call.

Atomicity of YAGI Action Execution

We specify the execution of a YAGI action (or more precisely, a procedure that has been generated from a YAGI action) to be atomic. The atomicity of the execution of a YAGI action is particularly important in the context of search since our implemented search strategy considers the execution of a YAGI action as fundamental step that shall not be interrupted. Moreover, data generated by exogenous events is assimilated after a YAGI action has been executed (as specified in Section 3.9.2), which guarantees that one single YAGI action always gets executed with respect to one specific model of the world. Lifting the restriction of atomicity of YAGI action executions might lead to inconsistent and/or undefined behavior. Finally, we want to note that we consider this level of atomicity as the most natural from a user's perspective, which influenced that decision as well. Still, there might be arguments for making the atomicity level more fine-grained than the execution of a single YAGI action, but due to the fact that we are not able to foresee the theoretical and practical implications of such a decision we defer this discussion to future work.

Recursion

Even though ConGolog (De Giacomo et al., 2000) as well as IndiGolog (De Giacomo et al., 2009) have definitions for unbounded recursive calls we decided to forbid recursive procedure calls in YAGI for the time being. Unbound recursive calls require second-order logic extensions for Trans and Final as discussed by (De Giacomo et al., 2000), which we want to avoid for the sake of simplicity of our specification.

3.21.3. Implementation Remarks

To avoid any ambiguities about whether to call a procedure that has been automatically created from a YAGI action or a procedure that has been explicitly declared by the programmer any implementation shall check that names of YAGI actions and YAGI procedures are distinctly and mutually unique, i.e. any two YAGI actions must not be equal, any two YAGI procedures must not be equal and any YAGI action must not be equal. For equality comparison we use name-arity tuples as defined for procedure uniqueness in Section 3.20.3. Any violation of this uniqueness property shall result in an error.
3.22. Fluent/Fact Query

3.22.1. Syntax

⟨fluent_query⟩ ::= ⟨id⟩ ;

3.22.2. Semantics

The fluent query command has no influence on the underlying domain theory or program execution whatsoever, its purpose is solely to echo the state (i.e. the assignment) of the fluent that is being queried. Note that due to the fact that ⟨fluent_query⟩ is a ⟨statement⟩ it could be part of any program ⟨block⟩. For the time being, we only define that if the whole program consists of just a ⟨fluent_query⟩ its semantics is that it returns the state of the queried fluent to the caller (i.e. the front-end) or false if the fluent (or fact) doesn’t exist. Any use of a fluent query inside a more complex YAGI program should be ignored gracefully.

3.23. Include

3.23.1. Syntax

⟨include⟩ ::= @include ⟨string⟩ ;

3.23.2. Semantics

The include command has no direct influence on the underlying domain theory or program execution whatsoever, its purpose is solely to import YAGI code from different files into a single file. More precisely, the semantics of ⟨include⟩ is that the whole include command gets replaced by the YAGI code from the file which name is provided via the ⟨string⟩ value. This semantics is identical to the semantics of macro replacement performed by the preprocessor in the C language.

3.24. Sequence

3.24.1. Syntax

⟨block⟩ ::= ⟨statement⟩+

⟨statement⟩ ::= ⟨test⟩
| ⟨proc_call⟩
| ⟨choose⟩
| ⟨pick⟩
| ⟨conditional⟩
| ⟨while_loop⟩
| ⟨for_loop⟩
| ⟨search⟩
| ⟨fluent_query⟩
3.24.2. Semantics

A sequence of YAGI statements. Semantically, it’s the counterpart of IndiGolog’s sequence \( \delta_1; \delta_2 \). Note that the only valid statements in a YAGI ⟨block⟩ are exactly the control flow statements (with their defined IndiGolog counterparts) as specified in the sections above, hence we can establish this correspondence between a YAGI ⟨block⟩ and an IndiGolog sequence.

3.25. YAGI Program

3.25.1. Syntax

\[
⟨\text{program}⟩ ::= ((⟨\text{declaration}⟩ | ⟨\text{block}⟩ | ⟨\text{include}⟩)^+ \\
⟨\text{declaration}⟩ ::= ⟨\text{fluent}\_\text{decl}⟩ | ⟨\text{fact}\_\text{decl}⟩ | ⟨\text{action}\_\text{decl}⟩ | ⟨\text{proc}\_\text{decl}⟩ | ⟨\text{exogenous}\_\text{event}\_\text{decl}⟩ | ⟨\text{sensing}\_\text{decl}⟩ | ⟨\text{assignment}⟩
\]

3.25.2. Semantics

Finally, we call arbitrary sequences of YAGI lines of code \( ⟨l_1, \ldots, l_n⟩ \) a YAGI program. A line in a YAGI program can either be a declaration modifying the state of the YAGI world (except ⟨proc_decl⟩), i.e. the underlying theory (as specified in the first part of this chapter) or any sequence of program flow statements (as specified in the second part of this chapter) that specify the program, i.e. a ⟨block⟩ or a ⟨proc_decl⟩. The semantics of a YAGI program is then given by the consecutive execution of its lines of code in their given order according to the transition semantics of IndiGolog defined by (De Giacomo et al., 2009). We restate the exact definitions of the IndiGolog transition semantics and show their correspondence to YAGI in Chapter ??.
Bibliography


Bibliography
