A language for application specific linguistic content.

Draft for comments.

Version 1.1 Steve Pulman, Jan/Feb 2008
Changes:
Added suggestion for emotion annotation of subsentential units.

Motivation

Excerpt from Jan’s minutes of the Sheffield 7.1.08 meeting:

2. The idea of a formal interface between the internal (inferencing, reasoning, planning) part of the DM and the language analysis/generation was deemed interesting. It should be

   a. Language independent (predicates, arguments, names)
   b. Formally defined (and specified in some language, presumably XML)
   c. Contain emotional and other modality information, or at least allow for easy addition later when we are able to work with such information
   d. Open enough to add additional predicates, names etc. later - we cannot expect to have closed domain, for neither of the two Companions scenarios (the less for general dialogue systems)

Stephen Pulman agreed to circulate draft very soon. It will be probably designed around 1st order logic; closeness to/compatibility with KIF should be important. "Final" draft should be presented and discussed further in Edinburgh at the workshop.

I’m a bit late, I’m afraid ...

Syntax definition

I suggest we start with what is essentially first order logic in the KIF notation.

We use the following KIF constructs (logic.stanford.edu):

A word is a sequence of alphanumeric characters.

- should we also include ‘,’ for convenience? KIF does not directly.
should case differences matter? In KIF they don’t, unless an escaping mechanism is used.

An **individual variable** (indvar) is a word prefixed with ‘?’. (KIF also has ‘sequence’ variables: I don’t think we’ll need them).

A **logical constant** is \(=\) | \(\neq\) | not | and | or | => | <= | <=> | forall | exists

(These are KIF’s ‘sentops’, without ‘holds’. \(\Rightarrow\) is material implication, \(<=\) is reverse material implication, i.e. \(<= p q\) \(<=>\) \((\Rightarrow q p)\)).

(Non-logical) constants are: **words** - **variables** - **logical constants**.

\[
\text{term ::= } \text{indvar} | \text{constant} | \text{funterm}
\]

This is a subset of KIF terms.

\[
\text{funterm ::= (constant term*)}
\]

This is a subset of KIF funterms. Functions in KIF (and FOL) are total: KIF provides the constant `bottom` to be the value of functions that would otherwise be undefined for certain arguments: e.g. `father of(X)=bottom` for any non-living value of X.

We adopt the KIF definition of sentence:

\[
\text{sentence ::= constant | equation | inequality} \\
| \text{logsent | relsent | quantsent}
\]

Two constants that count as sentences have special semantics: `true` and `false`, which are always true and false respectively.

\[
\text{equation ::= (= term term)}
\]

\[
\text{inequality ::= (/= term term)}
\]

\[
\text{rehsent ::= (constant term*)}
\]

\[
\text{logsent ::= (not sentence)} \\
| (and sentence*) \\
| (or sentence*) \\
| (\Rightarrow sentence* sentence) \\
| (\Rightarrow sentence sentence*) \\
| (\Leftrightarrow sentence sentence)
\]

\[
\text{quantsent ::= (forall (varspec+) sentence)} \\
| (exists (varspec+) sentence)
\]
The more complex varspec is illustrated by something like:

\[(\forall (\langle ?x \text{ dog} \rangle) \text{ (barks } ?x))\]

This is equivalent to \[(\forall (?x)(\Rightarrow (\text{dog } ?x)(\text{bark } ?x))).\]

There is an equivalent ‘infix’ version of KIF syntax which is less lisp-like: however, the web page describing it contains several errors and typos. I think that something of the form of the preceding example would be written:

\[\forall (?x). \text{ dog}(?x) \Rightarrow \text{bark}(?x).\]

If we need an infix version we could easily specify one.

**Semantics**

The semantics of this subset of KIF - let’s call it CSKIF, for ‘Companions subset of KIF’ - is essentially that of first order logic. I won’t repeat that here.

I will assume that we will carry out inference using expressions of CSKIF, via a theorem prover or model builder of some kind, operating on normalised forms of CSKIF expressions. We should therefore probably augment the language with some ‘performative’ operators of the type used in various agent communication languages. I suggest that at least the following would be useful:

*\(\text{(assert sentence)}\) - add the contents of the sentence to the current set of axioms.*

*\(\text{(retract sentence)}\) - remove the contents of the sentence from the current set of axioms.*

Note that this operation will have to be defined carefully. For example, if the database contains \((\text{likes john jane})\), should that only be retracted by \((\text{retract (likes john jane)})\), or also by \((\text{retract (forall (?x ?y) (likes ?x ?y)})\)? The former is an instance of the latter. Note also that we can only efficiently retract forms that unify, subsume, or are identical: we cannot do full-blown truth maintenance or belief revision. Thus if we retract \(p\), but we have \((\Rightarrow q p)\) and \(q\) in the axioms, the fact that we can still deduce \(p\) will not be affected.

Querying will depend on the inference method being used, but it would be nice to support the following performative query operators:

*\(\text{(ask_if sentence)}\) - do the axioms entail the sentence (alternatively, is the expression \((\text{and (not sentence) axioms})\) unsatisfiable?)
(ask_one sentence) - do the axioms entail the sentence, and if so, return one set of bindings for variables in sentence. It may be useful to define ‘which’ as a synonym for ‘exists’ indicating which variables are of interest: thus

\[(\text{which } (?x) (\text{exists } (?y) (\text{brother } ?x ?y)))\]  would only return bindings for \(?x\), even though \(?y\) would get instantiated in the course of a proof.

(ask_all sentence) - the same, but return all bindings (multiple answers).

**XML encoding**

As far as I know, there is no generally agreed DTD for an XML encoding of KIF, but it should not be too difficult to come up with one. It may even be possible to squeeze it into the RDF framework.

**Example translations**

The GATE system used by Sheffield outputs meaning representations in a Description Logic. The actual output is in the form of RDF triples, which are a rather verbose XML format. In their pure Description Logic format, some examples (courtesy of Alexei Dingli) are:

For the sentence ...

My name is Denise. This is my mother Mary. She is the sister of Jack.
That is her boyfriend, Ted.

We would get

- `isSpeaker(Denise)`
- `hasMother(Denise, Mary)`
- `hasSister(Jack, Mary)`
- `hasBoyFriend(Ted, Mary)`

Presumably this is after the operation of reference resolution etc. Denise, Mary etc are individual constants, ‘isSpeaker’ is a ‘class’ expression (denoting a set) and ‘hasMother’ is a ‘role’ expression denoting a set of pairs.

These expressions of Description Logic need only a trivial syntactic rearrangement to be expressed in CSKIF:

\[(\text{isSpeaker Denise})\]
Other constructs of Description Logic also have a straightforward mapping into CSKIF. So where C and D are classes and R is a role, DL allows for (among others) these more complex classes to be built up:

<table>
<thead>
<tr>
<th>Class meaning example</th>
<th>CSKIF</th>
</tr>
</thead>
<tbody>
<tr>
<td>C ( \cap ) D ( \lambda x. C(x) \land D(x) )</td>
<td>cat ( \cap ) black(Fido)</td>
</tr>
<tr>
<td>C ( \cup ) D ( \lambda x. C(x) \lor D(x) )</td>
<td>cat ( \cup ) dog(Fido)</td>
</tr>
<tr>
<td>( \exists R.C ) ( \lambda x. \exists y. R(x,y) \land C(y) )</td>
<td>( \exists ) hasChild.Doctor(John)</td>
</tr>
<tr>
<td>( \forall R.C ) ( \lambda x. \forall y. R(x,y) \rightarrow C(y) )</td>
<td>( \forall ) hasChild.Doctor(John)</td>
</tr>
</tbody>
</table>

Oxford parser

The CCG parser that we intend to use provides various kinds of output. For purposes of compatibility with CSKIF the one to use is probably the Discourse Representation Structure, which is easily convertable into standard first order logic and KIF.

Example: Who is that at the bottom of the picture?

Using the whcih synonym for exists suggested earlier this will be represented as:

\( (\text{which} (?x0) \land (\text{thing} ?x0)(\text{person} ?x0)) \land (\exists ?x1 ?x2 ?x4) \land (\text{picture} ?x1)(\text{bottom} ?x2)(\text{at} ?x4 ?x2))) \)
Note that the CCG parser has interpreted the ‘at the bottom of the picture’ PP as a VP modifier, i.e. as in ‘at the bottom of the picture, who is that?’ rather than the perhaps more natural reading ‘who is [that (person) at the bottom of the picture]?’

**Prague Dependency Structure**

As I understand it from Silvie, the current Prague system does not output a semantic representation as such. However, from the dependency trees it is possible to see how something semantic-like could be produced: we assume that verbs introduce events or states, as in the DRS above.

“Who is that at the bottom of the picture?”

Bracketing:

\[
\text{root} -- \text{is.PRED} [\text{who.PAT that.ACT} [\text{bottom.LOC} [\text{picture.APP}]]]
\]

Something like the following could be produced fairly easily directly from the bracketed structure, but the semantics is a little obscure:

\[
\text{(exists} \ (?e) \\
\quad \text{(and} \ (\text{PRED} ?e \text{ is}) \ (\text{PAT} ?e \text{ who}) \\
\quad \quad \quad \text{(ACT} ?e \text{ that}) \ (\text{LOC} \text{ that bottom})(\text{APP} \text{ bottom picture}))
\]

One would have to write semantic rules to produce something more like the CCG output above, but I am not sure how easy that would be. A representation with more transparent semantics might be as follows, on the assumption that ‘that’ has been resolved to refer to ‘who’: (can the same entity occur as ACT and PAT of the same verb?)

\[
\text{(which} \ (?x) \\
\quad \text{(and} \ (\text{person} ?x) \\
\quad \quad \text{(exists} \ (?e \ ?y \ ?z) \\
\quad \quad \quad \text{(and} \ (\text{be} \ ?e)(\text{PRED} \ ?e) \\
\quad \quad \quad \quad \text{(PAT} \ ?x \ ?e) \ (\text{ACT} \ ?x \ ?e) \\
\quad \quad \quad \quad \text{(bottom} \ ?y)(\text{picture} \ ?z) \\
\quad \quad \quad \quad \text{(LOC} \ ?x \ ?y)(\text{APP} \ ?y \ ?z))))
\]

In either case, what is not displayed in the dependency diagrams is the internal structure of nodes, in which number, tense, focus etc are represented.
Other semantic representations

I don’t have an example from the H&F analyser, but I remember their output being some kind of attribute-value notation. This translates easily, if not very revealingly:

att1=val1, att2=val2, ....
(and (= att1 val1)(= att2 val2)....)

Emotion annotation

I have no idea. There is a proposal here:

http://emotion-research.net/projects/humaine/earl/proposal

However, one consideration that might be important is the ability to annotate individual components of a sentence with emotion: for example, I might want to express that ‘motorbike’ should be uttered in tone of surprise in:

You want to buy a motorbike?

If we have a CSKIF representation like:

(want you '(exists (?x) (and (motorbike ?x)(buy you ?x))))

it is not obvious how to achieve this. (Note that our formulation uses the KIF quote operator: the semantics of this a a little tricky). One line of exploration might be to use a ‘flat’ representation of logical expressions. In a flat representation embedded structures are represented indirectly via numbered labels: e.g.

(and (likes jane john)(likes john jane))

becomes:

{1:and(2,3),2:likes(4,5),3:likes(6,7),4:jane,5:john,6:john,7:jane}

It is easy to see how to inter-translate between the flat and the recursive representations. Flat representations have been used in various theorem proving and computational linguistics applications: shake and bake translation, hole semantics, etc. They have several useful properties: often they can be quite economical - the above for example could be simplified to

{1:and(2,3),2:likes(4,5),3:likes(5,4),4:jane,5:john}

They can also be used to represent underspecified meanings: for example, take the two scopings of ‘everyone loves someone’:
1. \( \exists x \, \forall y \, \text{likes } x \, y) \)
2. \( \forall y \, \exists x \, \text{likes } x \, y) \)

We can represent what they have in common:

\[ \{1: \exists x 3, 2: \forall y 4, 5: \text{like } x \, y) \}\]

and obtain the fully scoped versions from that by adding constraints: adding \(3=2, 4=5\) will give scoping 1, and adding instead \(3=5, 4=1\) will give 2.

The main attraction of this kind of representation for emotion annotation is that it allows us very cleanly to single out parts of a meaning representation for special treatment:

0: (want 1 2)
1: you
2: \((\exists x 3 \, 4)\)
3: (?x)
4: (and 5 6)
<surprise>5: (motorbike 3)</surprise>
6: (buy 1 3))

References

...later