Go! – A Logic Programming Language for Implementing Multi-threaded Agents

K.L. Clark*  F.G. McCabe
klc@doc.ic.ac.uk  fgm@fla.fujitsu.com

March 17, 2003

Abstract

Go! is a multi-paradigm programming language that is oriented to the needs of programming secure, production quality, agent based applications. It is multi-threaded, strongly typed and higher order (in the functional programming sense). It has relation, function and action procedure definitions. Threads execute action procedures, calling functions and querying relations as need be. Threads in different agents communicate and coordinate using asynchronous messages. Threads within the same agent can also use shared dynamic relations acting as Linda-style tuple stores.

In this paper we introduce the essential features of Go!. We then illustrate them by programming a simple multi-agent application comprising hybrid reactive/deliberative agents interacting in a simulated ballroom. The dancer agents negotiate to enter into joint commitments to dance a particular dance (e.g. polka) they both desire. When the dance is announced, they dance together. The agents’ reactive and deliberative components are concurrently executing threads which communicate and coordinate using belief, desire and intention memory stores. We believe such a multi-threaded agent architecture represents a powerful and natural style of agent implementation, for which Go! is well suited.

*Alphabetical order of authors
1 Introduction


Go! is a logic programming descendant of the multi-threaded symbolic programming language April[16]. April was initially developed as the implementation language for the much higher level MAI²L[10] agent programming of the EU Imagine project. It has more recently been used to implement one of the FIPA compliant agent platforms of the EU AgentCities project[23], and the agent services running on that platform at Imperial College and Fujitsu.

A significant theme in the design of Go! is software engineering in the service of high-integrity systems. To bring the benefits of logic programming to applications developers requires fitting the language into current best-practice; and, especially since applications are increasingly operating in the public Internet, security, transparency and integrity are critical to the adoption of logic programming technology.

Although Go! has many features in common with Prolog, particularly multi-threaded Prolog’s such as Qu-Prolog[6], there are significant differences related to transparency of code and security. Features of Prolog that mitigate against transparency, such as the infamous cut (!) primitive, are absent from Go!. Instead, its main uses are supported by higher level programming constructs, such as single solution calls, iff rules, and the ability to define ‘functional’ relations as functions.

In Prolog, the same clause syntax is used both for defining relations, with a declarative semantics, and for defining procedures, say that read and write to files, which really only have an operational semantics. In Go!, behaviours are described using action rules which have a different syntax.

While Prolog is a meta-order language, Go! is higher-order (in the functional programming sense). Functions can be defined using rewrite rules and code can be treated as a value. A key feature of Go! is the theta construct used for structuring programs. It is analogous to the let ... in ... construct of some functional programming languages. It is used in Go! both as a module mechanism as well as a lexical scoping mechanism.

Go! is strongly typed – using a modified Hindley/Milner style type inference technique [11], [18] to reduce the programmer’s burden. The programmer can declare new types and thereby introduce new data constructors.

\footnote{Go! is not higher order in the higher order logic sense. There is no higher order unification of function or relation definitions, as there is in lambda-Prolog[17]}
Indeed, all uses of functors as data constructors need to be declared by being introduced in some type definition. Compile time type checking improves code safety.

Go! also has an optional object oriented syntax based on McCabe’s L&O [15] extension of Prolog. This is a class based OO syntax allowing the grouping together of a set of definitions that share updatable state. Each instance of a class has its own state. State can be recorded as values held in instances of two primitive object classes - a cell class, instances of which hold a single updatable value, and a dynamic relations class, instances of which hold sets of terms of the same type. Subclasses can be defined using multiple inheritance. This OO layer gives a rich knowledge representation notation akin to that of frame systems.

Go! is also multi-threaded, a feature which we consider is essential for building sophisticated agents. Threads primarily communicate through asynchronous symbolic message passing as in April[16]. Threads, executing action rules, react to received messages using pattern matching and pattern based message reaction rules. A communications daemon enables threads in different Go! processes to communicate transparently over a local network. Typically, each agent will comprise several threads, each of which can separately communicate with threads in other agents. Alternatively, each agent can have an external interface thread and all inter-agent communication is via this thread which acts as a message router for the agents internal threads.

Threads in the same Go! process, hence in the same agent, can also communicate by manipulated shared cell or dynamic relation objects. Updates of these objects are atomic. Moreover, threads can be made to suspend until a term unifying with some given term is added to the relation by some other thread. This enables dynamic relations to be used like Linda tuple stores[3] to coordinate the activities of different threads within an agent. This is a powerful implementation abstraction for building multi-threaded agents, which we will illustrate with our example application of Go!.

We illustrate Go! using a simple multi-agent application comprising hybrid reactive/deliberative agents interacting at a simulated ball. In this ballroom there are male and female dancers. The dancers have desires that reflect – somewhat – a typical ballroom situation with the different desires corresponding to individual preferences. Dancers negotiate with each other over up-coming dances and also make counter proposals – such as going to the bar instead of having a dance.

Each dancer agent is programmed using multiple concurrently execut-
2 Key Features of Go!

Go! is a multi-paradigm language with a declarative subset of function and relation definitions and an imperative subset comprising action procedure definitions.

2.1 Function, relation and action rules

Functions are defined using sequences of conditional rewrite rules of the form:

\[ f(A_1, \ldots, A_k) :: \text{Test} \Rightarrow \text{Exp} \]
The guard Test is optional. The :: can be read as such that.

As in most functional programming languages, the testing of whether a function rule can be used to evaluate a function call uses matching not unification. Once a function rule has been selected there is no backtracking to select an alternative rule.

An example function definition is:

\[
\begin{align*}
nrev([]) & => [] . \\
nrev([E,..X]) & :: append(nrev(X), [E], R) => R. \\
\end{align*}
\]

.. is Go!‘s list constructor, equivalent to the LISP cons and the Prolog |.

Relation definitions comprise sequences of Prolog-style :- clauses; with some modifications – such as permitting expressions as well as data terms, and no cut. An example is the familiar append definition:

\[
\begin{align*}
append([], X, X). \\
append([E,..X], Y, [E,..Z]) & :- append(X, Y, Z). \\
\end{align*}
\]

Instead of the cut, Go! has negation-as-failure (\+) and a conditional test as primitives, and relations can be defined using sequences of iff rules. These have the syntax:

\[
p(A_1,..,A_k):: Test ::-- Condition
\]

where Test is optional, but not Condition. As with function rules, once an \(iff \) rule has been found that unifies with a relation call and any associated test succeeds, there is a commitment to use just that rule. \(iff \) rules allow relations to be defined using disjoint cases, each case being represented as a different head pattern/test of a rule. An example of a definition using \(iff \) rules is:

\[
\begin{align*}
partition([], _, [], [], []) & ::-- true. \\
partition([E,..X], Pivot, [E,..A], B):: E<Pivot ::-- partition(X, Pivot, A, B). \\
partition([E,..X], Pivot, A, [E,..B]):: E>=Pivot ::-- partition(X, Pivot, A, B). \\
\end{align*}
\]

This is the definition of a relation that can be used to partition a list of numbers into those less that some \(Pivot\) value and those greater than or equal to its value.

The locus of action in Go! is a thread; each Go! thread executes a procedure. Procedures are defined using non-declarative action rules of the form:
As with equations, the first action rule that matches some call, and whose test is satisfied, is used; once an action rule has been selected there is no backtracking on the choice of rule.

An example action rule is:

```go
runMe(Count) ->
    stdout.outLine(nrev(iota(1,Count))^0)
```

This defines a single number argument procedure `runMe`. The body of the rule is the single action:

```go
stdout.outLine(nrev(iota(1,Count))^0)
```

This uses the Go! primitive function `iota` to construct a list of numbers from 1 to `Count`, uses `nrev` to reverse this list, and uses the standard operator `^` to convert this list into a string. `^` will convert any Go! data value into a string. This string, followed by a newline, is written to the standard output channel (usually the terminal window from which the program was invoked) using the procedure `outLine` of the `stdout` file object.

More generally, the permissible actions of an action rule include: message dispatch and receipt, I/O, updating of dynamic relations, the calling of a procedure, and the spawning of any action, or sequence of actions, to create a new action thread.

**Invoking queries from actions** The declarative part of a Go! program can be accessed from action rules in a number of ways:

- Any expression can invoke functions.
- An action rule guard \((A_1,..,A_k)::Q\) can extend the argument matching with a query \(Q\).
- If \(Q\) is a query, \(\{Q\}\), indicating a single solution to \(Q\), can appear as an ‘action’ in an action rule body.
- We can use a set expression \(\{Trm \mid Q\}\) to find all solutions to some query. This is Go!’s `findall`.
• We can use Go!’s `forall` action. \((Q \ast \to A)\) iterates the action \(A\) over all solutions to query \(Q\).

• We can use a conditional action. \((Q ? A_1 \mid A_2)\) executes \(A_1\) if \(Q\) succeeds, else \(A_2\).

**Invoking actions from queries** Occasionally it is necessary to execute an action inside a query or relation definition although this style of programming is discouraged in Go!. If it is done, then the action being called must be prefixed by the `action` operator to signal its use. We shall give an example of this in section 3.2.

### 2.2 Message communication

Threads in a single Go! invocation can communicate either by thread-to-thread message communication or by synchronisable access and update of shared data, such as dynamic relations. Threads in different Go! invocations can only communicate using messages. To support thread-to-thread communication, each thread has its own buffer of messages it has not yet read, which are ordered in the buffer by time of arrival.

**Sending messages** The message send action:

\(\text{Msg} \gg \text{To}\)

sends the message \(\text{Msg}\) to the thread identified by the handle \(\text{To}\). Go!’s message send is asynchronous. Successful dispatch means neither that the recipient task has ‘accepted’ the message nor that it actually received it. This is because message acceptance is an active process on the part of the receiver. Also, the destination task may be no longer executing. Every thread has a single associated message queue into which all sent messages are placed in time order of arrival.

**Multi-casting** The message send primitive sends to just one other thread, there is no primitive multicast or broadcast facility in Go!. However, a message can be sent to all the threads whose handles are in some list \(\text{Handles}\) using Go!’s `forall` operator:

\((H \text{ in } \text{Handles} \ast \to \text{Exp} \gg H)\)
This is read as:

\[ \text{forall } H \text{ in the list Handles, send value of Exp to } H. \]

in is Go!’s list membership primitive. More generally, we can use \( \ast \ast \) to send a message \( M \) to thread \( H \) for each \( M, H \) satisfying some arbitrary query condition \( \text{Cond}(M,H) \):

\[ (\text{Cond}(M,H) \ast \ast M \ast \ast H) \]

For example:

\[ (a_{\text{current\_sub\_task}}(Task), I_{\text{cannot\_do}}(Task), cando(Ag,Task) \ast \ast ('request',Task) \ast \ast Ag) \]

**Receiving messages** A thread can search for and remove a message from its message queue using a simple \( \ll \) message receive action, or by executing a choice message receive action. A \( \ll \) action has the form:

\[ \text{MsgPtn} \ll \text{Sender} \]

It removes from the message queue the first message that matches \( \text{MsgPtn} \) which comes from the thread identified by \( \text{Sender} \). Typically \( \text{Sender} \) is an unbound variable which will be bound to the identity of the sender. If not, it serves as a check on the identity of the sender, and only messages from that sender will be accepted.

If there is no message in the thread’s message queue that matches \( \text{Msg} \), and is from \( \text{Sender} \), then the \( \ll \) call suspends until such a message is received. The message receive is a match operation, not a unification, because no variable in the message term can be bound by the message receive. Any attempt to bind a variable in a message results in the match failure, and the next message in the queue is tested for a match.

Note that it is often useful to use a test pattern, with an associated condition, as the message term pattern. For example, the message receive action:

\[ \ldots; N::N<10 \ll S; \ldots \]

looks for a number from thread \( S \) whose value is less than 10. Numeric messages that are greater than 10, or messages from threads whose handles do not unify with \( S \), will not be picked up by this message receive. Using test patterns in message receive actions is quite powerful — as it allows an action to search for, and if necesary wait for a particular form of message, from a specified thread.
Choice message receive  Go! also has a more powerful mechanism for receiving messages: the choice message receive. These allow a thread to search for one of several different forms of message in a single sweep of its message queue and to execute a different action depending on which form of message it finds.

The semaphore procedure:

```prolog
semaphore(F) ->
  ( 'req':F>0 << H -> 'ok'>H; semaphore(F-1)
    | 'free' << H -> semaphore(F+1)
  ).
```

uses a choice message receive comprising a disjunction of two message rules. Each (recursive) execution of the choice receive causes a search of the thread’s message buffer, starting at the front of the buffer, looking for either a 'req' message, or a 'free' message. If a 'req' from a sender H – and the value of F is greater than zero – then the 'req' message is removed from the queue and the procedure sends an 'ok' acknowledgement back to the sender of the message. The semaphore procedure then recurses\(^3\) with a decremented value for F.

Otherwise, if a 'free' message is found first, or if F is not currently positive causing all 'req' messages to be skipped over until a 'free' is found, the 'free' message is removed and we enter a tail recursive call with an incremented value of F. The execution of the procedure will suspend if there is no message satisfying the message pattern of either rule.

More generally, a choice message receive action operates by examining each of the messages in the message queue; applying each of the message patterns of its disjunction of message rules to each message in turn. As soon as a message is found that satisfies a message pattern of some rule \(R\), the message is removed from the queue and the action sequence of rule \(R\) is executed – none of the other actions in the choice message receive will be executed. Execution of the choice message receive will suspend if there is currently no message satisfying any of the alternative message patterns. A message receive terminates only when such a message is received.

\(^2\) Notice that 'req" is quoted. Go! does not use the Prolog convention that all alphanumeric names beginning with a lower case letter are symbols and all those beginning with an upper case letter are variables. Symbols have to be explicitly quoted and unquoted alphanumeric name can be used as a variable.

\(^3\) Go! is fully tail recursive, so such a recursion is equivalent to an iteration.
Message passing is the primary way that Go! threads communicate with each other. They can also communicate using a shared cell variable or a shared dynamic relation. More generally, we can use one thread as a data manager. The other threads then send data update and data access messages to the data manager thread. This approach allows threads executing in different Go! invocations to access and update shared information.

2.3 Multi-threading

Any Go! thread can spawn new threads. These are calls to other action procedures that are executed concurrently with the spawning thread as new execution threads. (The evaluation of all the threads is time shared by the Go! run-time system.) Each spawned thread also has its own message buffer and handle identity.

As an example:

```
sem = spawn semaphore(3)
```

launches a new thread executing the `semaphore` procedure to control three resources. The identity of the thread, its handle is returned by the spawn and assigned to variable `sem`. The parent thread, and any other thread given access to the handle `sem`, can send messages to the new thread using:

```
...; 'req' >> sem; 'ok' << sem; ...
```

The 'ok' message receive action will suspend until reply from `sem` is sent.

The handle identity of `sem` is a term of the form:

```
hdl(symbol,symbol)
```

where the symbol components are generated by Go!'s runtime system. The programmer can also assign these components when a thread is launched, this giving a 'public' name to the thread which can be used throughout the program.
spawn semaphore(3) as hdl('interface','semaphore');

...;

'req' >> hdl('interface','semaphore');

2.4 Theta environments

In many ways, theta environments form the 'heart' of Go! programs: they are where most programs are actually defined; they are also the only place where new types may be defined. The scope of the type definition is the theta expression in which it appears.

A theta environment is of a set of definitions, each of which is either a

\[ \text{Var} = \text{Expression} \]

assignment definition, a

\[ \text{Type ::= TypeExpression} \]

or a sequence of relation, function, action rules or DCG grammar rules[?]. The rules and definitions are separated by the '.␣' operator. Communication daemons and a special external communications system module allow threads in different invocations of Go! to communicate using the same message send and receive actions as are used between threads of a single invocation, see [5].

This features makes use of programmer assigned public names. It allows an application comprising several modules, developed and tested as one multi-threaded Go! invocation, to be converted into a distributed application with minimal re-programming. The following is a module the body of which is a theta environment. The module exports the higher order function sort. The definitions of the auxiliary relations srt and partition are local to the theta environment and not visible outside. <> is Go!'s list append primitive. The . can be read as where. A module is a where expression that evaluates to a tuple of values, the values exported from the module.

sort .. { 
  sort(list,less) => srt(list) .. { 
    srt([]) => [].
    srt([E,..X]) :: partition(X,E,A,B) => srt(A)<>[E,..srt(B)].
  }
}

---

4Where '.␣' means a period followed at least one whitespace character.
2.5 Higher order values

The `sort` function is *parameterized* with respect to the ordering relation used to compare elements of the list. `sort` is further defined in terms of the auxiliary definitions for `srt` and `part`, themselves introduced in a subsidiary theta expression. This illustrates how theta expressions may be used at many levels – not just the top-level of a program. Note that the `less` variable – which holds the ordering relation – is only mentioned where it is important: where it is introduced as a parameter of `sort` and where it is used in `part`. This is an example of variables having a somewhat extended scope compared to `Prolog`. In `Prolog`, to achieve the same effect, we would have had to ‘pass down’ the `less` relation through all the intermediate programs from the top-level to where it is needed; this is a significant source of irritation in `Prolog` programming.

A call to `sort` must supply the list to be sorted and an ordering relation. In many cases the ordering relation is given as the value of a variable with a higher order value\(^5\); however, it is also possible to use a lambda rule, or a disjunction of such rules, to give an on-the-fly definition of the relation. For example, the call:

```
sort([(3,5),(-1,2),(10,12),...],
     ((X1,_),(X2,_)):-X1=<X2 | ((X,Y1), (X,Y2)) :-Y1=<Y2))
```

Here the expression

```
((X1,_),(X2,_)):-X1=<X2 | ((X,Y1), (X,Y2)) :-Y1=<Y2)
```

is a disjunction of lambda relation rules that uses the standard `=<` relation to define an ordering on pairs of numbers.

\(^5\)Note the contrast with `Prolog`. In `Prolog` a relation is passed as argument by passing in its name - which is an atom. `Prolog`'s metal level `call` is then used to map the name to the value at run-time by accessing a run-time dictionary linking atom names with code values. In `Go!` the code value is passed, not the name. More on this later.
2.6 Executable programs

An executable program is a theta expression that evaluates to a single procedure value, this is the procedure that will be called when the program is run. As an example:

```go
runMe .. {
    include "sys:go/fileio.gof".
    -- defn of nrev function
    -- def of app relation
    runMe(Count) ->
        stdout.outLine(nrev(iota(1,Count))^0)
}
```

is an executable program that can be run from the command line which includes the integer value to be used as the `Count`. The `include` statement loads the system I/O module that amongst other things exports the `stdout` object.

2.7 Types

Go! is a strongly typed language; using a form of Hindley/Milner’s type inference system[18]. For the most part it is not necessary for programers to associate types with variables or other expressions. However, all constructors and `unquoted` symbols are required to be introduced using type definitions. If an identifier is used as a function symbol in an expression it is assumed to refer to an ‘evaluable’ function unless it has been previously introduced in a type definition.

The type of a non-polymorphic relation, such as a relation whose extension is a set of pairs of lists of `numbers`, is denoted by the type expression:

```
(number[],number[]){}
```

Here, `number[]` is the Go! type expression for list of numbers, and

```
(number[],number[])
```

is the type expression for a pair of lists of numbers. The postfix `{}` following this type expression signals a set of pairs of lists of numbers, i.e. a binary relation over lists of numbers.
In Go! the type expressions for polymorphic relation, function and procedure types contain type variables instead of specific type names, and they are quantified with respect to these type variables. Type quantification is denoted by the \([vars]−type\) form, read as: \textit{forall vars type}.

Both the \texttt{mrev} function and \texttt{app} relations defined earlier are polymorphic - they are programs which involve lists of arbitrary type. The type of the \texttt{mrev} function can therefore be denoted by the quantified type expression:

\[
\[t\]-(t[]=>t[])
\]

where the type expression \(t[]\) denotes a list of elements of type \(t\), and \([t]−…\) introduces a \textit{type quantifier variable} \(t\). Since the type expression is closed, we could use a different variable in place of \(t\). This type expression tells us that it is a function over lists of type \(t\), \textit{for any} \(t\).

The type of \texttt{app} is:

\[
[t]-(t[],t[],t[]){}
\]

The type of \texttt{sort} is

\[
[t]-(((t[]),(t,t){})=>t[])
\]

i.e., it is a polymorphic function – it can sort lists of any type. \texttt{sort} takes a list and a relation as arguments and returns a list of the same type.\(^6\)

**Type inference and safety**  
The goal

\[
\texttt{app}(X,Y,[’bill’,’tom’,...])
\]

is a call to the \texttt{app} relation to find some splitting of the list of symbols

\[
[’bill’,’tom’,...]
\]

into contiguous sublists. Every element of a list has to have the same type. The \textit{type} of \(X\) and \(Y\) in the goal is inferred by the Go! system as \texttt{symbol[]} because the type of each element of the given list is \texttt{symbol}; variables have types even if they are unbound.

\(^6\)Note that while \texttt{sort} is polymorphic, its argument relation is not. This must be a relation over the values on its argument list. If \texttt{sort} required a polymorphic relation in its second argument, its type would be:

\[
[t]---((t[]),[s]−(s,s){})=>t[])
\]
2 KEY FEATURES OF GO!

Any values  Heterogeneous types of values can be included in a list of type any[]. The list:
[??1,??’a’,??[]]
is such a list. The ?? prefix operator maps any data value into the Go! standard type any.

any is an existential type [?]. The type of a value V in a ??V term is not visible outside the term, but its type is remembered as a hidden second argument of the term.

When unifying two ?? terms, the hidden types are also unified – at runtime if necessary. Thus the unification of two ?? terms succeeds only if both the hidden types and the encapsulated values unify.

As an example, suppose A is an variable of type any holding some ?? term. The unification:
A= ??(W:number)
where W is an unbound variable, will succeed if and only if A encapsulates a number. Thereafter, W, with type number, can be used in any context where a number is required and we have effectively extracted the number value from A. Existential types are very important in enabling Go! to be applied to many real world situations. Fundamentally, ‘encapsulating’ values in type safe ways is critical when a type safe program has to interact with the rather untyped (or at least unsafely typed) data present on public communications networks.

Programmer defined types  The pair of type definitions:
dance::= polka | jive | waltz | tango | quickstep | samba.
Desire::= toDance(dance,number) | barWhen(dance).
introduce two new types. An enumerated type dance, which has 6 literal values:
polka, jive, waltz, tango, quickstep, samba
and a Desire type that has two constructor functions toDance and barWhen.

These types will be used in our ballroom application. The Desire type are the terms that will be stored in the dynamic relation representing the agents desires. Type inference will check that only terms such as:
toDance(polka, 3), barWhen(samba)
are used to represent an agent’s desires.
2.8 Dynamic relations

In Prolog we can use assert and retract to change the definition of a dynamic relation whilst a program is executing. The most frequent use of this feature is to modify a definition comprising a sequence of unconditional clauses. Go! has a system module that can be imported and used to create, update and call dynamic relations defined using unconditional clauses. In addition, lists of higher-order values, such a relations of the same type, can be stored in cells. This allows us to store, modify and call dynamic relations defined by sequences of rules.

The dynamic relations module is imported by using:

`include "sys:go/dynamic.gof"`

This exports a dynamic relations class definition. A dynamic relation is an object with methods: `add`, for adding a fact term to the end of the current sequence of facts in the relation object, `del` for removing the first fact term in a dynamic relation object that unifies with some pattern, `delall` for removing all unifying fact terms, `delc` and `delallc` variants that we shall explain below, `mem`, for accessing the current fact terms in some dynamic relation component, and finally `ext` for retrieving the current extension as a list of fact terms.

Creating a new dynamic relation  A *dynamic relation* object can be created and initialised using:

```
desire = $dynamic([toDance(jive,2), toDance(waltz,1),
                    ..., barWhen(polka)])
```

dynamic takes a list of terms that are the initial extension of the relations. This list could be empty. The above initialisation is equivalent to giving the following the sequence of clauses for a Prolog dynamic relation:

```
desire(toDance(jive,2)).
desire(toDance(waltz,1)).
...
desire(barWhen(polka)).
```

Querying a dynamic relation  If we want to query such a dynamic relation we use the `mem` method as in:

```
desire.mem(todance(D,N)), N>2
```
Modifying a dynamic relation  To modify a dynamic relation we can use the add, and del action methods. For example:

```go
desire.add((barWhen(quickstep))
```

and:

```go
desire.del(toDance(jive,N)); desire.add(toDance(jive,N-1))
```

The second is analogous to the following sequence of Prolog calls:

```prolog
retract(desire(toDance(jive,N)),NewN is N+1,
assert(toDance(jive,NewN)
```

One difference is that we cannot backtrack on a del call to delete further matching facts. This is because it is an action, and all Go! actions are deterministic. A del call always succeeds, even if there is no matching term. The delall method deletes all unifying facts as a single action:

```go
desire.delall(barWhen(_))
```

will delete all current barWhen desires.

2.9 Classes and objects

Classes in Go! are labeled theta environments, which we can view as labeled theories as in L&O[15]. The labels can contain variables, which are global to all the definitions of the theory. By default, every definition inside the labeled theory is exported from the theory. To prevent a definition being exported, for example a relation definition, we prefix its name with a double underscore, as in \_privateR. The labeled theories may include dynamic relations and cells.

We can create an instance of a labeled theory by giving values to the global variables of the theory label. The instance is an object characterised by these global variable values - they define its static state. Different object instances of the theory will generally have different values for these global variables. If the theory contains dynamic relations or cells, the object will also have its own copies of these. They record its dynamic state. Finally, inheritance can be used to define a new labeled theory. This is done using inheritance rules using the theory labels.

The following set of definitions include constitute a mini-theory of a person:
2  KEY FEATURES OF GO!

person(Nm,Age,Sx,Hm)
    __Age=$cell(Age).
    age() => __Age.get().
    birthday() => __Age.set(__Age.get()+1).
    sex=Sx.
    name=Nm.
    lives(Hm)

The persons age is recorded in a cell valued variable \( __\text{Age} \) that is private to the theory. It is update by the \texttt{birthday} action method of the theory. \texttt{person(Nm,Age,Sx,Hm)} is the theory label and \( \text{Nm, Age, Sx, Hm} \) are the theory parameters charactering each instance when it is created.

We can create two instances of the theory and query them as follows:

\begin{verbatim}
P1=$person('bill',23,'male','london').
P2=$person('jane',20,'female','cardiff').
\end{verbatim}

\begin{verbatim}
P1.name -- returns name 'bill' of P1
P2.age() -- returns age 20 of P2
P1.birthday() -- adds one to age of P1
P1.age() -- returns 24 as age of P1
P2.lives(Place) -- gives solution: Place='cardiff'
\end{verbatim}

The following is labeled theory definition of a student that inherits from the \texttt{person} theory.

\begin{verbatim}
student(N,Age,Sx,Hm,_,_)<=person(Nm,Age,Sx,Hm).
student(_,_,_,_,Cge,Sbj)
    lives(P1):-location_of(Cge,P1).
    studies_at(Sbj,Cge).
\end{verbatim}

The separate \( \leq \) rule says that this theory inherits from the \texttt{person} theory with non-overriding inheritance. Non-overriding inheritance mans that any relation defined in the super class is extended by clauses give in the sub-class rather than being redefined by those clauses. (Functions and action procedures are always over-ridden if redefined.) In this case, there is only one relation, \texttt{lives}, which is so extended. If we had used \( \ll \) rather than \( \leq \) in the rule, the new definition of \texttt{lives} would override the inherited one. Note that this theory makes use of a global relation \texttt{location_of} defined outside the theory. It has a normal definition such as:
3 Multi-threaded applications and data sharing

It is often the case, in a multi-threaded Go! application, that we want the different threads to be able to share information. For example, in a multi-threaded agent we often want all the threads to be able to access the beliefs of the agent, and we want to allow some or all these threads to be able to update these beliefs.

We can represent the relations for which we will have changing information as dynamic relations. These can be defined at the top level of the theta environment that is the agent program and accessed by the different agent threads as global variables.

Each of the agent’s threads can then access and update these shared dynamic relations. An individual add or del action is atomic, as is access to a single dynamic relation. That is, if we have a call:

\[ DRel \cdot \text{mem}(Trm) \]

all the solutions of this dynamic call, even if there is backtracking, correspond to the value the relation had at the time the call was initially tried. So, the backtracking evaluation will see no changes to the relation that might be made by this, or any other thread.

However, a conjunction of calls to several dynamic relations is not necessarily evaluated using all the dynamic relation extensions that prevail when the evaluation of the conjunction starts. For example, suppose we have a conjunctive query:

\[ DRel_1 \cdot \text{mem}(Trm1), DRel_2 \cdot \text{mem}(Trm2) \]
Between the call to $DRel1$ and the call to $DRel2$ another thread could update $DRel2$. This lack of atomicity also applies to a sequence of updates, even of one dynamic relation.

### 3.1 Synchronising access using a lock

Each dynamic relation actually has an associated lock value that can be accessed using the function method `lock` applied to the dynamic relation. A lock is a special Go! data value that can be used to synchronise action sub-sequences in threads in the same Go! invocation and to ensure atomic execution of such sub-sequences when access to a shared updatable resource is involved.

A sync action:

\[
\text{sync}(Lk\{A1;A1;\ldots\})
\]

in some thread $Th$ will not be executed if any other thread is currently executing a sync action sequence using the same lock $Lk$. The execution of thread $Th$ will suspend in this case. Its 'request' to execute the action sequence:

\[
\{A1;A1;\ldots\}
\]

using the lock $Lk$ will be queued on the lock. (Other threads may also be waiting to sync using $Lk$.) When its 'request' reaches the head of the queue, and the previous sync action using $Lk$ terminates, execution of $Th$ will automatically resume and the above sync action will be executed.

We can use the lock value associated with a dynamic relation to ensure atomicity of a complex query or a sequence of updates on that relation. In each thread that accesses the relation we wrap the query or update sequence inside a sync action. Then:

\[
\text{sync}(\text{desire.lock})\{\text{desire.del(toDance(jive,N))};
\]

\[
\text{desire.add(toDance(jive,N+1))}\}
\]

The action procedure:

\[
\text{replace}(\text{Rel,Trm1,Trm2}) \rightarrow \text{sync}(\text{Rel.lock})
\]

\[
\{\text{Rel.del(Trm1)};\text{Rel.add(Trm2)}\}\]
will do such an an atomic update for any relation object $\text{rel}$.
Synchronising, always using the lock of one specific dynamic relation $\text{R}$, will also guarantee atomicity of update and query of a set dynamic relations. But one must always synchronise using $\text{R.lock}$, whether or not the relation $\text{R}$ is involved in the query or update.

### 3.2 Example shared relation - an agent belief store

As an example of the use of a shared dynamic relation let us consider a dynamic relation that stores the set of beliefs of a multi-threaded agent. These beliefs could be terms such as:

\[
\text{CanDo}(\text{agent}, \text{task}) \mid \text{CommittedTo}(\text{agent}, \text{task}) \mid \text{Trusted}(\text{agent})
\]

representing an agent's current beliefs about the capabilities of certain agents (including itself), its current commitments to other agents, and which other agents it currently trusts. \text{task} is the type of the terms representing tasks and \text{agent} is the type of the terms representing agents, which could just be the type handle.

In the agent program we would need type a definition such as:

\[
\text{Belief}::= \text{CanDo}(\text{agent}, \text{task}) \mid \text{CommittedTo}(\text{agent}, \text{task}) \mid \text{Trusted}(\text{agent}) \mid \ldots
\]

and a top level dynamic relation initialisation such as:

\[
\text{belief}=\text{dynamic}([\text{CanDo}(\ldots,\ldots),\text{Trusted}(\ldots,\ldots),\ldots])
\]

The \text{Belief} type definition defines the syntax of all the beliefs that might be stored and the assignment creates a new dynamic relation which will act as the belief store, seeding it with a list of initial beliefs.

From any thread of this program, executing a procedure defined within the program, we can execute update actions and query this belief store relation. Using synchronisation on its lock, we can ensure atomicity of update and access. For example, a planning thread might call:

\[
\text{belief.mem(\text{CanDo}(\text{SomeAg}, \text{Tsk}))}
\]

to find a task \text{Tsk} that \text{SomeAg} is currently believed to be capable of. Another thread might execute:
3 MULTI-THREADED APPLICATIONS AND DATA SHARING

sync(believe.lock)\{belief.del(Trusted(Ag1));
    belief.add(CanDo(Ag2, Task))\}

to atomically remove a belief about the trustworthiness of some agent, and
to add one about the capability of another agent.

Waiting for an update Sometimes an activity being executed by a thread
must wait until a particular item is added to a shared dynamic relation. A
test variant of the sync action caters for this. It has the general form:

\[ \text{sync(lock::Test)}\{\text{Action}\}; \]

The action is executed only when the lock is available and Test succeeds. As
an example:

\[ \text{sync(believe.lock::believe.mem(CommittedTo(AgId,Task)))\{.....\};} \]

will cause the thread to suspend until the believe lock is free and there is
a belief added (by some other thread) to the effect that AgId has committed
to doing some task Task. If, when the lock first becomes free, this test fails,
the test will be retried each time the lock again becomes free\(^7\) until the test
succeeds. At that point the action will be executed and the thread will
resume execution.

One action that we might execute, when the belief has been added, is the
deletion of the belief. The procedure:

\[ \text{delw(DRel,Term) \rightarrow \}
    \text{sync(DRel.lock::DRel.mem(Term))\{DRel.del(Term)\};} \]

is a general procedure for doing this for any dynamic relation DRel. As an
example:

\[ \text{delw(believe,CommittedTo(AgId,Task))} \]

will cause the thread that executes the action call to suspend until a fact
unifying with:

\(^7\)That is, it is re-tested each time some other thread has just terminated a synchronised
access to the belief relation. This happens even when another thread does a single update
of the relation since the add and del methods of the dynamic relations module synchronise
using the lock for that relation.
CommittedTo(AgId, Task)
is added to the believe relation by another thread. It will then immediately
delete that fact. This delw procedure gives us the functionality of the inw
action of the Linda[3] tuple store coordination model, where DRel is the tuple
store.

A generic memw relation, that suspends, if need be, until some term uni-
fying with a given Term is added to a dynamic relation, can be defined as:

\[
\text{memw}(\text{DRel}, \text{Term}) :-
\text{action} \{ \text{sync} (\text{DRel.lock}::\text{DRel.mem(Term)})\}\;
\]

Here the action associated with the test lock is empty. The synchronizing
action has to be prefixed by the action operator as it is used inside a relation
definition.

This is analogous to the Linda readw. The dual relation, that will sus-
pend, if need be, until some unifying tuple has been removed is:

\[
\text{notw}(\text{DRel}, \text{Term}) :-
\text{action} \{ \text{sync} (\text{DRel.lock}::!\text{DRel.mem(Term)})\}\;
\]

4 Example programs

4.1 Two threaded reactive agents

A style of reactive agent implementation facilitated by Go! is that of a two
threaded agent. One thread deals with the environment monitoring, the other
with the action behaviour of the agent. The threads interact via a shared
belief store dynamic relation. The environment monitoring thread maps
new data about the current state of the environment into current beliefs -
adding and deleting beliefs as appropriate, and the action behaviour thread
determines what actions to take by querying the repeatedly querying the
belief store, suspending if need be until certain queries are answered.

As an example, the following procedure might be executed by the action
thread of an reactive agent with a simple mission: to achieve some fixed goal
by the repeated execution of an appropriate action. The two action rule
procedure:

\[
\text{performMission}()::\text{Goal} \rightarrow \{\}.
\text{performMission}() \rightarrow \text{chooseAndDoNextStep}; \text{performMission}().
\]
captures the essence of this goal directed activity. ({} is the empty action.) This procedure would be executed by one thread within an agent whilst another concurrently executing thread is monitoring its environment, constantly updating the agent’s beliefs about the environment; these beliefs being queried by Goal, and by doNextStep. performMission is tail recursive procedure and will be executed as an iteration by the Go! engine.

Some missions – such as survival – do not have a termination goal but rather one or more continuation actions:

\[
\begin{align*}
\text{survive}() &::= \text{detectDanger}(D) \rightarrow \text{hideFrom}(D); \text{survive}(). \\
\text{survive}() &::= \text{detectFood}(F) \rightarrow \text{eat}(F); \text{survive}(). \\
\text{survive}() &\rightarrow \text{wanderFor}(\text{safeTime}()); \text{survive}().
\end{align*}
\]

The order of the rules prioritises avoiding danger. safeTime is a function that queries the belief store to determine a ‘safe’ period to wander, given current knowledge about the environment, before re-checking for danger. Again we assume the belief store is being concurrently manipulated by an environment monitoring thread within the agent. hideFrom(D) would typically cause the survival thread to suspend until the monitoring thread deletes those beliefs that made detectDanger(D) true.

### 4.2 A directory server example

Our directory server stores descriptions as arbitrary length lists of attribute terms. These are terms of the form:

\[
\text{attr}(\text{symbol}, \text{any})
\]

where the symbol is the attribute name, such as ‘id’ or ‘desires’ and the any value is just that, any type of value for the attribute wrapped in an ?? term. A key benefit of this directory structure is that it is open-ended: we do not need to fix at design time all of the attributes that we may be interested in. On the other hand, Go!’s any type, and associated run-time time checking, help to ensure that the directory will be accessed in a type safe way. The different entries do not need to have the same attributes or the same number of attributes.

\[
\text{directory_server}.{}
\]
include "sys:go/dynamic.gof".

attribute ::= attr(symbol,any).
DSmessage ::= register(attribute[]) 
  | subscribe(attribute[],symbol[]) 
  | inform(attribute[]).

description=dynamic([]).
subscription=dynamic([]).

matching descr(PDescr,RDescr) :-
  (Attr in PDescr -> Attr in RDescr).
inform descr(RDescr,Names) =>
  {attr(Nm,V)|Nm in Names,attr(Nm,V) in RDescr}.
inform if match(PDescr,RDescr,AttrNms,S) ->
  (matching descr(PDescr,RDescr) 
   ?
    inform(inform descr(RDescr,AttrNms)) >> S 
   | {}).

directory server() ->

  ( register(RDescr) << _ ->

    description.add(RDescr);
    (subscription.mem(S,PDescr,AttrNms) ->
      inform if match(PDescr,RDescr,AttrNms,S)
    | subscribe(PDescr, AttrNms) << S ->

      subscription.add((S,PDescr, AttrNms);
      (description mem(RegDescr) ->
        inform if match(PDescr,RDescr,AttrNms,S))
    );
    directory server()
  )

This program is a separately compilable module that exports a single tail recursive and non-terminating procedure directory_server. It imports a
system module – using the include statement – which implements and exports the function dynamic for creating dynamic relations components.

The DSmessagetype defines the type of the message protocol of the directory server. This is a common use of new type. A register message is declared to have an argument that is a list of attributes, that is the meaning of the type term register(attribute[]). A subscribe message term has two arguments, a list of attribute terms and a list of symbols.

The module has two dynamic relations description and subscription. The first holds the lists of attributes contained in each received registration message, and the second holds triples - handles of subscribers, partial descriptions, and lists of symbols (attribute names). A dynamic relations component holds a relation as a sequence of facts (there are no rules), and has methods for querying and manipulating the relation. For example,

description.mem(RegDesc)

is a query to the description dynamic relation.

Using the directory server The directory server is designed to be executed in a separate thread, and clients communicate with it using message send and message receive actions.

Clients register with the directory with a registration message, such as:

register(
    [attr('name',??'sally'),attr('role',??'dancer'),
     attr('sex',??'female'),...]) >> DS

This will record a list of attributes in the directory server DS. Note that our simple program does not record the sender of the registration! A more complete directory server would, of course, support this and more functionality.

A client of the server sends a subscribe message such as:

subscribe([attr('role',??'dancer')],[‘name’]) >> DS

which contains a list of attributes or attribute patterns, and a list of attribute names.
Such a message is understood as a request for a stream of inform messages to be sent as response. In this case, the client will be informed about each current and future registration which has a ‘role’ attribute value ‘dancer’ by being sent the value of its ‘name’ attribute.

The directory server module can be imported into another program, and its exported procedure spawned as a directory server for all the other threads spawned in the importing program. It can also be executed as a separate stand-alone Go! process[5]. Then other Go! processes, on the same or other hosts, can use it to advertise their attributes and to discover one another.

This directory server is by no means complete; a more elaborate directory server would also accept de-registration messages and un-subscribe messages. If intended for a public environment it would also require a technique for authenticating clients. However, this program is sufficient for the purposes of coordinating the activities of the dancer agents in our ‘ballroom’ scenario.

5 Multi-threaded dancer agents at a ball

In our agents’ ball, we have male and female dancer agents that are attempting to dance with each other and a band that ‘plays’ music for different kinds of dances. The two kinds of dancer agents are required to discover like-minded agents and to negotiate over possible dance engagements. In addition to dancing, dancer agents may have additional goals – such as getting refreshed at the bar. This scenario is a compact use case that demonstrates many of the aspects of building intelligent agents and of coordinating their activities.

Following a BDI model[2][21], each agent has a belief, a desire and an intention relation. The belief relation contains beliefs about what other dancers there currently are and what dances they like to do. The desire relation contains the goals each dancer would like to achieve, for example, which dances it would like to dance. The intention relation holds its current intentions – these normally represent the agent’s commitments to perform some particular dance with some partner agent; however, it can also be an intention to go to the bar when a dance is announced.

The dancers use the directory server we saw above to discover one another. As each dancer agent ‘arrives’ at the dance in some random and phased order, it registers with the directory server. The dancers also subscribe in order to be informed about other dancers that are already ‘at the dance’, and those
that will arrive later.

The internal execution architecture of each dancer agent comprises three threads – corresponding to the three key activities of the agent: a directory server interface thread, a negotiations thread and an intention execution thread. The directory server interface interacts with the directory server to publish its own description and to subscribe for the descriptions of other dancer agents. The negotiations thread communicates with other dancer agents in order to agree joint intentions to dance the next dance of a particular kind. The intentions execution thread coordinates the actual dance activities and any ‘drinking’ activities.

These threads communicate using the shared dynamic relations: belief, desire and intention. Note that while all the dancers could be executed in a single invocation of the Go! engine, they will not have direct access to each others’ beliefs, desires and intentions. Furthermore, it is a simple task to distribute the program across multiple invocations and machines, making each dancer a separate Go! process.

5.1 A dancer’s directory interface thread

The role of the directory interface thread is to allow the dancer agent to publish its description – to allow other agents to discover it – and to record the results of its subscription – to allow it to learn of other agents’ existence. Other agents’ characteristics are recorded in the belief dynamic relation – which is shared by the other threads in the agent.

Each dancer registers with the directory its name, gender, negotiation thread handle, intention execution thread handle, and a list of their initial desires. These are the desires that will be used to initialise their own desires relation.

```plaintext
DSInterface..{
  ... -- include statements and type defs
  sex ::= male | female.
  Belief::=hasDesires(symbol,handle,handle,Desire[])
  | ... -- other Belief terms
  dance ::= polka | jive | waltz | ...
  Desire::= toDance(dance,number) | barWhen(dance)...

  opposite sex(male)=>female.
```
opposite_sex(female) => male.

DSinterface(belief, MyNm, MySex, MyNgtTh, MyExecTh, MyDesires, DS) ->
  register([attr('role', ??'dancer'),
             attr('sex', ??MySex),
             attr('ngtTh', ??MyNgtTh),
             attr('execTh', ??MyExecTh),
             attr('desires', ??MyDesires)]) >> DS;
  subscribe([attr('role', ??'dancer'),
             attr('sex', ??opposite(MySex))],
             ['name', 'ngtTh', 'danceTh', 'desires']) >> DS;
  acceptDSinforms(DS).

acceptDSinforms(DS) ->
  inform([attr('name', ??Nm), attr('ngtTh', ??NgtTh),
          attr('execTh', ??ExecTh),
          attr('desires', ??Dsrs)]) << DS;
  add(belief, hasDesires(Nm, NgtTh, ExecTh, Dsrs));
  acceptDSinforms(DS). }

The acceptDSinforms procedure – which is the continuation of the interface thread after the initial registration – waits for inform messages from the directory server. Each contains part of another agent’s description of itself. It records these as hasDesires terms in the belief dynamic relation. The type definition of the allowed belief terms means that the compiler can check that only ‘allowed’ beliefs are added to the belief dynamic relation.\(^8\)

### 5.2 A dancer’s intention execution thread

A dancer’s intention execution thread handles the execution of intentions when they are triggered by dance announcements. We assume a band agent which sends an announcement message to every currently registered dancer when it starts, and when it later stops playing each dance ‘number’.

The procedures for the intention execution threads of the male and female dancers are very similar with respect to how they ‘listen’ for announcements

\(^8\)Type inference will infer that belief is a dynamic relation over Belief terms.
from the band. They differ in what happens when a dance is starting and there is an intention to do that dance. We present here only the male case—as the male dancer is expected to take the initiative during the dance.\footnote{This symmetry is an aspect of the ballroom scenario; one that we would not expect for general agent systems.}

\begin{verbatim}
maleIntention..{
  ... -- include statements
  dance::= polka | jive | waltz | ...
  bandMessage::= starting(dance) | stopping(dance) | 
                ball_over.
  Belief::= hasDesires(symbol,handle,handle,Desire[]) |
            bandPlaying(dance) |
            ballOver |
            haveDanced(dance,symbol) |
            ... -- other Belief terms
  Intention ::= toDanceWith(dance,symbol,handle) |
               ... -- other intention terms
  maleIntention(belief,desire,intention) ->
    ( starting(D) << _ ->
      belief.add(bandPlaying(D));
      check_intents(D,belief,desire,intention);
      maleIntention(belief,desire,intention) |
    ) 
  | stopping(D) << _ ->
    belief.del(bandPlaying(D));
    maleIntention(belief,desire,intention) |
  | ball_over << _ -> add(belief,ballOver)
  ).

  check_intents(D,belief,desire,intention) ->
    ( intention.mem(toDanceWith(D,PtnrNm,PtnrExecTh)) ?
      intention.del(toDanceWith(D,PtnrNm,\_));
      maleDance(D,PtnrNm,PtnrExecTh,belief,desire));
  | (intention.mem(...) ? ..
    -- else if branches for other intentions
    -- triggered by starting(D) announcement)).
}
\end{verbatim}
The maleIntention action procedure is a recursive loop that listens for messages; in this case messages from the band that signal the starting of a new tune, the stopping of the current tune, and the ball_over message. This last message terminates the loop.

When it receives a starting(D) message, and there is an intention to do that dance, the maleDance procedure is executed. This is given access to the dancer’s beliefs and desires as it may need to modify them. The intended partner should similarly have called its corresponding femaleDance procedure and the interaction between the dance threads of the two dancers is the joint dancing activity. Each procedure will terminate when the stopping(D) message is received by the intention execution threads and picked up by a message receive in the maleDance and femaleDance procedures.

Notice that the maleIntention procedure reflects its environment by maintaining an appropriate belief regarding what the band is currently doing and when the ball is over.

The maleDance procedure is:

\[
\text{maleDance}(D, \text{PtnrNm}, \text{PtnrTh}, \text{belief}, \text{desire}) \rightarrow \\
\text{shallWeDance} \gg \text{PtnrTh}; \\
(\text{myPleasure} \ll \text{PtnrTh} \rightarrow \\
\quad \text{stopping}(D) \ll _- \rightarrow \text{dance till end of } D \\
\quad \text{belief}.\text{add}\text{(haveDanced}(D, \text{PtnrNm})) \\
\mid \text{stopping}(D) \ll _- \rightarrow \text{-- did not manage to dance } D \text{ with PtnrNm} \\
\quad \{\text{desire}.\text{mem}(\text{Dance}(D,N))\}; \\
\quad \text{replace}(\text{desire}, \text{Dance}(D,N), \text{Dance}(D,N+1)) \\
); \\
\text{belief}.\text{del}(\text{bandPlaying}(D)).
\]

replace is an atomic update action on a dynamic relation that was defined in section 3.

5.3 A dancer’s negotiation thread

The procedures executed by the negotiation threads of our dancers are the most complex. They represent the rational and pro-active activity of the agent for they convert desires into intentions using current beliefs. In contrast, the intentions execution and directory interface threads are essentially reactive activities.
A male dancer’s negotiation thread must decide which uncommitted desire to try to convert into an intention, and, if this is to do some dance the next time it is announced, which female dancer to invite to do the dance. This may result in negotiation over which dance they will do together, for the female who is invited may have a higher priority desire. Remember that each dancer has a partial model of the other dancer in that it has beliefs that tell it the desires the other dancer registered with the directory server on arrival. But it does not know the priorities, or which have already been fully or partially satisfied.

The overall negotiation procedure is \texttt{satisfyDesires}:

\begin{verbatim}
SatisfyDesires()::belief.mem(ballOver) -> {}. 
SatisfyDesires() -> 
{notW(belief,bandPlaying(_));
 (chooseDesire(Des,FNm),\+ belief.mem(bandPlaying(_)),
 \+ belief.mem(ballOver) *>
 negotiateOver(Des,FNm));
{callW(belief,bandPlaying(_))};
SatisfyDesires().
\end{verbatim}

The \texttt{satisfyDesires} procedure terminates when there is a belief\textsuperscript{10} that the band has finished – a belief that will be added by the intentions execution thread when it receives the message \texttt{ball\_over}. If not, the first action of \texttt{satisfyDesires} is the \texttt{notW} call. This is a query action to the \texttt{belief} relation that will suspend, if need be, until any \texttt{bandPlaying(_)} belief is removed by the intention execution thread that is ‘listening’ to the band. This relation was defined in section 3. For our dancers we only allow negotiations when the band is not playing. This is not a mandatory aspect of all scenarios – other situations may permit uninterrupted negotiations over desires.

There is then an attempt to convert into commitments to dance as many unsatisfied dance desires as possible, before the band restarts. This is done by negotiation with a named female whom the male dancer believes shares that dance desire. When that iterative action terminates, the dancer checks that the band has restarted and waits if not. It then recurses for a new round of negotiation over unsatisfied desires. The next time the band stops playing, the answers returned by \texttt{chooseDesire} will almost certainly be different.

\textsuperscript{10}All the procedures for this thread access the dynamic relations as global variables since the procedures will be defined in the module where these relations are introduced.
because the beliefs, desires and intentions of the dancer will have changed. Even if one of the answers is the same, a re-negotiation with the same female may now have a different outcome because of changes in her mental state.

\[
\text{chooseDesire(Dance(D,N),FNm) :-}
\]
\[
\text{uncmtdFeasibleDesire(Dance(D,N),FNm),}
\]
\[
\text{(desire.mem(toDance(OthrD,OthrN)),OthrD\neq D}
\]
\[
\text{\rightarrow OthrN < N).}
\]
\[
\text{chooseDesire(Dance(D,N),FNm) :-}
\]
\[
\text{uncmtdFeasibleDesire(Dance(D,N),FNm),}
\]
\[
\text{\neg belief.mem(haveDanced(D,_))).}
\]
\[
\text{chooseDesire(Dance(D,N),FNm) :-}
\]
\[
\text{uncmtdFeasibleDesire(Dance(D,N),FNm),}
\]
\[
\text{\neg belief.mem(haveDanced(D,FNm))).}
\]
\[
\text{chooseDesire(Dance(D,N),FNm) :-}
\]
\[
\text{uncmtdFeasibleDesire(Dance(D,N),FNm).}
\]

\[
\text{uncmtdFeasibleDesire(Dance(D,N),FNm) :-}
\]
\[
\text{uncmtdDesire(Dance(D,N)),}
\]
\[
\text{belief.mem(hasDesires(FNm,_,_,FDesires)),}
\]
\[
\text{toDance(D,_) in FDesires.}
\]
\[
\text{...}
\]
\[
\text{uncmtdDesire(Dance(D,N)):-}
\]
\[
\text{desire.mem(toDance(D,N)), N>0,}
\]
\[
\text{\neg intention.mem(toDanceWith(D,_,_)).}
\]
\[
\text{...}
\]

The above clauses are tried in order, which reflects priorities. All the clauses return a dance desire only if it is currently uncommitted and the male believes some female desires to do that dance. It is an uncommitted desire if it is still desired to perform the dance at least once, and there is not a current intention to do that dance. (We allow a dancer to enter into at most one joint commitment to do a particular type of dance since this is understood as a commitment to do the dance with the identified partner the next time that dance is announced.)

The first rule selects a dance if, additionally, it is desired more times than any other dance. The second selects a dance if it has not so far been danced with any partner. The third rule selects it if it has not so far been
danced with the female who will now be asked. The last, default rule, selects any desired, uncommitted feasible dance. A male with this `chooseDesire` definition only actively tries to satisfy dance desires, but it could still end up with an intention of going to the bar as a result of negotiation with a female dancer.

Below is a `negotiateOver` procedure embodying a simple negotiation strategy:

```plaintext
ngtMess::= willYouDance(dance) | okDance(dance) | sorry | barWhen(dance) | okBar(dance).

negotiateOver(Dance(D,N),FNm) ->
{belief.mem(hasDesires(FNm,FNgTh,FExecTh,_))};
ngtOverDance(D,N,FNm,FNgTh,FExecTh,[]).

ngtOverDance(D,N,FNm,FNgTh,FExecTh,PrevDs) ->
| willYouDance(D) >> FNgTh;
| ( okDance(D) << FNgTh ->
|     replace(desire,toDance(D,N),toDance(D,N-1));
|     intention.add(toDanceWith(D,FNm,FExecTh))
| | sorry << FNgTh -> {}
| | willYouDance(D2)::uncmtdDesire(Dance(D2,N2))
|     << FNgTh ->
|     intention.add(toDanceWith(D2,FNm,FExecTh));
|     replace(desire,toDance(D2,N2),toDance(D2,N2-1));
|     okDance(D2) >> FNgTh
| | willYouDance(D2) ->
|     counterP(FNm,FNgTh,FExecTh,[D,D2,..PrevDs])
| | barWhen(D2)::uncmtdDesire(barWhen(D2))
|     << FNgTh ->
|     intention.add(toBarWith(D2,FNm,FExecTh));
|     desire.barWhen(D2));
|     okBar(D2) >> FNgTh
| | barWhen(D2) << FNgTh ->
|     counterP(FNm,FNgTh,FExecTh,[D,D2,..PrevDs])
| )).

counterP(FNm,FNgTh,FExecTh,PrevDs)::
chooseDesire(Dance(D,N),FNm),\+(D in PrevDs)--
```
ngtOverDance(D,N,FNm,FNgTh,FExecTh,PrevDs).
counterP(_,FNgTh,_,_) ->
    sorry >> FNgTh.

The negotiation is with the corresponding thread FNgTh in the female dancer with name FNm.

The negotiation to fulfill a dance desire with a named female starts with the male sending a \texttt{willYouDance(D)} message to her negotiation thread. There are four possible responses: an \texttt{okDance(D)} accepting the invitation, a \texttt{sorry} message declining, or a counter proposal to do another dance, or to go to the bar when some dance is played. A counter proposal is accepted if it is currently an uncommitted desire. Otherwise, the \texttt{counterP} procedure is called to suggest an alternative dance. This calls \texttt{chooseDesire} to try find another feasible dance \( D \) for female \( FNm \), different from all previous dances already mentioned in this negotiation (the \texttt{PrevDs} argument). If this succeeds, the dance negotiation procedure is re-called with \( D \) as the new dance to propose. If not, a \texttt{sorry} message is sent and the negotiation with this female ends.

\section{5.4 The male dancer agent}

Below we give the overall structure of the main module for a male dancer. It imports the modules defining the \texttt{maleIntention} and \texttt{DSinterface} procedures and it spawns them as separate threads.

\texttt{maleDancer..{ .. -- include statements and type defs belief=dynamic([]). desire=dynamic([]). intention=dynamic([]).}
maleDancer(MyNm,MyDesires,DS) ->
    (Des on MyDesires *> desire.add(Des));
    ExecTh = spawn maleIntention(belief,desire,
        intention);
    _ = spawn DSinterface(belief,MyNm,male,
        self,ExecTh,MyDesires,DS);
    satisfyDesires().
    ... -- defs for satisfyDesires etc}
The exported procedure of this module is the one called to launch a male dancer. It takes three arguments: the symbol name of the dancer, such as ‘bill’, a list of its desires expressed as Desire terms such as toDance(jive,3) - the desire to dance three jives, and the handle of the directory server for the ball.

The procedure immediately adds each desire of its argument MyDesires to the dancer’s desire relation. It then spawns the intention execution and directory server interface threads before calling satisfyDesires. Thus, the main thread of this program becomes the negotiation thread of the dancer and so its handle is passed as the identity of the negotiation thread to the directory interface procedure\textsuperscript{11}

The negotiation thread executes concurrently with the other two threads. The directory interface thread will be adding beliefs about other agents to the shared belief relation as it receives inform messages from the directory server, and the execute intentions thread will be concurrently accessing and updating all three shared relations.

The female dancer is similar to the male dancer; we assume that the female never takes the initiative. The female negotiation thread must wait for an initial proposal from a male but thereafter it can make counter proposals. It might immediately counter propose a different dance or to go to the bar, depending on its current desires and commitments. It can handle negotiations with male proposers one at a time, or have simultaneous negotiations by spawning auxiliary negotiation threads. This requires another dynamic relation to keep track of the latest proposal of each negotiation so that they do not result in conflicting commitments.

\section{Related Work}

\subsection{Logic Programming Languages}

Qu-Prolog\cite{6}, BinProlog\cite{22}, CIAO Prolog\cite{4}, SICStus-MT Prolog\cite{7} are all multi-threaded Prolog systems. The closest to Go! is Qu-Prolog. Threads in Qu-Prolog communicate using messages or via the dynamic data base. As in Go!, threads can suspend waiting for another thread to update some dynamic

\footnote{The identity of any thread is held in a special variable self. Hence it is the value of this variable that is passed to the directory interface thread as the handle of the dancer’s negotiation thread.}
6 RELATED WORK

relation. However, Qu-Prolog has no higher order features or type checking support, and all threads in the same Qu-Prolog invocation share the same global dynamic data base. In Go!, using modules, dynamic relations can be restricted to a specified set of threads.

SICStus-MT\[7\] Prolog threads also each have a single message buffer, which they call a port, and threads can scan the buffer looking for a message of a certain form. But this buffered communication only applies to communication between threads in the same Prolog invocation.

In BinProlog\[22\], threads in the same invocation communicate through the use of Linda tuple spaces\[3\] acting as shared information managers. BinProlog also supports the migration of threads, with the state of execution remembered and moved with the thread\[12\]. The CIAO Prolog system \[4\] uses just the global dynamic Prolog database for communicating between thread’s in the same process. Through front end compilers, the system also supports functional syntax and modules.

Mercury\[24\] is a pure logic programming language with polymorphic types and modes. The modes are used to aid efficient compilation. It is not multi-threaded.

6.2 Logic and action agent languages

Vip[13], AgentSpeak(L)[20], 3APL[12], Minerva[14] and ConGolog[9] are all proposals for higher level agent programming languages with declarative and action components. We are currently investigating whether the implied architectures of some of these languages can be readily realised in Go!. Vip and 3APL have internal agent concurrency.

These languages typically have plan libraries indexed by desire or event descriptors with belief pre-conditions of applicability. Such a plan library can be encoded in Go! as a set of planFor and reactTo action rules of the form:

\[
\text{planFor}(\text{Desire})::\text{beliefCond} \rightarrow \text{Actions} \\
\text{reactTo}(\text{Event})::\text{beliefCond} \rightarrow \text{Actions}
\]

\[12\] A Go! thread executing a recursive procedure can also be migrated by sending a closure containing a 'continuation' call to this procedure in a message. The recipient then spawns the closure allowing the threads computation to continue in a new location. The original thread can even continue executing, allowing cloning.
The actions can include updates of the belief store, or the generation of new desires whose fulfillment will complete the plan. Calls to `planFor` or `reactTo` can be spawned as new threads, allowing concurrent execution of plans.

7 Conclusions

Go! is a multi-paradigm programming language – with a strong logic programming aspect – that has been designed to make it easier to build intelligent agents while still meeting strict software engineering best practice. Although many AI practitioners find the restrictions imposed by strong typing and other SE-oriented disciplines to be irksome, we find that we are not significantly hindered. In part this has been because we had the requirements for agent programming in mind in the design of Go!.

There are many other important qualities of a production environment that we have not had the space to explore – for example the I/O model, permission and resource constrained execution and the techniques for linking modules together in a safe and scalable fashion. We have also omitted any discussion of how Go! applications are distributed and of how Go! programs interoperate with standard technologies such as DAML, SOAP and so on. For a more complete description of some of these topics see [5].

The ballroom scenario is an interesting use case for multi-agent programming. Although the agents are quite simple, it encompasses key behavioural features of agents: autonomy, adaptability and responsibility. Our implementation features inter-agent communication and co-ordination via messages, multi-threaded agents, intra-agent communication and co-ordination via shared memory stores. We believe these features, which are so easily implemented in Go!, are firm foundations on which to explore the development of much more sophisticated deliberative multi-threaded agents.

8 Acknowledgments

The first named author wishes to thank Fujitsu Labs of America for a research contract that supported the collaboration between the authors on the design of Go! and the writing of this paper.
References


REFERENCES


