1 Introduction

Predicate construction is a guided search process: a space of predicates must be enumerated in a systematic way in order to find, according to some criterion, a suitable predicate for some purpose. Often the search space is large and heuristics are used to guide the search. Typically, predicate construction is studied and implemented in the context of first-order logic. In this paper, I outline the foundations of predicate construction in higher-order logic. The main contribution is a method of incremental construction of predicates on higher-order terms that represent individuals. As will be explained, the higher-order nature of the logic is exploited in several fundamental ways. First, the logic will provide in a simple way sets, multisets, and similar types, for representing individuals. Second, the logic will provide a convenient framework for constructing predicates by composing certain functions, some of which are themselves higher-order.

The primary application domain of predicate construction is machine learning and an account of the application of the techniques of this paper to machine learning can be found in [1] and [2]. Generally speaking, the application domain is the same as that of the field of inductive logic programming [8], but the methods used there are rather different, relying as they do on (usually, untyped) first-order logic, in fact, the Horn clause subset. For comparison, the first-order analogue of much of the material in this paper can be found in [9].

The setting is a typed, higher-order logic that provides a variety of important data types, including sets, multisets, and graphs for representing individuals. This logic also provides highly expressive hypothesis languages. Induced definitions in such hypothesis languages are comprehensible, thus providing insight into the nature of the application data. For reasons of space, it is not possible to give a detailed account of the logic here. Fortunately, the features that are relevant to the material of this paper can be given in the following three paragraphs. A detailed account of this logic, including the proofs of all the propositions in this paper, is contained in [7].

The logic is based on Church's simple theory of types [3] with several extensions. First, I assume there is given a set of type constructors $\mathfrak{T}$ of various arities. Included in $\mathfrak{T}$ are the constructors $\mathit{1}$ and Ω both of arity 0. The domain corresponding to $\mathit{1}$ is some canonical singleton set and the domain corresponding to Ω is the set containing just *True* and *False*, i.e. the boolean values. The *types* of the logic are built up from the set of type constructors and a set of parameters (that is, type variables), using the symbol $\rightarrow$ (for function types) and $\times$ (for product types). The set of all closed types is denoted by $\mathfrak{S}^c$. Standard types include *Nat* (the type of natural numbers), *Int* (the type of integers), and *Float* (the type of floating-point numbers). Note that the logic is polymorphic, an extension not considered by Church.

There is also a set $\mathfrak{C}$ of constants of various types. Included in $\mathfrak{C}$ are $\top$ (true) and $\bot$ (false). Two different kinds of constants, *data constructors* and *functions*, are distinguished. In a knowledge representation context, data constructors are used to represent individuals. In a programming language context, data constructors are used to construct data values. In contrast, functions are used to compute on data values; functions have definitions while data constructors do not. In the semantics for the logic, the data constructors are used to construct models.

The *terms* of the logic are the terms of the typed λ-calculus, which are formed in the usual way by abstraction, tupling, and application from constants in $\mathfrak{C}$ and a set of variables. $\mathfrak{L}$ denotes the set of all terms (obtained from a particular alphabet). A term of type Ω is called a *formula*. A function whose codomain type is Ω is called a *predicate*. In the logic, one can introduce the usual connectives and quantifiers as functions of appropriate types. Thus the connectives conjunction, $\wedge$, and disjunction, $\vee$, are functions of type $\Omega \to \Omega \to \Omega$ and the (generalised) existential quantifier, Σ, and universal quantifier, Π, have type $(a \to \Omega) \to \Omega$. (The $\to$ is right associative.) Terms of the form $\Sigma(\lambda x.t)$ are written as $\exists x.t$ and terms of the form $\Pi(\lambda x.t)$ are written as $\forall x.t$. In addition, if t is of type Ω, the abstraction $\lambda x.t$ is written $\{x \mid t\}$ to emphasise its intended meaning as a set. A set abstraction of the form $\{x \mid \mathit{if}\ x = t_1\ \mathit{then}\ \top\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ \top\ \mathit{else}\ \bot\}$ is abbreviated to $\{t_1, \ldots, t_n\}$, where x does not occur in any t_i. There is also a tuple-forming notation $(\ldots)$. Thus, if $t_1, \ldots, t_n$ are terms of type $\tau_1, \ldots, \tau_n$, respectively, then $(t_1, \ldots, t_n)$ is a term of type $\tau_1 \times \cdots \times \tau_n$.

In the next section, the terms used to represent individuals are defined. In the third section, the procedure for constructing predicates on individuals is given. The fourth section gives some indication of the applications of the approach to machine learning.

2 Representation of Individuals

I identify a class of terms, called *basic* terms, suitable for representing individuals in diverse applications. For example, this class is suitable for machine learning applications [1, 2]. From a (higher-order) programming language perspective, basic terms are data values [6]. The most interesting aspect of the class of basic terms is that it includes certain abstractions and therefore is much wider than is normally considered for knowledge representation. These abstractions allow one to model sets, multisets, and similar data types, in an elegant way. Of course, there are other ways of introducing (extensional) sets, multisets, and so on, without using abstractions. For example, one can define abstract data types or one can introduce data constructors with special equality theories. The primary advantage of the approach adopted here is that one can define these abstractions *intensionally*.

2.1 Normal Terms

The definition of basic terms is given in several stages: first I define normal terms, then define an equivalence relation on normal terms, and finally define basic terms as distinguished representatives of equivalence classes. Before getting down to the first step of giving the definition of normal terms, some motivation will be helpful. How should a (finite) set or multiset be represented? First, advantage is taken of the higher-order nature of the logic to identify sets and their characteristic functions, that is, sets are viewed as predicates. With this approach, an obvious representation of sets uses the connectives, so that $\lambda x.(x = 1) \vee (x = 2)$ is the representation of the set $\{1, 2\}$. This was the kind of representation used in [6] and it works well for sets. But the connectives are, of course, not available for multisets, so something more general is needed. An alternative representation for the set $\{1, 2\}$ is the term

$$\lambda x.\mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \bot,$$

and this idea generalises to multisets and similar abstractions. For example,

$$\lambda x.\mathit{if}\ x = A\ \mathit{then}\ 42\ \mathit{else}\ \mathit{if}\ x = B\ \mathit{then}\ 21\ \mathit{else}\ 0$$

is the multiset with 42 occurrences of A and 21 occurrences of B (and nothing else). Thus I adopt abstractions of the form

$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$$

to represent (extensional) sets, multisets, and so on.

However, before giving the definition of a normal term, some attention has to be paid to the term s_0 in previous expression. The reason is that s_0 in this abstraction is usually a very specific term. For example, for finite sets, s_0 is $\bot$ and for finite multisets, s_0 is 0. For this reason, the concept of a default term is now introduced. The intuitive idea is that, for each closed type, there is a (unique) default term such that each abstraction having that type as codomain takes the default term as its value for all but a finite number of points in the domain, that is, s_0 is the default value. The choice of default term depends on the particular application but, since sets and multisets are so useful, one would expect the set of default terms to include $\bot$ and 0. However, there could also be other types for which a default term is needed. For each type constructor T, I assume there is chosen a unique *default data constructor* C such that C has signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$. For example, for Ω, the default data constructor could be $\bot$, for *Int*, the default data constructor could be 0, and for *List*, the default data constructor could be [].

Definition. The set of *default terms*, $\mathfrak{D}$, is defined inductively as follows.

1. If C is a default data constructor having signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$ and $t_1, \ldots, t_n \in \mathfrak{D}$ $(n \geq 0)$ such that $C\ t_1 \ldots t_n \in \mathfrak{L}$, then $C\ t_1 \ldots t_n \in \mathfrak{D}$.
2. If $t \in \mathfrak{D}$ and x is a variable, then $\lambda x.t \in \mathfrak{D}$.
3. If $t_1, \ldots, t_n \in \mathfrak{D}$ $(n \geq 0)$ and $(t_1, \ldots, t_n) \in \mathfrak{L}$, then $(t_1, \ldots, t_n) \in \mathfrak{D}$.

There may not be a default term for some closed types. However, if it exists, one can show the default term for each closed type is unique.

Proposition 2.1. *For each $\alpha \in \mathfrak{S}^c$, there exists at most one default term having type more general than α.*

Now normal terms can be defined. In the following, $\lambda x.s_0$ is regarded as the special case of

$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$$

when $n = 0$.

Definition. The set of *normal terms*, $\mathfrak{N}$, is defined inductively as follows.

1. If C is a data constructor having signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$ and $t_1, \ldots, t_n \in \mathfrak{N}$ $(n \geq 0)$ such that $C\ t_1 \ldots t_n \in \mathfrak{L}$, then $C\ t_1 \ldots t_n \in \mathfrak{N}$.
2. If $t_1, \ldots, t_n \in \mathfrak{N}$, $s_1, \ldots, s_n \in \mathfrak{N}$ $(n \geq 0)$, $s_0 \in \mathfrak{D}$ and

$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{L},$$

then

$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{N}.$$

3. If $t_1, \ldots, t_n \in \mathfrak{N}$ ($n \geq 0$) and $(t_1, \ldots, t_n) \in \mathfrak{L}$, then $(t_1, \ldots, t_n) \in \mathfrak{N}$.

Part 1 of the definition of the set of normal terms states, in particular, that individual natural numbers, integers, and so on, are normal terms. Also a term formed by applying a constructor to (all of) its arguments, each of which is a normal term, is a normal term. As an example of this, consider the following declarations of the data constructors *Circle* and *Rectangle*.

$$\begin{aligned}&\mathit{Circle} : \mathit{Float} \to \mathit{Shape}\\&\mathit{Rectangle} : \mathit{Float} \to \mathit{Float} \to \mathit{Shape}.\end{aligned}$$

Then (Circle 7.5) and ($\mathit{Rectangle}$ 42.0 21.3) are normal terms of type *Shape*. However, ($\mathit{Rectangle}$ 42.0) is not a normal term as not all arguments to *Rectangle* are given. Normal terms coming from Part 1 of the definition are called *normal structures* and always have a type of the form $T \alpha_1 \ldots \alpha_n$.

The abstractions formed in Part 2 of the definition are "almost constant" abstractions since they take the default term s_0 as value for all except a finite number of points in the domain. They are called *normal abstractions* and always have a type of the form $\beta \to \gamma$. This class of abstractions includes useful data types such as (finite) sets and multisets (assuming $\perp$ and 0 are default terms). More generally, normal abstractions can be regarded as lookup tables, with s_0 as the value for items not in the table.

Part 3 of the definition of normal terms just states that one can form a tuple from normal terms and obtain a normal term. These terms are called *normal tuples* and always have a type of the form $\alpha_1 \times \cdots \times \alpha_n$.

Proposition 2.2. $\mathfrak{D} \subseteq \mathfrak{N}$.

It will be convenient to gather together all normal terms that have a type more general than some specific closed type.

Definition. For each $\alpha \in \mathfrak{S}^c$, define $\mathfrak{N}_\alpha = \{t \in \mathfrak{N} \mid t \text{ has type more general than } \alpha\}$.

The intuitive meaning of $\mathfrak{N}_\alpha$ is that it is the set of terms representing individuals of type α. Note that $\mathfrak{N} = \bigcup_{\alpha \in \mathfrak{S}^c} \mathfrak{N}_\alpha$. However, the $\mathfrak{N}_\alpha$ are not (usually) disjoint. For example, if the alphabet includes *List*, then $[] \in \mathfrak{N}_{\mathit{List}\ \alpha}$, for each closed type α.

2.2 An Equivalence Relation on Normal Terms

Several syntactically distinct terms in $\mathfrak{N}$ can represent the same individual. For example,

$$\begin{aligned}&\lambda x.\mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \perp,\\&\lambda x.\mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \perp,\end{aligned}$$

and

$$\lambda x.\mathit{if}\ x = 3\ \mathit{then}\ \perp\ \mathit{else}\ \mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \perp$$

all represent the set $\{1, 2\}$. To reflect this, a relation $\equiv$ is defined on $\mathfrak{N}$.

Definition. The binary relation $\equiv$ on $\mathfrak{N}$ is defined inductively as follows. Let $s, t \in \mathfrak{N}$. Then $s \equiv t$ if there exists $\alpha \in \mathfrak{S}^c$ such that $s, t \in \mathfrak{N}_\alpha$ and one of the following conditions holds.

1. $\alpha = T\ \alpha_1 \ldots \alpha_k$, for some $T, \alpha_1, \ldots, \alpha_k$, and s is $C\ s_1 \ldots s_n$, t is $C\ t_1 \ldots t_n$ and $s_i \equiv t_i$, for $i = 1, \ldots, n$.
2. $\alpha = \beta \to \gamma$, for some β, γ, and
s is $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$,
t is $\lambda y.\ \mathit{if}\ y = u_1\ \mathit{then}\ v_1\ \mathit{else}\ \ldots\ \mathit{if}\ y = u_m\ \mathit{then}\ v_m\ \mathit{else}\ s_0$
and, $\forall r \in \mathfrak{N}_\beta$,
$(\exists i, j.\ r \equiv t_i\ \wedge\ r \not\equiv t_k (\forall k < i)\ \wedge\ r \equiv u_j\ \wedge\ r \not\equiv u_m (\forall m < j)\ \wedge$
$s_i \equiv v_j) \vee$
$(\exists i.\ r \equiv t_i\ \wedge\ r \not\equiv t_k (\forall k < i)\ \wedge\ r \not\equiv u_j (\forall j)\ \wedge\ s_i \equiv s_0) \vee$
$(\exists j.\ r \not\equiv t_i (\forall i)\ \wedge\ r \equiv u_j\ \wedge\ r \not\equiv u_m (\forall m < j)\ \wedge\ s_0 \equiv v_j) \vee$
$(r \not\equiv t_i (\forall i)\ \wedge\ r \not\equiv u_j (\forall j))$.
3. $\alpha = \alpha_1 \times \cdots \times \alpha_n$, for some $\alpha_1, \ldots, \alpha_n$, and s is $(s_1, \ldots, s_n)$, t is $(t_1, \ldots, t_n)$ and $s_i \equiv t_i$, for $i = 1, \ldots, n$.

Proposition 2.3. *For each $\alpha \in \mathfrak{S}^c$, $\equiv|_{\mathfrak{N}_\alpha}$ is an equivalence relation on $\mathfrak{N}_\alpha$.*

2.3 A Total Order on Normal Terms

The equivalence relation $\equiv$ was introduced because several syntactically distinct terms in $\mathfrak{N}$ can represent the same individual. Rather than deal with all the normal terms in an equivalence class in some $\mathfrak{N}_\alpha$, it is preferable to deal with a single representative from the equivalence class. For this purpose, a (strict) total order on normal terms is introduced.

Recall that a (strict) partial order on a set A is a binary relation $<$ on A such that, for each $a, b, c \in A$, $a \not< a$ (irreflexivity), $a < b$ implies $b \not< a$ (asymmetry), and $a < b$ and $b < c$ implies $a < c$ (transitivity). In addition, a (strict) partial order is a (strict) total order if, for each $a, b \in A$, exactly one of $a = b$ or $a < b$ or $b < a$ holds.

If $<$ is a (strict) total order on a set A, then $<$ can be lifted to (strict) total order, also denoted by $<$, on the set of sequences of elements in A by $a_1 \ldots a_n < b_1 \ldots b_m$ if either
(i) $a_1 = b_1, \ldots, a_n = b_n$ and $n < m$, or
(ii) there exists j such that $1 \leq j \leq n$, $a_1 = b_1, \ldots, a_{j-1} = b_{j-1}$ and $a_j < b_j$.
The order $<$ on the sequences is called the *induced lexicographic* ordering.

In the definition of the binary relation $<$ below, it is assumed that, for each $T \in \mathfrak{T}$, there is defined a (strict) total order $\prec_T$ on the set of all data constructors associated with the type constructor T. For standard types, such as *Int* and *Float*, the usual order provides an appropriate total order. To simplify the statement of the definition, the concept of the trace of an abstraction will be useful.

Definition. Suppose that s is a normal abstraction

$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0.$$

Then the *trace* of s, $\mathit{trace}(s)$, is the sequence $t_1\, s_1\, t_2\, s_2 \ldots t_n\, s_n$. (For the normal abstraction $\lambda x.s_0$, the trace is the empty sequence ε.)

Definition. The binary relation $<$ on $\mathfrak{N}$ is defined inductively as follows. Let $s, t \in \mathfrak{N}$. Then $s < t$ if there exists $\alpha \in \mathfrak{S}^c$ such that $s, t \in \mathfrak{N}_\alpha$ and one of the following conditions holds.

1. $\alpha = T\ \alpha_1 \ldots \alpha_k$, for some $T, \alpha_1, \ldots, \alpha_k$, and s is $C\ s_1 \ldots s_n$, t is $D\ t_1 \ldots t_m$ and either $C \prec_T D$ or $C = D$ and there exists j such that $1 \leq j \leq n$, $s_1 = t_1, \ldots, s_{j-1} = t_{j-1}$ and $s_j < t_j$.

2. $\alpha = \beta \to \gamma$, for some β, γ, and $\mathit{trace}(s) < \mathit{trace}(t)$, where $<$ is the induced lexicographic ordering.
3. $\alpha = \alpha_1 \times \cdots \times \alpha_n$, for some $\alpha_1, \ldots, \alpha_n$, and s is $(s_1, \ldots, s_n)$, t is $(t_1, \ldots, t_n)$ and there exists j such that $1 \leq j \leq n$, $s_1 = t_1, \ldots, s_{j-1} = t_{j-1}$ and $s_j < t_j$.

Proposition 2.4. *For each $\alpha \in \mathfrak{S}^c$, $<|_{\mathfrak{N}_\alpha}$ is a (strict) total order on $\mathfrak{N}_\alpha$.*

2.4 Basic Terms

Finally, the definition of the key concept of a basic term can be given.

Definition. The set of *basic terms*, $\mathfrak{B}$, is defined inductively as follows.

1. If C is a data constructor having signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$ and $t_1, \ldots, t_n \in \mathfrak{B}$ $(n \geq 0)$ such that $C\ t_1 \ldots t_n \in \mathfrak{L}$, then $C\ t_1 \ldots t_n \in \mathfrak{B}$.
2. If $t_1, \ldots, t_n \in \mathfrak{B}$, $s_1, \ldots, s_n \in \mathfrak{B}$, $t_1 < \ldots < t_n$, $s_i \notin \mathfrak{D}$, for $1 \leq i \leq n$ $(n \geq 0)$, $s_0 \in \mathfrak{D}$ and
$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{L},$$
then
$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{B}.$$
3. If $t_1, \ldots, t_n \in \mathfrak{B}$ $(n \geq 0)$ and $(t_1, \ldots, t_n) \in \mathfrak{L}$, then $(t_1, \ldots, t_n) \in \mathfrak{B}$.

The basic terms from Part 1 of the definition are called *basic structures*, those from Part 2 are called *basic abstractions*, and those from Part 3 are called *basic tuples*.

Proposition 2.5. $\mathfrak{D} \subseteq \mathfrak{B} \subseteq \mathfrak{N}$.

As for normal terms, the basic terms of a particular type can be gathered together.

Definition. For each $\alpha \in \mathfrak{S}^c$, define $\mathfrak{B}_\alpha = \{t \in \mathfrak{B} \mid t \text{ has type more general than } \alpha\}$.

Proposition 2.6. *For each $\alpha \in \mathfrak{S}^c$, $\mathfrak{B}_\alpha \subseteq \mathfrak{N}_\alpha$.*

The next result shows that, for basic terms, the equivalence relation $\equiv$ reduces to the identity relation.

Proposition 2.7. *Let $s, t \in \mathfrak{B}$. Then $s \equiv t$ iff $s = t$.*

The next proposition justifies restricting attention to basic terms for knowledge representation purposes.

Proposition 2.8. *If $s \in \mathfrak{N}_\alpha$, for some $\alpha \in \mathfrak{S}^c$, then there is a unique $t \in \mathfrak{B}_\alpha$ such that $s \equiv t$.*

Definition. Let $s \in \mathfrak{N}_\alpha$, for some $\alpha \in \mathfrak{S}^c$. The unique $t \in \mathfrak{B}_\alpha$ such that $s \equiv t$ is called the *basic form* of s.

Here is an example to illustrate how the basic form can be computed.

Example. Let s be the normal term

$$\lambda x.\mathit{if}\ x = 3\ \mathit{then}\ \bot\ \mathit{else}\ \mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}$$
$$\mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 3\ \mathit{then}\ \top\ \mathit{else}\ \bot.$$

Assume that the total order on the integers is the usual order. In the first step, each of $1, 2, 3$, and $\top$ and $\bot$ are replaced by their basic form. Since each of these is already a basic term, this step has no effect. Second, the component of the if-then-else containing the duplicated occurrence of $x = 3$ is dropped to obtain

$$\lambda x.\mathit{if}\ x = 3\ \mathit{then}\ \bot\ \mathit{else}\ \mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \bot.$$

Third, the component containing the occurrence $x = 3$ is dropped since the corresponding value is $\bot$ to obtain

$$\lambda x.\mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \bot.$$

Finally, the sequence 2 1 is ordered according to the total order and the components of the if-then-else are reordered accordingly to obtain

$$\lambda x.\mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \bot,$$

which is the basic form of s.

3 Predicate Construction

Now I turn to the main topic of predicate construction. Essentially, there are two parts to the process of predicate construction. The first part involves the systematic enumeration of a suitable space of predicates. The second part involves the use of heuristics to efficiently search this space in order to find, according to some criterion, a suitable predicate. Here I concentrate on the first part of predicate construction. The general idea is to construct predicates by composing certain functions from a possibly large set of more basic functions.

3.1 Transformations

Composition is handled by the (reverse) composition function

$$\cdot : (\rho \to \sigma) \to (\sigma \to \tau) \to (\rho \to \tau)$$

defined by

$$((f \cdot g)\ x) = (g\ (f\ x)).$$

Predicates are built up by composing transformations, which are defined as follows.

Definition. A *transformation* f is a function having a signature of the form

$$f : (\rho_1 \to \Omega) \to \cdots \to (\rho_k \to \Omega) \to \mu \to \sigma,$$

where any parameters in $\rho_1, \ldots, \rho_k$ and σ appear in μ, and $k \geq 0$. The type μ is distinguished and is called the *source* of the transformation, while the type σ is called the *target* of the transformation. The number k is called the *rank* of the transformation.

The intuitive idea behind this definition is that, given predicates $p_i : \rho_i \to \Omega$, for $i = 1, \ldots, k$, $f\ p_1 \ldots p_k$ is a function that takes individuals of type μ to individuals of type σ. By composing (generally) several such functions, the last of which is a predicate, a predicate on individuals of the desired type is obtained.

Before giving some examples, a word on notation. Sets and predicates have been identified, so that a set has type $\mu \to \Omega$, for some type μ. However, in practice, even though sets and predicates have been identified, for a term of type $\mu \to \Omega$, it is sometimes convenient to make an informal distinction depending upon whether one is thinking of it as a 'set of elements' or as a 'condition'. In the former case, the synonym $\{\mu\}$ for $\mu \to \Omega$ is used to indicate this. To emphasise, there is no mathematical distinction between $\{\mu\}$ and $\mu \to \Omega$, but there is a change in the intuitive role the corresponding term is understood to be playing. Similarly, the notation '$x \in t$' is used if one is thinking of t as a 'set of elements' and the notation '$(t\ x)$' is used if one is thinking of t as a 'condition'. This distinction will be very convenient in the following discussion.

Example. Consider the transformation

$$domCard : (\mu \to \Omega) \to \{\mu\} \to Nat$$

defined by

$$domCard\ b\ t = card\ \{x \mid (b\ x) \wedge x \in t\},$$

where *card* computes the cardinality of a (finite) set. Given a predicate b on the type μ and a unary predicate on Nat such as (> 42), which returns $\top$ iff its argument is strictly greater than 42, one can construct a predicate $(domCard\ b) \cdot (> 42)$ on sets of type $\{\mu\}$ which selects the subset of elements that satisfy the predicate b and then checks that the cardinality of this subset is greater than 42.

Example. Consider the transformation

$$setExists_1 : (\mu \to \Omega) \to \{\mu\} \to \Omega$$

defined by

$$setExists_1\ b\ t = \exists x.(b\ x) \wedge (x \in t).$$

The predicate $(setExists_1\ b)$ checks whether a set has an element x that satisfies b. More generally, for $n \geq 1$, one can define

$$setExists_n : (\mu \to \Omega) \to \cdots \to (\mu \to \Omega) \to \{\mu\} \to \Omega$$

by

$$\begin{aligned} setExists_n\ p_1 \ldots p_n\ t = \exists x_1. \cdots \exists x_n.\ &(p_1\ x_1) \wedge \cdots \wedge (p_n\ x_n) \wedge \\ &(x_1 \in t) \wedge \cdots \wedge (x_n \in t) \wedge (x_1 \neq x_2) \wedge \cdots \wedge (x_{n-1} \neq x_n). \end{aligned}$$

Note the overlap between $(domCard\ b) \cdot (> 0)$ and $(setExists_1\ b)$. Typically, $setExists_n$ is used for small values of n, say, 1, 2 or 3, while *domCard* is used in conjunction with $(> n)$ for larger values of n.

Example. Consider the transformation

$$setAll : (\mu \rightarrow \Omega) \rightarrow \{\mu\} \rightarrow \Omega$$

defined by

$$setAll\ b\ t = \forall x.(x \in t \longrightarrow (b\ x)).$$

The predicate ($setAll\ b$) checks whether all elements in a set satisfy the predicate b.

Example. The transformation

$$\wedge_n : (\mu \rightarrow \Omega) \rightarrow \cdots \rightarrow (\mu \rightarrow \Omega) \rightarrow \mu \rightarrow \Omega$$

defined by

$$\wedge_n\ p_1 \dots p_n = \lambda x.((p_1\ x) \wedge \cdots \wedge (p_n\ x)),$$

where $n \geq 2$, provides a "conjunction" with n conjuncts. Transformations analogous to the other connectives can be defined similarly.

Example. Following the style of the previous example, the transformation

$$\sim\, : (\mu \rightarrow \Omega) \rightarrow \mu \rightarrow \Omega$$

defined by

$$\sim\ p = \lambda x.\neg(p\ x),$$

provides negation. But negation can also be introduced directly by the transformation

$$\neg : \Omega \rightarrow \Omega,$$

which is the usual negation connective. Then the transformation ($\sim\ p$) can be written equivalently as $p \cdot \neg$. (In practice, just one of these methods of introducing negation would be used.)

Example. Each projection

$$proj_i : \mu_1 \times \cdots \times \mu_n \rightarrow \mu_i$$

defined by

$$proj_i\ (t_1, \dots, t_n) = t_i,$$

for $i = 1, \dots, n$, is a transformation of rank 0.

Example. For each type μ, there are two fundamental transformations $top : \mu \rightarrow \Omega$ and $bottom : \mu \rightarrow \Omega$ defined by $top\ x = \top$ and $bottom\ x = \bot$, for each x. Each of *top* and *bottom* is a constant predicate, with *top* being the weakest predicate on the type μ and *bottom* being the strongest.

Example. Let μ be a type and suppose that $A : \mu$, $B : \mu$, and $C : \mu$ are constants. Then, corresponding to A, one can define a transformation

$$(= A) : \mu \rightarrow \Omega$$

by

$$((= A)\ x) = \top \ \text{ iff } \ x = A,$$

with analogous definitions for $(= B)$ and $(= C)$. Similarly, one can define the transformation

$$(\neq A) : \mu \to \Omega$$

by

$$((\neq A)\ x) = \top \ \text{ iff } \ x \neq A.$$

Example. Consider a type such as *Int* that has various order relations defined on it. Then, for any integer N, one can define the transformation

$$(< N) : \mathit{Int} \to \Omega$$

by

$$((< N)\ m) = \top \ \text{ iff } \ m < N.$$

In a similar way, one can define the transformations $(> N)$, $(\geq N)$, and $(\leq N)$.

Example. The if-then-else transformation

$$\mathit{ite} : (\mu \to \Omega) \to (\mu \to \Omega) \to (\mu \to \Omega) \to \mu \to \Omega$$

is defined by

$$\mathit{ite}\ p\ q\ r = \lambda x.\mathit{if}\ (p\ x)\ \mathit{then}\ (q\ x)\ \mathit{else}\ (r\ x).$$

As an illustration, one could define the predicate

$$\mathit{ite}\ ({\sim}\ \mathit{null})\ (\mathit{head} \cdot (= 42))\ \mathit{bottom},$$

where *null* checks whether a list is empty or not and *head* extracts the head of a non-empty list. This predicate returns $\top$ iff its argument is a non-empty list with 42 as its first item.

Example. Suppose that $\mathit{Graph}\ \nu\ \varepsilon$ is the type of (undirected) graphs whose vertices have type $\mathit{Vertex}\ \nu\ \varepsilon$ and edges have type $\mathit{Edge}\ \nu\ \varepsilon$, where ν is the type of the information at a vertex and ε is the type of information (for example, the weight) on an edge. Consider the following transformations on graphs.

$$\begin{aligned}
&\mathit{vertices} : \mathit{Graph}\ \nu\ \varepsilon \to \{\mathit{Vertex}\ \nu\ \varepsilon\}\\
&\mathit{edges} : \mathit{Graph}\ \nu\ \varepsilon \to \{\mathit{Edge}\ \nu\ \varepsilon\}\\
&\mathit{vertex} : \mathit{Vertex}\ \nu\ \varepsilon \to\ \nu\\
&\mathit{edge} : \mathit{Edge}\ \nu\ \varepsilon \to\ \varepsilon\\
&\mathit{connects} : \mathit{Edge}\ \nu\ \varepsilon \to (\mathit{Vertex}\ \nu\ \varepsilon \to \mathit{Nat})\\
&\mathit{edgesAtVertex} : \mathit{Vertex}\ \nu\ \varepsilon \to \{\mathit{Edge}\ \nu\ \varepsilon\}\\
&(\mathit{subgraphs}\ N) : \mathit{Graph}\ \nu\ \varepsilon \to \{\mathit{Graph}\ \nu\ \varepsilon\}.
\end{aligned}$$

Here *vertices* returns the set of vertices of a graph, *edges* returns the set of edges of a graph, *vertex* returns the information at a vertex, *edge* returns the information on an edge, *connects* returns the unordered pair of vertices joined by an edge, *edgesAtVertex* returns the set of edges ending at a vertex, and (*subgraphs* N) returns the set of (connected) subgraphs containing N vertices of a graph.

With these transformations, the predicate

$$\mathit{vertices} \cdot (\mathit{domCard}\ (\mathit{vertex} \cdot (\wedge_2\ b\ c))) \cdot (> 7)$$

on individuals of type *Graph* ν ε can be constructed, where b and c are predicates on individuals of type ν. This predicate is true for a graph iff the cardinality of the subset of vertices in the graph whose information satisfies the predicates b and c is greater than 7.

One can also construct the predicate

$$(\mathit{subgraphs}\ 3) \cdot (\mathit{domCard}\ (\mathit{edges} \cdot (\mathit{setExists}_2 \\ (\mathit{edge} \cdot b)\ (\mathit{edge} \cdot c)))) \cdot (> 3),$$

where b and c are predicates on the type ε. This predicate is true for a graph iff the cardinality of the subset of (connected) subgraphs in the graph that have three vertices and two (distinct) edges, one of which has information satisfying b and one satisfying c, is greater than 3.

Example. Consider again the type *Shape* and the data constructors

$$\mathit{Circle} : \mathit{Float} \rightarrow \mathit{Shape}$$
$$\mathit{Rectangle} : \mathit{Float} \rightarrow \mathit{Float} \rightarrow \mathit{Shape}.$$

The function *isCircle* : $(\mathit{Float} \rightarrow \Omega) \rightarrow \mathit{Shape} \rightarrow \Omega$ defined by

$$\mathit{isCircle}\ b\ t = \exists x.(t = \mathit{Circle}\ x) \wedge (b\ x)$$

is a transformation. Similarly, the function *isRectangle* : $(\mathit{Float} \rightarrow \Omega) \rightarrow (\mathit{Float} \rightarrow \Omega) \rightarrow \mathit{Shape} \rightarrow \Omega$ defined by

$$\mathit{isRectangle}\ b\ c\ t = \exists x.\exists y.(t = \mathit{Rectangle}\ x\ y) \wedge (b\ x) \wedge (c\ y)$$

is a transformation. For predicates b and c, (*isRectangle* b c) is a predicate on geometrical shapes which returns $\top$ iff the shape is a rectangle whose length satisfies b and whose breadth satisfies c.

3.2 Standard Predicates

Next the definition of the class of predicates formed by composing transformations is presented. First, a larger class of predicates, the standard predicates, is defined and then the desired subclass, the regular predicates, which has much less redundancy, is defined. In the following definition, it is assumed that some (possibly infinite) class of transformations is given and all transformations considered are taken from this class. A standard predicate is defined by induction on the number of transformations it contains as follows.

Definition. A *standard predicate* is a term of the form

$$(f_1\ b_{1,1} \ldots b_{1,k_1}) \cdot \ldots \cdot (f_n\ b_{n,1} \ldots b_{n,k_n}),$$

where f_i is a transformation of rank k_i ($i = 1, \ldots, n$), the target of f_n is Ω, b_{i,j_i} is a standard predicate ($i = 1, \ldots, n$, $j_i = 1, \ldots, k_i$), $k_i \geq 0$ ($i = 1, \ldots, n$) and $n \geq 1$.

The set of all standard predicates is denoted by $\mathbf{S}$.

Definition. For each $\alpha \in \mathfrak{S}^c$, define $\mathbf{S}_\alpha = \{p \in \mathbf{S} \mid p$ has type $\mu \to \Omega$ and μ is more general than $\alpha\}$.

The intuitive meaning of $\mathbf{S}_\alpha$ is that it is the set of all predicates of a particular form given by the transformations on individuals of type α.

The class of standard predicates just defined contains some redundancy in that there are syntactically distinct predicates that are semantically equivalent, in the sense of the following definition. Let $\mathcal{B}$ denote the theory consisting of the definitions of the transformations (and associated functions). This theory is usually called the *background theory*.

Definition. Let $p, q \in \mathbf{S}$. Then p and q are *semantically equivalent* if the types of p and q are the same (up to variants) and $p = q$ is a logical consequence of $\mathcal{B}$.

Semantic equivalence is an equivalence relation.

Example. Without spelling out the background theory $\mathcal{B}$, one would expect that the standard predicates $(\wedge_2\ p\ q)$ and $(\wedge_2\ q\ p)$ are semantically equivalent, where p and q are standard predicates. Similarly, $(\mathit{setExists}_2\ p\ q)$ and $(\mathit{setExists}_2\ q\ p)$ are semantically equivalent. Less obviously, $(\mathit{domCard}\ p) \cdot (> 0)$ and $(\mathit{setExists}_1\ p)$ are semantically equivalent.

It is important to remove as much of this redundancy in standard predicates as possible. Ideally, one would like to be able to determine just *one* representative from each class of equivalent predicates. However, determining the semantically equivalent predicates is undecidable, so one usually settles for some easily checked syntactic conditions which reveal equivalence of predicates. Thus these syntactic conditions are sufficient, but not necessary, for equivalence. These considerations motivate the next definition.

Definition. A transformation f is *symmetric* if it has a signature of the form

$$f : (\rho \to \Omega) \to \cdots \to (\rho \to \Omega) \to \mu \to \sigma,$$

and $f\ p_1 \ldots p_k$ and $f\ p_{i_1} \ldots p_{i_k}$ are semantically equivalent, for all standard predicates $p_j : \rho \to \Omega$, for $j = 1, \ldots, k$ and permutations i of $\{1, \ldots, k\}$, where k is the rank of f.

Clearly, every transformation of rank k, where $k \leq 1$, is (trivially) symmetric. Furthermore, the transformations $\mathit{setExists}_n$ and $\wedge_n$ are symmetric. However, *isRectangle* is not symmetric since it distinguishes the length argument from the breadth argument.

Since any permutation of the predicate arguments of a symmetric transformation produces a semantically equivalent predicate, it is advisable to choose one particular order of arguments and ignore the others. For this purpose, a total order on standard predicates is defined and then arguments for symmetric transformations are chosen in increasing order according to this total order. To define the total order on predicates, one must start with a total order on transformations. Therefore, it is supposed that, for each type μ, the transformations having source μ are ordered according to some (arbitrary) strict total order $<$.

In preparation for the definition of $\prec$, a structural result about standard predicates is needed.

Proposition 3.1. *Let $p, q \in \mathbf{S}$, where p is $(f_1\ b_{1,1} \ldots b_{1,k_1}) \cdot \ldots \cdot (f_n\ b_{n,1} \ldots b_{n,k_n})$ and q is $(g_1\ c_{1,1} \ldots c_{1,s_1}) \cdot \ldots \cdot (g_r\ c_{r,1} \ldots c_{r,s_r})$. Suppose that p and q are (syntactically) distinct. Then exactly one of the following alternatives holds.*

1. *One of p or q is a (strict) prefix of the other.*
2. *There exists i such that the transformation f_i in p and the transformation g_i in q are distinct, and p and q agree to the left of f_i and g_i.*
3. *There exist i and j_i such that b_{i,j_i} in p and c_{i,j_i} in q are distinct, and p and q agree to the left of b_{i,j_i} and c_{i,j_i}.*

Example. As an illustration of the first alternative in the preceding proposition, consider p and $p \cdot \neg$, for some standard predicate p. For the third alternative, consider $(\wedge_2\ p\ q)$ and $(\wedge_2\ (p \cdot \neg)\ q)$, for some standard predicates p and q.

The following definition of the relation $p \prec q$ uses induction on the maximum of the number of transformations in p and the number in q.

Definition. The binary relation $\prec$ on $\mathbf{S}$ is defined inductively as follows. Let $p, q \in \mathbf{S}$, where p is $(f_1\ b_{1,1} \ldots b_{1,k_1}) \cdot \ldots \cdot (f_n\ b_{n,1} \ldots b_{n,k_n})$ and q is $(g_1\ c_{1,1} \ldots c_{1,s_1}) \cdot \ldots \cdot (g_r\ c_{r,1} \ldots c_{r,s_r})$. Then $p \prec q$ if there exists $\alpha \in \mathfrak{S}^c$ such that $p, q \in \mathbf{S}_\alpha$ and one of the following holds.

1. p is a (strict) prefix of q.
2. There exists i such that $f_i < g_i$, and p and q agree to the left of f_i and g_i.
3. There exist i and j_i such that $b_{i,j_i} \prec c_{i,j_i}$, and p and q agree to the left of b_{i,j_i} and c_{i,j_i}.

Proposition 3.2. *For each $\alpha \in \mathfrak{S}^c$, $\prec|_{\mathbf{S}_\alpha}$ is a (strict) total order on $\mathbf{S}_\alpha$.*

The relation $\preceq$ is defined by $p \preceq q$ if either $p = q$ or $p \prec q$. Clearly, $\preceq$ is a total order on each $\mathbf{S}_\alpha$.

3.3 Regular Predicates

Now the class of regular predicates can be defined by induction on the number of transformations in a predicate.

Definition. A standard predicate $(f_1\ b_{1,1} \ldots b_{1,k_1}) \cdot \ldots \cdot (f_n\ b_{n,1} \ldots b_{n,k_n})$ is *regular* if b_{i,j_i} is a regular predicate, for $i = 1, \ldots, n$ and $j_i = 1, \ldots, k_i$, and f_i is symmetric implies that $b_{i,1} \preceq \cdots \preceq b_{i,k_i}$, for $i = 1, \ldots, n$.

The set of all regular predicates is denoted by $\mathbf{R}$.

Example. Going back to a previous example,

$$vertices \cdot (domCard\ (vertex \cdot (\wedge_2\ b\ c))) \cdot (> 7)$$

is a regular predicate iff $b \preceq c$, and b and c are regular predicates.

The next result shows that each standard predicate is semantically equivalent to a regular predicate and hence attention can be confined to the generally much smaller class of regular predicates.

Proposition 3.3. *For every $p \in \mathbf{S}$, there exists $q \in \mathbf{R}$ such that p and q are semantically equivalent.*

One cannot expect the regular predicate q in Proposition 3.3 to be (syntactically) unique. To see this, simply note that $(\mathit{domCard}\ \mathit{top}) \cdot (> 0)$ and $(\mathit{setExists}_1\ \mathit{top})$ are regular predicates, and are semantically equivalent.

On the other hand, using the proof of Proposition 3.3 as a basis, one can give an algorithm for constructing a regular predicate semantically equivalent to some given standard predicate. This algorithm is given in Figure 1 below. The regular predicate q given by this algorithm is called the *regularisation* of the standard predicate p. The regularisation of p is denoted by $\overline{p}$. The regularisation $\overline{p}$ of a standard predicate p is in fact a regular predicate, and $\overline{p}$ and p are semantically equivalent. It is also clear that the regularisation of a regular predicate is itself.

function $\mathit{Regularise}(p)$;
returns a regular predicate q such that p and q are semantically equivalent;
input: p, a standard predicate
$(f_1\ b_{1,1} \ldots b_{1,k_1}) \cdot \ldots \cdot (f_n\ b_{n,1} \ldots b_{n,k_n})$;

for $i = 1, \ldots, n$ **do**
  for $j_i = 1, \ldots, k_i$ **do**
    $\overline{b_{i,j_i}} := \mathit{Regularise}(b_{i,j_i})$;
  if f_i is symmetric
  then
    $[c_{i,1}, \ldots, c_{i,k_i}] := \mathit{Sort}([\overline{b_{i,1}}, \ldots, \overline{b_{i,k_i}}])$;
    $p_i := (f_i\ c_{i,1} \ldots c_{i,k_i})$;
  else
    $p_i := (f_i\ \overline{b_{i,1}} \ldots \overline{b_{i,k_i}})$;
return $p_1 \cdot \ldots \cdot p_n$;

Figure 1: Algorithm for regularising a standard predicate

Before moving on to other issues, I note that there is scope for employing syntactic conditions other than symmetry to determine semantic equivalence of predicates. Furthermore, notwithstanding the undecidability of the problem, one could even employ a theorem prover to attempt to establish semantic equivalence. If the cost of proving a theorem to show that two predicates are semantically equivalent is less than the cost of redundantly using both predicates in some application, then the theorem-proving approach is worthwhile.

The next issue is that of the finiteness of the class of standard predicates. If there are infinitely many transformations, then clearly there can be infinitely many standard predicates. For example, if all predicates on integers of the form $(> n)$, for $n = 1, 2, 3, \ldots$, are included in the class of transformations, then (trivially) there are infinitely many standard (and also regular) predicates. Even in the case when there only finitely many transformations, there may still be infinitely many standard predicates. To see this, consider the transformation *leftTree* : *BTree* $\mu \rightarrow$ *BTree* μ. Here *BTree* μ is the type of binary trees whose nodes have type μ and the meaning of *leftTree* is that it returns the left subtree of its argument. Then compositions of *leftTree* with itself of unbounded length can be formed, and so there are infinitely many standard predicates. The next proposition gives the obvious conditions under which the class of standard predicates is finite.

Proposition 3.4. *Suppose that the class of transformations is finite and the number of times each transformation may appear in a standard predicate is bounded. Then the number of standard, and hence regular, predicates is finite.*

3.4 The Implication Preorder

The next major issue is that of determining whether one predicate implies (that is, is stronger than) another. This relationship between predicates can play a crucial role in structuring the search space of predicates in applications. As an example of this, see [2]. I begin with the definition of the implication preorder for standard predicates. The binary relation $\Leftarrow$ on $\mathbf{S}$ is defined as follows.

Definition. Let $p, q \in \mathbf{S}$. Then $p \Leftarrow q$ if the type of q is more general than the type of p and $\forall x.((p\ x) \longleftarrow (q\ x))$ is a logical consequence of $\mathcal{B}$.

Of course, this definition could be given for any predicates, not just standard ones, but that generality is not needed here.

Proposition 3.5. *For each* $\alpha \in \mathfrak{S}^c$, $\Leftarrow\!|_{\mathbf{S}_\alpha}$ *is a preorder on* $\mathbf{S}_\alpha$.

The relation $\Leftarrow$ is called the *implication preorder*. Like semantic equivalence, determining that $p \Leftarrow q$ is undecidable. Consequently, another relation, which is sufficient but not necessary for implication and which can be easily checked, is desirable. With such a relation available, implication can be quickly checked, with the disadvantage that sometimes an implication will be missed. I now turn to the definition of a suitable relation. For this, some machinery is needed.

Definition. A *skeleton* is a term of the form

$$(f_1\ x_{1,1} \ldots x_{1,k_1}) \circ \cdots \circ (f_n\ x_{n,1} \ldots x_{n,k_n}),$$

where f_i is a transformation of rank k_i, for $i = 1, \ldots, n$, the target of f_n is Ω, and the x_{i,j_i} are distinct variables, for $i = 1, \ldots, n$ and $j_i = 1, \ldots, k_i$.

The use of distinct variables in the previous definition is somewhat arbitrary; they merely indicate a position in the skeleton, as in the next definition.

Definition. A variable $x_{r,s}$, for some $r \in \{1, \ldots, n\}$ and some $s \in \{1, \ldots, k_r\}$, is a *monotone argument* of a skeleton

$$(f_1\ x_{1,1} \ldots x_{1,k_1}) \circ \cdots \circ (f_n\ x_{n,1} \ldots x_{n,k_n})$$

if, for any standard predicate p of the form

$$(f_1\ b_{1,1} \ldots b_{1,k_1}) \circ \cdots \circ (f_n\ b_{n,1} \ldots b_{n,k_n})$$

which is an instance of the skeleton and for any standard predicate c such that $b_{r,s} \Leftarrow c$, it follows that $p \Leftarrow p[b_{r,s}/c]$.

Here $p[b_{r,s}/c]$ denotes the standard predicate obtained from p by replacing the distinguished occurrence of $b_{r,s}$ in p by c.

Example. For the skeletons $(setExists_n\ x_1 \dots x_n)$ and $(\wedge_n\ x_1 \dots x_n)$, each x_i is a monotone argument, for $i = 1, \dots, n$.

Example. The variable x is a monotone argument in the skeleton $(domCard\ x) \cdot (> N)$, where N is a non-negative integer. Note that, if $(> N)$ is replaced by $(< N)$ in the predicate, then x no longer has this property.

The next result shows that if one argument of a symmetric transformation is monotone in some skeleton, then all arguments of the transformation are monotone.

Proposition 3.6. *Let $(f_1\ x_{1,1} \dots x_{1,k_1}) \cdot \dots \cdot (f_n\ x_{n,1} \dots x_{n,k_n})$ be a skeleton, f_r a symmetric transformation, and $x_{r,s}$ a monotone argument of the skeleton, for some $r \in \{1, \dots, n\}$ and some $s \in \{1, \dots, k_r\}$. Then x_{r,j_r} is a monotone argument of the skeleton, for $j_r = 1, \dots, k_r$.*

Now I give the key definition of a monotone position in a standard predicate by induction on the number of transformations in the predicate.

Definition. Assume that there is given a set of skeletons together with their monotone arguments. Let p be a standard predicate $(f_1\ b_{1,1} \dots b_{1,k_1}) \cdot \dots \cdot (f_n\ b_{n,1} \dots b_{n,k_n})$ and q a standard predicate that has a distinguished occurrence as a subterm of p. Then q is in a *monotone position* in p if either

1. q is a suffix $(f_t\ b_{t,1} \dots b_{t,k_t}) \cdot \dots \cdot (f_n\ b_{n,1} \dots b_{n,k_n})$ of p, for some $t \geq 1$, or
2. there is a skeleton $(f_t\ x_{t,1} \dots x_{t,k_t}) \cdot \dots \cdot (f_n\ x_{n,1} \dots x_{n,k_n})$, for some $t \geq 1$, having a monotone argument $x_{r,s}$, where $r \in \{t, \dots, n\}$ and $s \in \{1, \dots, k_r\}$, such that q is in a monotone position in $b_{r,s}$.

In the case when a skeleton $(f_t\ x_{t,1} \dots x_{t,k_t}) \cdot \dots \cdot (f_n\ x_{n,1} \dots x_{n,k_n})$ has the same sequence of transformations $[f_t, \dots, f_n]$ as a suffix $(f_t\ b_{t,1} \dots b_{t,k_t}) \cdot \dots \cdot (f_n\ b_{n,1} \dots b_{n,k_n})$ of the predicate, I say that the skeleton *matches* the suffix.

The next proposition establishes some basic properties of the concept of monotone position.

Proposition 3.7. *Assume that there is given a set of skeletons together with their monotone arguments. Let $p, q, r \in \mathbf{S}$. Then the following hold.*

1. *p is in a monotone position in p.*
2. *If q is in a monotone position in p, then q is in a monotone position in $s \cdot p$, for any prefix s such that $s \cdot p \in \mathbf{S}$.*
3. *If r is in a monotone position in q and q is in a monotone position in p, then r is in a monotone position in p.*

Example. The predicate b is in a monotone position in $vertices \cdot (setExists_2\ (\wedge_2\ b\ c)\ d)$. This can be proved using Proposition 3.7 as follows. First, b is in a monotone position in b. Then, since x is a monotone argument in $(\wedge_2\ x\ y)$, it follows that b is in a monotone position in $(\wedge_2\ b\ c)$. Since x is a monotone argument in $(setExists_2\ x\ y)$, b is in a monotone position in $(setExists_2\ (\wedge_2\ b\ c)\ d)$. Finally, by Part 2 of Proposition 3.7, it follows that b is in a monotone position in $vertices \cdot (setExists_2\ (\wedge_2\ b\ c)\ d)$.

The key property of a monotone position is given by the following proposition.

Proposition 3.8. *Assume that there is given a set of skeletons together with their monotone arguments. Let $p, q \in \mathbf{S}$, where q has a distinguished occurrence as a subterm of p. If q is in a monotone position in p then, whenever q' is a standard predicate such that $p[q/q']$ is a term and $q \Leftarrow q'$, it follows that $p \Leftarrow p[q/q']$.*

Given below, in Figure 2, is an algorithm that determines the monotone positions in a standard predicate. The input to this algorithm is a standard predicate and a set of skeletons together with their monotone arguments indicated. The output is the set of monotone positions in the predicate. The algorithm follows closely the earlier definition of a monotone position in a predicate.

function $Positions(p, \mathcal{S})$;
returns the set of monotone positions in p;
input: p, a standard predicate
$(f_1\ b_{1,1} \ldots b_{1,k_1}) \cdot \ldots \cdot (f_n\ b_{n,1} \ldots b_{n,k_n})$;
$\mathcal{S}$, a set of skeletons together with their monotone arguments;

$pos := \{\}$;
for each suffix s of p **do**
  $pos := pos \cup \{s\}$;
for each skeleton in $\mathcal{S}$ **do**
  if the skeleton matches a suffix of p
  then
    for each $b_{r,s}$ corresponding to a monotone argument in skeleton **do**
      $pos := pos \cup Positions(b_{r,s}, \mathcal{S})$;
return pos;

Figure 2: Algorithm for finding the monotone positions in a standard predicate

Example. Consider the predicate p defined by

$$vertices \cdot (setExists_2\ (\wedge_2\ (proj_1 \cdot (= A))\ (proj_2 \cdot (= 3)))\ (proj_1 \cdot (= B))).$$

Suppose the skeletons are $(setExists_2\ x\ y)$ with x and y being monotone arguments, and $(\wedge_2\ x\ y)$ with x and y being monotone arguments. Then the set of monotone positions in p returned by the algorithm is

$$\begin{aligned}
\{&\mathit{vertices} \cdot (\mathit{setExists_2}\ (\wedge_2\ (\mathit{proj_1} \cdot (= A))\ (\mathit{proj_2} \cdot (= 3)))\ (\mathit{proj_1} \cdot (= B))),\\
&\mathit{setExists_2}\ (\wedge_2\ (\mathit{proj_1} \cdot (= A))\ (\mathit{proj_2} \cdot (= 3)))\ (\mathit{proj_1} \cdot (= B)),\\
&\wedge_2\ (\mathit{proj_1} \cdot (= A))\ (\mathit{proj_2} \cdot (= 3)),\\
&\mathit{proj_1} \cdot (= A),\\
&(= A),\\
&\mathit{proj_2} \cdot (= 3),\\
&(= 3),\\
&\mathit{proj_1} \cdot (= B),\\
&(= B)\}.
\end{aligned}$$

3.5 The Relation $\ll$

One more ingredient is needed in order to define the relation that approximates the implication preorder.

Definition. A binary relation $\rightarrowtail$ on $\mathbf{S}$ is *implication-consistent* if, for each $p, q \in \mathbf{S}$, $p \rightarrowtail q$ implies $p \Leftarrow q$.

Example. For an application involving chemical structure using (undirected) graphs [2], the implication-consistent relation $\rightarrowtail$ includes the following relationships. (Recall that *top* is the predicate defined by $\mathit{top}\ x = \top$, for all x.).

$$\begin{aligned}
&\mathit{top} \rightarrowtail (\mathit{subgraphs}\ 3) \cdot (\mathit{setExists_1}\ (\wedge_2\ (\mathit{edges} \cdot \mathit{top})\ (\mathit{vertices} \cdot \mathit{top})))\\
&\mathit{top} \rightarrowtail (\mathit{domCard}\ \mathit{top}) \cdot (> 0)\\
&\mathit{top} \rightarrowtail \wedge_2\ (\mathit{connects} \cdot (\mathit{msetExists_2}\ \mathit{top}\ \mathit{top}))\ (\mathit{edge} \cdot \mathit{top})\\
&\mathit{top} \rightarrowtail \mathit{vertex} \cdot (\wedge_3\ (\mathit{projElement} \cdot \mathit{top})\ (\mathit{projAtomType} \cdot \mathit{top})\\
&\qquad\qquad\qquad\qquad (\wedge_2\ (\mathit{projCharge} \cdot \mathit{top})\ (\mathit{projCharge} \cdot \mathit{top})))\\
&\mathit{top} \rightarrowtail (= 1)\\
&\mathit{top} \rightarrowtail (= S)\\
&\mathit{top} \rightarrowtail (\leq 0.013)\\
&(> 0) \rightarrowtail (> 1)\\
&(> 1) \rightarrowtail (> 2).
\end{aligned}$$

For the above $p \rightarrowtail q$, where p is *top*, the property $p \Leftarrow q$ is obvious. For $(> 0) \rightarrowtail (> 1)$ and $(> 1) \rightarrowtail (> 2)$, this property is also obvious.

I can now define the desired relation $\ll$ on $\mathbf{S}$.

Definition. Assume that there is given an implication-consistent relation $\rightarrowtail$ on $\mathbf{S}$ and a set of skeletons together with their monotone arguments. Let $p, q \in \mathbf{S}$. Then $p \ll q$ if there exists $\alpha \in \mathfrak{S}^c$ such that $p, q \in \mathbf{S}_\alpha$ and either $p = q$ or there is a predicate r in a monotone position in p and a predicate s such that $r \rightarrowtail s$ and $q = p[r/s]$.

A good way to understand the relation $\rightarrowtail$ is to regard $r \rightarrowtail s$ as a *rewrite*, for which the redex r (in some predicate) is rewritten to s. In other words, p and q are in the relation $p \ll q$ if either $p = q$ or there is a redex r in a monotone position in p and q is the predicate obtained from p by applying the rewrite $r \rightarrowtail s$ at this redex.

Example. Let ε be the type of information on edges in a graph, and b and c predicates on the type ε. Then the relation

$$\begin{aligned}
&(\mathit{subgraphs}\ 3) \cdot (\mathit{domCard}\ (\mathit{edges} \cdot (\mathit{setExists}_2 \\
&\qquad\qquad\qquad\qquad (\mathit{edge} \cdot b)\ (\mathit{edge} \cdot c)))) \cdot (> 1) \\
\ll\ &(\mathit{subgraphs}\ 3) \cdot (\mathit{domCard}\ (\mathit{edges} \cdot (\mathit{setExists}_2 \\
&\qquad\qquad\qquad\qquad (\mathit{edge} \cdot b)\ (\mathit{edge} \cdot c)))) \cdot (> 2)
\end{aligned}$$

holds between the two predicates, assuming that $(> 1) \rightarrowtail (> 2)$.

Also the relation

$$\begin{aligned}
&(\mathit{subgraphs}\ 3) \cdot (\mathit{domCard}\ (\mathit{edges} \cdot (\mathit{setExists}_2 \\
&\qquad\qquad\qquad\qquad (\mathit{edge} \cdot b)\ (\mathit{edge} \cdot c)))) \cdot (< 2) \\
\ll\ &(\mathit{subgraphs}\ 3) \cdot (\mathit{domCard}\ (\mathit{edges} \cdot (\mathit{setExists}_2 \\
&\qquad\qquad\qquad\qquad (\mathit{edge} \cdot b)\ (\mathit{edge} \cdot c)))) \cdot (< 1)
\end{aligned}$$

holds between the two predicates, assuming that $(< 2) \rightarrowtail (< 1)$.

Finally, the relation

$$\begin{aligned}
&(\mathit{subgraphs}\ 3) \cdot (\mathit{domCard}\ (\mathit{edges} \cdot (\mathit{setExists}_2 \\
&\qquad\qquad\qquad\qquad (\mathit{edge} \cdot b)\ (\mathit{edge} \cdot c)))) \cdot (> 1) \\
\ll\ &(\mathit{subgraphs}\ 3) \cdot (\mathit{domCard}\ (\mathit{edges} \cdot (\mathit{setExists}_2 \\
&\qquad\qquad\qquad\qquad (\mathit{edge} \cdot b')\ (\mathit{edge} \cdot c)))) \cdot (> 1)
\end{aligned}$$

holds, assuming that $b \rightarrowtail b'$.

Clearly $\ll$ is reflexive and so $\ll^*$, the transitive closure of $\ll$, is a preorder. The next proposition shows that $\ll^*$ is a suitable relation for approximating the implication preorder.

Proposition 3.9. *For each $p, q \in \mathbf{S}$, $p \ll^* q$ implies $p \Leftarrow q$.*

To what extent does $\ll^*$ closely approximate $\Leftarrow$? This depends largely on the choice of $\rightarrowtail$. If this relation is small, in some sense, then there may not be enough rewrites to establish the relation $p \ll^* q$ for some p and q which satisfy $p \Leftarrow q$. At the other extreme, one can guarantee that $\ll^*$ is equal to $\Leftarrow$: simply choose $\rightarrowtail$ to be $\Leftarrow$! While mathematically correct, this remark is of no practical use. It would be interesting to know if there were some practically useful sufficient conditions on $\rightarrowtail$ (and the set of skeletons) to ensure that $\ll^* = \Leftarrow$.

3.6 Enumerating Regular Predicates

A much more common task than checking that $p \ll q$ or $p \ll^* q$, for some predicates p and q, is to efficiently enumerate (a large subset of) the set of regular predicates in such a way as to keep track of the implication preorder between predicates. I now discuss an algorithm, given in Figure 3 below, to do this.

The input to this algorithm is an implication-consistent relation $\rightarrowtail$ on $\mathbf{S}$ and a set of skeletons together with their monotone arguments. Since the algorithm is intended to produce regular predicates only, the restriction is made that $\rightarrowtail$ holds only for regular predicates, as given by the following definition.

Definition. A binary relation $\rightarrowtail$ on $\mathbf{S}$ is *regular* if, for each $p, q \in \mathbf{S}$, $p \rightarrowtail q$ implies $p, q \in \mathbf{R}$.

The output of the algorithm is a set of regular predicates. With some choices of $\rightarrowtail$ the algorithm may not terminate. One could ensure termination, if necessary, by restricting the number of times each transformation can appear in a predicate, for example. Since the open list is treated as a queue, the predicates are produced in a breadth-first fashion. In typical applications, the relation $\rightarrowtail$ will be finite. (This is not a requirement of the theory, but it is hard to imagine an application that required $\rightarrowtail$ to be infinite.) In this case, the branching factor of the search space is bounded.

function *Enumerate*$(\rightarrowtail, \mathcal{S})$;
returns a set of regular predicates;
input: $\rightarrowtail$, a regular, implication-consistent relation;
$\mathcal{S}$, a set of skeletons together with their monotone arguments;

regular $:= \{\}$;
openList $:= [top]$;
seenSet $:= \{top\}$;
while *openList* $\neq []$ **do**
  $p :=$ *head*(*openList*);
  openList $:=$ *tail*(*openList*);
  regular $:=$ *regular* $\cup \{p\}$;
  for each r such that $r \in$ *Positions*$(p, \mathcal{S})$ and $r \rightarrowtail s$, for some s, **do**
    $q :=$ *Regularise*$(p[r/s])$;
    if $q \notin$ *seenSet*
    then
      seenSet $:=$ *seenSet* $\cup \{q\}$;
      openList $:=$ *openList* $++ [q]$;
return *regular*;

Figure 3: Algorithm for enumerating regular predicates

There are some complications that this algorithm has to deal with. The first is that there may be many paths from *top* to some predicate p and so p could be obtained many times. In other words, the problem is that the search space may be a graph rather than a tree. The classical solution to this problem, also adopted here, is to simply keep a record of all the predicates seen so far. This is done with the set *seenSet*, which would be implemented as a hash table. Each time a child predicate is generated, the algorithm checks to see if it is in this set, adding it if it is not. This tactic essentially turns the search space back into a tree. The second complication is that $p[r/s]$ may not be regular and hence there is a need to regularise it.

It is important to establish that the enumeration algorithm does actually return a large class of regular predicates.

Proposition 3.10. *Assume that there is given a regular, implication-consistent relation $\rightarrowtail$ on* $\mathbf{S}$ *and a set of skeletons together with their monotone arguments. Suppose p is a regular predicate for which there is a standard predicate q such that $top \ll^* q$ and $p = \overline{q}$. Then p is in the set returned by the enumeration algorithm.*

A major feature of the enumeration algorithm is that the children of a predicate all imply the parent. In some applications, especially for (top-down) machine learning algorithms this feature is crucial. Typically in such applications, once predicates become sufficiently strong (in the sense of the implication preorder $\Leftarrow$), they should be pruned from the search space. If a predicate is sufficiently strong to be pruned, then so are all its successors. Hence it is safe to remove the predicate from the open list and prune all successors (by not generating its children). For more details, see [1] and [2].

At this point, it is worthwhile reflecting on the predicate construction process. I have advocated the utility of the class of regular predicates for building hypothesis languages. So, first, how general is the class of regular predicates? Regular predicates are defined relative to some set of transformations and every predicate is a transformation, thus *every* predicate is a regular predicate with respect to a suitable set of transformations. The more interesting question is how effective and convenient is the predicate construction process for building hypothesis languages? One cannot prove a theorem to answer this question – the answer must come from the applications and experience. The concept of a transformation itself came from analysing a wide variety of applications and trying to find an elegant way of specifying the corresponding hypothesis languages. The conclusion from this analysis is that it does indeed seem to be the appropriate higher-order concept for this purpose. Furthermore, my experience is that the incremental construction of predicates, as described above and illustrated in the next section (and in [1] and [2]), does give the user precise and explicit control over the search space of predicates. Space restrictions preclude giving further examples here, but the reader can consult [1] and [2] for applications involving a wide variety of types that illustrate the utility of the ideas.

4 Applications to Machine Learning

The predicate construction process described in the previous section has applications throughout machine learning. Perhaps the most obvious of these is to decision-tree learning. To build a decision tree, the fundamental step in the algorithm is to find a predicate that splits some given set of training examples into two sets. The split must be a good one in the sense that each of the two subsets is "purer" in the distribution of classes than the original set. This paper provides a convenient and effective method for finding such a predicate for individuals with complex structure. Given the type of the individuals, one first decides on a suitable set of transformations and then a suitable set of rewrites for enumerating predicates. Then, using the algorithm in Figure 3, one can enumerate a large class of regular predicates. This section contains a worked example for a simple application to illustrate these ideas. The application chosen is interesting because it illustrates the multiple-instance problem that is discussed in [4]. From a knowledge representation point of view, a multiple-instance problem is indicated by having a term which is a set at the top level representing an individual.

Example. Consider the problem of determining whether a key in a bunch of keys can open a door. More precisely, suppose there are a number of bunches of keys and a particular door which can be opened by a key. For each bunch of keys either no key opens the door or there is at least one key which opens the door. For each bunch of keys it is known whether there is some key which opens the door, but it is not known precisely which key does the job, or it is known that no key opens the door. The problem is to find a classification function for the bunches of keys, where the classification is into those which contain a key that opens the door and those that do not. This problem is prototypical of a number of important practical problems such as that of pharmacophore discovery [4, 5].

A characteristic property of the multiple-instance problem is that the "individuals" are actually a collection of similar entities and the problem is to try to find a suitable entity in each collection satisfying some predicate. Thus the kind of predicate one wants to find has the form ($setExists_1$ b), for a suitable predicate b on the entities in the collection. Expressed this way, it is clear that the multiple-instance problem is just a special case of the general framework that is developed in this paper in which the type of the individuals is a set. Since the individuals to be classified have a set type, following the principles proposed here, one should go inside the set to apply predicates to its elements.

To illustrate the multiple-instance problem, consider the following application concerning bunches of keys. The following declarations are made.

$$\begin{aligned}
&\mathit{Abloy}, \mathit{Chubb}, \mathit{Rubo}, \mathit{Yale} : \mathit{Make}\\
&\mathit{Short}, \mathit{Medium}, \mathit{Long} : \mathit{Length}\\
&\mathit{Narrow}, \mathit{Normal}, \mathit{Broad} : \mathit{Width}.
\end{aligned}$$

The following type synonyms are helpful.

$$\begin{aligned}
&\mathit{NumProngs} = \mathit{Nat}\\
&\mathit{Key} = \mathit{Make} \times \mathit{NumProngs} \times \mathit{Length} \times \mathit{Width}\\
&\mathit{Bunch} = \{\mathit{Key}\}.
\end{aligned}$$

Thus the individuals in this case are sets whose elements are 4-tuples. The function to be learned is

$$\mathit{opens} : \mathit{Bunch} \to \Omega.$$

Here are two typical examples.

$$\begin{aligned}
\mathit{opens}\ \{&(\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad}),\\
&(\mathit{Chubb}, 3, \mathit{Long}, \mathit{Narrow}),\\
&(\mathit{Abloy}, 3, \mathit{Short}, \mathit{Normal})\} = \top\\
\mathit{opens}\ \{&(\mathit{Chubb}, 3, \mathit{Medium}, \mathit{Broad}),\\
&(\mathit{Rubo}, 5, \mathit{Long}, \mathit{Narrow}),\\
&(\mathit{Abloy}, 4, \mathit{Short}, \mathit{Broad})\} = \bot.
\end{aligned}$$

The hypothesis language contains the following transformations.

$$\begin{aligned}
&(= \mathit{Abloy}) : \mathit{Make} \to \Omega\\
&(= \mathit{Chubb}) : \mathit{Make} \to \Omega\\
&(= \mathit{Rubo}) : \mathit{Make} \to \Omega
\end{aligned}$$

$$
\begin{aligned}
&(= \mathit{Yale}) : \mathit{Make} \to \Omega \\
&(= 2) : \mathit{NumProngs} \to \Omega \\
&(= 3) : \mathit{NumProngs} \to \Omega \\
&(= 4) : \mathit{NumProngs} \to \Omega \\
&(= 5) : \mathit{NumProngs} \to \Omega \\
&(= 6) : \mathit{NumProngs} \to \Omega \\
&(= \mathit{Short}) : \mathit{Length} \to \Omega \\
&(= \mathit{Medium}) : \mathit{Length} \to \Omega \\
&(= \mathit{Long}) : \mathit{Length} \to \Omega \\
&(= \mathit{Narrow}) : \mathit{Width} \to \Omega \\
&(= \mathit{Normal}) : \mathit{Width} \to \Omega \\
&(= \mathit{Broad}) : \mathit{Width} \to \Omega \\
&\mathit{prMake} : \mathit{Key} \to \mathit{Make} \\
&\mathit{prNumProngs} : \mathit{Key} \to \mathit{NumProngs} \\
&\mathit{prLength} : \mathit{Key} \to \mathit{Length} \\
&\mathit{prWidth} : \mathit{Key} \to \mathit{Width} \\
&\mathit{setExists_1} : (\mathit{Key} \to \Omega) \to \mathit{Bunch} \to \Omega \\
&\wedge_2 : (\mathit{Key} \to \Omega) \to (\mathit{Key} \to \Omega) \to \mathit{Key} \to \Omega \\
&\wedge_3 : (\mathit{Key} \to \Omega) \to (\mathit{Key} \to \Omega) \to (\mathit{Key} \to \Omega) \to \mathit{Key} \to \Omega \\
&\wedge_4 : (\mathit{Key} \to \Omega) \to (\mathit{Key} \to \Omega) \to (\mathit{Key} \to \Omega) \to (\mathit{Key} \to \Omega) \to \\
&\hspace{20em} \mathit{Key} \to \Omega.
\end{aligned}
$$

Here *prMake* is the projection from the type *Key* onto the first component of type *Make*. Similarly, for *prNumProngs*, *prLength* and *prWidth*. Starting from the weakest predicate *top* (having type $\mathit{Bunch} \to \Omega$), I show how to systematically enumerate predicates using a system of rewrites. First, here are the rewrites for this application.

$$
\begin{aligned}
&\mathit{top} \rightarrowtail \mathit{setExists_1}\ (\wedge_4\ (\mathit{prMake} \cdot \mathit{top})\ (\mathit{prNumProngs} \cdot \mathit{top}) \\
&\hspace{16em} (\mathit{prLength} \cdot \mathit{top})\ (\mathit{prWidth} \cdot \mathit{top})) \\
&\mathit{top} \rightarrowtail (= \mathit{Abloy}) \\
&\mathit{top} \rightarrowtail (= \mathit{Chubb}) \\
&\mathit{top} \rightarrowtail (= \mathit{Rubo}) \\
&\mathit{top} \rightarrowtail (= \mathit{Yale}) \\
&\mathit{top} \rightarrowtail (= 2) \\
&\mathit{top} \rightarrowtail (= 3) \\
&\mathit{top} \rightarrowtail (= 4) \\
&\mathit{top} \rightarrowtail (= 5) \\
&\mathit{top} \rightarrowtail (= 6) \\
&\mathit{top} \rightarrowtail (= \mathit{Short}) \\
&\mathit{top} \rightarrowtail (= \mathit{Medium}) \\
&\mathit{top} \rightarrowtail (= \mathit{Long}) \\
&\mathit{top} \rightarrowtail (= \mathit{Narrow}) \\
&\mathit{top} \rightarrowtail (= \mathit{Normal}) \\
&\mathit{top} \rightarrowtail (= \mathit{Broad})
\end{aligned}
$$

The relation defined by this set of rewrites is clearly implication-consistent. The skeletons are ($setExists_1\ x$), for which x is a monotone argument, and ($\wedge_4\ x_1\ x_2\ x_3\ x_4$), for which x_i is a monotone argument, for $i = 1, \ldots, 4$.

Because of the nature of the rewrite set, the only order of interest is that for the projections and, for this, I define

$$prMake < prNumProngs < prLength < prWidth.$$

The procedure for constructing predicates is as follows. Starting with the predicate top (of type $Bunch \rightarrow \Omega$), all rewrites whose head matches top are applied. There is only one of these,

$$\begin{aligned} top \rightarrowtail setExists_1\ (\wedge_4\ (prMake \cdot top)\ (prNumProngs \cdot top)& \\ (prLength \cdot top)\ (prWidth \cdot top)),& \end{aligned}$$

which gives the new predicate

$$\begin{aligned} setExists_1\ (\wedge_4\ (prMake \cdot top)\ (prNumProngs \cdot top)& \\ (prLength \cdot top)\ (prWidth \cdot top)).& \end{aligned}$$

There are four redexes in this predicate, each having the form top. When the rewrites whose heads match these redexes are applied, the following predicates are obtained.

$$\begin{aligned} setExists_1\ (\wedge_4\ (prMake \cdot (= Abloy))\ (prNumProngs \cdot top)& \\ (prLength \cdot top)\ prWidth \cdot top))& \\ setExists_1\ (\wedge_4\ (prMake \cdot (= Chubb))\ (prNumProngs \cdot top)& \\ (prLength \cdot top)\ (prWidth \cdot top))& \\ setExists_1\ (\wedge_4\ (prMake \cdot (= Rubo))\ (prNumProngs \cdot top)& \\ (prLength \cdot top)\ (prWidth \cdot top))& \\ setExists_1\ (\wedge_4\ (prMake \cdot (= Yale))\ (prNumProngs \cdot top)& \\ (prLength \cdot top)\ (prWidth \cdot top))& \\ setExists_1\ (\wedge_4\ (prMake \cdot top)\ (prNumProngs \cdot (= 2))& \\ (prLength \cdot top)\ (prWidth \cdot top))& \\ setExists_1\ (\wedge_4\ (prMake \cdot top)\ (prNumProngs \cdot (= 3))& \\ (prLength \cdot top)\ (prWidth \cdot top))& \\ \vdots\qquad& \\ setExists_1\ (\wedge_4\ (prMake \cdot top)\ (prNumProngs \cdot top)& \\ (prLength \cdot top)\ (prWidth \cdot (= Narrow)))& \\ setExists_1\ (\wedge_4\ (prMake \cdot top)\ (prNumProngs \cdot top)& \\ (prLength \cdot top)\ (prWidth \cdot (= Normal)))& \\ setExists_1\ (\wedge_4\ (prMake \cdot top)\ (prNumProngs \cdot top)& \\ (prLength \cdot top)\ (prWidth \cdot (= Broad))).& \end{aligned}$$

Taking the first of these children,

$$\begin{aligned} setExists_1\ (\wedge_4\ (prMake \cdot (= Abloy))\ (prNumProngs \cdot top)& \\ (prLength \cdot top)\ (prWidth \cdot top)),& \end{aligned}$$

and applying the rewrites at each available redex, one obtains

$$
\begin{aligned}
&\mathit{setExists}_1\ (\wedge_4\ (\mathit{prMake} \cdot (= \mathit{Abloy}))\ (\mathit{prNumProngs} \cdot (= 2)) \\
&\qquad\qquad\qquad\qquad (\mathit{prLength} \cdot \mathit{top})\ (\mathit{prWidth} \cdot \mathit{top})) \\
&\mathit{setExists}_1\ (\wedge_4\ (\mathit{prMake} \cdot (= \mathit{Abloy}))\ (\mathit{prNumProngs} \cdot (= 3)) \\
&\qquad\qquad\qquad\qquad (\mathit{prLength} \cdot \mathit{top})\ (\mathit{prWidth} \cdot \mathit{top})) \\
&\qquad\quad \vdots \\
&\mathit{setExists}_1\ (\wedge_4\ (\mathit{prMake} \cdot (= \mathit{Abloy}))\ (\mathit{prNumProngs} \cdot \mathit{top}) \\
&\qquad\qquad\qquad (\mathit{prLength} \cdot \mathit{top})\ (\mathit{prWidth} \cdot (= \mathit{Narrow}))) \\
&\mathit{setExists}_1\ (\wedge_4\ (\mathit{prMake} \cdot (= \mathit{Abloy}))\ (\mathit{prNumProngs} \cdot \mathit{top}) \\
&\qquad\qquad\qquad (\mathit{prLength} \cdot \mathit{top})\ (\mathit{prWidth} \cdot (= \mathit{Normal}))) \\
&\mathit{setExists}_1\ (\wedge_4\ (\mathit{prMake} \cdot (= \mathit{Abloy}))\ (\mathit{prNumProngs} \cdot \mathit{top}) \\
&\qquad\qquad\qquad (\mathit{prLength} \cdot \mathit{top})\ (\mathit{prWidth} \cdot (= \mathit{Broad}))).
\end{aligned}
$$

This process continues, producing successively stronger predicates.

For this application, the learning system in [1] and [2] found the following definition for the function *opens*.

$$
\begin{aligned}
&\mathit{opens}\ b = \\
&\quad \mathit{if}\ \ \mathit{setExists}_1\ (\wedge_2\ (\mathit{projMake} \cdot (= \mathit{Abloy})) \\
&\qquad\qquad\qquad\qquad (\mathit{projLength} \cdot (= \mathit{Medium})))\ b \\
&\quad \mathit{then}\ \top \\
&\quad \mathit{else}\ \bot.
\end{aligned}
$$

"A bunch of keys opens the door if and only if it contains an Abloy key of medium length".

The appearance of $\wedge_2$ in the definition is due to a final simplification step:

$$
\begin{aligned}
&\wedge_4\ \ (\mathit{prMake} \cdot (= \mathit{Abloy}))\ (\mathit{prNumProngs} \cdot \mathit{top}) \\
&\qquad\qquad\qquad (\mathit{prLength} \cdot (= \mathit{Medium}))\ (\mathit{prWidth} \cdot \mathit{top})
\end{aligned}
$$

can be simplified to

$$\wedge_2\ (\mathit{prMake} \cdot (= \mathit{Abloy}))\ (\mathit{prLength} \cdot (= \mathit{Medium})).$$

This completes the example.

At this point, it is instructive to compare the higher-order approach to machine learning advocated here with the more usual first-order approach in the field of Inductive Logic Programming (ILP), the theory of which is covered comprehensively in [9].

First, representation of individuals is handled differently in each approach, but there is a correspondence between them. In the higher-order case, individuals are represented by basic terms. In the first-order case, abstractions such as sets and multisets are not directly available and so the usual approach is to name the individual by a constant and then have a relational database of facts about this constant that records the same information as the higher-order term does. An outline of the mapping from a basic term to the corresponding relational database is given in [7] and a example involving the representation of a chemical molecule as both a term and a database is given in [2].

The higher-order approach is arguably more direct than the first-order one and corresponds precisely to the principle in logic that (closed) terms denote individuals. Furthermore, since higher-order logic includes all the features of first-order logic and more besides, it is clearly advantageous to work in higher-order logic. However, it is true that all individuals that can be represented in higher-order logic can also be represented in first-order logic, so nothing essential is lost by staying in first-order logic.

Second, the approach to theory construction here and that in ILP are different, but there is a correspondence between them. To see this, consider the general statement of the problem of inductive learning for the first-order case, as expounded in [9, p.167].

> **Given:** A background theory $\mathcal{B}$, a set E^+ of positive examples, and a set E^- of negative examples.
> **Find:** A theory Σ such that $\Sigma \cup \mathcal{B}$ is correct with respect to E^+ and E^-.

Correctness means that $\Sigma \cup \mathcal{B}$ logically implies all the positive examples but none of the negative ones.

Most ILP systems proceed by starting with some initial theory Σ and iteratively specialising $\Sigma \cup \mathcal{B}$, if it is too strong, or generalising $\Sigma \cup \mathcal{B}$, if it is too weak, until $\Sigma \cup \mathcal{B}$ is correct. Typically, both Σ and $\mathcal{B}$ are Horn clause theories and Σ is the definition of some predicate. Such systems are often either top down (that is, only ever specialise) or are bottom up (that is, only ever generalise). A key feature of ILP systems is the use of refinement operators. An upward refinement operator computes generalisations of a clause, while a downward refinement operator computes specialisations of a clause. Much more detail is contained in [9], but this is enough to be able to draw out the comparison between the first-order and higher-order approaches.

Now I return to the extended example of this section to draw out the comparison. The induced definition for the function *opens* was found with a decision-tree system, which is a top-down learner. Starting from the most general definition that all bunches of keys open the door, the learner specialises the predicate in the definition until a suitable one is found. The corresponding search tree has the most general predicate *top* at the root of the tree and has successively stronger predicates down the branches of the tree. For this example, the decision-tree has a single split and hence the corresponding definition contains just one *if_then_else* function. More generally, the decision-tree algorithm does a hill-climbing search, at each stage committing to some split of the examples at that node. For each split, there is a corresponding occurrence of the *if_then_else* function in the definition.

To make the comparison of the first-order and higher-order cases clearer, it will be convenient to replace the occurrences of the *if_then_else* function in the definition by occurrences of the if-then-else transformation *ite*. As an example of this, the definition for *opens* can be reformulated as follows.

$$\begin{aligned} opens = ite\ (setExists_1\ (\wedge_2\ (projMake \cdot (= Abloy))\\ (projLength \cdot (= Medium))))\ top\ bottom. \end{aligned}$$

More generally, definitions produced by the decision-tree learner will have as many occurrences of *ite* as there are splits in the corresponding decision tree.

With this change, solving the keys problem can be thought of as a top-down search for a definition for *opens* that starts from the most general definition

$$opens = top,$$

and uses the rewrites

$$\begin{aligned}
&top \rightarrowtail ite\ top\ top\ top\\
&top \rightarrowtail setExists_1\ (\wedge_4\ (prMake \cdot top)\ (prNumProngs \cdot top)\\
&\qquad\qquad\qquad\qquad (prLength \cdot top)\ (prWidth \cdot top))\\
&top \rightarrowtail (= Abloy)\\
&\qquad\vdots\\
&top \rightarrowtail (= Broad)\\
&top \rightarrowtail bottom.
\end{aligned}$$

In this formulation, there would need to be careful control of the use of the first rewrite – this control is effectively built into the decision-tree algorithm.

As well as this top-down approach, there are bottom-up ones. In this case, the starting definition for *opens* is the most specific definition

$$opens = bottom,$$

and the rewrites have the form

$$\begin{aligned}
&bottom \rightarrowtail ite\ bottom\ bottom\ bottom\\
&bottom \rightarrowtail setExists_1\ (\wedge_4\ (prMake \cdot bottom)\ (prNumProngs \cdot bottom)\\
&\qquad\qquad\qquad\qquad (prLength \cdot bottom)\ (prWidth \cdot bottom))\\
&bottom \rightarrowtail (= Abloy)\\
&\qquad\vdots\\
&bottom \rightarrowtail (= Broad)\\
&bottom \rightarrowtail top.
\end{aligned}$$

For bottom-up algorithms, the definition of implication-consistent is reversed, so that the head of a rewrite implies the body. With this set of rewrites, the definition for *opens* could be found by a covering algorithm that successively constructs the following definition bodies.

$$\begin{aligned}
&bottom\\
&ite\ bottom\ bottom\ bottom\\
&ite\ (setExists_1\ (\wedge_4\ (prMake \cdot bottom)\ (prNumProngs \cdot bottom)\\
&\qquad (prLength \cdot bottom)\ (prWidth \cdot bottom)))\ bottom\ bottom\\
&ite\ (setExists_1\ (\wedge_4\ (prMake \cdot (= Abloy))\ (prNumProngs \cdot bottom)\\
&\qquad (prLength \cdot bottom)\ (prWidth \cdot bottom)))\ bottom\ bottom\\
&\qquad\vdots\\
&ite\ (setExists_1\ (\wedge_4\ (prMake \cdot (= Abloy))\ (prNumProngs \cdot top)\\
&\qquad (prLength \cdot (= Medium))\ (prWidth \cdot top)))\ top\ bottom.
\end{aligned}$$

The comparison between the first-order approach and higher-order approach should be clear by now. In the top-down case, the definition body *top* corresponds to the most general theory and applying a rewrite corresponds to applying a downward refinement operator. In the bottom-up case, the definition body *bottom* corresponds to the most specific theory and applying a rewrite corresponds to applying an upward refinement operator.

To some extent, whether one uses the the first-order approach or the higher-order approach is a matter of taste – except for some specifically higher-order tasks, such as explicitly learning the definition of a higher-order function, whatever can be done in the higher-order setting can be done in the first-order one. Nevertheless, there are clear advantages in using the higher-order setting. First, in addition to the first-order facilities, there are the additional higher-order ones that prove to be difficult to live without once one gets used to their convenience. Second, the technique of specifying the hypothesis language by means of rewrites has proved to be highly successful for decision-tree learning. Evidence for this is given in [1] and [2]. Constructing theories by specifying a set of rewrites gives the user precise and explicit control over the search space of predicates. Furthermore, this method banishes troublesome variables altogether from the set of rewrites and the predicates constructed – variables are needed to define the transformations, but after that they do not appear.

Finally, I make a remark about types. The logic of this paper is typed and much essential use is made of this here, especially in the rewriting process where the types ensure that meaningless predicates are not constructed. The first-order approach is based on the logic of Prolog, that is, untyped first-order logic. Not having a type system creates problems (mostly by an unwarranted explosion of the search space with meaningless clauses) and this has been recognised by many ILP researchers who have added *ad hoc* type systems to their learning systems. Types really do make a difference both for knowledge representation and predicate (or theory) construction, so the only truly satisfactory solution is to adopt a typed logic, whether first- or higher-order, as the foundation of the learning system.

5 Conclusion

Typed higher-order logics provide powerful tools for applications in computational logic and machine learning. This paper has illustrated this in the context of predicate construction for machine learning. I hope other researchers take up the challenge of putting these logics to practical use.

Acknowledgements

This work has greatly benefited from a collaboration over the last four years with Antony Bowers and Christophe Giraud-Carrier on the development of a higher-order inductive learning system.

References

[1] A.F. Bowers, C. Giraud-Carrier, and J.W. Lloyd. Classification of individuals with complex structure. In P. Langley, editor, *Machine Learning: Proceedings of the Seventeenth International Conference (ICML2000)*, pages 81–88. Morgan Kaufmann, 2000.

[2] A.F. Bowers, C. Giraud-Carrier, and J.W. Lloyd. A knowledge representation framework for inductive learning. Available at `http://csl.anu.edu.au/~jwl`, 2001.

[3] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[4] T.G. Dietterich, R.H. Lathrop, and T. Lozano-Pérez. Solving the multiple instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89:31–71, 1997.

[5] P. Finn, S. Muggleton, D. Page, and A. Srinivasan. Pharmacophore discovery using the inductive logic programming system PROGOL. *Machine Learning*, 30:241–270, 1998.

[6] J.W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1999(3), March 1999.

[7] J.W. Lloyd. Knowledge representation, computation, and learning in higher-order logic. Available at `http://csl.anu.edu.au/~jwl`, 2001.

[8] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.

[9] S.H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. Lecture Notes in Artificial Intelligence, 1228. Springer-Verlag, 1997.