Appl Intell (2006) 24: 189–204
DOI 10.1007/s10489-006-8511-x

Ontology oriented programming in Go!

K. L. Clark · F. G. McCabe

Abstract In this paper we introduce the knowledge representation features of a new multi-paradigm programming language called `Go!` that cleanly integrates logic, functional, object oriented and imperative programming styles. Borrowing from *L&O* [1], `Go!` allows knowledge to be represented as a set of labeled theories incrementally constructed using multiple-inheritance. The theory label is a constructor for instances of the class. The instances are `Go!`'s objects.

A `Go!` theory structure can be used to characterize any knowledge domain. In particular, it can be used to describe classes of things, such as people, students, etc., their subclass relationships and characteristics of their key properties. That is, it can be used to represent an ontology. For each ontology class we give a type definition—we declare what properties, with what value type, instances of the class have—and we give a labeled theory that defines these properties. Subclass relationships are reflected using both type and theory inheritance rules. Following [2], we shall call this *ontology oriented programming*.

This paper describes the `Go!` language and its use for ontology oriented programming, comparing its expressiveness with Owl, particularly Owl Lite[3]. The paper assumes some familiarity with ontology specification using Owl like languages and with logic and object oriented programming.

K. L. Clark (✉)
Department of Computing, Imperial College, 180, Queen's Gate, London SW7 2BZ
e-mail: klc@doc.ic.ac.uk

F. G. McCabe
Fujitsu Labs of America, 1240 East Arques Ave, M/S 345, Sunnyvale, CA 94085
e-mail: frank.mccabe@us.fujitsu.com

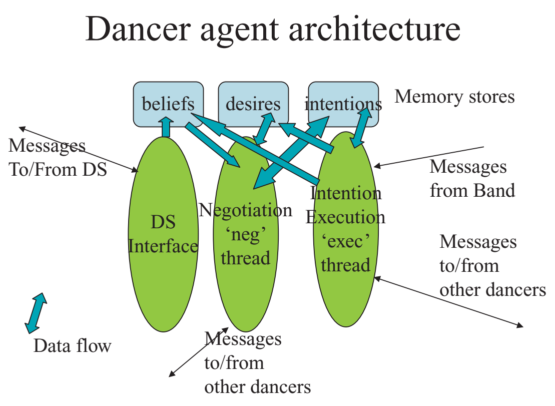
1. Introduction

`Go!` has many features in common with the *L&O* [1] object oriented extension of `Prolog`. Both languages allow the grouping of a set of relation and function definitions into a lexical unit called a labeled theory that characterise some class of 'things'. `Go!` extends *L&O* in also having action procedures defined by action rules. It also differs from *L&O* in having moded type declarations for programs with compile time type checking. Mercury [7] also has types and modes but these differ from `Go!`'s.

`Go!` is multi-threaded with asynchronous message communication between the threads using mailboxes. A mailbox is essentially a queue object shared by the communicating threads. Typically only one thread has read access to a given mailbox, while several threads can have write access.

`Go!` has been primarily designed to allow fast development of intelligent agent based applications involving multi-threaded agents. A `Go!` agent typically comprises several threads that implement different aspects of the agent's behaviour and which share a set of updatable objects, usually dynamic relations or hash tables. These are used to represent the agent's changing *beliefs*, *desires* and *intentions*. As an example, the dancer agents described in [4] have the architecture depicted in the Fig. 1 below.

In the dancer agent application beliefs are just a set of facts, but in a more complex agent application it is useful to structure the beliefs in terms of an ontology. The beliefs then record descriptions of individuals belonging to different ontology classes and must be consistent with the ontology. We can also augment the extensional held partial descriptions with inferences that are licensed by the ontology. For example, to infer that bill is a child of mary if we believe that mary is a parent of bill, where the ontology tells us that 'child of' and 'parent of' are inverse properties.



The emphasis of this paper is Go!'s class type and labeled theory notation and its use for representing ontological concepts. Its multi-threading and thread coordination and communication features are described in [4]. We introduce these ontology related features through a series of increasingly complex examples. As we do so, we shall compare the way ontological concepts can be represented in Go! with the way they can be represented in Owl Lite [3, 5]. We use the Owl abstract syntax of [3] rather than the XML syntax of [5].

Owl Lite, and its extension Owl DL, are ontology definition languages in which classes of things are characterised in terms of subclass and identity relationships with other classes, and by restrictions on unary properties for instances of the class. They are based on description logics [6]. These are logics with fast tailored inference procedures that support reasoning about the subsumption relationships between classes—inferring that all instances of one class must also be instances of another given their respective class descriptions, as well as reasoning about individuals of a class. They are more declarative than Go!, but they are not general purpose programming languages.[1] We shall mostly make comparisons with Owl Lite rather than Owl DL as the mapping between Owl Lite and Go! class notation is more direct, and, according to [6], Owl Lite has nearly all the expressive power of Owl DL.

We shall see that in Go! many of the restrictions on property values that one can express in Owl Lite become type constraints for the properties captured in a type definition. These can be checked at compile time. Others become runtime constraints that are checked when we try to create instances of a class. Ontological concepts such as transitivity of a property are implemented in Go! as explicit recursive definitions of the closure relation. This approach to representing ontologies is what Goldman [2] calls "ontology oriented programming". He shows how a hierarchy of ontology classes, and implementations of their respective interface properties, can be reflected in the class and interface type hierarchy of a C# or Java application.

In the next section we give a brief introduction to the basic elements of Go!- introducing the different forms of definition and Go!'s dynamic relations, which are objects. In Section 3 we introduce labeled classes. In Section 4 we illustrate the building of new classes as extensions of existing classes using inheritance. Section 5 gives an example of a recursive class—one that must make use of the very class concept it is defining. Section 6 covers multiple inheritance. In Section 7 we introduce the use of dynamic relations in a class to give us objects with changeable state. In Section 8 we investigate using Go! rules to define *n*-ary relations over objects allowing us to define relationships that can only be captured using a rule extension of Owl. We summarise and discuss related work in Section 9.

2. Base elements 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 rewrite rules of the form:

$$f(A_1, \ldots, A_k)::\mathtt{Test} \Rightarrow Exp$$

where the guard *Test* is omitted if not required. For each function there must also be an associated type definition of the form:

$$f:[t_1, \ldots, t_k]\texttt{=>}t$$

where t_i is the type of the i'th argument and t is the type of the value. These must all be data types. Go! is not higher order but we can program in a higher order way by passing in and returning object values.

As in most functional programming languages, the testing of whether a rule can be used to evaluate a function call uses *matching* not unification. The first function rule to match some function call, whose test also succeeds, is used to evaluate the call.

Example function definitions are:

```
father_of:[person]=>person.
father_of(C)::
   C.parent(F), F.gender()==male  => F.
```

[1] For example, a communicating agent that reasons using an ontology cannot be implemented in Owl. An Owl reasoner would have to be embedded inside an outer wrapper written in a language such as Java, Prolog or Go!. In contrast, the entire agent can be implemented in Go!.

```
number_of_children:[person]=>integer.
number_of_children(P) => len({C || P.child(C)}).
len:[list[T]]=>integer.
len([]) => 0.
len([Hd,..Tl]) => len(Tl)+1.
```

The operator `::` can be read as *such that*. An expression of the form:

$\{T \;||\; Cond\}$

is a set expression, it is `Go!`'s equivalent to the Prolog `findall`. `len` is declared to be a polymorphic function from a list of any type `T` to an integer. `,..` is `Go!`'s list data constructor to be read as *followed by*. It is the same as the Prolog `|`, which in `Go!` has other uses.

Relation definitions comprise sequences of Prolog-style `:-` (*if*) clauses of the form:

$r\ (A_1, .., A_k)$:- $Cond_1, \dots, Cond_n$

or sequences of `:--` (*iff*) committed choice clauses of the form:

$r\ (A_1, .., A_k)$: : $Test$:-- $Cond_1, \dots, Cond_n$

with an associated type definition of the form:

$r : [t_1, \dots, t_k]\{\}$

Prolog's cut (`!`) is not allowed[2] and evaluable expressions may be used as condition arguments inside the bodies of the clauses. The type expressions may be moded using annotations. We can say that an argument of type `t` is input using `t+`, that it is output using `t-`. In a relation type expression no annotation means that the argument may be input or output, allowing multiple uses. In contrast, an un-annotated argument type in a function or action procedure type expression means that the argument is input. The mode information is used by the type inference system to reason about sub-types. For an input argument a sub-type value can be given in the call, for an output argument or a function value a sub-type value can be generated.

The following is a single clause relation definition:

```
takes_only_maths_courses:[student+]{}.
takes_only_maths_courses(S) :-
  (S.takes(C) *> C.dept()='maths').
```

This defines a property that holds of any `student S` such that every `C` that `S` takes has `dept()` attribute `'maths'`[3]. The preceding mode annotated type definition tells us that this is a test relation. The type expression `student+` signals that the argument must be given when the relation is called and be an object of type `student`, or an object with a type that is a declared sub-type of `student`, say a `married_student`. `*>` is `Go!`'s *forall*. A condition:

($Cond1$ `*>` $Cond2$).

holds if for every solution to *Cond1*, there exists a solution to *Cond2*. *Cond1* and *Cond2* typically share variables.

The locus of action in `Go!` is a *thread*; each `Go!` thread executes an action procedure. These are defined using non-declarative *action* rules of the form:

$a(A_1, .., A_k)$::$Test$ -> $Action_1$; ... ;$Action_n$

with associated type definitions of the form:

a:$[t_1, \dots, t_k]$*

`*` is the annotation for an action type. We use ":" rather than "," to separate the action calls in the body of an action rule to emphasise the imperative aspect of the rule.

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 should one of its actions fail. Failure to find a rule for an action call is a run-time error.

The permissible actions of an action rule include: message dispatch and receipt, *I/O*, updating of dynamic relations, the calling of an action procedure, and the spawning of any action, or sequence of actions, to create a new action thread. The new thread executes concurrently with the spawning thread. The two threads can communicate using shared objects—typically mailboxes.

An example action procedure definition is:

```
display_info_about:[person]*.
display_info_about(P) ->
case P.age() in
(A::A>=18 -> stdout.outLine(
        P.name()<>" is an adult")
|A::A>12 -> stdout.outLine(
        P.name()<>" is a teenager")
|_ -> stdout.outLine(
         P.name()<>" is a child").
```

[2] We have found that all our uses of the cut when programming in Prolog may be achieved in `Go!` using function rules, `:--` clauses and other high level control features such as conditionals and single solution conditions.

[3] Note that `'maths'` is singly quoted. This is because, unlike Prolog, `Go!` does not have a variable name convention—most identifiers can be used as variable names, so must be quoted when used as a symbol.

This procedure is defined using one action rule. It is a procedure for displaying on the standard output channel, usually a terminal window, the values of the name and age attributes of any P that is a person or a sub-type of person. stdout is a Go! system object with various methods for sending strings to the standard output channel. <> is a polymorphic primitive for concatenating lists of any values. Go! strings are lists of single character symbols.

2.2. Go! dynamic relations

In Prolog we can use assert and retract clauses 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 unconditonal clauses. In Go!, such a dynamic relation is an object with updateable state. It is an instance of a polymorphic system class with interface type dynamic[T], T being the type of the argument of the dynamic relation. All Go! dynamic relations are unary, but the unary argument can be a tuple of terms.

The dynamic relations class has methods: add, for adding an argument term to the end of the current extension of the relation, del for removing the first argument term that unifies with a given term, delall for removing all argument terms unifying with a given term, mem, for accessing terms in the current extension using unification, and finally ext for retrieving the current extension as a list of terms.

A *dynamic relation* object can be created and assigned to a variable as in:

```
eats:dynamic[(symbol,symbol,integer)].
eats = $dynamic([('peter','apples',2),
                 ('john','icecream',1)])
```

The given list of 3-tuples is the initial extension. The preceding type declaration tells us that eats is a dynamic relation object comprising three-tuples—two symbols and an integer. We can now manipulate and query the relation using:

```
eats.del(('peter','apples',N));
   deletes tuple, binds N to 2
eats.add(('peter','apples',N+1));
   add new tuple ( ... , ... ,3)
(eats.mem(('john',F,K)),K>1 ? ... | ... );
```

The last action is a conditional action. ? can be read as *then*, | as *else*.

State information can also be recorded in special cell objects and in hash table objects. cell objects have set and get methods for updating and accessing a single stored value. hash tables are like dynamic relations except that every stored value must have a unique associated key which can be used for fast access to the value.

3. Labeled theories

The following set of definitions constitute a mini-theory of a person:

```
Gender::= male | female.
person <~{dayOfBirth:[]=>day.
           age:[]=>integer.
          gender:[]=>Gender. name:[]=>string.
          home:[]=>string. lives:[string]{}}.
person:[string,day,Gender,string]$=person.
person(Nm,Born,Sx,Hm)..{
  dayOfBirth()=>Born.
  age() => yearsBetween(now(),Born).
  gender()=>Sx.
  name()=>Nm.
  home()=>Hm.
  lives(Pl) :- Pl=home().

  yearsBetween:[integer,day]=>integer.
  yearsBetween(...) => ..
  }.
newPerson:[string,day,Gender,string]=>person.
newPerson(Nm,Born,Sx,Hm)
  =>$person(Nm,Born,Sx,Hm).
```

The ::= rule defines a new algebraic type—a data type with only data constructors. The <~ rule defines an interface type—it tells us what properties are characteristic of a person and also gives us type constraints on these properties. It tells us that age is a functional property with an integer value, that lives is a unary relation over strings, and that dayOfBirth is a functional property with a value that is an object of type day.[4]

The $= type rule tells us that there is also a theory label, with the functor person, for a theory that defines the characteristic properties of the person type—implements the person interface—in terms of four given parameters of types string, day, Gender and string. This overloading of the type name person is allowed, but not required. We could equally have used personC, or any other name, as the label functor.[5]

The theory labeled person(Nm,Born,Sx,Hm) is an implementation of the person interface type. The label parameters Nm, Born, Sx, Hm, are global variables of the theory.

[4] This is an object type that we do not define. It will have interface properties year, month etc that are used by the yearsBetween utility function.

[5] Go! allows us to give several different labeled theories implementing the same interface type, all with different labels. For purposes of this paper we shall only need one labeled theory per interface type so we shall always re-use the type name as the label functor.

Their values, given when an instance is created, transform the template theory into a mini-theory of a specific person. The characteristic properties `dayOfBirth`, `gender`, `name`, `home`, `age`, and `lives` are defined in terms of these parameters. The compiler will check that the given definitions conform to the type signatures of the `person` type. `yearsBetween` is a function used to implement the changing `age` property. It is not an externally visible property of a `person`. `now` is a system function for returning the Unix time.

The `newPerson` function is not strictly necessary as a $*label* expression, as used in the function definition, can be used to generate an instance of any labeled theory. However, using explicitly defined functions to construct objects has certain advantages. For one thing it allows us to hide or add default values for some of the label parameters. We could, for example, also define `newMalePerson` and `newFemalePerson` that do not need to be given the `Gender` argument.

Creating class instances. We can create two instances of the `person` class, i.e. two `person` objects, and query them as follows:

```
P1=newPerson("Bill",$day(1982,3,15),
   male,"London,England")
P2=newPerson("Jane",$day(1980,11,23),female,
   "Cardiff,Wales")
P1.name() returns name "Bill" of P1
P2.age() returns current age, say 25, of P2
P2.lives(Place)  gives solution:
   Place="Cardiff, Wales"
```

The expression:

```
(P1.name(),P1.dayOfBirth().year(),P1.home())
```

will evaluate to the tuple:

```
("Bill",1982,"London,England")
```

Ontological reading. In ontological terms, the `person` interface type defines a person as a 'thing' that has:

- a functional property `dayOfBirth` with a value that belongs to the `day` class/type
- a functional integer valued property `age`
- a functional string valued property `name`
- a functional string valued property `home`
- a functional property `gender` with a value from the data type `Gender`
- a multi-valued property `lives` with values that are strings

In addition, its associated labeled theory tells us that:

- the property `age` is dependent upon the value of their `yearOfBirth`
- that one value for the `lives` property is the value for their `home` property

3.1. Class definition in Owl

Using Owl Lite concrete abstract syntax [3], the above 'ontological' reading can in part be captured by the Owl class axiom:

```
Class(person partial
      restriction(dayOfBirth Cardinality(1)
                      allValuesFrom(day))
      restriction(age Cardinality(1)
                      allValuesFrom(integer)
      restriction(name Cardinality(1)
                      allValuesFrom(string))
      restriction(home Cardinality(1)
                      allValuesFrom(string))
      restriction(lives Cardinality(1)
                      allValuesFrom(string))
      restriction(gender maxCardinality(1)
                      allValuesFrom(Gender))
Datatype(Gender).
```

Alternatively, if we are prepared to 'globalise' the cardinality and range constraints of the property names so that they apply to every use of these property names, in every class of the ontology, we can use a much simplified class axiom and several property axioms:

```
Class(person partial)
ObjectProperty(dayOfBirth range(day)
                          Functional)
DatatypeProperty(age range(number)
                     Functional)
DatatypeProperty(name range(string)
                      Functional)
DatatypeProperty(home super(lives)
                 range(string) Functional)
DatatypeProperty(lives range(string))
DatatypeProperty(gender range(Gender)
                        Functional)
Datatype(Gender).
```

In Owl a distinction is made between data valued properties, properties that have scalar values such as strings and numbers, and object properties which have instances of some ontology class as values. Note that in Owl Lite we can only say that values for the functional property `gender` are from a data type called `Gender`. We cannot further constrain this data set. In Owl DL we can; we can explicitly enumerate the two allowed values for the gender property:

```
DatatypeProperty(gender
    range(oneOf(malefemale)) Functional)
```

For none of the property axioms have we given a domain restriction. This allows them to be used as properties of any Owl Lite class. The equivalent of this type of globalisation of property names in Go! is a self imposed constraint that whenever we use the same name, such as age, in a class interface type definition, we always give it the same type. However, Go! does not allow us to declare that age will *always* be functional with an integer value. As in most OO programming languages, the same property/method name can be used with a quite different associated type in different class interface types. This is an *intended* feature of the language. The only constraint on re-use in Go! is in a sub-class definition. Any re-definition of age in a sub-class of the person class must define it to have the same type.

Notice that in the first Owl formulation we do not capture the restriction that one value for the lives property is the value of the home property. We cannot express this sub-property relationship using the class specific property restrictions of Owl Lite or Owl DL. By using separate property axioms, we can capture it by saying that home has lives as a super-property. In other words, that every value of the home property of an object is a value of the lives property of that object. Capturing this restriction comes at the cost of 'globalising' these two properties. As far as we understand, Owl does not allow us to express the restriction that age is functional dependent upon dayOfBirth, we can only express the restriction that age is functional.

In Owl we can tighten the restriction on the age attribute and say that its range is the data type nonNegativeInteger. Since nonNegativeInteger is not a base type of Go!, to capture this restriction we must add a constraint to the class label parameter Born. The theory label becomes:

```
person(Nm,(Born::
        yearsBetween(now(),Born) >=0),Sx,Hm)
```

The test will be applied to the given Born value when an instance of the person class is created—when we instantiate the theory to describe a particular person. Note that the test uses the yearsBetween function defined inside the class which is in scope for the label.

3.2. Owl complete class axioms

The class axiom for person has modality partial. In Owl this means that when an individual is known to be a member of the class we can infer that it belongs to any super-classes mentioned in the axiom, and that its properties satisfy the extra restrictions given in the class axiom.

The other Owl class axiom modality is complete. This tells us that the membership of the super classes, and satisfaction of the restrictions given for the properties, may also be considered as defining restrictions—that any 'thing' satisfying all the restrictions of the class axiom can be inferred to be an instance of the class.

An example would be:

```
Class(marriedPerson complete person
       restriction(spouse Cardinality(1))
       allValuesFrom(marriedPerson))
```

This says that a married person is a person with exactly one married person spouse. It also says that any person with a married person spouse is, *ipso facto*, a married person. It gives defining characteristics for a married person. So, even if some object is not known to be a marriedPerson, it can be inferred to be one if they are known to be a person, perhaps because they belong to a subclass of person, and they have a spouse that is a marriedPerson.

To define the marriedPerson type in Go! we can use two <~ rules:

```
marriedPerson <~ person.
marriedPerson <~ {spouse:[]=> marriedPerson}.
```

The first rule says that marriedPerson includes all the properties, with the same type signature, as the person type—that marriedPerson is a *sub-type* of person. The second says that, in addition, marriedPerson includes a spouse functional property returning a marriedPerson value. The first rule allows us to use a marriedPerson object where ever a person object is required as a given value. The second marriedPerson rule tells the compiler about the additional spouse property of a marriedPerson object.

The complete class concept of Owl does not have a direct mapping into Go!. In Go! programming terms it means that any other type that has all the properties of the marriedPerson interface must be such that the Go! compiler treats it as a sub-type of marriedPerson. Suppose we want to characterize in Go! some new class otherPerson which happens to include all the properties of the marriedPerson type as well as some additional properties. We could give a single interface type definition for otherPerson that explicitly enumerates all its properties and associated types, but the Go! compiler would treat this as a completely separate type not related to the marriedPerson type. To ensure that the compiler will treat objects of type otherPerson as objects of type marriedPerson, we must explicitly declare that otherPerson is a sub-type marriedPerson, and in a separate type rule enumerate its extra properties and their types. That is, we define the interface for otherPerson indirectly by referring to the marriedPerson type. So, the complete class concept of Owl is captured in Go! as an ontological programming pattern—always define a new interface type that includes all the properties of a type that is completely characterized by its interface, by explicitly declaring that the new type is a sub-type of this 'complete' type.

As an example, suppose we want to characterize the marriedStudent class in Go!. Instead of using one type definition rule:

```
marriedStudent <~
    {dayOfBirth:[]=>day.
     age:[]=>integer. ...
     lives:[string]{}.
     spouse:[]=> marriedPerson.
     college:[]=>string. ... }.
```

or even the two rules:

```
  marriedStudent <~ person.
  marriedStudent <~ {spouse:[]=>marriedPerson.
                     college:string. ... }.
```

that tell us marriedStudent is a sub-type of the person type, we define the marriedStudent interface type using:

```
  marriedStudent <~ person.
  marriedStudent <~ marriedPerson.
  marriedStudent <~ {college:[]=>string. ... }.
```

or, more concisely as:

```
  marriedStudent <~ marriedPerson.
  marriedStudent <~ {college:string, ... }.
```

The two rule definition is equivalent to the three rule definition since:

```
  marriedStudent <~ person.
```

can be inferred from:

```
  marriedStudent <~ marriedPerson.
  marriedPerson <~ person.
```

by transitivity of <~. This enables a marriedStudent object to be used wherever a marriedPerson or a person object is required. The Go! compiler does this class membership inference using the type inheritance rules.

3.3. Describing class instances in Owl

In Owl, class instances, called individuals, are created and given properties as follows:

```
Individual(person1 person
      value(name "Bill")
      value(dayOfBirth Individual(day
                       value(year 1982)
                       value(month 3)
                       value(day 15)))
     value(gender male)
     value(age 23)
     value(home "London,England"))
```

The value terms give the property values for the instance. Giving an individual an indentifier, such as person1, is the analogue of assigning an object to a variable, such as P1 in Go!. Note that the individual that is the day is not given an identifier, it is an anonymous individual. Also note that age has to be given a value. Owl does not allow us to define the function that computes the value of age using dayOfBirth, just as it does not allow us to state the functional dependency between age and dayOfBirth. In this respect Owl is weaker than the frame concept for knowledge representation [8]. We do not need to give a value for the lives property if the Owl axiomatization with separate property axioms is used. An Owl inference engine will infer the value "London,England" for the lives property from the axiom:

```
  DatatypeProperty(home super(lives)
    range(string) Functional)
```

by making use of the super(lives) declaration. This is the equivalent of Go!'s use of the rule:

```
  lives(Pl) :- Pl=home().
```

given in the person labeled theory to infer that the home location is a place where a person lives.

3.4. Querying on Owl ontology

There is no specific Owl query language but Owl-QL [9] is a recent proposal for a language that could be used to query an Owl ontology held inside some ontology server. A Owl-QL query essentially comprises an answer template, which is usually a list of variables appearing inside the query pattern, and a query pattern, which is a list of query conditions. Variables are prefixed with ?. A query condition is a term of the form:

```
(propertyId propertyValue propertyValue)
```

or the form:

```
(type propertyValue classId)
```

An example query, in pseudo Owl-QL is:

```
Answer Pattern: {(?N ?Y ?H)}
Query Pattern: {(name person1 ?N)
                 (dayOfBirth person1 ?D)
               (year ?D ?Y)(home person1 ?H)}
```

This queries the description of the individual named person1 to find some of its property values. It is the equivalent of the Go! expression:

```
(P1.name(),P1.dayOfBirth().year(),P1.home())
```

given earlier.

More generally, in Owl-QL, one can use type conditions to find the property values of all the individuals of some class.

```
Answer Pattern: {(?N ?Y ?H)}
Query Pattern: {(type ?P person)(name ?P ?N)
(dayOfBirth ?P ?D)(year ?D ?Y)(home ?P ?H)}
```

can be used to find the name, year of birth and home location of all instances of the person class described in the ontology.

3.5. Class search queries in Go!

In Go!, to be able to find property values of all instances of a class, or to find all the instances that have particular property values, we need to be able to iterate over all the created objects of the class. One way to do this is to store each one, when it is created, in a dynamic relation:

```
Person:dynamic[person]=$dynamic([]).
isaPerson(P) :- Person.mem(P).
```

Person:dynamic[person] declares the type of the global variable Person as a dynamic relation object holding person objects. We must now add each person object to the dynamic relation as it is created. We can do this by adding the action:

```
${Person. add(this)}
```

to the person class. Any $ prefixed action, or action sequence, inside a class is executed each time an object of the class is created. this denotes the created object.

The equivalent of the second Owl-QL query is now the succinct Go! set expression:

```
{(P.name(),P.dayOfBirth().year(),P.home())
  || isaPerson(P)}
```

4. Theory and type inheritance

We may define a new class as a modification/extension of an existing class using inheritance.

Below we give an interface type definition and a labeled theory characterizing the student class. The first type rule says that student is a sub-type of person. The <= theory inheritance rule says that when an instance student(Nm,Born,Sx,Hm,_,_) of the student labeled theory is created all the definitions for the instance person(Nm,Born,Sx,Hm) of the person labeled theory, not over-ridden in the student theory, are implicitly added to the student theory instance. In addition, any $ action of the person theory is to be executed before and in addition to any $ action of the student theory.

There is a $ action inside the student theory that adds each new student to the extension of an associated Student dynamic relation. We also define an auxiliary class college—the class of values for the enrolled property of a student.

```
  student <~ person.
  student <~ {enrolled:[]=>college. studies:[string]{}}.
  student:[string,day,Gender,string,college,
         list[string]]$= student.
  student(Nm,Born,Sx,Hm,_,_)<=
          person(Nm,Born,Sx,Hm).
  student(_,_,_,_,Cge,Sbjs)..{
  lives(Pl) :- Pl=Cge.location().
  lives(Pl) :- person.lives(Pl).
  enrolled()=>Cge.
  studies(Sbj):-Sbj in Sbjs.
  ${Student.add(this)}.
   }.
Student:dynamic[student]=$dynamic([]).
isaStudent:[student]{}.
isaStudent(S) :- Student.mem(S).
newStudent:[string,day,Gender,string,college,
  list[string]]=> student.
newStudent(Nm,Born,Sx,Hm,Cge,Sbs) =>
  $student(Nm,Born,Sx,Hm,Cge,Sbs).

college <~ {name:[]=>string,location:[]=>string}.
```

```
college:[string,string]$=college.

college(Nm,Lct)..{
  name()=>Nm.
  location()=>Lct
  ${College.add(this)}.
}.
... -- defs for isaCollege and newCollege
```

In the student theory the relation lives is redefined. However, the second clause for this relation explicitly invokes the over-ridden definition in the person class. This means that the student lives relation extends the person lives relation.

We can create a specific student description and query it as follows:

```
  S1=newStudent("june",$day(1984,4,3),female,
        "Bath,England",
        $college("Imperial","London,England"),
        ["computing","mathematics"])

  S1.lives(Place)    has two answers:
                     Place="Bath,England",
                     Place="London,England"

  S1.studies(Sub)    has two answers:
                     Sub="computing",
                     Sub="mathematics"

  {(S.name(),S.enrolled().name(),
    S.age(),{Sb||S.studies(Sb)})||
      isaStudent(S)}
is list of 4-tuples giving name, enrolled
college name, age and the list of study sub-
jects of all current students
```

Finding a person that is a student. Every student can also be treated as a person because we have declared that student is a sub-type of person. In addition, because the student theory inherits from the person theory, each time we create a new student we will not only execute the $ action of the student theory, to add it to the Student dynamic relation, we shall also first execute the $ action of the person theory, which adds it to the Person dynamic relation. When we are searching for a person using isaPerson we will thus have automatic access to the set of student objects—viewed as person objects.

4.1. Inheritance in Owl

In Owl the student class could be axiomatised as:

```
  Class(student complete person
           restriction(studies allValues
              From(string))
           restriction(enrolled
              Cardinality(1)
              allValuesFrom(college)))

  Class(college partial
           restriction(location
              Cardinality(1))
              allValuesFrom(string)))
```

Note that the above does not capture the information expressed in the Go! class that the location of a student's enrolled college is a value for their lives property. To capture this restriction we would have to lift the name and location properties of a college to make them direct properties of a student, perhaps naming them collegeName and collegeLocation. In a separate property axiom we can then say that collegeLocation is a subproperty of lives. This is a bit convoluted and loses the separate concept of a college as a property value for a student.

```
  Class(student complete person
        restriction(studies
              allValues From(string))
      restriction(collegeName Cardinality(1)
                      allValuesFrom(string)))
DatatypeProperty(collegeLocation super(lives)
                    range(string) Functional)
```

When querying an Owl ontology the condition (type ?P person) will include all individuals declared to be instances of the class student because the student class axiom says that this is a sub-class of the person class.

5. Recursive classes, symmetric properties

A married person is a person who has a spouse, that spouse being a *married person*. This is a recursive class since we cannot properly characterise a married person without making use of the concept being defined.

spouse is also a symmetric property. Symmetry is a meta-property of a property that can be declared in an Owl axiom. The declaration enables an Owl reasoner to infer that "peter" is married to "mary", when all that is explicitly recorded is that "mary" is married to "peter".

The following Go! marriedPerson definition implicitly uses symmetry of the spouse property in the second rule for the spouse() function definition, as described below. Note the recursive characterisation of the type marriedPerson in the second type rule.

```
marriedPerson <~ person.
 marriedPerson <~
   {spouseName:[]=>string.
    spouse:[]=>marriedPerson)}.
 marriedPerson(string,day,Gender,
               string,string)$=marriedPerson.

 marriedPerson(Nm,Born,Sx,Hm,_)<=person
   (Nm,Born,Sx,Hm).
 marriedPerson(Nm,_,Sx,Hm,SpNm)..{
   spouseName()=>SpNm.

   defaultSpouseFor:[marriedPerson]=>
      marriedPerson.
   defaultSpouseFor(MP)=>
          $person(SpNm,$day(0,0,0),
                   oppGender(Sx),Hm)..{
          spouseName()=>Nm.
          age()=>0.
           spouse()=>MP}.

   spouse()::SpNm!="",isaMarriedPerson(MP),
      MP.name()==SpNm =>MP.
   spouse()::SpNm=="",isaMarriedPerson(MP),
      MP.spouseName()==Nm =>MP.
   spouse()=>defaultSpouse(this).

    ${MarriedPerson.add(this)}
  }.
 oppGender:[Gender]=>Gender.
  oppGender(male) => female.
  oppGender(female) => male.

  ... -- defs for isaMarriedPerson
         and newMarriedPerson
```

The spouse function is defined by three rules. The first is used when the name of the spouse was known when the instance of the class was created—`SpNm` is not the empty string `""`—and a `marriedPerson` `MP` with the name `SpNm` has been created and hence can be accessed using the `isaMarriedPerson` relation. `MP` is returned as the spouse. The second rule is used if `SpNm` was unknown when `this` `marriedPerson` was created—so its `SpNm` is the empty string—but again the spouse has been created. In this case we can find the spouse by using the `isaMarriedPerson` relation to look for one that has `Nm`, the name of the married person whose spouse we want to find, as `spouseName`. This second rule makes use of the symmetry of the `spouse` relationship. The last rule is used only when the tests of the first two rules fail. It returns a default spouse object. Note that as soon as information about the spouse becomes available, and the appropriate instance of the `marriedPerson` class is created to record this information, the third rule will no longer be used.

The `defaultSpouse` function returns an instance of a `Go!` anonymous class. This instance is a modification and extension of the instance:

```
$person(SpNm,$day(0,0,0),oppGender(Sx),Hm)
```

of the `person` class that implements the `marriedPerson` interface type. It has its own definition of the age function, that returns a default age of 0, and definitions for the `spouseName` and `spouse` functions as required for the `marriedPerson` type. Otherwise, the `defaultSpouse` for a married person `MP` is given opposite gender, name `SpNm` as recorded in `MP`, the name `Nm` of `MP` as its spouse name, and `MP`, as its spouse. It is given the same home location `Hm` as `MP`. The day of birth has a default value `$day(0,0,0)`. Note that `oppGender` is defined outside the class so is a global utility function that can be used in other labeled theories.

An example use is:

```
H=newMarriedPerson("peter",$day(1976,5,16),
                   male,‘‘Bath,UK","");
H.spouseName()           has value ""
H.spouse().name()        has value ""
H.spouse().age()         has value 0
H.spouse().spouseName()  has value "peter"
H.spouse().home()        has value "Bath,UK"
W=newMarriedPerson("mary", $day(1978,2,24),
                   female,"Bath,UK","peter");
W.spousename()         has value "peter"
W.spouse().name()       has value "peter"
H.spouse().name()     now has value "mary"
H.spouse().age()      now has value, say 27
```

Because of the inference capability programmed into `spouse()`, we should use `spouse().name()` when querying a `marriedPerson` object to find the name of its spouse, and not `spouseName`, which might have been unknown when the object was created.

In Owl the `marriedPerson` class and the symmetry of the `spouse` property would be axiomatised as:

```
Class(marriedPerson complete person
        restriction(spouse marriedPerson))
DatavaluedProperty(spouseName
        domain(marriedPerson)
        range(string) Functional)
ObjectProperty(spouse Functional
        Symmetric)
```

An Owl reasoner will also use the Symmetry property to infer values for the spouse property which are not explicitly recorded. However, when no details of the spouse are known, it will not return a default description.

6. Multiple inheritance

In both Go! and Owl a class may inherit from more than one superclass. To illustrate multiple inheritance, we define marriedStudent as a class that inherits from both student and marriedPerson.

```
marriedStudent <~ student.
marriedStudent <~ marriedPerson.
marriedStudent:[string,day,Gender,string,
                string,college,list[string]]
               $=marriedStudent.

marriedStudent(Nm,Born,Sx,Hm,Cge,Sbjs,_)<=
        student(Nm,Born,Sx,Hm,Cge,Sbjs).
marriedStudent(Nm,Born,Sx,Hm,_,_,SpNm)<=
        marriedPerson(Nm,Born,Sx,Hm,SpNm).
marriedStudent(_,_,_,_,_,_,_)..{
  lives(Pl) :- student.lives(Pl).
  $ {MarriedStudent.add(this)}
  }.
...
```

The marriedStudent type is characterised by the two <~ rules. Its labeled theory is defined using two inheritance rules and a small auxiliary labeled theory. The type rules say that marriedStudent is a sub-type of both the student and marriedPerson types. The theory inheritance rules say that it inherits all the definitions from both the student class and the marriedPerson class unless these are overridden in the marriedStudent class. Where there is duplication in the super classes, there is an arbitrary selection of which definition is inherited. In this case all duplicated definitions are the same except that for lives which is different in the two inherited classes. This is the only definition to be overridden. The overriding definition selects the definition of the student super class as the one to be used for a marriedStudent.

We can also use a overriding definition to union definitions from super classes. Suppose the marriedPerson class had itself extended the lives relation, say by a definition:

```
lives(Pl) :- Pl=home().
lives(SpH) :-
SpH=spouse().home(),SpH! =home().
```

This has a married person also living in the home location of their spouse, if it is different from their own home location. We might then use the definition:

```
lives(Pl) :- student.lives(Pl).
lives(Pl) :- marriedPerson.lives(Pl),
             \+student.lives(Pl).
```

in the marriedStudent class. \+ is Go!'s negation-as-failure operator[10]. The second rule picks up as extra lives locations all those that can be inferred using the marriedPerson definition that cannot be inferred using the student definition.

6.1. Multiple inheritance in Owl

The Owl Lite axiom for marriedStudent is:

```
Class(marriedStudent complete
         student marriedPerson)
```

This says that marriedStudent is defined to be the intersection of the student and marriedPerson classes. Any restrictions on properties expressed in the class axioms for student and marriedPerson will also apply to properties of marriedStudent.

However, to express the concept that the home of the spouse is a possible extra value the lives property of a marriedPerson we must make spouseHome a property. We then use the property axiom:

```
DatavaluedProperty(spouseHome
         super(lives) Functional)
```

Now, any recorded value for the spouseHome property of a married person will automatically be returned as a value for the lives property of that married person. Unfortunately we cannot in Owl define spouseHome as the home of the spouse, since the language does not allow us to define properties using rules. We will be forced to explicitly add values for this property to descriptions of individual married persons whenever its value is different from their home location. There is no way of stating in Owl that the value of the spouseHome property must be the same as the home of the spouse.

7. Objects with changeable state

We can use dynamic relations, cells and re-assignable variables inside a Go! labeled theory. Instances of such a class are objects that have changeable state. Below we define the familyPerson subclass of the person class. This has two dynamic relations that can be accessed and augmented for storing other familyPerson objects which are the children and parents of the familyPerson. It also has a cell that can be used to store a spouse, if and when the person marries, and a re-assignable integer valued variable NumP.

```
  familyPerson <~ person.
  familyPerson <~
      {addSpouse:[familyPerson]*.
      spouse:[familyPerson]{}.
      addChild:[familyPerson]*.
      child:[familyPerson]{}.
      addParent:[familyPerson]*.
      parent:[familyPerson]{}.
      ancestor:[familyPerson]{}.
      descendant:[familyPerson]{}}.
  familyPerson:[string,day,Gender,string]
            $=familyPerson.
  familyPerson(Nm,Born,Sx,Hm)<=
            person(Nm,Born,Sx,Hm).
  familyPerson(_,_,_,_)..{
     Spouse:cell[familyPerson]=$cell(_).
    Child:dynamic[familyPerson]=$dynamic([]).
    Parent:dynamic[familyPerson]=$dynamic([]).
    NumP:integer:=0.

    addSpouse(Sp)::var(Spouse.get()) ->
        Spouse.set(Sp);Sp.addSpouse(this).
    addSpouse(Sp) -> {}.
    spouse(Sp) :- Sp=Spouse.get(),nonvar(Sp).

    addChild(C)::\+child(C) ->
        Child.add(C);C.addParent(this).
    addChild(C)->{}.
    child(C) :- Child.mem(C).

    addParent(P)::NumP<2,\+parent(P) ->
          Parent.add(P);NumP:=NumP+1;
          P.addChild(this).
    addParent(P) -> {}.
    parent(P):- Parent.mem(P).

    ancestor(P)   :- parent(P).
    ancestor(A)   :- parent(P), P.ancestor(A).
    descendant(C) :- child(C).
    descendant(D) :- child(C),C.descendant(D).

    ${FamilyPerson.add(this)}
     }.
... -- defs for isaFamilyPerson etc
```

Two new dynamic relations Child and Parent and a cell Spouse are created for each new instance of the class, and are private to the object. These can only be indirectly accessed using the methods addSpouse, spouse, addChild, child and addParent, parent of the familyPerson interface. Each instance also has its own private copy of the variable NumP that keeps a count of the number of recorded parents. The Spouse cell is initialised to an anonymous (hence unbound) variable. spouse is a relation rather than a function because not every familyPerson will have a spouse. When there is no spouse a call to spouse will fail.

Note the recursive definitions of the transitive ancestor and descendant relations. They allow us to walk over a family tree from any familyPerson on the tree.

We create an instance of the class without giving a spouse, parents or children. We add them to the object after creation using the addSpouse, addChild and addParent action methods. A call P.addSpouse(Sp) will, if no spouse is yet recorded (the Spouse cell contains an unbound variable), store Sp in the cell and then call Sp.addSpouse(P) to automatically add P as the recorded spouse of Sp, if this is not yet recorded[6]. Similarly, a call P.addChild(C) will automatically update the recorded parents of child C to include P, if need be, and a C.addParent(P) call will update the recorded children of P to include C, if need be. These automatically updates implement forward chaining inference using ontological knowledge that the spouse relation is symmetric and that parent and child are inverses. In addition, the addParent method will ignore an attempt to add an extra parent if two are already recorded, implementing an ontology restriction that a person has at most two parents.

We can construct a small family tree as follows:

```
  J=newFamilyPerson("john",$day( ... ),
                    male," ... ");
  M=newFamilyPerson("mary",$ day( ... ),
                   female," ... ");
    creates J and M named "john", "mary"
  J.addSpouse(M);
    records M as spouse of J and vice versa
  S=newFamilyPerson("sally",$day( ... ),
                    female," ... ");
  J.addChild(S); M.addChild(S);
    creates S named "sally", records S as a
    child of both J and M and records J,M as
     parents of S
  S.addChild(newFamilyPerson("paul",
              $day( ... ),male," ... "));
    adds a new family person named "paul" as
    child of S and adds S as a parent of the
    child

  (J.descendant(D) *>
          stdout.outLine(D.name ()));
```

[6] Sp.addSpouse(P) may now result in a recall of P.addSpouse(Sp) but this time the second action rule for addSpouse will be used, ending the interaction.

```
    will display names "sally", "paul" of
    descendants of J

{(FP.name(), {A.name()||FP.ancestor(A)},
              {D.name()||FP.descendant(D)})
    || isaFamilyPerson(FP) }
    is a list of triples containing for each
    family person their name, names of
    ancestors, names of descendants
```

7.1. Incremental data and inverse and transitive properties in Owl

In an Owl ontology we can add information about a property value of an individual at any time. Indeed we can add an extra axiom for a class at any time. So an Owl ontology is inherently dynamic. Instead of explicitly recursive definitions, in Owl we simply declare that descendant is a super property of child and that it is transitive. We can also declare that parent is the inverse of child, and that descendant is the inverse of ancestor.

```
  Class(familyPerson partial person)
  ObjectProperty(child super(descendant)
                 domain(familyPerson)
                 range(familyPerson)
                 inverseOf(parent))
  ObjectProperty(descendant
                 domain(familyPerson)
                 range(familyPerson)
                 Transitive
                 inverseOf(ancestor))
```

An Owl reasoner makes use of these declarations to infer information about the child property from given information about the parent property, and vice versa, and to infer values for the descendant and ancestor properties from inferred or given values for these child and parent base properties. By not requiring an explicit recursive definition of each transitive relation, and in doing automatic inferences about properties that are declared as symmetric or have inverses, an Owl reasoner is more high level than Go!.

The restriction that a family person has at most two parents cannot be expressed as an Owl Lite restriction on the parent property in the familyPerson class axiom. We can only require a maximum cardinality of 0 or 1 in Owl Lite. In Owl DL we could specify a maximum cardinality of 2.

Other restrictions that we might want to enforce, such as the restriction that the age of a child must be less than that of each of its parents, can be implemented in Go! as extra tests in the addChild and addParent procedures. This restriction cannot be expressed in Owl Lite or Owl DL, as numerical inequality constraints cannot be expressed.

8. General relations involving objects

So far we have only illustrated the use of binary relations as (unary) properties of objects of a class. Both Owl Lite and Owl DL can only use unary properties in an ontology.

Sometimes it is useful to be able to use and to characterise more general relations between classes. As an example, consider the ternary relationship between three family persons C, P, A, that holds when A is an aunt of C because A is a female sibling of a parent P of C.

Let us assume that we have added another property sibling to the familyPerson type with the definition:

```
sibling:[familyPerson]{}.
sibling(S) :- parent(P)!,P.child(S),S!=this.
```

! is a postfix operator[7] used to indicate that only one solution of a condition is required. This defines a sibling, sufficient for our purposes, as any *other* child of one of the parents—it does not matter which. (So, for us a sibling has to share both parents.)

This property cannot be fully characterised in Owl. We can add a sibling property to the familyPerson class, but we cannot restrict each value given to this property to be a different child of a parent. This is similar to the inability to restrict the lives property of a student to include the location of their college, that we mentioned in Section 4.1.

We can now define the ternary relation connecting a child C to an aunt A via a parent P of the child. This can be defined as a global relation outside the familyPerson class:

```
auntSinceSiblingOf:[familyPerson+,
            familyPerson,familyPerson]{}.
auntSinceSiblingOf(C,P,A) :-
  C.parent(P),P.sibling(A),A.gender()=female.
```

In a call to this relation, the argument C must be given. The second and third arguments may be given, or may be unbound variables to be bound by the call. To generate all instances of the relation we can use a query conjunction:

```
isaFamilyPerson(C),auntSinceSiblingOf(C,P,A)
```

where C, P, A are all unbound variables.

[7] Same symbol but very different semantics from the Prolog cut.

The relation can also be defined as a binary relation inside the class providing we add:

```
auntSinceSiblingOf:[familyPerson,
                    familyPerson]{}.
```

to the interface type definition for `familyPerson`. The internal class definition is then:

```
auntSinceSiblingOf(P,A) :-
     parent(P),P.sibling(A),A.gender()=female.
```

The above query becomes:

```
isaFamilyPerson(C),C.auntSinceSiblingOf(P,A)
```

In `Go!` class properties do not need to be binary relations between an instance of the class and the instance of another class or data value. They can be $(n+1)$-ary relations relating a class instance to n other class instances or data values. To represent the above relationship as a class property in Owl we would have to add to the ontology an artificial class of `familyPersonPairs`, with two functional properties `first` and `second` to access the components of each pair. `auntSinceSiblingOf` can then be included as a property of a family person with values from the class of `familyPersonPairs`. Values of the property for a given individual would have to be explicitly given. There is no way of specifying the restriction that the first component of each pair must be a parent of the individual and the second must be a female sibling of that parent. The most we could do is to restrict the ranges of the selector properties so that `first` has values from the `hasChild` class defined by the axiom:

```
Class(hasChild complete familyPerson
       restriction child
           someValuesFrom(familyPerson))
```

and `second` has values from the `hasSibing` class:

```
Class(hasSibling complete familyPerson
       restriction sibling
           someValuesFrom(familyPerson))
```

These restrictions ensure that each value for `auntSinceSiblingOf` comprises a parent and a sibling, but do not ensure they are related to the individual in question.

Here are some other definitions of non-binary relations involving objects. We define them as class independent relations but each could be defined, with one less argument, inside the class corresponding to their first argument.

```
bothAged:[person+,person+,integer]{}.
bothAged(FP1,FP2,A) :- A=FP1.age(),A=FP2.age().

bothLiveIn:[person+,person+,string]{}.
bothLiveIn(P1,P2,T) :- P1.lives(T),P2.lives(T).

bothStudySubjectAt:[student+,student+,
                    symbol,college]{}.
bothStudySubjectAt(S1,S2,Sbj,Cge) :-
     S1.studies(Sbj),S2.studies(Sbj),
     Cge=S1.enrolled(),Cge=S2.enrolled().
```

9. Summary and related work

We hope we have convinced the reader that `Go!` is a rich language for building executable ontologies—for ontology oriented programming. By drawing comparison with the features of Owl Lite, we have demonstrated that much of what is considered necessary in an ontology description language can be expressed directly or indirectly in `Go!`.

However, Owl Lite ontology specifications are more high level than the class definitions in `Go!`. In Owl Lite characteristics of properties, such as symmetry or transitivity, are declaratively specified. Owl DL allows one to say that classes are disjoint and that the union of a set of classes is equivalent to some other class. For example, one can say that animals, plants and inanimates are disjoint and together cover all things. A full Owl inference engine, provided by the translation of Owl into a description logic [3, 11], can reason using the class definitions themselves, and the statements about relationships between classes. It can determine that one class is a subset of another, using the descriptions of the classes provided in the class axioms.

Inference about the subsumption relationships between classes is not possible with the direct representation of ontology classes as `Go!` classes that we have illustrated in this paper. A `Go!` class is a not a data value. It is code. An alternative, meta-level representation, is investigated in [12]. There, each Owl Lite class axiom becomes a `Go!` fact describing the named class. Each property axiom similarly becomes a `Go!` fact about the named property. Individuals are represented as instances of a single generic object class. This generic class has meta-methods that give access to the names of the ontology classes to which the individual has been declared or inferred to belong, the names of all its defined properties, and a generic method for accessing the current values of a given named property. New ontology class memberships are inferred using range or domain constraints and complete class axioms. Using this representation we can also reason, if need be, about subsumption relationships between classes.

So, the direct representation of an ontology as Go! classes has weaker inference capabilities when compared with a meta level representation in Go! [12], or its representation in Owl Lite using description logic inference. In compensation, the more direct representation has many ontology constraints checked at compile time and property value access and update is direct and fast. It is the far better approach to Go! based ontology oriented programming when the extra inferences afforded by the meta-level representation are not required, or are dispensable. This is the case when one only wants to infer extra property values for instances of classes using ontological concepts, and not to reason about the concepts themselves.

Ontologies with rules. We have shown how Go!'s logic rules can be used to extend the range of ontological relationships that can be expressed. They can be used to define quite general relationships between class properties (the definition of the marriedStudent lives relation being an example) and to define *n*-ary relations. The ontology community recognizes the benefit of augmenting ontology languages based on description logics with rules [13].

The SWRL language [14] (Semantic Web Rule Language) is an extension of Owl DL to include a Horn clause rule language. Using such as extension, one can augment an Owl ontology with *n*-ary relations. Here is the abstract syntax version of a SWRL rule that defines the bothAged relation:

```
Implies(Antecedent(age(i-variable(FP1)
          d-variable(A)) age(i-variable(FP2)
          d-variable(A)))
        Consequent(bothAged(i-variable(FP1)
           i-variable(FP2)))
```

SWRL indicates that a variable ranges over individuals or data values by wrapping it by the functors i-variable, d-variable respectively.

WRL [15] (Web Rule Language), which builds upon F-Logic [16], a frame based logic programming language, is another recent proposal to complement Owl with rules. In WRL surface syntax the bothAged relation is defined by the rule:

```
bothAged(?FP1,?FP2,?A) :-
     ?FP1[age hasValue ?A] and ?FP2
        [age hasValue ?A].
```

That age is a property of an individual is indicated by the use of the keyword hasValue. Juxtaposition of an hasValue condition to a variable or name indicates access to a property value of an individual. It is similar to Go!'s use of dot as in FP1.age().

KIF[17], which is based on full first order logic, is used for ontology specification. It has no restrictions on the arity of ontology relations that can be axiomatised. The KIF definition of bothAged is:

```
(defrelation bothAged (?FP1 ?FP2 ?A) :=
      (and (age ?FP1 ?A) (age ?FP1 ?A)))
```

Flora-2 [18], is another development of F-Logic that can be used for rule based ontological knowledge representation. Like Go!, Flora-2 is a OO logic programming language with multiple inheritance. Type information, analogous to Go!'s interface type declarations, can be asserted as facts. The Flora-2 equivalent of the Go! type declaration:

```
person <~ {name:()=>string. age:[]=>integer.
          home:[]=>string. lives:[string]{}}.
```

is the assertion:

```
person[name=>string, age=>integer,
        home=>string, lives=>>string].
```

The =>> indicates that the lives attribute is multi-valued. The type assertions do not seem to be used for type checking but they can be queried (see below).

As in Go!, one can also define attributes using rules. For example, we can state that for any person the value of the home property is a value of the lives property using the rule:

```
P[lives-->H] :- person::P, P[home->H].
```

The infix :: is used to indicate class membership and is the equivalent to our use of the isPerson predicate.

Flora-2 has two other components. One is a higher order, or perhaps more accurately a meta-order component, called HLog. This enables one to query an object to find its attribute names, and whether they are single and multiple valued, or to query a class type declaration to find the methods for that class. Using this component one can reason about the relationships between classes, as in Owl. To do this in Go! one has to use the meta-level representation of classes and objects [12]. The other Flora-2 component is transaction logic rules for specifying updates. Transaction logic rules are similar to Go!'s action rules except that Flora-2 transaction rules canfail and any updates already performed by the rule are then automatically undone. A Go! action rule should not fail. It is an error if it does.

Finally, *L&O* [1], and two other object oriented extensions of Prolog, Prolog++ [19] and Logtalk [20], allow similar representation of ontological concepts using a combination of class encapsulated rules, inheritance, and meta-level inference. None of these languages is typed.

References

1. McCabe FG (1992) L&O: Logic and objects. Prentice-Hall International.
2. Goldman NM (2003) Ontology oriented programming—static typing for the inconsistent programmer. In The Semantic Web, Proceedings of ISWC 2003, Sanibel Island, Florida, Springer-Verlag, LNAI, Vol 2870.
3. Patel-Schneider PF et al (2003) Owl web ontology language—semantics and abstract syntax. W3C Candidate Recommendation, http://www.w3.org/TR/2004/REC-owl-semantics-20040210/.
4. Clark KL and McCabe FG (2004) Go! – a Multi-paradigm programming language for implementing Multi-threaded agents. Annals of Mathematics and Artificial Intelligence, 41(2-4):171–206.
5. McGuiness S et al (2000) Owl web ontology language—overview. W3C candidate recommendation, http://www.w3.org/TR/owl,
6. Horrocks I, Peter F, Patel-Schneider, Frank van Harmelen (2003) From $\mathcal{SHIQ}$ and RDF to OWL: The making of a web ontology language. J. of Web Semantics, 1(1):7–26.
7. Henderson F, Somogyi Z, Conway T (1995) Mercury: an efficient purely declarative logic programming language. In Proceedings of the Australian Computer Science Conference, pages 499–512.
8. Minsky M (1975) A framework for representing knowledge. In P. Winston, editor, Psychology of Computer Vision, pages 211–277. MIT Press.
9. Fikes R, et al (2003) OWL-QL—A language for deductive query answering on the semantic web. SL Technical Report 03-14, http://ksl.stanford.edu/KSL_Abstracts/KSL-03-14.html.
10. Clark KL (1978) Negation as failure. In H. Gallaire and J. Minker, editors, Logic and Databases, pages 293–322. Plenum press.
11. Horrocks I, Tessaris S (2002) Querying the semantic web: a formal approach. In Ian Horrocks and James Hendler, editors, Proc. of the 13th Int. Semantic Web Conf. (ISWC 2002), number 2342 in Lecture Notes in Computer Science, pages 177–191. Springer-Verlag.
12. Clark KL, McCabe FG (2006) Ontology schema for an agent belief store, to appear in International Journal of Human-Computer Studies, special issue on Ontologies.
13. Grosof B, et al (2003) Description logic programs: combining logic programs with description logics. In G. Hencsey and B. White, editors, Proc. of the WWW-2003.
14. Horrocks I, Peter F, Patel-Schneider, Bechhofer S, Tsarkov D (2005) OWL rules: A proposal and prototype implementation. J. of Web Semantics, 3(1):23–40.
15. Jos de Bruijn et al (2005) Web Rule Language (WRL), version 1.0. Rule Markup Initiative Technical Report, http://www.wsmo.org/wsml/wrl/wrl.html.
16. Kifer M, Lausen G, Wu J (1995) Logical foundations of object-oriented and frame-based languages. Journal of the ACM, 42:741–843.
17. Gruber TR (1993) Toward principles for the design of ontologies used for knowledge sharing. Technical report, Stanford University, http://ksl-web.stanford.edu/KSL_Abstracts/KSL-93-04.html.
18. Yang G, Kifer M, Zhao C (2003) Flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web. In R. King, M. Orlowska, and R. Studer, editors, Proceedings on Ontologies, Databases and Applications of Semantics'03, LNAI 2888, pages 671–688. Springer Verlag,
19. Moss C (1994) Prolog++: The power of object-oriented and logic programming. Addison-Wesly.
20. Lopes de Moura PJ (2003) Logtalk: Design of an object-oriented logic programming language. PhD Thesis, Departamento de Informatica, Universidade da Beira Interior, Portugal.