FOL Contexts – the Data Structures

Richard W. Weyhrauch
Ibuki Inc.
rww@ibuki.com

Carolyn Talcott
Stanford University
clt@sail.stanford.edu

Prolog

This note describes the FOL context data structure. It is a reference manual, rather than a primer. We make no attempt to motivate or justify the choices here.

1. A birds-eye view of FOL Contexts

We start with a set of symbols, *FOLsym*, a set of labels *Lab*, as well as numbers and the ability to form finite lists of things. These are our basic building blocks. We let s range over *FOLsym* and *lab* range over *Lab*.[1] We also assume we have a computation system (see [2]) that provides us with programs and computations.

Forms are the primeval syntactic structures. A form is either a symbol or a list of forms. We let *Form* be the set of forms so generated, and let *form* range over *Form*.

An FOL context is a data structure that contains a label, *lab*, a language, L, a simulation structure, S, and some facts, F. Informally we write,

$$C = \langle lab : L, S, F \rangle$$

to describe such an FOL context. This note describes the data structures (languages, simulation structures, facts) that are used to build contexts, and describes the basic operations for constructing and manipulating FOL contexts. The computation system provides finite data structures from which we build simulation structures. The purpose

[1] By convention whenever we introduce a metavariable to range over some sort, then we also may use subscripted or superscripted variants range over that sort. Thus lab' and lab_0 will also range over *Lab*.

of simulation structures is to tell us how we can compute the value of expressions in the language of a context.

2. Languages

An `FOL` language contains a finite set of symbols each with an associated syntactic type—*SortSym*, *RelSym*, *FunSym*, *IndSym*—which tells us how it is used to form terms and formulas. We factor the description of an `FOL` language into two parts: a *similarity type* and its corresponding *symbol declarations*. A similarity type specifies the number of symbols of each syntactic type and the number of arguments (arity) of each relation and function symbol. The symbol declarations specify the actual symbols of the language. In the case of sorts, the declaration consists of the sort symbol together with a (possibly empty) list of symbols usable as variables ranging over that sort.

2.1. Similarity Types

A *similarity type* is a data structure of the form

$$T = \mathit{simtypeMake}(j, p, q, n)$$

where

(s) $j = \mathit{simtypeSorts}(T)$ is a positive integer, the number of sort symbols;

(r) $p = \mathit{simtypeRels}(T)$ is a list of positive integers, $p = \langle p_1, \ldots, p_k \rangle$, where the length, k, of p is the number of relation symbols, and p_i is the arity of the i-th relation symbol;

(f) $q = \mathit{simtypeFuns}(T)$ is a list of positive integers, $q = \langle q_1, \ldots, q_l \rangle$, where the length, l, of q is the number of function symbols, and q_i is the arity of the i-th function symbol; and

(i) $n = \mathit{simtypeInds}(T)$ is a natural number, the number of individual constant symbols.

We let the symbol T range over the sort *Simtype* of similarity types. In places where a similarity type is expected, we write

$$T = \langle j, p, q, n \rangle$$

to describe a similarity type with components j, p, q, n as above.

2.2. First order languages

An `FOL` language is a data structure of the form

$$L = langMake(T, svDecs, syms_r, syms_f, syms_i)$$

where

(t) $T = langSimtype(L)$ is a similarity type,

(s) $svDecs = langSortVarDecs(L)$, the sort and variable symbol declarations, is a non-empty list of non-empty lists of symbols

$$svDecs = \langle\langle s_1, v_{1,1}, \ldots, v_{1,m_1}\rangle, \ldots, \langle s_j, v_{j,1}, \ldots, v_{j,m_j}\rangle\rangle,$$

where s_i is declared a sort symbol and v_{i,m_i} is declared a variable symbol ranging over s_i for $1 \leq i \leq j$.

(r) $syms_r = langRelDecs(L)$, the relation symbol declarations, is a list of symbols $syms_r = \langle r_1, \ldots, r_k\rangle$ declaring each r_i to be a relation symbol.

(f) $syms_f = langFunDecs(L)$, the function symbol declarations, is a list of symbols $syms_f = \langle f_1, \ldots, f_l\rangle$ declaring each f_i to be a relation symbol.

(i) $syms_i = langIndDecs(L)$, the individual constant symbol declarations, is a list of symbols $syms_i = \langle c_1, \ldots, c_n\rangle$ declaring each c_i to be an individual constant symbol.

and the lists of symbols are pairwise disjoint. We let L range over the sort *Lang* of `FOL` languages. We write

$$\begin{aligned} L = \langle\ & T, \\ & \langle\langle s_1, v_{1,1}, \ldots, v_{1,m_1}\rangle, \ldots, \langle s_j, v_{j,1}, \ldots, v_{j,m_j}\rangle\rangle, \\ & \langle r_1, \ldots, r_k\rangle, \\ & \langle f_1, \ldots, f_l\rangle, \\ & \langle c_1, \ldots, c_n\rangle\ \rangle \end{aligned}$$

to describe a language, L, of similarity type T with symbol declarations as above. The sort symbol s_1 is called the *mostgeneral* or *universal* sort symbol. Note that the above definition requires every context to have a universal sort. This sort plays a distinguished role as will be explained in later sections.

By the *symbols* of L we mean the set

$$\{s_1, \ldots, s_j, v_{1,1}, \ldots v_{j,m_j}, r_1, \ldots, r_k, f_1, \ldots, f_l, c_1, \ldots, c_n\}$$

Well-formed *terms* and *formulae* of L are subsets of FOL forms defined in the usual way from the individual variable and constant symbols, `termIf` (conditional term formation); the sentential constants, `True` and `False`; the sentential connectives, `and` ($\wedge$), `or` ($\vee$), `imp` ($\supset$), `iff` ($\equiv$), `not` ($\neg$), `wffIf` (conditional formula formation); and the quantifiers, `all` ($\forall$), and `exists` ($\exists$) (see Prawitz [1]). We say that e is an *expression* of (the language) L, when e is either a term or a formula of L. A form *form* is called a *formula* if it has the shape of a formula, that is if there is a language L such that *form* is a (well-formed) formula of L. We let *Formula* be the set of formulas, and let *formula* range over *Formula*.

We have chosen to define a language in terms of two lists—a similarity type and symbol declarations—rather than as a set of symbols and an associated arity map. Since a similarity type also describes the structure of a model, independent of the choice of symbols used to describe terms and formulas, similarity types types provide a link between the syntactic and semantic structure of `FOL` contexts. For the purpose of recognizing the set of well-formed terms and formulas, it is easy to map one form of language presentation to the other.

3. Computation Systems

In [2] we introduced the notion of a *computation system* as a certain kind of partial structure in order to give a clear understanding of what we mean by "program". A computation system has as its domain a set, $CsysU$, of computational entities (the computational universe). The basic sorts of a computation system include:

- Pgm the set of programs;
- Cmp the set of restartable computations;
- Env the set of environments

The reification of computations and the notion of a restartable computation (using a stepper semantics) is described in detail in [2]. Here we explain just enough to allow us to make sense of the notion of simulation structure.

A computation system has a relation *run* and the functions *apply*, and *call*.

$$run(cmp, u)$$

means that if you start the computation, *cmp* the computation completes and the result is u. We require that *run* behave like a function with respect to its first argument, i.e. there is at most one value u such that $run(cmp, u)$.

$$apply(P, env, cmps)$$

produces a restartable computation that computes the result of applying the program P to the list of arguments resulting from the list of computations $cmps$.

$$call(P, env, args)$$

is like apply, but expects as its third argument a list of argument values, not a list of computations producing these values.

In a computation system, we also have available some specific data structures:

yes and *no* representing the booleans *True* and *False*;

mt-env, the empty environment;

nc-pgm, a program that fails to complete its computation for any argument list; and

nc-cmp, a restartable computation that is never done.

Note that *nc-pgm* and *nc-cmp* are specific data structures and can be tested against for equality.

4. Simulation Structures

An `FOL` simulation structure is a data structure of the form

$$S = ssMake(T, env, reps, satts, ratts, fatts, catts)$$

where

(t) $T = ssSimtype(S)$ is a similarity type,

(r) $reps = ssReps(S)$ is a list of representation types (*FOLsyms*),

(e) $env = ssEnv(S)$ is an environment,

(sa) $satts = ssSortAtts(S)$ is a list, $\langle satt_1, \ldots, satt_j \rangle$, of sort attachments, where each sort attachment, $satt_i$, is a (possibly empty) list of pairs, each consisting of a sort representation type and a program,

- $satt_i = \langle \langle srep_{i,x}, P_{S_{i,x}} \rangle \mid 1 \leq x \leq ms_i \rangle$
- $srep_{i,x} \in reps$

(ra) $ratts = ssRelAtts(S)$ is a list, $\langle ratt_1, \ldots, ratt_k\rangle$, of relation attachments, where each attachment, $ratt_i$, is a (possibly empty) list of pairs, each consisting of a relation representation type and a program,

- $ratt_i = \langle\langle rrep_{i,x}, P_{R_{i,x}}\rangle \mid 1 \leq x \leq mr_i\rangle$
- $rrep_{i,x} \in reps^{p_i}$

(fa) $fatts = ssFunAtts(S)$ is a list, $\langle fatt_1, \ldots, fatt_l\rangle$, of function attachments, where each attachment, $fatt_i$, is a (possibly empty) list of pairs, each consisting of a function representation type and a program,

- $fatt_i = \langle\langle frep_{i,x}, P_{F_{i,x}}\rangle \mid 1 \leq x \leq mf_i\rangle$
- $frep_{i,x} \in reps^{q_i+1}$

(ia) $catts = ssIndAtts(S)$ is a list, $\langle iatt_1, \ldots, iatt_n\rangle$, of individual constant attachments, where each attachment, $iatt_i$, is either the empty list or a list with one element –an indconst representation type paired with a computation,

- $iatt_i = \langle\langle irep_{i,x}, cmp_{i,x}\rangle \mid 1 \leq x \leq mc_i\rangle$
- $irep_{i,x} \in reps$, mc_i is 0 or 1

where a given representation type (sort, relation, function or individual) appears at most once as the first element of an element of an attachment list.

We let S range over the sort Ss of `FOL` simulation structures. We write

$$\begin{aligned}
S = \langle\ & \langle T, reps, env\rangle, \\
& \langle satt_1, \ldots, satt_j\rangle, \\
& \langle ratt_1, \ldots, ratt_k\rangle, \\
& \langle fatt_1, \ldots, fatt_l\rangle, \\
& \langle iatt_1 \ldots, iatt_n\rangle\ \rangle
\end{aligned}$$

to describe a simulation structure, S, of similarity type T, with environment, representations, and attachments as above.

Even though representation types may seem to be one of the more bizarre and complex features of `FOL` contexts, we do not try to motivate their use here. As mentioned earlier, the purpose of this note is to carefully *describe* the data structures we call `FOL` contexts. The motivation for representations will be discussed elsewhere.

5. Assertions and Facts

The facts of an `FOL` context are collections of assertions that are required to satisfy certain well-formedness conditions. A *assertion* is a data structure of the form

$$assert = assertMake(lab, formula, deps, just)$$

where

(1.) $lab = assertLab(assert)$ is a label,

(2.) $formula = assertForm(assert)$ is an FOL formula,

(3.) $deps = assertDeps(assert)$ is a list of labels called dependencies,

(4.) $just = assertJust(assert)$ is a justification.

Justifications, $Just$, are data structures used to record additional information about an assertion. Among other things, a justification might say why an assertion is asserted in a context, or what it might be used for and how it might be used in the process of reasoning. We leave justifications unspecified for the present, simply requiring that they be finite data structures.

We let $assert$ range over the sort $Assert$ of assertions. We write

$$assert = \langle lab, formula, deps, just\rangle$$

to describe an assertion with components as above.

If the formula of an assertion, $assert$, is a formula of the language L, then we say $assert$ is an assertion of L, and write $Assert\{L\}(assert)$.

We let $ListOf[Assert]$ be the the set of lists of assertions, and let as range over $ListOf[Assert]$. $assertsLabs(as)$ is the set of labels of assertions occurring in as, and $assertsCons(assert, as)$ adds $assert$ to the list as. $assertsGet(as, lab)$ is the first assertion in the list as with label lab. We write $ListOf[Assert]\{L\}(as)$, if as is a list assertions over L, i.e. if each element of as is a assertion over L.

Assertions may have no dependencies—for example axioms and tautologies. If the label of an assertion, $assert$, appears in its dependency list and its label is the only dependency of that assertion, then we say that $assert$ is an *assumption* and write $Assume?(assert)$.

6. Contexts

An `FOL` context is a data structure of the form

$$C = cxtMake(lab, L, S, F)$$

where

(1.) $lab = cxtLab(C)$ is a label,

(2.) $L = cxtLang(C)$ is a language,

(3.) $S = cxtSs(C)$ is a simulation structure,

(4.) $F = cxtFacts(C)$ is an assertion list over L ($ListOf[Assert]\{L\}(F)$).

We further require that every label occurring as a dependency of some assertion in F is the label of an assumption of F. We let C range over the sort Cxt of `FOL` contexts. As stated in the beginning, informally, we write

$$C = \langle lab, form, deps, just \rangle$$

to describe a context with components as above.

Notice that, if the computation system has only finite data structures, i.e. programs, arguments, environments, computations, ..., are all finite data structures, then `FOL` contexts are finite data structures. An example of such a computation system is the HGKM computation system [`fol-little-hgkm`].

7. References

[1] D. Prawitz. *Natural Deduction: A Proof-theoretical Study.* Almquist and Wiksell, 1965.

[2] C. L. Talcott and R. W. Weyhrauch. Computation systems with restartable computations, 1996. URL = http://www-formal.stanford.edu/FOL/home.html.

[3] R. W. Weyhrauch and C. L. Talcott. FOL home page, 1995. URL = http://www-formal.stanford.edu/FOL/home.html.