A Semantics for the Knowledge Interchange Format

Patrick Hayes
IHMC, University of West Florida
phayes@ai.uwf.edu

Christopher Menzel
Texas A&M University
cmenzel@philebus.tamu.edu

Abstract

We give a precise semantics for a proposed revised version of the Knowledge Interchange Format. We show that quantification over relations is possible in a first-order logic, but sequence variables take the language beyond first-order.

Introduction

The Knowledge Interchange Format (KIF) has been widely used in the fields of knowledge engineering and artificial intelligence. Most notably, perhaps, it is currently being used actively in the development of the Standard Upper Ontology (SUO) project. There is good reason for this, as there is need for a rigorous, computer-readable knowledge representation framework like KIF that is based clearly upon the solid foundations of first-order logic. Due to its growing importance, there is now a renewed push to make KIF an official international standard. There is, in fact, a draft standard for KIF (Genesereth 98) that was developed by Michael Genesereth, who has been KIF's central developer to this point. And it is a good start. However, we feel that the version of KIF in that document falls short in several respects. First, it is not sufficiently "modular". That is, a number of features of KIF – notably, metalinguistic features, and number systems – are better off separated from the basic logical apparatus of the language. Second, the grammar of the language needs to be updated to allow for such things as a arbitrary URIs and Unicode characters. And finally, and perhaps most critically, a comprehensive semantics for KIF needs to be developed – this in turn will reflect back on the second task, as semantical considerations will impact the grammatical structure of the language. In this short paper, we will focus on the last of these. That is, we will provide a definition of the notion of a KIF language (somewhat simplified for purposes here) and we will then propose a semantics for that language. To avoid confusion, we will refer to the new version of KIF here proposed as "SKIF".

Overview

Before giving formal definitions we review the aspects of SKIF that are of particular interest and chiefly serve to distinguish it from earlier versions of KIF. SKIF follows KIF in many respects. Notably,

1. SKIF uses a LISP-compatible syntax where every well-formed expression is represented as an Sexpression containing its immediate syntactic constituents as elements (although a SKIF expression is not assumed to *be* a LISP datastructure);

2. SKIF allows relations to be *variably polyadic* – i.e., it allows them to be true of varying numbers of arguments;

3. A syntactic corollary of the preceding feature is that SKIF puts no restrictions on the number of terms that can follow the term occurring in predicate position in an atomic formula, that is, terms that can occur in predicate position are not assigned a unique *arity*.

4. SKIF has a provision for quantifying over finite sequences of arguments, using a special form of variable. In SKIF these are called *row variables*, which generalize the sequence variables in KIF.

As we will show, the full use of row variables in fact takes the language beyond first-order expressiveness, but there is a useful sublanguage which is strictly first-order.

However, some aspects of KIF have been omitted. Notably:

1. SKIF has no special provision for stating definitions, since definitions have no semantic content.

2. The semantics of row variables is worked out in much greater detail than is the semantics of sequence variables

3. Most notably, SKIF uses a *free syntax*, by which we mean that no distinction is made between object

constants, function symbols, and relation symbols. There are only *terms*, and any term may occur in predicate position in an atomic sentence, in function position in a function term, or in argument position in at atomic sentence or function term.

This last difference between KIF and SKIF gives the language a superficial appearance of a higher-order logic. However, as we will show, this syntactic freedom in and of itself (i.e., without the addition of row variables) does not take SKIF beyond first-order expressibility. The purpose of SKIF's free syntax is to put as few syntactic burdens as possible on users, and as few parsing burdens as possible on compilers and other syntax processors.

SKIF syntax

SKIF is written using character sequences called *words*. For now, we define a word to be any sequence of ASCII characters which does not contain quote marks, parentheses or whitespace, and which starts with some character other than an integer or the symbols ? or @. A more detailed syntax, based on Unicode, will be provided in a fuller account of SKIF. This syntax will be also extended in later versions to include such things as documentation forms, restricted quantifiers, quoted strings, importation conventions for including one ontology inside another, and so on. To simplify the statement of the semantics below, we will ignore these extensions in this paper.

An ontology namespace N consists of a recursive set of words.(Some words, e.g. the quantifier names, 'and', 'or' and 'not', and variable names, are reserved for use in SKIF and cannot be included in a namespace.) The elements of this namespace are *constant names* that refer to the distinguished individuals, properties, classes, and relations in the ontology. There are two kinds of variables; a word prefixed with ? is an *object variable*; a word prefixed by @ is a *row variable*.

The syntax for SKIF expressions is:

```
<term>     ::= <objterm>|<rowvariable>
<objterm>    ::= <objvariable>|
       <constant> |<fnterm>
<fnterm>     ::= (<objterm> <term>*)
<sentence>   ::= <atomsent> |<boolsent>
       |<quantsent>| <equalsent>
<equalsent> ::= (=<objterm> <objterm>)
<atomsent>   ::= (<objterm> <term>*)
<boolsent>   ::= (not <sentence>)|
       ({and|or} <sentence>*)|
      ({=>|<=>} <sentence> <sentence>)
<quantsent> ::=
({forall|exists}({<objvar>|<rowvar>}+)
                 <sentence>)
```

The SKIF *language* λ_N *generated by the namespace* N is simply the set of expressions generated by the above BNF when N is the set of all <constant>s. In particular, the expressions generated by <sentence>, <constant>, <term>, <objterm>, and <fnterm> are known as the *sentences*, *constants*, *terms*, *object terms*, and *function terms* of λ_N, respectively.

SKIF Semantics

We begin with an informal discussion of the semantics of SKIF. There are two particularly important features of KIF's proposed syntax that deserve attention: (i) its free syntax and its semantics, and (ii) the semantics of row variables.

SKIF's Free Syntax and Its Semantics

The model theory for SKIF is very close to a conventional model theory for first-order logic, but has a few unusual features which arise from the need to cope its free syntax. In particular, since any name can be used as a function or a relation name, the semantics must assign a truth-value to expressions even if their function or relation symbol does not denote a function or relation, and it must make sense of terms whose function name refers to a relation which is not functional. It must also be able to handle self-application (i.e., an expression where the same name is used to refer to a relation or function as to one of its arguments.)

To handle all these issues, SKIF utilizes a basic semantic device which associates an *extension* with every object in the domain. The extension of an object is a set of *n*-tuples. The extensions of individuals are always empty. Structurally, therefore, in a given model, individuals are indistinguishable from relations with empty extensions; they are individuals rather than relations with empty extensions in the model simply in virtue of having been *declared* to be individuals.

The basic semantic rule is that an atomic sentence of the form (t_0 t_1 ... t_n) is interpreted as true just when *the extension associated with* the denotation of t_0 contains the *n*-tuple consisting of the denotations of t_1, ..., t_n, respectively. Sentences whose interpretation would violate the "normal" interpretation rules (such as the use of an individual name in a relation position) simply turn out to be false, since the relevant extensions fail to supply suitable values to make them true.

Since relations are distinguished from their extensions, the extension of a relation may contain the relation itself (or an n-tuple which contains it). This provides a way to describe self-application without violating the axiom of foundation. (Although the model theory is in fact completely agnostic about whether relations and extensions are distinct. The association could be identity, although in that case the models would have be understood relative to some non-well-founded set theory, in general. We discuss this issue in more detail below.)

Notice that while a SKIF interpretation must provide a denotation for every term in the language, and hence an extension for every term used as a relation or function, there is no presumption that the interpretation must contain any particular universe of relations. In contrast, classical higher-order logic requires that every domain contain all relations over the domain of individuals; higher-order syntax with the Henkin semantics, while expressively only first-order, requires that they contain all definable relations. This is why SKIF is not in any sense a higher-order logic. In fact, as we will show, SKIF without row variables is really only a notational variation of conventional first-order logic.

One other complication needs attention. Since the syntactic specification of the classes <atomsent> of atomic sentences and <fnterm> of function terms are identical, expressions of the form (<pred> <term>*) can be used both to refer to an object (and hence take on the guise of function term in an atomic sentence) and to make an assertion (and hence themselves occur as an atomic sentence). For instance, suppose that 'father-of' denotes a 2-place relation, i.e., something whose associated extension contains only pairs. If we want to attribute some property to Cain's father, it is convenient to use functional notation:

```
(gardener (father-of cain)).
```

But simply to indicate that Cain's father is Adam, an atomic sentence is preferable:

```
(father-of cain adam).
```

SKIF follows KIF 3 in allowing both forms. However, since SKIF does not declare symbols to be relational or functional, there are two potentially troublesome cases to consider: when a function is used inappropriately in an atomic sentence, and when a term denoting a non-functional relation is used in a function term. For an instance of the first, what if we assert

```
(exists (?x) (father-of ?x))?
```

The answer is straightforward: taken as a sentence (as it must be here), since '`father-of`' expresses a 2-place (functional) relation, '`(father-of ?x)` is simply false for all values of '`?x`' – father-of is not a *property* of any single thing – and hence the quantified sentence above evaluates to false as well.

The second potentially problematic case arises if an expression denoting a non-total function or a non-function (i.e., a relation which is not functional, or a non-relation) is used in a function position in a function term $(f\, t_1 \ldots t_n)$. In such cases, the term in question will not have a "natural" denotation for at least some values of $t_1 \ldots t_n$. For instance, if '`father-of`' denotes the father-of relation in a model of the Genesis myth, then '`(father-of adam)`' is undefined, as Adam had no biological father. This is problematic, as undefined terms are not permitted in classical logic. Hence, in such cases, an arbitrary value is value is assigned to the function term when it occurs in an atomic sentence – we will choose the denotation of the term occurring in function position. This choice could lead to some odd side-effects – for instance, '`((father-of adam) cain adam)`' and '`(exists (?x) (= plus (plus ?x ?x))`' turn out true. However, such side-effects are either innocuous, or can be rendered innocuous in an ontology simply by restricting the quantifiers in axioms carefully to ensure that one is only talking things that are "genuinely" in the range of the function in question. (The things that are genuinely in the range of a function F are simply those things b such that $\langle a_1,\ldots,a_n,b\rangle \in ext(F)$ for some $a_1,\ldots,a_n$. These objects can be identified in SKIF by means of row variables:

```
(forall (?r ?y)
   (=> (relation ?r)
       (<=> (in-range ?y ?r)
            (exists (@args)
                    (?r @args ?y))))).
```

The functionality of a relation can be expressed as follows:

```
(forall (?r)
  (=> (relation ?r)
     (<=> (functional ?r)
        (forall (@args ?y ?z)
           (=> (and (?r @args ?y)
                    (?r @args ?z))
               (= ?y ?z))))))
```

These two axioms together, then, enable one to identify the objects in the range of a function (i.e., functional relation) and talk about them alone in axioms involving terms that denote that function.

The semantics of row variables

As noted, like KIF, SKIF allows relations and functions to be *variably polyadic*. Hence, to capture this fact syntactically, the same symbol can legitimately occur in predicate position in different atomic sentences with different numbers of arguments. To state general properties of such relations, one needs to be able to quantify over a variable number of things at once. In KIF, this was achieved by assuming that expressions were LISP list structures and allowing a form of quantification over the 'tails' (CDRs) of these expressions. SKIF uses a similar but somewhat more abstract device by allowing quantification over arbitrary n-tuples of objects. Such a n-tuple corresponds to a row of symbols in an expression, such as the row of three symbols '`a b c`' in the expression '`(R a b c d)`'. A row variable has such rows of terms as instances in exactly the same way as an object variable has single terms as instances, so that the above expression would be an instance of '`(R @x d)`'. Row variables provide a mechanism for talking about the series of objects indicated by arbitrary rows of terms, without actually putting those series of objects themselves in the universe of discourse. Thus, quantification of row variables provides roughly the same expressive power in the language as is often achieved informally in the metalanguage by the use of three dots to indicate a missing sequence of arbitrary length.

Rows can be thought of simply as a special category of n-tuples. Several distinctive properties of rows should be highlighted here, however. First, they are finite. Since finiteness is not first-order expressible, one might expect that introducing a quantifier which ranges over finite entities will take the logical language beyond first-order expressiveness, and indeed this is the case. Second, a singleton row is indistinguishable from its sole element. Third, there is no way to include one row as a single item in another row: rows have no nested structure, unlike lists, sets, or logical terms.

Formally Speaking

Rows

First, some definitions. For any set A, let A^n be the set of sequences of length n of members of A, i.e. functions from the set $\{0, 1, \ldots, n\text{-}1\}$ of ordinals $<n$ into A. (It follows that $A^0 = \{\varnothing\}$ since the set of ordinals < 0 is empty.) We call the members of A^n *n-tuples over A*. We will write the n-tuple $\{\langle 0,a_0\rangle, \langle 1,a_1\rangle, \ldots, \langle n\text{-}1,a_{n-1}\rangle\}$ as $\langle a_0,a_1,\ldots,a_{n-1}\rangle$. In particular, we will sometimes refer to $\varnothing$ as '$\langle\rangle$' when thinking of it as a 0-tuple. Let A^* be the set of all n-tuples over A, for all n, i.e., $A^* = \sim_{n<\omega} A^n$. We will call the members of A^* tuples over A. We want to consider n-tuples in which there is no distinction between 1-tuple and its (sole) member, so we define an equivalence relation $\approx$ on $A \cup A^*$ by : $\langle x\rangle \approx x$, and let A^{**} be the set of equivalence classes on $A \cup A^*$ under $\approx$. We call the members of A^{**} *rows over A* or *A-rows*. Notice that any element of A constitutes a singleton row (this makes the technical statement of the model theory somewhat simpler). If $s_1, \ldots, s_n$ are rows, then $cc(s_1,\ldots,s_n)$ is the concatenation of those rows; that is, the row whose length is the sum of the component rows and which consists of the elements of those rows in consecutive order. Notice that a concatenation of objects is simply the row of those objects, so that cc is also the row-forming function. We define the value of cc on an empty list of arguments to be the 0-tuple $\langle\rangle$.

These conventions have some consequences for the semantics. In particular, if a SKIF object expression could denote a row, the language would be ambiguous; so we impose a semantic restriction that a SKIF domain cannot contain the rows that are used in the semantic definition. (It may contain isomorphic structures, but they cannot be *identical*.) If this restriction is felt to be onerous, we can simply declare rows to be a special category of structures declared by *fiat* to be isomorphic to the rows described here, but not defined to be identical to their set-theoretic description.

Interpretations

Now, let N be an ontology namespace, and let λ_N be the language generated by N. An *interpretation* of λ_N is a triple $\langle D,ext,V\rangle$ such that

- $D = E \cup R$ is a nonempty set, where $E \cap R$ is empty.
- ext is a function on D such that, for all r in R, $ext(r) \subseteq D^{**}$, and for all e in E, $ext(e) = \perp$.
- V is a function that maps the constants and object variables of λ_N into D^1 and the row variables of λ_N into D^{**}.

Intuitively, R is the set of relations in an ontology (functions are a particular kind of relation, and properties can be thought of as 1-place relations), and E is the set of individuals. Thus, ext maps each relation r to its *extension*, i.e., to a set of rows of elements of D. Notice that the lengths of the rows in the extension of a relation needn't be identical; relations, that is to say, can be *variably polyadic*. For convenience, individuals are assigned an empty extension. Finally, V is the semantical mapping from the terms into entities of an appropriate sort. Specifically, V maps constants and object variables to objects in the domain of discourse D, and row variables into rows, i.e., n-tuples of objects in D. Since $D \subseteq D^{**}$, i.e., single objects are identical to their singleton rows, V can be said to map all terms to rows of objects.

Truth under an Interpretation

Several more definitions will be helpful for defining truth. First, for an interpretation $I = \langle D, ext, V \rangle$, and for any variables $\nu_1, \ldots, \nu_n$, say that an interpretation $I' = \langle D, ext, V' \rangle$ is an *I-variant on* $\nu_1, \ldots, \nu_n$ if V' differs from V at most in what it assigns to one or more of $\nu_1, \ldots, \nu_n$. Second, define a relation r to be *functional on* $\langle e_1, \ldots, e_n \rangle$ just in case there is exactly one object e in the domain of discourse D such that $\langle e_1, \ldots, e_n, e \rangle$ is in the extension $ext(r)$ of r.

Now let φ be an expression of λ_N. We have already defined $V(\varphi)$ when φ is a name or a variable. But we also need to define it for the case when φ is a functional term. This will be included in the following general definition of truth for λ_N.

1. If φ is a functional term $(F\ t_1 \ldots t_n)$, then there are two cases to consider. If $V(F)$ is functional, then $V(\varphi)$ is the unique e such that $cc(V(t_1),\ldots,V(t_n),e) \in ext(V(F))$; otherwise, $V(\varphi) = (V(F))$. This latter choice is arbitrary, as noted above.

2. If φ is an atomic sentence $(P\ \ t_1\ \ \ldots\ \ t_n)$, then φ is *true under I* , just in case $cc(V(t_1),\ldots,V(t_n)) \in ext(V(P))$.

3. If φ is `(=` $t_1\ \ t_2$`)` , then φ is true under I just in case $V(t_1) = V(t_2)$.

4. If φ is `(not` ψ`)`, then φ is true under I just in case ψ is not true under I.

5. If φ is `(and` $\psi_1 \qquad \psi_n$`)`, $n \geq 0$, then φ is true under I just in case, for all positive integers $i \leq n$, ψ_i is true under I.

6. If φ is `(or` $\psi_1 \qquad \psi_n$`)`, $n \geq 0$, then φ is true under I just in case, for some positive integer $i \leq n$, ψ_i is true under I.

7. If φ is `(=>` $\psi\ \ \theta$`)`, then φ is true under I just in case ψ is not true under I or θ is true under I.

8. If φ is `(<=>` $\psi\ \ \theta$`)`, then φ is true under I just in case either both ψ and θ are true under I or neither is.

9. If φ is `(forall` $(\nu_1 \ \ldots\ \nu_n)\ \ \theta$`)`, then φ is true under I just in case θ is true under all I-variants on $\nu_1, \ldots, \nu_n$.

10. If φ is `(exists` $(\nu_1 \ \ldots\ \nu_n)\ \ \theta$`)`, then φ is true under I just in case θ is true under some I-variant on $\nu_1, \ldots, \nu_n$.

Most of the above is standard; all the originality is in the first two clauses. The complexity of the clause for object terms arises from the need to specify a semantic value for the functional term when it has no intuitively correct value, as discussed earlier. Both clauses use the notion of a concatenation of rows to form the appropriate n-tuple of arguments which is supplied to the function or relation. For conventional first-order logical syntax this would be needlessly complicated (although correct), but the use of row variables requires this much care. Notice that this definition extends to empty rows, so that an atomic sentence of the form (P) will be true just in case the empty sequence $\langle\rangle \in ext(r)$. Note also that as special cases of clauses 5 and 6, `(and)` is vacuously false, and `(or)` vacuously true, under any interpretation.

Extensional and intensional interpretations

This semantics makes a conceptual distinction between relations and their extensions. This has a number of technical advantages, notably that of giving a denotation to expressions involving circular applications, such as (in the most extreme case)

```
(R R)
```

This is true under I just in case I(R) denotes e and $e \in ext(e)$, which of course is quite an unexceptional condition. However, if we were to insist that relations were identical to their extensions, this would amount to the requirement that $e \in e$, in direct contradiction with the axiom of foundation. There are several possible responses to this observation, and the authors do not agree on which is preferable. They do agree, however, that a merit of the SKIF model theory is that it is entirely agnostic about which stance to take.

On one view, to distinguish between relations and their extensions is philosophically proper, and so nothing more needs to be said; particularly as the SKIF model theory can be derived from a more conventional Tarskian interpretation of a sublanguage, as we show below.

On another view, it is proper to think of relations in a first-order model theory as being identical with their extensions. This can be partly expressed by an axiom of extensionality:

```
(forall (?r ?s)
  (=> (forall (@x)(<=>(?r @x)(?s @x)))
      (= ?r ?s)))
```

which asserts, in effect, that *ext* is 1:1; call this a *weakly* extensional interpretation, and an interpretation in which *ext* is identity a *strongly* extensional interpretation. There is no way in SKIF to write an axiom which guarantees strong extensionality, but on the other hand any weakly extensional interpretation defines a strongly extensional one in which every sentence has the same truth value, so if strong extensionality is considered desirable, it may be imposed by stipulation. The cost, of course, is that one has then to understand the entire semantic description relative to a 'non-well-founded' set theory which rejects the axiom of foundation, such as that described by Peter Aczel (1988). This makes no difference to the actual model theory as stated, but some may find the conceptual burden too onerous.

To sum up, there are three options regarding the interpretation of the semantics. A nonextensional interpretation, or weak extensionality, are always options; if one wishes to impose strict extensionality, and if the language being considered uses circular application, then one must interpret the model theory relative to a non-well-founded set theory. The formal properties of SKIF and its model theory work the same way in all three cases.

Extensions to SKIF

The SKIF language described here is a minimal language; we envision that the final form of SKIF will contain several syntactic extensions designed for ease of use. A complete version of the language will be given elsewhere, but currently proposed extensions include, but are not limited to, the following. Only the last one changes the essential logical character of the language.

Documentation wrappers

The language will have special syntactic forms for attaching documentation strings and other 'tags' to expressions and assertions. (Although logically trivial, this complicates the syntax since such strings may contain otherwise 'illegal' characters.) We also expect to define the language to allow Unicode character strings in the UCS-2 subset, making it more appropriate for international use.

Restricted quantification

Restrictions on quantifiers can be written inside the quantifier syntax, so that for example one can write

```
(forall (?x (human ?x)) (bipedal ?x))
```

in place of

```
(forall (?x)(=>(human ?x)(bipedal
?x)))
```

Class heirarchies and sorting

SKIF follows normal logical practice in treating properties as unary relations, but deviates from the usual first-order logical tradition by also treating them as objects. Other languages treat properties as classes[1], and some logical languages allow certain classes to be distinguished as 'sorts'(where sort membership can be checked by the parser without performing general inference.)

Much of this can be axiomatized within SKIF as described here, but a fuller version of SKIF will provide syntactic devices to allow the user to specify sort information in a uniform framework.

Namespaces and importation

We will provide syntactic machinery for declaring namespaces, maintaining namespace separation, and importing ontologies from remote sources (such as web sites or external files). None of these change the basic logical properties of the language. However, some extensions are proposed which do give the language significantly greater expressive power, particularly a meta-extension:

Meta-extension

This will provide syntactic machinery for describing and quantifying over linguistic expressions, including those of the language itself, and for asserting the truth of such expressions. We expect to use the '`wtr`' predicate from KIF as a suitable truth predicate to avoid the well-known paradoxes.

---

[1] We here use class as the term is used in the description logic literature; cf. (McGuinness & Patel-Schneider 1998) not in the sense of proper class from set theory.

Mapping SKIF into conventional logic

SKIF can be mapped into a more conventional logical language in two stages. We first describe an embedding from SKIF into a restricted SKIF language which has a conventional clear distinction betwen object, function and relation symbols of fixed arity, and then give a further truth-preserving mapping from SKIF into a similar logic without row variables. These mappings provide an intuitively useful account of SKIF, establish its basic logical properties, and illustrate how the semantic complexities (notably the relation/extension distinction and the central use of row-concatenation) arise.

Holding and applying

To 'tidy up' the free syntax of SKIF we utilize a familiar trick for writing 'higher-order' syntax into a first-order notation, which makes use of a denumerable set of special relations `Holds-0`, `Holds-1`, ... and function symbols `App-0,App-1,...` (one for each natural number), and, for a given sentence φ we simply rewrite every term occurrence of an expression of the form

$(t_0 \ t_1 \ \ldots \ t_n)$

as

$(\texttt{App-}n \ t_0 \ t_1 \ \ldots \ t_n)$

and every sentential occurrence as

$(\texttt{Holds-}n \ t_0 \ t_1 \ \ldots \ t_n)$,

with these translations applied recursively to every subexpression in the obvious way. If θ (or λ_N) is an expression (or a language on N), let $H(\theta)$ (or $H(\lambda_N)$) be the expression (or language) defined by the above translation; we will call this the *holds expression* (*language*) corresponding to θ (λ_N) .

In any holds language, all the symbols of the original SKIF language have become individual names; all the function, relation and individual names are distinct, and each function symbol and relation symbol has a unique arity; and no variables or nonatomic terms occur in the relation or function position. Apart from the presence of row variables, therefore, a holds language is a conventional first-order language. To prove this, it can easily be shown that any interpretation *I* of an SKIF language can be transformed into a standard first-order interpretation *I** of the corresponding holds language in a truth preserving way (i.e., sentences of SKIF have the same truth value in *I* that their counterparts under H have in *I**).

SKIF and Infinitary Logic

As noted, row variables extend SKIF's expressive power beyond that of simple first-order logic. Specifically, they give it the expressive power of an infinitary sublanguage of the infinitary logic $L\omega_1\omega$. In this section we make this explicit.

To keep the exposition simple, we will first define a truth-preserving mapping from SKIF to an infinitary logic without row variables. The resulting language retains the other odd features of SKIF, such as variable polyadicity, so is not strictly a sublanguage of $L\omega_1\omega$, but if this mapping is composed with the transformation H described in the previous section, the result is strictly within $L\omega_1\omega$. We will call this mapping K; the mappings H and K commute.

The intuitive idea of the K mapping is that any instance of a row variable @x is also a row of instances of a row of object variables $\nu_1 \ldots \nu_n$, so the effect of universally quantifying a row variable can be obtained by conjoining an infinite series of expressions representing all the possible universal quantifications of those rows using object variables; and similarly , of course, by disjoining such a series for an existential quantifier. An expression of the form

```
(forall (@x)(foo @x))
```

thus will map into an infinite conjunction of the form

```
(and (foo)
     (forall (?x1)(foo ?x1))
     (forall (?x1 ?x2)(foo ?x1 ?x2))
     (forall (?x1 ?x2 ?x3)(foo ?x1 ?x2
?x3))
...)
```

To state this formally it is convenient to assume that row quantifiers are treated separately, with each quantifier binding only a single row variable. Obviously every expression has an equivalent expression in this *row-separated* form.

Let λ_N be a SKIF language in row-separated form. We will map λ_N into an infinitary language Λ_N over the same vocabulary. The complex formulas of Λ_N will be as usual for such a language; specifically, Λ_N allows countable conjunctions and disjunctions (but only finite quantifier strings). Given a countable set S of formulas of Λ_N, we will indicate their conjunction by $\wedge S$ and their disjunction by $\vee S$. If S is indexed by the finite ordinals, we will indicate the conjunction of the members of S by $\wedge_{i<\omega}S$ and their disjunction by $\vee_{i<\omega}S$.

For any SKIF expression θ and for any row variable ρ and object variables $\nu_1, \ldots, \nu_m$, let $\theta\,[\rho/\nu_1 \ldots \nu_m]$ be the result of replacing every occurrence of ρ in θ with the row $\nu_1 \ldots \nu_m$. (Notice that this refers to all occurrences of ρ, even those inside quantifiers.) Then the translation scheme K from λ_N into Λ_N is defined recursively as follows:

- If φ is any word, then $K[\varphi] = \varphi$.
- if ρ is a row variable and φ is `(forall (ρ) ψ)`, then $K[\varphi] = \wedge_{m<\omega}\{$ `(forall (`$\nu_1 \ldots \nu_m$`) K[` $\psi[\rho/\nu_1 \ldots \nu_m]$`) ]` $\}$, where, for each $m<\omega$, $\nu_1, \ldots, \nu_m$ are the m alphabetically earliest object variables that do not occur in ψ.
- If ρ is a row variable and φ is `(exists (ρ) ψ)`, then $K[\varphi] = \vee_{m<\omega}\{$ `(exists (`$\nu_1 \ldots \nu_m$`) K[`$\psi[\rho/\nu_1 \ldots \nu_m]$`) ]` $\}$, where, for each $m<\omega$, $\nu_1, \ldots, \nu_m$ are the m alphabetically earliest object variables that do not occur in ψ.
- Otherwise, $K[\varphi]$ is the expression got by applying K to every well-formed immediate subexpression of φ.

It is easy to see that K is truth-preserving: the statement of the truth-conditions for universal row quantification refer to all finite rows, which can be transcribed directly into the truth-conditions for the infinitary conjunction, each of whose conjuncts states the analogous condition for one of the possible finite lengths of the row; and similarly for the existential case.

It is almost equally clear that the full meaning of the row quantifier could not be captured by any finite subexpression, since any such expression would omit some case that might falsify the infinite conjunction. Hence the row quantifiers take the language beyond strict first-order definability. We have not yet fully investigated the extent to which this extra expressiveness may take the language, but some facts are clear.

Given any SKIF language, applying both the mappings K and H (in either order) yields a sublanguage of the well-known infinitary logic $L\omega_1\omega$. However, these mappings only utilize a sublanguage of $L\omega_1\omega$, which we will call SKIF$\omega_1\omega$, rather than the full language, and there is some hope that this sublanguage may be computationally more tractable than full $L\omega_1\omega$.

$L\omega_1\omega$ allows arbitrarily complex conjunctions and disjunctions, while SKIF$\omega_1\omega$ only uses a highly simplified subcase in which each component can be generated from the previous one by a simple addition of one new variable. We have not yet fully investigated the tractability of the SKIF$\omega_1\omega$ sublanguage relative to $L\omega_1\omega$, but there is some hope that computationally useful properties may be obtained. $L\omega_1\omega$ itself is among the more tractable infinitary languages, and for example admits a completeness theorem from countable sets of premises. To illustrate the problems, however, we note that skolemisation is not possible in general for row quantifiers, as can easily be seen by applying the K mapping to a sentence containing an existential row quantifier inside the scope of a universal row quantifier.

A useful sublanguage and its limitations.

The fact that SKIF (and indeed KIF, as may be easily checked by considering a similar mapping from sequence variables into $L\omega_1\omega$) is not first-order, may be reasonably considered to be Bad News. Moreover, the generalization of KIF sequence variables to SKIF row variables, although theoretically cleaner, has some computational disadvantages, since the use of row variables in general position makes unification very difficult: because row-concatenation is associative, simple pairs of expressions, such as `(r a @x)` and `(r @x a)`, may have infinitely many most general unifiers. However, there is a restricted sublanguage of SKIF which avoids most of these difficulties and but may still be of general utility, which we will call 'schema form' since it corresponds to the use of schemas in first-order logic.

An SKIF sentence is in *schema form* if row variables are 1. restricted to universal quantifiers at the top level of any assertion (i.e., not within the scope of any expression other than a universal quantifier), and

2. only occur in the final position of any atomic sentence or relational term

and 3. if the sentence is used only as an assertion, and never posed as a theorem to be proved.

The second restriction makes row variables similar in usage to sequence variables in KIF, i.e., they always intuitively refer to the 'tails' of argument lists. In practice this is not usually felt to be an onerous restriction, although it does sometimes force axioms to be written in a more complex style. It has the merit of guaranteeing that unification patterns that create the above-mentioned difficulties cannot arise. (There may be less restrictive conditions which also supply such a guarantee: we have not investigated this in detail.)

The first restriction is rather more dramatic, in that it makes this sublanguage first-order expressible. To see this, notice that any such 'top-level' universal quantifications can be mapped by K into a top-level infinitary conjunction, which in turn is equivalent to an infinite set of first-order sentences. By the compactness theorem, any consequence of this set is also a consequence of some finite subset of them, so this infinite set can be treated as strictly first-order, when used as an assertion. (Notice however that this argument would break down if the sentence were posed as a theorem, since it could only be proved from an infinite number of premises.) Since free variables in assertions are taken to be universally quantified, an assertion in schema form may be thought of as having only free row variables. Notice also that skolemisation can be applied to expressions in schema form, uisng a polyadic skolem function:

```
(forall (@r) (exists (?x)(...?x...)
```

can be transformed into

```
(forall (@r) (... (f @r) ...)
```

All this amounts to the observation that a first-order *schema* does not extend the language beyond first order if it used solely as a 'source' of axioms, standing in place of a recursively infinite set of assumptions. Schema form provides a general-purpose notation for writing computationally tractable first-order schemas in this sense, and also makes clear their expressive limitations.

In particular, notice that there is no way to describe the integers, or to guarantee finiteness, using such a schema. For example, since rows are defined to be finite, one might think that one could use unary arithmetic to define integers, using a row of length *n* as a numeral to represent the number *n*. It is indeed easy to write the relevant axioms in SKIF:

```
(equinum 0)

(forall (@r ?x ?y)
(<=> (equinum (succ ?x) ?y @r)
   (equinum ?x @r)))

(<=> (finite ?x) (exists (@r)(equinum
?x @r)))
```

but, as is easily checkable, the final biconditional cannot be put into schema form.

In general, the common use of 'recursive' definitions in SKIF (and KIF), as in the second axiom above, is always hostage to the expressive limitations of the language. If row quantification is considered to be part of the language, then finiteness is expressible and such recursive definitions can be considered to be fully specified; but if the language is restricted to first-order, then the language cannot possibly express their full import. Just as with schemas, they can be considered to be 'generators' of an infinite set of first-order axioms, but this only provides a lower bound on their first-order models, and cannot rule out nonstandard models in which quantification over the 'recursively defined' constructs (integers, in the above case) also range over nonstandard elements which would be excluded by a least-fixed-point semantics or by the expressive power of an infinitary logic. So for example, the definition of lists in KIF (reference KIF3 manual) is in fact incomplete, in that it fails to restrict the models to only the finite lists.

Acknowledgements

This work arose from a group effort to create a revised version of KIF and has benefitted from input from our colleagues in the group.The extensionality axiom was provided by Bill Andersen.

Hayes recognises the partial support of DARPA under contract # F30602-00-2-0577.

References

Aczel, P. (1988) "Non-well-founded Sets", CSLI Lecture Notes, number 14

Bell, J. (2000) "Infinitary Logic", Stanford Encyclopedia of Philosophy, http://plato.stanford.edu/entries/logic-infinitary.html

Genesereth, M. et. al. (1998), “Knowledge Interchange Format,” draft proposed American National Standard (dpANS), NCITS.T2/98-004, http://logic.stanford.edu/kif/dpans.html

McGuinness, D. L. and Patel-Schneider, P. F. (1998) “Usability Issues in Description Logic Systems,” *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, Madison, Wisconsin.