SCL: A LOGIC STANDARD FOR SEMANTIC INTEGRATION

CHRISTOPHER MENZEL AND PATRICK HAYES

The Knowledge Interchange Format (KIF) [2] is an ASCII-based framework for use in exchanging of declarative knowledge among disparate computer systems. KIF has been widely used in the fields of knowledge engineering and artificial intelligence. Due to its growing importance, there arose a renewed push to make KIF an official international standard. A central motivation behind KIF standardization is the wide variation in quality, style, and content — of logic-based frameworks being used for knowledge representation. Variations of all three types, of course, hinder the possibility of semantic integration. A well-crafted logic standard for the representation of declarative knowledge would impose some greatly needed syntactic and semantic uniformity on the current somewhat chaotic situation, uniformity that would in turn greatly enhance the capacity for semantic integration.

For all its potential advantages, however, the idea of a logic standard is problematic for at least two reasons:

- Standardization of a single syntax forces comformant users to write their logic in a form that is likely to be, at the least, unfamiliar, and, at worst, may in fact not be optimal for their representational needs. Call this the *uniformity* problem
- The standard might involve constructs that are neither needed nor desired for one's representational purposes. Call this the *excess baggage* problems.

KIF in fact seems particularly vulnerable to objections along these lines. Its LISP-like syntax is not universally held in high esteem. Moreover, it includes a variety of constructs that researchers find quirky and unnecessary, notably:

- Variable polyadicity — predicate constants and function symbols have not fixed arity, but can take any number of arguments;
- Pseudo-higher-order constructs — bound variables can occur in predicate position in atomic formulas.
- Type-freedom — predicates can occur as arguments to other predicates; semantically speaking, properties and relations are "first-class" objects that can be referred to and quantified over like any other individuals.
- Non-first-order expressiveness — KIF includes "sequence variables", the presence of which raises its expressive power beyond first-order to that of a weak infinitary logic.

To add to the confusion, KIF has lacked a rigorous model theory for its distinctive constructs.

Nevertheless, the idea of standardization is still a good one — widespread conformance to such a standard would go a long way toward enabling semantic integration between diverse knowledge bases. Moreover, something on the order of KIF's full first-order expressive power, at the least, is still needed, especially for the metalinguistic constructs that are inevitably needed to enable semantic integration. Finally, though superfluous in some context, KIF's additional constructs prove useful and convenient in others.

The solution sketched in this brief technical paper — the Simplified Common Logic (SCL) framework[1] — addresses the uniformity problem by defining a purely abstract syntax that specifies only the underlying structure that a conformant language must exhibit, leaving the concrete specifics of any given manifestation to the discretion of the user. SCL addresses the excess baggage problem by defining the grammatical framework flexibly enough to allow users to pick and choose from a variety of syntactic constructs depending on their representational needs and preferences. Finally, a rigorous general model theory is provided that yields definitions of denotation and truth for any given SCL language.[2]

1. LEXICONS

An SCL language is based upon an initial stock of primitive syntactic entities. Specifically, an SCL *lexicon* λ will consist of the following sets:

- A countable set *PCon* called the *predicate constants* of λ. This set will include a distinguished predicate *Id*. (Predicate constants will also be referred to simply as *predicates*.)
- A countable set *ICon* called the *individual constants* of λ.
- A countable set *FnSym* called the *function symbols* of λ.
- A denumerable set *GVar* called the *general variables* of λ;
- A set *SVar* called the *sequence variables* of λ. *SVar* will be either empty or denumerable.

If *SVar* is empty, then λ is known as a *first-order* lexicon.

[1] SCL is part of the Common Logic Standard effort; see [1]. The present paper is a distillation of some of the current SCL working document [4].

[2] We have recently been made aware of the language HiLog [3], which purportedly is syntactically and semantically quite similar to SCL (without sequence variables) . We have not had the time yet to study the framework full, so we will have to report on the similarities and differences in a further paper.

$Con = PCon \cup ICon$ is known as the set of *constants* of λ. $Var = GVar \cup SVar$ is known as the set of *variables* of λ. *GVar* and *SVar* shall be disjoint, and *Var* shall be disjoint from $Con \cup FnSym$. Let $PrimTrm = ICon \cup GVar$. *PrimTrm* is known as the set of *primitive terms* of λ.

Lexicons λ also come with a function *arity* that maps each predicate constant and function symbol into the set $\mathbb{N} \cup \omega$, where $\mathbb{N}$ is the set of natural numbers and ω is any object not in $\mathbb{N}$. For predicates π, *arity* will indicate the number of arguments π will take. (This will of course be expressed explicitly in the grammar below.) If $arity(\pi) = n \in \mathbb{N}$, then π is said to be an n-place predicate; otherwise π is *variably polyadic*. Variably polyadic predicates will be able to take any number of arguments. We let $PCon_n$ be the set of n-place predicates, and $PCon_\omega$ the set of variably polyadic predicates. Because we will interpret function symbols as functional relations, we will let the arity of a function symbol correspond to the arity of the relation it denotes rather than to the number of arguments it takes. This will also enable the predicates of an SCL lexicon to do double duty as function symbols — note that there is no requirement that *PCon* and *FnSym* be disjoint. Accordingly, for function symbols α, if $arity(\alpha) = n+1$, we say that α is an n-place function symbol; otherwise α is variably polyadic. We stipulate that $arity(\alpha) \neq 0$, for any function symbol α. We let $FnSym_n$ be the set of n-place function symbols, and $FnSym_\omega$ the set of variably polyadic function symbols.

Over and above presence of sequence variables, SCL lexicons differ from traditional first-order lexicons in three important ways. First, SCL generalizes the notion of arity by allowing (though not requiring) variably polyadic predicates and function symbols, i.e., predicate constants and function symbols that can take arbitrarily many arguments. Variably polyadicity is especially useful and appropriate in SCL languages containing sequence variables.

Second, it is not required that *PCon*, *ICon*, and *FnSym* be pairwise disjoint. This reflects SCL's goal of generality. Many knowledge representation languages are "type-free" to one extent or another; that is, they treat properties, propositions, classes, functions, and other so-called "higher-order" entities as "first-class citizens" in their own right, capable of being referred to and quantified over along with individuals. Natural language itself reflects this "dual role" that properties and their ilk can play in the gerundive construction, whereby verb phrases expressing properties and relations — e.g., *is a linguist* — are transformed into noun phrases — *being a linguist*. By allowing predicate constants and function sybols simultaneously to serve as individual constants, and by allowing variables to serve as predicable terms, SCL provides a formal correlate to these constructions and thereby provides a rigorous framework in which this common knowledge representation construction is fully sanctioned.

To illustrate SCL's flexibility, we explicitly pick out several important limiting cases of SCL languages that are determined by minimally or maximally tweaking arity and the degree of overlap among constants and function symbols. Thus, say that an SCL lexicon λ is *fully typed* if $(PCon \cup FnSym) \cap ICon = \varnothing$ (i.e., if there is no overlap between the predicates constants, function symbols, and individual constants of λ); *arity-fixed* if, for all predicate constants and function symbols κ, $arity(\kappa) = n$, for some $n \in \mathbb{N}$ (i.e., if every predicate constant and function symbol has a fixed arity); and *traditional first-order* (*TFO*) if λ is both fully-typed and arity-fixed. By contrast, say that λ is *arity-free* if, for all predicate constants and function symbols κ, $arity(\kappa) = \omega$; *type-free* if $PCon \cup FnSym \subseteq ICon$; and *unconstrained* if λ is both arity-free and type-free. In between the extremes of TFO and unconstrained lexicons, of course, lie any number of interesting intermediate possibilities.

2. GRAMMARS

2.1. **Terms.** Given an SCL lexicon λ, we define the notion of a term class based on λ. Intuitively, a term is either a primitive term (constant or variable) or the result of "applying" a function symbol to some nonempty sequence of terms. Because we are defining an abstract syntax, we do not want to specify the exact form that the application of a function symbol to its arguments should take. Hence, we simply specify the general constraints than any syntax of application must satisfy; we do this in terms of a certain type of syntactic function.

As groundwork for this definition, for any set M, let M^ω be the set of finite sequences of elements of M, i.e., $M^\omega = \bigcup n \in \mathbb{N} M^n$, where M^n is the set of all n-tuples of elements of M. Given this, say that T is a *term class for* λ if T contains all of the primitive terms of λ and is the smallest class closed under a one-to-one operation **App** — called a *term generator* for λ — such that

$$\textbf{App} : \bigcup_{n\in\mathbb{N}}\{FnSym_n \times T^n \cup (FnSym_\omega \times (T^\omega \cup (T^\omega \times SVar)))\} \longrightarrow T.$$

That is, for $\tau_1, ..., \tau_n \in T$, if α is an n-place function symbol, then $\textbf{App}(\alpha, \tau_1, ..., \tau_n) \in T$, and if α is a variably polyadic functional, then in addition for any sequence variable σ, $\textbf{App}(\alpha, \tau_1, ..., \tau_n, \sigma) \in T$;

We say that **App** *generates* the corresponding term class T. For any term generator **App** for λ, let $FnTrm = Range(\textbf{App})$. $FnTrm$ is the set of *function terms* of λ (relative to **App**).

So, for example, if `a` and `b` were among the constants of a lexicon λ and `f` and `g` among its function symbols, then any of the following might among the function terms

produced by different generators: `f(a,g(b),s)`, `(f a (g b) s)`, `s[bg]af` (somewhat perversely) and even the XML'ish

```
<term>
  <fnsym>f</fnsym>
  <indcon>a</indcon>
  <term>
    <fnsym>g</fnsym>
    <indcon>b</indcon>
  </term>
  <seqvar>s</seqvar>
</term>
```

2.2. **Type-Freedom and Predicability.** As hinted at above, and as will be spelled out in more detail in the model theory below, one of the important features of SCL is that it allows for a "type-free" semantics in which properties and relations are treated as first-class individuals. Languages with such a semantics will there be allowed to refer to and quantify over such "reified" entities directly. In particular, it is important to allow such languages to quantify over them in their predicative roles. Syntactically speaking, this means that we must allow variables to occur in predicate position in atomic formulas, e.g., in KIF:

```
(forall (?x ?y ?F)
        (impl (Symmetric ?F)
              (impl (?F ?x ?y) (?F ?y ?x))))
```

However, because it is important that SCL encompass more traditional first-order languages as well, type-freedom should be optional. Accordingly, whether or not variables (and other expressions, more generally) can occur in predicative position along with predicate constants will be specified in the grammar for a language, rather than being predetermined by the chosen lexicon. Consequently, the set $Pred_n$ of n-place predicables in an SCL grammar is allowed to be either simply the set $PCon_n \cup Pred_\omega$ (since variably polyadic predicates be predicated of any finite number of arguments — hence, in particular of n) or the set $PCon_n \cup Pred_\omega \cup GVar$. A similar generalization that allows variables to occur in function position in complex terms adds a certain elegance and convenience at the cost of a great deal of semantic complexity, but the gains are minimal for the purposes envisioned for SCL.

2.3. **Formulas.** In light of the above, we now do for formulas what we did for terms. Let λ be an SCL lexicon, and let *Trm* be the term class for λ generated by some term generator **App**. First, we need a class of basic formulas. Let **Holds** be a one-to-one function on $\bigcup_{n\in\mathbb{N}}\{Pred_n \times T^n \cup (Pred_\omega \times (T^\omega \cup (T^\omega \times SVar)))\}$. That is, given an n-place predicable and n terms, or a variably polyadic predicable, n terms and a sequence variable, **Holds** returns a unique formula. Any such function **Holds** is said to be a *predication operation for λ based on* **App**. As with term generators, the outputs of different predication functions might take very different forms. The only constraint is that distinct inputs always yield distinct outputs. Given a term generator, the range of a predication operation **Holds** for λ is said to be the class of *atomic formulas* for λ generated by **Holds**.

Let *At* be the class of atomic formulas for λ based on a predication operator **Holds**. Say that F is a *formula class* for λ, relative to **Holds**, if it is the smallest class that includes *At* and is closed under a set *Op* — known as a *formula generator* for λ based on **Holds** — of operations **Id**, **Neg**, **Conj**, **Disj**, **Cond**, **Bicond**, **EQ**, **UQ** that satisfy the following conditions:

- Each operation is one-to-one;
- The ranges of the operations are pairwise disjoint, and disjoint from *Trm*
- **Id** : $Trm \times Trm \longrightarrow F$
- **Neg** : $F \longrightarrow F$
- **Conj** : $F^* \longrightarrow F$
- **Disj** : $F^* \longrightarrow F$
- **Cond** : $F \times F \longrightarrow F$
- **Bi** : $F \times F \longrightarrow F$
- **EQ** : $(GVar \cup (GVar \times (PCon_1 \cup PCon_\omega)))^* \times F \longrightarrow F$
- **UQ** : $(GVar \cup (GVar \times (PCon_1 \cup PCon_\omega)))^* \times F \longrightarrow F$

Let *Fla* be range of the operations in *Op*. We say that *Fla* is the formula class *generated by Op*.

As with terms, depending on one's choice of term generator, predication operation, and generator set, SCL languages can come in many different concrete forms. So, for example, the standard, first-order "logical form" of 'Every boy kissed a girl' in terms of our abstract syntax is

UQ$(\nu_1,$ **Cond**$($**Holds**$(\pi_1,\nu_1),$ **EQ**$(\nu_2,$ **Conj**$($**Holds**$(\pi_2,\nu_2),$ **Holds**$(\pi_3,\nu_1,\nu_2)))))$,

where π_1, π_2, and π_3, are "slots" for the predicates constants of the appropriate arity chosen from any particular lexicon to represent boyhood, girlhood, and kissing, and ν_1 and ν_2 represent some choice of variables. In one SCL language, this form might be realized by its familiar introductory text-book form:

$$(\forall x)(Boy(x) \rightarrow (\exists y)(Girl(y) \wedge Kissed(x,y))).$$

A conceptual graph interchange form (CGIF) implementation has a rather different appearance:

```
[@every*x][If:(Boy ?x)[Then:[*y](Girl ?y)(Kissed ?x ?y)]].
```

As does a KIF-like implementation:

```
(forall (?x ?y)
        (impl (Boy ?x))
              (exists (?y)
                      (and (Girl ?y)
                           (Kissed ?x ?y))))
```

not to mention the following XML'ish monstrosity:

```
<formula>
  <forall>
    <var>x</var>
    <formula>
      <implies>
        <formula>
          <atom>
            <con>Boy</con>
            <var>x</var>
          </atom>
        </formula>
        <formula>
          <exists>
            <var>y</var>
            <formula>
              <and>
                <formula>
                  <atom>
                    <con>Girl</con>
                    <var>x</var>
                  </atom>
                </formula>
                <formula>
                  <atom>
                    <con>Kissed</con>
                    <var>x</var>
                    <var>y</var>
                  </atom>
                </formula>
              </and>
            </formula>
          </exists>
        </formula>
      </implies>
   </forall>
</formula>
```

It is important to observe that, because the operations in a generator set for a formula class *Fla* for λ are all one-to-one and disjoint in their ranges, every element of *Fla* will have exactly one "decomposition" under the inverses of those operations, and that all such decompositions are finite. Let $\varphi \in Fla$. An object ϵ in the decomposition of φ is an *atom* of φ just in case ϵ is an element of the lexicon λ. ψ is a *subformula* of φ if $\psi \in Fla$ and ψ is in the decomposition of φ.

2.4. **Languages.** Let **App** be a term generator for λ, where *Trm* is the set generated by **App**, and let **Holds** be based upon **App**. Let *Op* be a formula generator for λ based on **Holds**, and let **L** be the formula class generated by *Op*. We define any such set **L** to be an *SCL language* for the SCL lexicon λ, and we say that λ *underlies* **L**. *Trm* is said to be the set of *terms* of **L**. If λ and λ' are SCL lexicons with the same sets of constants and function symbols, and **L** and **L**' are SCL languages for λ and λ', respectively, then **L** and **L**' are said to be *equivalent*. If λ is a first-order lexicon, then a language for λ is said to be a *first-order* SCL language. In particular, on ths definition, every familiar first-order language turns out to be an instance of an SCL language whose underlying lexicon is traditional first-order (i.e., "TFO" — see the end of Section 1 above). We therefore call any such language a *TFO* language.

3. Interpretations

Let λ be an SCL lexicon. An *SCL interpretation* **I** for λ is a 4-tuple $\langle I, R, ext, V\rangle$ satisfying the following conditions. First, I and R are nonempty sets. Intuitively, I represents the set of *individuals* of **I**, and will serve as the range of the quantifiers and its members will serve as the denotations of terms. R is the set of relations[3] whose members serve as possible denotations of predicate constants. To allow for type-freedom, there is no requirement that I and R be disjoint; indeed any degree of overlap, from partial to complete, is allowed. Those relations that are also members of I are said to be *reified*. Intuitively, reified relations are relations that can also be thought of as individuals. Accordingly, they can also be the values of individual constants and individual variables.

R is itself the union of countable sets R_ω, R_1, R_2, R_3, ... All are possibly empty with the exception of R_2, which contains a distinguished element **Id**, intended to serve as the identity relation. Intuitively, R_ω is the set of variably polyadic relations, and each R_n the set of n-place relations. Accordingly, *ext* is a corresponding extension function from R into $Pow(I^\omega)$ subject to the constraint that, for any natural number $n > 0$, if $r \in R_n$, then $ext(r) \subseteq I^n$; in particular, $ext(\mathbf{Id}) = \{\langle a, a\rangle : a \in I\}$.

Intuitively, of course $ext(r)$ represents the extension of r. For elements r of R_ω, if $ext(r)$ is a total (extensional) function on I^ω, then we say that r is a *function* on I^ω. For $n+1$-place relations r, if $ext(r)$ is a total (extensional) function on I^n, then we also say that r is a *function on* I^n, or an *n-place function*.

Finally, V is a "denotation" function that assigns appropriate values to the constants and function symbols of **L**. Specifically,

- If κ is an individual constant, then $V(\kappa) \in I$;
- If π is a predicate constant, then $V(\pi) \in R_{arity(\kappa)}$.
- If α is a function symbol, then $V(\alpha)$ is a function on $I^{arity(\alpha)}$.

Note, importantly, that it is not required that I and R be disjoint. This is the semantic correlate of the type-freedom permitted (though not required) in SCL languages. Specifically, an SCL language **L** can allow a primitive term κ to do double duty as both a predicate constant

[3]It is possible to model of the members of R extensionally as sets, though this will in general require non-well-founded set theory, since a relation, qua individual, can be in its own extension.

and an individual constant. Consequently, the denotation function V in any interpretation **I** of **L** must by definition map κ, qua predicate constant, to an element of R; it must also map κ, qua individual constant, qua individual constant, to an element of I. Consequently, to satisfy these constraints, **I**, $V(\kappa)$ will have to be in both I and R, i.e., it will have to be both a relation and an individual. And this is just what the semantics allows. In a similar fashion, predicate constants can do double duty as function symbols.

A question might arise about interpretation in which only some members of R are members of I. In fact, it is likely that in the most common intended interpretations overlap will either be nonexistent or complete. However, there is a reasonably natural idea corresponding to partial overlap, namely, that some predicates indicate real properties of things and others are just convenient ways of categorizing things. For example, in an biological ontology, "is an arm" may not be thought of as a genuine property of anything, but only a convenient way of classifying things that play a certain functional role in a biological organism. By contrast "is a cell" might be thought of as indicating a genuine biological property of a thing that one might wish to include the genuine inventory of one's ontology. Partial overlap provides a natural way of preserving this distinction.

4. Denotations and Truth

Given the notion of an interpretation for a lexicon λ, we can now define what it is for a formula of an SCL language L based on λ to be *true* in an interpretation.

Some additional apparatus will be useful in defining truth for quantified formulas (i.e., formulas in the range of **EQ** and **UQ**). First, given an interpretation **I**, define a *variable assignment* for **I** to be a function that maps individual variables into I and sequence variables into I_ω. To define the semantics of quantification, what we need is the notion of a variable assignment v' that is exactly like a given assignment v except that it might not agree with v on what to assign to some finite set of individual variables. The idea is straightforward, but the presence of restricted quantifiers forces us to proceed with some care. Let $\mathbf{I} = \langle I, R, ext, V\rangle$ be an interpretation for **L**, and let v be a variable assignment for **I**. In our syntax, a quantifier can bind an entire sequence consisting of (individual) variables and variable/predicate pairs. So let $\chi_1, ..., \chi_n$ be such a sequence, and say that a variable assignment v' for **I** is a $[\chi_1, ..., \chi_n]$-*variant* of v iff

- if χ_i is a variable / predicate-constant pair $\langle \nu, \kappa\rangle$ and $V(\kappa)$ is a relation, then $v'(\nu)$ is in the extension of $V(\kappa)$; and
- $v'(\nu) = v(\nu)$, if ν is distinct from all the variables in the sequence $\chi_1, ..., \chi_n$ and all of the variables occurring in variable/constant pairs in the sequence.

So let **L** be an SCL language for a lexicon λ, where **App** generates the set *Trm* of terms of **L**, and let $\mathbf{I} = \langle I, R, ext, V\rangle$ be an interpretation for **L**. Given **I** and a variable assignment v, let V_v be $V \cup v$. Given **I** and a variable assignment v, the denotations of the function terms of **L** in **I** are completely determined by V_v. This can be expressed in terms of a unique extension $V_v^\#$ of V such that, for any term $\tau \in Trm$:

- If τ is an individual constant, then $V_v^\#(\tau) = V(\tau)$.
- If τ is a variable, then $V_v^\#(\tau) = v(\tau)$.
- If τ is a function term $\mathbf{App}(\alpha, \tau_1, ..., \tau_n)$, then:
  - If τ_n is a sequence variable and $v(\tau_n) = \langle e_1, ..., e_m\rangle$, then $V_v^\#(\tau) = V(\alpha)(V_v^\#(\tau_1), ..., V_v^\#(\tau_{n-1}), e_1, ..., e_m)$.
  - If τ_n is not a sequence variable, then $V_v^\#(\tau) = V^\#(\alpha)(V_v^\#(\tau_1), ..., V_v^\#(\tau_n))$;

Given V, we define satisfaction for the formulas of **L** by a variable assignment v for our interpretation **I** as follows. Let $\varphi \in \mathbf{L}$:

- If $\varphi = \mathbf{Holds}(\kappa, \tau_1, ..., \tau_n)$, then:
  - If τ_n is a sequence variable and $v(\tau_n) = \langle e_1, ..., e_m\rangle$, then v satisfies φ iff $V(\kappa) \in R_{arity(\kappa)}$ and $\langle V_v^\#(\tau_1), ..., V_v^\#(\tau_{n-1}), e_1, ..., e_m\rangle \in ext(V(\kappa))$.
  - If τ_n is not a sequence variable, then v satisfies φ iff $V(\kappa)$ is a relation and $\langle V_v^\#(\tau_1), ..., V_v^\#(\tau_n)\rangle \in ext(V(\kappa))$.
- If $\varphi = \mathbf{Id}(\tau, \tau')$, then v satisfies φ iff $V_v^\#(\tau) = V_v^\#(\tau')$.
- If $\varphi = \mathbf{Neg}(\psi)$, then v satisfies φ iff ψ is not true in **I**.
- If $\varphi = \mathbf{Disj}(\psi_1, ...\psi_n)$, then v satisfies φ iff v satisfies ψ_i for some i, $1 \leq i \leq n$.
- If $\varphi = \mathbf{Conj}(\psi_1, ...\psi_n)$, then v satisfies φ iff v satisfies ψ_i for each i, $1 \leq i \leq n$.
- If $\varphi = \mathbf{Cond}(\psi, \psi')$, then v satisfies φ iff v does not satisfy ψ or v satisfies ψ'.
- If $\varphi = \mathbf{Bi}(\psi, \psi')$, then φ v satisfies φ iff iff v either satisfies both ψ and ψ' or satisfies neither.
- If $\varphi = \mathbf{EQ}(\chi_1, ..., \chi_n, \psi)$, then v satisfies φ iff some $[\chi_1, ..., \chi_n]$-variant of v satisfies ψ.
- If $\varphi = \mathbf{UQ}(\chi_1, ..., \chi_n, \psi)$, then v satisfies φ iff every $[\chi_1, ..., \chi_n]$-variant of v satisfies ψ.

Finally, then, a formula φ is *true in* **I** iff every variable assignment for **I** satisfies φ.

Note that, on this semantics, free individual variables are implicitly universally quantified; that is, if φ is a formula containing a free individual variable ν, then φ is true in **I** iff $\mathbf{UQ}(\nu, \varphi)$ is true in **I**. We do not have a similar metatheorem for formulas with free sequence variables because sequence variables are not explicitly quantified. It should be clear, however, that the above definition of

truth treats free sequence variables as if they were universally quantified as well: a formula φ containing a free sequence variable σ will be true in an interpretation **I** iff every variable assignment v satisfies φ, and hence iff every $[\sigma]$-variant of every variable assignment satisfies φ.

5. SCL AND TRADITIONAL FOL

We conclude with an important observation about the relation between SCL and first order logic. Consider, the following sentence from an unconstrained SCL language **L**:

$$(\forall x)(Px \leftrightarrow \neg Qx) \wedge (\forall xy)x = y$$

Because **L** is unconstrained, there is no distinction between predicate constants and individual constants. Hence, all such terms denote individuals in the domain. Such languages are useful, recall, in contexts where properties and relations are themselves considered "first-class citizens" and hence are included the domain of individuals. By the first conjunct in the above sentence, the individuals p and q that 'P' and 'Q' denote individuals must be distinct, as they must differ in their extensions. By the second conjunct, however, there is exactly one individual, and hence p and q cannot be distinct. Therefore, the sentence is false in all interpretations of **L**.[4]

This might lead one to charge that SCL's model theory does violence to the logical properties of traditional first-order logic. But it does not. The logical properties of the sentence above change only with respect to SCL languages that incorporate features that extend traditional first-order languages. Considered as a sentence of a TFO language (and many others midway between TFO and unconstrained), the the sentence is satisfiable relative to SCL's model theory no less than it is in traditional "Tarskian" model theory. More generally, then: The logical properties of TFO languages — those SCL language with no sequence variables, no variably polyadic predicates, no type-freedom, and no variables in predicate position — are *identical* regardless of whether they are interpreted according to the usual Tarskian semantics or according to SCL semantics; a formula of such a language will be true in all SCL interpretation iff it is true in all Tarskian interpretations. (The proof of this is quite simple, as it is easy to transform one type of interpretation into the other in a way that preserves truth.) Moreover, if one is unhappy with the differences in logical properties that can arise in a less constrained SCL language, there is a simple translation function that maps such a language to a theory in TFO language that has exactly the same expressive power.[5]

SCL is thus in a very precise sense a "conservative" extension of traditional first-order logic; it encompasses traditional first-order logic in all its many guises, but allows as well for the definition of much more powerful and flexible comformant languages. SCL thereby provides elegant solutions to both the uniformity problem and the excess baggage problem.

REFERENCES

[1] The Common Logic Working Group, "Common Logic Standard," URL = `http://cl.tamu.edu`.

[2] KIF Working Group, "Knowledge Interchange Format: Draft proposed American National Standard (dpANS)," NCITS.T2/98-004, URL = `http://logic.stanford.edu/kif/dpans.html`.

[3] Chen, W., M. Kifer, and D. S. Warren, "HiLog: A Foundation for higher-order logic programming," *Journal of Logic Programming* **15**(3), February 1993, pp. 187–230.

[4] The Common Logic Working Group, "Abstract Syntax and Semantics for SCL," URL = `http://cl.tamu.edu/docs/scl/scl-latest.html`.

DEPARTMENT OF PHILOSOPHY, TEXAS A&M UNIVERSITY, COLLEGE STATION, TX 77843-4237
E-mail address: `cmenzel@tamu.edu`

IHMC, UNIVERSITY OF WEST FLORIDA, PENSACOLA, FL 32501
E-mail address: `phayes@ihmc.us`

[4] We thank Ian Horrocks for the example, who came up with it to illustrate his dissatisfaction with an earlier incarnation of SCL. In that incarnation, there was no distinction between predicate and individual constants in any SCL language, and hence the sentence above turned out to be logically false. This pointed out an admittedly disturbing disconnect between the logical properties of SCL sentences relative to SCL's model theory and their logical properties relative to traditional Tarskian model theory. Revisions since then have added flexibility to SCL that undermines this objection.

[5] Briefly, one introduces new predicates $Holds_n$ for all n and maps every atomic sentence '$P(t_1, \ldots, t_n)$' of the non-TFO language in which 'P' is serves as both an individual and predicate constant into the sentence '$Holds(P, t_1, \ldots, t_n)$'.