Ontology Representation and Inference in Go!

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

November 27, 2003

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. A key feature of `Go!` is that it allows knowledge to be represented as a set of theories incrementally constructed using multiple-inheritance. The theories are parameterised and can be instantiated by giving values to these parameters.

A 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, lecturers etc. That is, we can use theory inheritance to construct an ontology of classes. Instances of a theory then describe class instances, i.e. individual students and lecturers. Theory instances are `Go!`'s objects.

This paper illustrates the use of `Go!` for ontology construction and querying and compares its expressiveness with that of currently proposed ontology specification languages, using Oil as a representative.

1 Introduction

`Go!` has many features in common with McCabe's *L&O*[11] object oriented extension of `Prolog`. A key feature of both languages is the ability to group a set of relation and function definitions into a lexical unit called a labeled

*Alphabetical order of authors

theory. Go! extends *L&O* in also having imperative procedures defined by action rules.

Like Prolog, *L&O* is untyped, or rather has only run-time type checking. Go! is strongly typed with compile-time type checking – using a modified Hindley/Milner style type inference technique [7], [13] to reduce the programmer's burden. The programmer can declare new types and thereby introduce new data constructors. Indeed, all uses of functors as data constructors need to be declared by being introduced in some type definition. Go! is also higher order in the functional programming sense, in this it is similar to the other hybrid logic and functional programming languages Escher[10], Curry[6] and Oz[15]. Code, and code closures - code with variable bindings - can be treated as data.

Go! is also multi-threaded with asynchronous message communication between the threads modeled on the inter-thread communication of Erlang[1] and April[12]. It has been primarily designed to allow fast development of intelligent agent based applications involving multi-threaded agents. We will not illustrate the multi-threading nor agent building use of the language in this paper, but refer the reader to [4]. Here we concentrate on the use of labeled theories.

A theory label is just a term that can contain variables. These variables can appear throughout the definitions of the theory and are global variables of these definitions. Instances of the theory are created by given values to these label variables. An instance is a theory closure. New labeled theories can be defined as modifications and extensions of existing theories using multiple inheritance. In addition, a labeled theory implicitly introduces a new data type - the type of its instances. Theories are Go!'s class definitions and instances of theories are Go!'s objects. In this paper we shall use the terms *theory/class* and *theory closure/theory instance/object* as equivalents.

Go!'s labeled theories can be used to characterize sets of things and their properties, i.e. to represent an ontology. We illustrate their use by building a small ontology. As we do so, we shall make comparisons with the way ontologies can be specified in the current standard ontology languages in the Oil[2]/Owl[8] family. These are untyped ontology specification languages in which classes of things are characterised in terms of subclass and identity relationships with other classes, and by constraints on the attributes (also called properties or slots) that instances of the class must satisfy. We shall see that in Go! many of these attribute constraints become type constraints for variables of the class theory which can be checked at compile time. Others

become run-time constraints that are checked when we try to create instances of a theory. Yet others get mapped into relation or function definitions in the theory, or into `Go!` procedures to be used for creating theory instances.

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 theories. In section 4 we illustrate the building of new theories as extensions of one or more existing theories using inheritance. Section 5 gives an example of a recursive theory - one that must make use of the very class concept it is defining. In section 6 we introduce the use of dynamic relations in a theory to give us objects with updateable state. In section 7 we investigate the added flexibility given by introducing an overarching super class to which every object belongs, and an object identifier accessible as an attribute value. We summarise and conclude in section 8.

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`,..,`A_k`)::`*Test* `=>` *Exp*

where the guard *Test* is omitted if not required.

As in most functional programming languages, the testing of whether a 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.

Example function definitions are:

```
father_of(C)::(parent_of(F,C),male(F)) => F.

number_of_children(P) => listlen({C || parent_of(P,C)}).

age(DateOfBirth) => yearsBetween(now(),DateOfBirth).
```

The operator :: can be read as *such that*. listlen is Go!'s primitive for finding the length of a list and an expression of the form:

{*T || Cond*}

is a set expression, it is Go!'s equivalent to the Prolog findall. now is a system function for returning the Unix time from which the current date can be computed. yearsBetween is some other defined function.

Relation definitions comprise sequences of Prolog-style :- clauses with some modifications – such as permitting evaluable expressions as well as data terms, and no cut. We can also define relations using *iff* rules.

An example clause is:

```
takes_only_maths_courses(S:student) :-
  (S.takes(C:course) *> C.subject='maths').
```

This defines a property that holds of a student (object) if every course they take has subject 'maths'[1]. The first occurrences of S and C are : type annotated as instances of the student and course class respectively. Object attributes are accessed using '.', as in C.subject('maths'). The variables must have their types declared since the type inference scheme cannot infer the object type of a variable from method invocation use. This is because the same attribute/method names may be associated with different object types.

*> 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 a procedure. Procedures are defined using non-declarative *action* rules of the form:

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

We use ":" rather than "," to separate the actions to emphasise the imperative aspect o the rule.

[1]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.

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.

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.

An example procedure rule is:

```
display_name_age_of(P:person) ->
  stdout.outLine(Name is: "<>P.name<>"\nAge is: "<>P.age^0).
```

This 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 a person object[2]. `stdout` is a `Go!` system object with various methods for sending strings to the standard output channel. The operator ^ will convert any `Go!` data value to a display string, and `<>` is a standard library primitive for concatenating lists of any values, hence also strings.

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 `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 initialised using:

[2]More precisely, any object that implements the person class interface and so can be type caste to the `person` type. This includes all objects that are instances of subclasses of this class.

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

The type argument `(symbol,number,symbol)` says that the argument type of the relation is a 3-tuple comprising two `symbol`s and a `number`. The given list of 3-tuples is the initial extension.

We can now manipulate and query the relation using:

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

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

3 Labeled Theories

A labeled theory has the form:

```
LabelTerm{
  Def1.
  ...
  Defk
  }
```

Where the definitions of the theory define functions, relations, procedures an constants that are *local* to the theory. They are based on the labeled theories of L&O[11]. `LabelTerm` can contain variables, which are global to all the definitions of the theory. An instance of the theory is logically a copy the theory in which label variables are given values. It reality it is a `Go!` higher order closure value that can be held as the value of a variable, sent in a message, or passed as an argument. However, we cannot unify two theory instances. A labeled theory can be thought of as a class definition, the instantiations of the theory can be viewed as objects, with the theory definitions its methods.

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

```
date:> (number,number,number).     type macro
sex::= male | female.              new enumerated type
person(Nm:symbol,Born:date,Sx:sex,Hm:string){
   born=Born.
   age() => yearsBetween(now(),born).
   sex=Sx.
   name=Nm.
   home=Hm.
   lives(P):-P=this.home
   }.
```

The label arguments `Nm`, `Born`, `Sx`, `Hm` are parameters to the theory which, when known, make it the mini-theory of a specific person. These four values determine the values for the six attributes of the theory: the constants `born`, `sex`, `name` and `home`, the function `age` and the relation `lives`. The parameters are typed, and this type infomation determines the type of the six attributes of the theory. `yearsBetween` is a function defined outside the class. It is function mapping a pair of `date`s to a number. The class definition also implicilty introduces a new type `person` - with name the functor of the class label. We can used this for declaring that variables take a `person` as their values. parameter `Sp` of the class label is just the functor of the class label.

In ontological terms, the above class definition tells us that a person has a born attribute `born` that is single valued and its value type is a three-tuple of numbers. It tells us that `lives` is also an attribute that is multi-valued - since it is a relation - and that all its values are strings. In addition, it tells us that for any instance of the class one value for `lives` is the value of `home` attribute of that instance - the `this` key word. In this class definition only this one value for `lives` is defined, but by using a relation we leave open the possibility that subclasses can extend the definition. A subclass may characterise a special type of person that lives in multiple locations, such as a student.

Class definition in Oil Using Oil's publication syntax[2], such a class description would look like:

```
class-def person
   slot-constraint born
     value-type date
     properties functional
   slot-constraint age
     value-type number
     properties functional
   slot-constraint sex
     value-type string
     properties functional
   slot-constraint name
     value-type string
     properties functional
   slot-constraint home
     value-type string
     properties functional
     subslot-of lives
   slot-constraint lives
     value-type string
```

```
class-def date
   slot-constraint year
   value-type number
 slot-constraint month
   value-type number
 slot-constraint day
   value-type number
```

In Oil a slot is assumed to be multi-valued, i.e. represents a many-many relationship between objects and values, unless it is declared to have the property functional. Oil only has two primitive data types, number and string, all other 'types' have to be introduced as classes - hence the definition of the `date` class.

We could also have defined such a class in `Go!`:

```
date(Yr:number,Mn:number,D:number){
  year=Yr.
  month=Mn.
  day=D.
}
```

instead of defining the `date` type as a type macro.

Notice that the Oil definition expresses the constraint that the `home` attribute should be a value of the `lives` attribute by saying that is is a subslot of that slot. In other words, that the value of the `home` slot is a value of the `lives` slot. However it does not express the property that `age` is a function

of `born`, only that `age` is functional. That it is a function of `born` cannot be expressed in Oil.

Oil and Owl do allow range constraints on property/slot values. For example, in Oil one can say:

```
slot-constraint age
  value-type (min 0)
```

To reflect that constraint in `Go!` we can add a constraining test to argument `Born` in the class label that ensures that any subsequently computed age will be non-negative:

```
person(Nm:symbol,(Born:date::(yearsBetween(now(),Born)>=0)),
       Sx:sex,Hm:string)
```

The test will be applied to the class parameter when an instance is created. It will prevent the creation of an instance of the class with a `Born` date later than the date when the object is created. This ensures that any subsequently computed `age` will be positive. This test also enforces as constraint that the value of the `born` attribute cannot be later than the data at which the `person` instance was created. This is a constraint that cannot be expressed in Oil.

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

```
P1=$person('Bill',(1982,3,12)[3],male,"London,England")
P2=$person('Jane',(1980,11,23),female,"Cardiff,Wales")
P1.name              returns name 'Bill' of P1
P2.age()             returns current age,  say 23, of P2
P2.lives(Place)      gives solution: Place="Cardiff,Wales"
```

Instances in Oil In Oil, instances are created using:

```
instance-of date1 date
   related date1  year 1982
   related date1  month 3
   related date1  day 23
```

[3]If we had given a class definition for the `date` type the birth-date would be given as $date(1982,3,12), an instance of the `date` class.

```
instance-of person1 person
   related person1  name "Bill"
   related person1 born date1
   related person1 sex "male"
   related person1 age 21
   related person1 home "London,England"
```

The `related` statements give the slot values for the instance. Note that the instance has to be given an explicit identifier that uniquely identifies this object. Also note that `age` has to be given a value. Oil does not allow us define the function that computes the value of `age` using `born`, just as it does not even allow us to state the functional dependency between `age` and `born`. In this respect Oil is weaker than the frame concept for knowledge representation [14]. We do not need to give a value for the `lives` slot because a value `"London,England"` can be inferred from the constraint:

```
slot-constraint home
  subslot-of lives
```

Oil queries There is no specific Oil query language but the most recent proposal for a language that could be used to query Oil and Owl ontologies held inside some ontology server is DQL[5]. A DQL query is essentially a conjunction of Oil statements in which some of the values and instance identifiers have been replaced by variables, coupled with a specification of which variables must be given in any answer. In addition, the language allows for pragmatic issues such as number of required answers to be specified. The assumption is that the ontology server will not only access the ontology instances but use general rules implicit in the ontology to answer the queries.

An example query, in pseudo DQL is:

```
{(born bill1 ?D)(year ?D ?Y)(home bill1 ?H)}
Must Bind: (?Y ?H)
```

This can be used to find the year of birth and home location of the instance with identifier `bill1`. Its class is not specified. More generally:

```
{(instance-of ?P person)(name ?P ?N)(born ?P ?D)
  (year ?D ?Y)(home ?P ?H)}
Must Bind: (?N ?Y ?H)
```

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

To be able to handle this form of query in `Go!`, we need to be able to search over all the created `person` objects. One way to do this, is to store each one, when it is created, in a dynamic relation. If we do that, it is also best to create instances using a defined procedure that will automatically add the new object to the dynamic relation.

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

create_person(Nm,Born,Sx,Hm,P) ->
  P=$person(Nm,Born,Sx,Hm);
  Person.add(P).
```

We now create `person` objects and query them as follows:

```
create_person('Jane',(1980,11,23),female,"Cardiff,Wales",P)

P.name              returns name 'Jane' of P
```

The equivalent of the second DQL query is now the set expression:

```
{(P.name,Yr,P.home) || a_person(P),P.born=(Yr,_,_)}
```

or, the even more succint:

```
{(P.name,P.born.year,P.home) || a_person(P)}
```

if we represent dates as objects.

4 Theory Inheritance

We can define a new theory as a modification/extension of an existing theory using inheritance. An inheritance relationship is expressed as a rule:

```
SubClassThLabel <= SuperClassThLabel
```

`<=` can be read as: *sub-theory of* or *sub-class of*. The following is a labeled theory for `student` that inherits from the `person` theory. It has an associated `Student` dynamic relation for keeping track of all the created instances, and a procedure for creating instances.

```
student(Nm,Born,Sx,Hm,_,_)<=person(Nm,Born,Sx,Hm).
student(_,_,_,_,Cge:string,Sbjs:list(string)){
  lives(Pl) :- Pl=location_of(this.college).
  lives(Pl) :- person.lives(Pl).
  college=Cge.
  studies(Sbj):-Sbj in Sbjs.
}.

Student=$dynamic[student]([]).
a_student(S) :- Student.mem(S).

create_student(Nm,Born,Sx,Hm,Cge,Sbjs,S) ->
  S=$student(Nm,Born,Sx,Hm,Cge,Sbjs);
  Student.add(S).
```

The infix predicate `in` is `Go!`'s list membership primitive.

The separate `<=` rule says that the `student` theory inherits from the `person` theory. Inheritance in `Go!` is overriding, which mans that any relation, function or action procedure defined in a super class is replaced by any re-definition in the inheriting class. In this case, the relation `lives` is redefined. However, the second clause for this relation explicitly invokes the over-ridden definition in the `person` class which means that in this case the new definition actually extends the old definition. Note that the `student` theory makes use of a global function `location_of` defined outside the theory. It has a normal definition such as:

```
location_of("Imperial") => "London,England".
location_of("UCLA") => "Los Angeles,CA".
...
```

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

```
create_student('june',(1984,4,3),female,"Reading,England",
               "Imperial",["computing","mathematics"],S1)

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.college,S.age()) || a_student(S)}
                  is list giving name, college and age
                  of all current students
```

Treating a student as a person Every new `student` object is also a person object, so we need to modify the definition of `a_person` to retrieve student objects from the `Student` dynamic relation as well as `person` objects from the qPerson dynamic relation.

To do this, we can add an extra clause to the definition of `a_person`:

```
a_person(P) :-
   a_student(S),
   P=(S~>person).
```

This clause finds an object `P`, which can be viewed as a `person` object, by finding a `student` object `S` and type casting it to its supertype `person` to give the value of `P`. `~>` is `Go!`'s type caste operator. We add such a clause each time we create a top level subclass of the person class. If and when we create a top-level subclass if the student class we do the same for the `a_student` relation. This not only ensures that all the objects in this student subclass can be accessed as `student` objects, but that they can all also be accessed as `person` objects.

Inheritance in Oil In Oil the `student` class would be defines something like:

```
class-def student
  subclass of person
  slot-constraint studies
     value-type string
  slot-constraint college
      value-type College
      property functional
```

```
class-def College
 slot-constraint name
   has-value string
   property functional
 slot-constraint location
   has-value string
   property functional
```

In Oil new constraints on a slot of a subclass are added to the constraints on the same slot in all the superclasses, so we do not need to repeat that `home` is a subslot of `lives`. The Oil definition also requires us to make the value type of the `college` slot a class which has a `location` property/slot. This is because

only an instance of a class can have a property such as a `location`. Note that we have not expressed the constraint that the `location` slot of the `college` slot value is a subslot of `lives`. In Oil we cannot express such indirect subslot relationships. To capture the constraint we would have to lift the `name` and `location` attributes of `college` into the `student` class, as `college-name` and `college-location` slots. We can then say that `college-location` is a subslot of `lives`.

5 Recursive Theories

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

```
married_person(Nm,Born,Sx,Hm,_)<=person(Nm,Born,Sx,Hm).
married_person(_,_,_,_,Sp:married_person){
  spouse=Sp.
}.

Married_person=$dynamic[married_person]([]).
a_married_person(MP) :- Married_person.mem(MP).

create_married_person(Nm,Born,Sx,Hm,Sp:married_person,MP) ->
  MP=$married_person(Nm,Born,Sx,Hm,_);
  check_spouses(MP,Sp);
  Married_person.add(MP).

check_spouses(MP:married_person,Sp:married_person) ->
  (var(Sp) ? {} |
    (Sp.spouse=MP ?
      MP.spouse=Sp
    | stdout.outLine((Sp,Sp.name)^0<>
                     " already married to another"))).
```

The procedure for creating and recording instances of the class is either given an unbound variable as the `Sp` parameter, or a previously created instance of the class. An unbound variable represents an as yet undetermined spouse, and the created instance will have an unbound variable as the value

of its spouse attribute. When an instance is created with Sp given, that Sp should have an undetermined spouse. In that case, the call to check_spouses will ensure that both the newly created married person, and Sp, will record each other as their spouse. The test Sp.spouse=P will bind the unbound variable value of Sp.spouse to P, and conversely, P.spouse=Sp will bind the unbound variable value of P.spouse to Sp.

An example use is:

```
create_married_person('john',(...),male,"Bath,England",_,H);
create_married_person('mary',(...),female,"Bath,England",H,W);
W.spouse.name     has value 'john'
W.spouse.spouse.name     has value 'mary'
H.spouse.age()       has value, say 27
```

Symmetric slot constraint in Oil Notice that the married_person theory does not tell us that spouse is symmetric, that from the information that Sp is the spouse of MP we can infer that MP is the spouse of Sp. We had to encode that knowledge, and make use of it, in the object creation procedure. In Oil we can express the symmetry property directly in the class definition:

```
class-def married-person
   subclass of person
   slot-constraint spouse
       value-type married-person
       property functional
       property symmetric
```

An Oil ontology server should be able to infer that some married person Sp is married to P even if it only has the explicit statement of the property that P is married to Sp.

In Go!, if we are to exploit the efficiency of attribute value computation by method invocation, we must explicitly record that Sp is married to P inside the Sp object. That is, we must do the 'inference' implied by the symmetry of the spouse relation when we are first informed of some spouse relationship. In effect, we do forward chaining reasoning as the data is entered to ensure that we can subsequently answer a query about the spouse of Sp, when we have only been told that P has spouse Sp.

Multiple Inheritance We can inherit from more than one superclass. To illustrate multiple inheritance, we can define a `married_student` as a class that inherits from `student` and `married_person`.

```
married_student(Nm,Born,Sx,Hm,Cge,Sbjs,_)<=
                student(Nm,Born,Sx,Hm,Cge,Sbjs).
married_student(Nm,Born,Sx,Hm,_,_,Sp)<=
                married_person(Nm,Born,Sx,Sp).
married_student(_,_,_,_,_,_,_){
   lives(Pl) :- student.lives(Pl).
   }.

Married_student=dynamic[married_student]([]).
a_married_student(MS) :- Married_student.mem(MS).

create_married_student(Nm,Born,Sx,Hm,Cge,Sbjs,
                       Sp:married_person,MS)->
  MS=$married_student(Nm,Born,Sx,Hm,_);
  check_spouses(MS~>married_person,Sp);
  Married_student.add(MS).
```

The procedure for creating instances requires the spouse argument to be of type `married_person` or, implicitly, any of its subtypes, such as `married_student` which can be type-caste to `married_person`. This way we do not constrain students to marry other students, but we allow it.

`married_student` is defined by two inheritance rules and a mini-theory that simply specifies that the `lives` attribute should inherit its definition from the student class. We can query an instance of the class using any of the attributes of `person`, `student` and `married_person`. The methods and attributes from the shared `person` super-class are are inherited only once. If the same attribute is defined in more than one super class, the different definitions must have the same type and it is not determined which definition will be inherited. If it matters, the attribute must be redefined in the inheriting class so as to call the definition to be inherited, indicated by a class name prefix. This is why we have re-defined `lives`, since it has a definition in both the `student` and `married_person` super-classes. If we want to 'union' the super-class definitions in some way, we redefine the attribute in the inheriting class to use all its super-class definitions.

Suppose the `married_person` class had itself extended the `lives` relation, say by a definition:

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

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

```
lives(Pl) :- student.lives(Pl).
lives(Pl) :- married_person.lives(Pl).
```

in the `married_student` class.

In Oil, the union of the super-class constraints on a slot, when a class has more than one super-class, is assumed. However, without making `spouse_home` a slot of the `married_person` class, we cannot express the constraint that its value is a possible extra value for the `lives` slot of a `married_person`. If we add a `spouse_home` slot, we cannot then express the constraint that the value of this new slot should be identical to the value of the `home` slot of the `spouse`.

6 Dynamic theories

We can have one or more dynamic relations inside a labeled theory. Instances of such a theory are objects that have updateable state.

```
family_person(Nm,Born,Sx,Hm,_,_)<=person(Nm,Born,Sx,Hm).
family_person(_,_,_,_,
        (Prnts:list(family_person)::listlen(Prnts)=<2){
  __Child=$dynamic[family_person]([]).
  add_child(C) ->
    __Child.add(C).
  child(C) :- __Child.mem(C).
  parent(A):- A in Prnts.
  ancestor(A) :- parent(A).
  ancestor(A) :- parent(P), P.ancestor(A).
  descendant(D) :- child(D).
  descendant(D) :- child(C),C.descendant(D).
  }.
```

```
Family_person=$dynamic[family_person])([]).
a_family_person(FP) :- Family_person.mem(FP).

create_family_person(Nm,Born,Sx,Hm,Prnts) ->
  FP=$dynamic(Nm,Born,Sx,Hm,Prnts);
  Family_person.add(FP);
  (Prnt in Prnts *> Prnt.add_child(FP)).
```

The dynamic relation `__Child` is private to the theory and will not be accessible outside the theory. It is a private attribute. The privacy is indicated by the __ prefix on the name. All instances will, however, have a private `__Child` dynamic relation unique to that instance.

Note the recursive definitions of the transitive `ancestor` and `descendant` relations. They allow us to walk over a family tree starting at any family person on the tree.

We can create an instance class without any known parents by giving the `Prnts` argument as the empty list, or we can make it a list containing one parent. Note that the test on `Prnts` in the class label restricts the number of parents to a maximum of 2. We do not initially give any children. When a new instance `NFP` is created which has `FP` specified as a parent, then `NFP` is automatically added as a child of `FP`. In effect, we are inferring a value for the `child` attribute of `FP`, given the information that `FP` is a parent of `NFP`, making use of the ontological knowledge that `child` is the inverse `parent`.

We can construct a family tree as follows:

```
create_family_person('john',(...),male,"...",[],J);
create_family_person('mary',(...),female,"...",[],M);
   creates J and M with no known parents
create_family_person('sally',(...),female,"...",[J,M],S);
  creates S with parents J,M
  and adds S as a child of J and M
create_family_person('paul',(...),male,"...",[S],P));
  creates P with single parent S
  and adds S as a child of P

(J.descendant(D) *> stdout.outLine(D.name^0));
  will display names 'sally', 'paul' of descendants of J
```

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

Dynamic Oil, inverse and transitive slots In Oil we can add information about a slot value of a class instance at any time. Indeed we can extend a class definition at any time. So an Oil ontology is inherently dynamic. We cannot give a recursive definition for a transitive slot, nor is it necessary. The slot is simply declared to be transitive. We can also declare that `child` is the inverse of `parent`. Note that the link between `parent` and `ancestor` is given by saying that the former is a sub-slot of the latter. Because of the inverse properties, it follows that `child` is a sub-slot of `descendant`.

```
class-def family-person
   subclass of person
   slot-constraint parent
      value-type family-person
      max cardinality 2
      property inverse child
      subslot-of ancestor
   slot-constraint ancestor
      value-type family-person
      property transitive
      property inverse descendant
```

7 Generalised slot constraints and runtime type tests

The above examples illustrate how many of the concepts that one wants to express in an ontology of things can be captured using `Go!`'s labeled theory notation and associated type and sub-type system. In the above examples, the class membership constraints on the values for slots/attributes that one can express in Oil mapped into simple type conditions possibly augmented with a run time test. For example, the constraints for the `parent` slot of a family person mapped to the `Go!` type condition that the class argument was of type `list(family_person)`, checkable at compile time, augmented with

the runtime test that the length of the given list of parents is not greater than 2 when an instance of the class is created.

In Oil, one can define slot constraints that in `Go!` terms have no type constraint, only a runtime test. For example, one can express the constraint that the values for a slot are either from one class C1 or class C2, that they can be from any class, or that they should *not* be from some specific class.

To capture the idea that a value can be from any class - essentially unconstrained - we can introduce the concept of an `object` superclass of every class. At the same time we can introduce the concept of an identifier for any object, which is the only attribute of this class.

```
object(I){
   identifier=I.
}.
```

Now, for every top level class of our ontology, such as `person`, we add a class rule:

```
person(...,I)<=object(I).
```

Every class is now a direct or indirect subclass of `object` and every instance of a class can be type caste to `object` and will be accessible via the `an_object` relation. We can also change the dynamic relation associated with each class so that it now stores a pair, the identifier of the object and the object. So for the `object` class we have:

```
Objects=$dynamic[(symbol,object)]([]).
an_object(I,O) :- Objects.mem((I,O)).
```

The instance accessing relations such as `an_object` can now find an instance of the class given its identifier[4]. Remember that for every top-level subclass of a class we need to have a clause accessing instances of the subclass as instances of the class. Thus, we need to add:

```
an_object(I,P) :- a_person(I,P).
```

etc. to the definition of `an_object`. A call:

[4]We can even use `Go!` hash tables instead of dynamic relations. These are similar to dynamic relations except that each stored term is associated with a key - in this case the object identifier - that can be used for fast access to a stored value.

```
a_person(I,P)
```

returns a value for `P` such that `P.identifier=I`. So, if `I` is give, it retrieves the `person` object with identifier `I`.

We can also define the relation:

```
a_personID(I) :-
  a_person(I,_).
```

for use when we do not want to retrieve an object with a particular identifier, just to find or check identifiers for the class. We can use it as a runtime test that an `object` is a member of a particular class - we check that its identifier is that of an instance of the class. Use of the `ID` relations also allows us to give the same object identifier to instances of different classes, as is allowed in Oil, and to check at runtime in which classes objects with that identifier exist.

Let is now make use of these ideas to implement more generalised slot constraints. A typical one, given in [2], is for the slot `eats` of a definition of the `herbivore` subclass of `animal`.

```
class-def animal.

class-def defined herbivore
   subclass-of animal
   slot-constraint eats
      value-type (plant or
                  slot-constraint is-part-of has-value plant)
```

The slot constraint expresses the condition that everything a herbivore eats must be a plant, or an object with an `is-part-of` slot with a value that is an object that is a plant. Separately, the ontology specifies that `is-part-of` has an inverse `has-part`, and that both are transitive. So the things eaten by a herbivore may be indirectly part of some plant. Notice that `animal` has no class specific slot constraints. This does not mean that a particular animal does not have slot values. In Oil slot values can be given to an instance of a class even when the slot name is not mentioned in the class definition. Slot constraints can also be given independently of the class definitions, as slot definitions. They then apply to any uses of the slot name in a class definition, and in instance data. In `Go!` we cannot specify attribute constraints outside a class definition, nor can we create instances of a class with attributes not

mentioned in its class definition, except if we are indirectly creating an instance of some ontology class by creating an instance of one of its subclasses which has extra attributes.

In `Go!` we can define the animal and herbivore classes as:

```
animal(Eats:list(object)){
  eats(E) :- E in Eats.
}.

 all_vegetarian(Eats) :-
   (E in Eats *>
     (is_plantID(E.identifier)
     |
      is_plant(_,Pl),Pl.has_part(E.identifier)).

herbivore(Eats::all_vegetarian(Eats)) <= animal(Eats).
```

The definition of `all_vegetarian` requires that the list of `objects` given as the diet of a particular herbivore be such that each one `E` either has an identifier that is that of a plant (so E is an instance of the plant class or one of its subclasses), or is such that there is an instance of the plant class with a part (of unspecified class) which has the same identifier attribute as `E`. We are here using the ontology 'knowledge' that `is_part_` and `has_part` are inverses.

When we create an instance of the herbivore class we must type caste each given food item to `object`. By doing so we can give as food items not only instances of the plant class, or any of its subclasses, but we can also give instances of non-plant classes such as a leaf, providing that the leaf is recorded as being part of some plant.

7.1 Defined classes

The Oil `herbivore` class definition includes the keyword `defined`. Its inclusion means that not only must every carnivore be an animal that only eats plants or parts of plants, but that every such animal, even if it is not explicitly declared to be an instance of the `carnivore` class, should non-the-less be classified as a `carnivore`.

Type casting from a superclass to a subclass is not supported in Go!, even when the subclass has the same attributes as the superclass. To capture the defined class idea in Go!, we extend the definition of the class associated relation that determines the object identifiers that belong to the class. So, for carnivore, we extend a_carnivoreID with an extra rule telling us that every identifier of an animal that satisfies the carnivore test, is also the identifier of a carnivore:

```
a_carnivoreID(I) :-
  a_carnivore(I,_).
a_carnivoreID(I) :-
  an_animal(I,A),
  all_vegetarian(A.eats).
```

If we use this relation to give us the identifiers of carnivores we must be aware that some of the identifiers returned may be those of objects of type animal, as well as type carnivore. So, when we want to access the object with a carnivore identifier we must use both a_carnivore and an_animal as access relations. As an example, we must use:

```
{(I,Eats) || a_carnivoreId(I),
             (a_carnivore(I,C),Eats={E || C.eats(E)}
             |
              an_animal(I,A),Eats={E || A.eats(E)})! }
```

to find the list of identifiers paired with lists of the objects eaten of each animal that is deemed to be a carnivore. ! is Go!'s single solution operator - it effectively says that the disjunction is an exclusive or.

8 Conclusions

We hope we have convinced the reader that Go! is a rich language for building executable ontologies. The class definitions act as a schema that their instances must satisfy and as a framework for defining rules that can be used to make inferences from the set of instances. By drawing comparison with the features of Oil, we have demonstrated that much of what is considered necessary in an ontology definition language can be expressed directly or indirectly in Go!.

Oil ontology specifications are more high level than the class definitions in `Go!`. All constraints are declaratively specified. One can also state relationships between classes, for example that animals, plants and inanimates disjointly cover all objects. A full Oil inference layer, provided by the translation of Oil into the extended predicate logic of KIF[5], or into a description logic[9][3], can reason using the class definitions themselves, and the statements about relationships between classes. Suppose that in an Oil ontology one creates an instance of the herbivore class and states that it eats something with an identity `I123`. A full Oil inference layer can conclude that `I123` is a plant, or part of a plant. In our `Go!` representation we cannot even create the herbivore instance without having first created an object with identity `I123` as an instance of the plant class, or as an instance of some other class with the property that, directly or indirectly, it is part of an instance of the plant class. As another example, suppose that we have defined in Oil two subclasses of animals. One, called carnivore, with an an eats constraint that requires that everything that is eaten is an animal, the other, called animal-type-1, with an eats constraint that everything that is eaten is not either a plant or an inanimate. A full Oil inference layer will conclude, using the property that animals, plants and inanimates disjointly cover all objects, that carnivore and animal-type-1 are equivalent classes. In `Go!` they would remain separate object types unless the ontology builder realises that animal-type-1 is a superfluous class definition. That is, the `Go!` ontology programmer has to do the inference that it is equivalent to the carnivore class.

So, the `Go!` class representation of an ontology has, in comparison, weaker inference capabilities. In compensation, a `Go!` ontology is directly usable as a compiled `Go!` program. It requires no inference layer that reasons using the class definitions and class relationship statements as axioms. Moreover, in one respect at least, Oil ontologies are weaker than those that can be expressed in `Go!` since Oil slots cannot have values computed by arbitrary function or relation definitions. Computed slot values were a feature of the older frame approach[14] to knowledge representation.

We are investigating compiling Oil or Owl ontologies expressed in XML into `Go!` programs. We could then use such programs as a knowledge component of a `Go!` agent knowing that it conforms to the ontology. Alternatively, we can use the translation to build an ontology server, running as a separate `Go!` process[4], that can be shared, updated and queried by a group of agents. The server will be useable as a repository for instances of the ontology classes, with consistency checking on update and inference on querying,

using the ontology.

References

[1] J. Armstrong, R. Virding, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall International, 1993.

[2] S. Bechhofer et al. An informal description of standard oil and instance oil. White paper, http://www.ontoknowledge.org/oil/, 2000.

[3] S. Bechhofer et al. Instance store - database support for reasoning over individuals. Technical report, http://instancestore.man.ac.uk/, 2002.

[4] K. Clark and F. McCabe. Go! – a multi-paradigm programming language for implementing multi-threaded agents. *Annals of Mathematics and Artificial Intelligence*, 2004. To appear.

[5] R. Fikes, P. Hayes, and I. Horrocks. DQL - a query language for the semantic web. In *Proceedings WWW 2003*. http://www2003.org/cdrom/html/refereed/index.html, 2003.

[6] M. Hanus. A unified computation model for functional and logic programming. In *Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 80–93, 1997.

[7] R. Hindley. The principal type scheme of an object in combinatory logic. *Trans. AMS*, 146:29–60, 1969.

[8] I. Horrocks, P. F. Patel-Schneider, and F. van Harmelen. From SHIQ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 2003. To appear.

[9] I. Horrocks and S. Tessaris. Querying the semantic web: a formal approach. In I. Horrocks and J. 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, 2002.

[10] J. Lloyd. Programming in an integrated functional and logic programming language. *Journal of Functional and Logic Programming*, pages 1–49, March 1999.

[11] F. McCabe. *L&O: Logic and Objects*. Prentice-Hall International, 1992.

[12] F. McCabe and K. Clark. April - Agent PRocess Interaction Language. In N. Jennings and M. Wooldridge, editors, *Intelligent Agents, LNAI, 890*, pages 324–340. Springer-Verlag, 1995.

[13] R. Milner. A theory of type polymorphism in programming. *Computer and System Sciences*, 17(3):348–375, 1978.

[14] M. Minsky. A framework for representing knowledge. In P. Winston, editor, *Psychology of Computer Vision*, pages 211–277. MIT Press, 1975.

[15] P. Van Roy and S. Haridi. Mozart: A programming system for agent applications. In *International Workshop on Distributed and Internet Programming with Logic and Constraint Languages*. http://www.mozart-oz.org/papers/abstracts/diplcl99.html, 1999. Part of International Conference on Logic Programming (ICLP 99).