Higher-order Computational Logic

J.W. Lloyd

Computer Sciences Laboratory
Research School of Information Sciences and Engineering
Australian National University
jwl@csl.anu.edu.au

Abstract. This paper presents a case for the use of higher-order logic as a foundation for computational logic. A suitable polymorphically-typed, higher-order logic is introduced and its syntax and proof theory briefly described. In addition, a metric space of closed terms suitable for knowledge representation purposes is presented. The approach to representing individuals is illustrated with some examples, as is the technique of programming with abstractions. The paper concludes by placing the results in the wider context of previous and current research in the use of higher-order logic in computational logic.

1 Introduction

In 1974, Robert Kowalski published the seminal idea that predicate logic could be used as a programming language [10]. The setting for Kowalski's idea was first-order logic, in fact, the Horn clause fragment of that logic. Since 1974, there has been an explosion of research activity that has pushed the fundamental idea in many different directions.

Once the idea that logic can be used as a programming language is appreciated, it is natural to ask what other logics might be useful in this context. Indeed, one can ask the question more generally: what logics are useful as a basis for the field of computational logic, where computational logic is to be understood broadly as the use of logic in Computer Science? There are a number of good answers to this question that depend to some extent on which part of computational logic one wishes to study. Some of these are explored elsewhere in this volume. In this paper, I make a case for the use of higher-order logic as the theoretical basis for computational logic.

I begin by giving an overview of a suitable higher-order logic. This logic has its origins in Church's simple theory of types [4], but is significantly extended by a polymorphic type system that is needed for its application to declarative programming languages. I briefly describe the syntax and proof theory of the logic. Next I introduce a metric space of certain closed terms. This space is highly suitable for modelling individuals in diverse applications and, unusually, includes certain abstractions that represent (finite) sets and multisets. A much more detailed account of the logic, including its semantics and the proofs of all the propositions in this paper, is contained in [13].

I then turn to some more practical issues. First, there is a discussion, illustrated by two examples, of the approach to the representation of individuals. This knowledge representation issue is crucial in many applications. For example, in machine learning, there are individuals about which something has to be learned (for example, a classification or a regression value) and these individuals can have complex internal structure for which the full power of the higher-order logic is needed to represent them. In particular, it is often necessary to use lists, sets and multisets, all of which are provided directly by the logic. The second practical issue is the technique of programming with abstractions which is illustrated with some examples. This technique provides an elegant and convenient method of processing sets and multisets.

The last section sets the contributions of this paper in the context of previous and current research in the use of higher-order logics for computation. In particular, I discuss briefly three higher-order declarative programming languages Haskell [9], λProlog [15], and Curry [8]. The paper ends with some suggestions for future research in higher-order computational logic.

2 Types

Definition. An *alphabet* consists of four sets:

1. A set $\mathfrak{T}$ of type constructors.
2. A set $\mathfrak{P}$ of parameters.
3. A set $\mathfrak{C}$ of constants.
4. A set $\mathfrak{V}$ of variables.

Each type constructor in $\mathfrak{T}$ has an arity. The set $\mathfrak{T}$ always includes the type constructors $\mathit{1}$ and Ω both of arity 0. $\mathit{1}$ is the type of some distinguished singleton set and Ω is the type of the booleans. The set $\mathfrak{P}$ is denumerable (that is, countably infinite). Parameters are type variables and are typically denoted by a, b, c, ... Each constant in $\mathfrak{C}$ has a signature (see below). The set $\mathfrak{V}$ is also denumerable. Variables are typically denoted by x, y, z, ... For any particular application, the alphabet is assumed fixed and all definitions are relative to the alphabet.

Types are built up from the set of type constructors and the set of parameters, using the symbols $\rightarrow$ and $\times$.

Definition. A *type* is defined inductively as follows.

1. Each parameter in $\mathfrak{P}$ is a type.
2. If T is a type constructor in $\mathfrak{T}$ of arity k and $\alpha_1, \ldots, \alpha_k$ are types, then $T\ \alpha_1 \ldots \alpha_k$ is a type. (For $k = 0$, this reduces to a type constructor of arity 0 being a type.)
3. If α and β are types, then $\alpha \rightarrow \beta$ is a type.
4. If $\alpha_1, \ldots, \alpha_n$ are types, then $\alpha_1 \times \ldots \times \alpha_n$ is a type. (For $n = 0$, this reduces to $\mathit{1}$ being a type.)

$\mathfrak{S}$ denotes the set of all types obtained from an alphabet ($\mathfrak{S}$ for 'sort'). The symbol $\rightarrow$ is right associative, so that $\alpha \rightarrow \beta \rightarrow \gamma$ means $\alpha \rightarrow (\beta \rightarrow \gamma)$.

Definition. A type is *closed* if it contains no parameters.

Notation. $\mathfrak{S}^c$ denotes the set of all closed types obtained from an alphabet.

Note that $\mathfrak{S}^c$ is non-empty, since $\mathit{1}$, $\Omega \in \mathfrak{S}^c$.

Example. In practical applications of the logic, a variety of types is needed. For example, declarative programming languages typically admit the following types (which are nullary type constructors): $\mathit{1}$, Ω, *Int* (the type of integers), *Float* (the type of floating-point numbers), *Char* (the type of characters), and *String* (the type of strings). Another useful type in applications is *Nat* (the type of natural numbers).

Other useful type constructors are those used to define lists, trees, and so on. In the logic, *List* denotes the (unary) list type constructor. Thus, if α is a type, then *List* α is the type of lists whose elements have type α.

Use will be made later of the concept of one type being more general than another.

Definition. Let α and β be types. Then α is *more general than* β if there exists a type substitution ξ such that $\beta = \alpha\xi$.

Note that "more general than" includes "equal to", since ξ can be the identity substitution.

Example. Let $\alpha = (\mathit{List}\ a) \times \Omega$ and $\beta = (\mathit{List}\ \mathit{Int}) \times \Omega$. Then α is more general than β, since $\beta = \alpha\xi$, where $\xi = \{a/\mathit{Int}\}$.

3 Terms

Definition. A *signature* is the declared type for a constant.

Notation. The fact that a constant C has signature α is sometimes denoted by $C : \alpha$.

I distinguish two different kinds of constants: *data constructors* and *functions*. 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. As examples, the constants $\top$ (true) and $\bot$ (false) are data constructors, as is each integer, floating-point number, and character. The constant $:$ (cons) used to construct lists is a data constructor. The constants $\subseteq$ and *concatenate* introduced in examples below are both functions.

The set $\mathfrak{C}$ always includes the following constants (where a is a parameter).

1. $()$, having signature $\mathit{1}$.
2. $=$, having signature $a \to a \to \Omega$.
3. $\top$ and $\bot$, having signature Ω.
4. $\neg$, having signature $\Omega \to \Omega$.
5. $\wedge$, $\vee$, $\longrightarrow$, $\longleftarrow$, and $\longleftrightarrow$, having signature $\Omega \to \Omega \to \Omega$.
6. Σ and Π, having signature $(a \to \Omega) \to \Omega$.

The intended meaning of $=$ is identity (that is, $=\ x\ y$ is $\top$ iff x and y are identical), the intended meaning of $\top$ is true, the intended meaning of $\bot$ is false, and the intended meanings of the connectives $\neg$, $\wedge$, $\vee$, $\longrightarrow$, $\longleftarrow$, and $\longleftrightarrow$ are as usual. The intended meaning of Σ is that it is true iff its argument is a predicate that is true on *some* domain element, and the intended meaning of Π is that it is true iff its argument is a predicate that is true on *every* domain element.

Note. In this paper, the equality symbol '=' is overloaded. On the one hand, '=' is a constant in the alphabet of a higher-order logic. On the other hand, '=' is a symbol of the informal meta-language in which the paper is written with the intended meaning of identity. The meaning of any occurrence of the symbol '=' will always be clear from the context. Equality is nearly always written infix.

Data constructors always have a signature of the form $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$, where T is a type constructor of arity k, $a_1, \ldots, a_k$ are distinct parameters, and all the parameters appearing in $\sigma_1, \ldots, \sigma_n$ occur among $a_1, \ldots, a_k$ ($n \geq 0,\ \ k \geq 0$). Furthermore, for each type constructor T, I assume that there does exist at least one data constructor having a signature of the form $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$.

Example. The data constructors for constructing lists are $[]$ having signature $\mathit{List}\ a$ and $:$ having signature $a \to \mathit{List}\ a \to \mathit{List}\ a$, where $:$ is usually written infix. $[]$ represents the empty list. The term $s : t$ represents the list with head s and tail t. Thus $4 : 5 : 6 : []$ represents the list $[4, 5, 6]$.

The next task is to define the central concept of a term. In the non-polymorphic case, a simple inductive definition suffices. But the polymorphic case is more complicated since, when putting terms together to make larger terms, it is generally necessary to solve a system of equations and these equations depend upon the relative types of free variables in the component terms. The effect of this is that to define a term one has to define simultaneously its type, and its set of free variables and their relative types.

Definition. A *term*, together with its type, and its set of free variables and their relative types, is defined inductively as follows.

1. Each variable x in $\mathfrak{V}$ is a term of type a, where a is a parameter.
   The variable x is free with relative type a in x.
2. Each constant C in $\mathfrak{C}$, where C has signature α, is a term of type α.
3. (Abstraction) If t is a term of type β and x a variable in $\mathfrak{V}$, then $\lambda x.t$ is a term of type $\alpha \to \beta$, if x is free with relative type α in t, or type $a \to \beta$, where a is a new parameter, otherwise.

A variable other than x is free with relative type σ in $\lambda x.t$ if the variable is free with relative type σ in t.

4. (Application) If s is a term of type $\alpha \to \beta$ and t a term of type γ[1] such that the equation
$\alpha = \gamma$,
augmented with equations of the form
$\rho = \delta$,
for each variable that is free with relative type ρ in s and is also free with relative type δ in t, have a most general unifier θ, then $(s\ t)$ is a term of type $\beta\theta$.
A variable is free with relative type $\sigma\theta$ in $(s\ t)$ if the variable is free with relative type σ in s or t.
5. (Tupling) If $t_1, \dots, t_n$ are terms of type $\alpha_1, \dots, \alpha_n$[2], respectively, such that the set of equations of the form
$\rho_{i_1} = \rho_{i_2} = \ldots = \rho_{i_k}$,
for each variable that is free with relative type ρ_{i_j} in the term t_{i_j} ($j = 1, \dots, k$ and $k > 1$), have a most general unifier θ, then $(t_1, \dots, t_n)$[3] is a term of type $\alpha_1\theta \times \ldots \times \alpha_n\theta$.
A variable is free with relative type $\sigma\theta$ in $(t_1, \dots, t_n)$ if the variable is free with relative type σ in t_j, for some $j \in \{1, \dots, n\}$.

The type substitution θ in Parts 4 and 5 of the definition is called the *associated* mgu.

Notation. $\mathfrak{L}$ denotes the set of all terms obtained from an alphabet and is called the *language* given by the alphabet.

Definition. A term is *closed* if it contains no free variables.

Example. Let M be a nullary type, and $A : M$ and $concatenate : List\ a \times List\ a \to List\ a$ be constants. Recall that $[] : List\ a$ and $(:) : a \to List\ a \to List\ a$ are the data constructors for lists. I will show that $(concatenate\ ([], [A]))$ is a term. For this, $([], [A])$ must be shown to be a term, which leads to the consideration of $[]$ and $[A]$. Now $[]$ is a term of type $List\ a$, by Part 2 of the definition of a term. By Parts 2 and 4, $(:\ A)$ is a term of type $List\ M \to List\ M$, where along the way the equation $a = M$ is solved with the mgu $\{a/M\}$. Then $((:\ A)\ [])$ (that is, $[A]$) is a term of type $List\ M$ by Part 4, where the equation $List\ M = List\ a$ is solved. By Part 5, it follows that $([], [A])$ is a term of type

[1] Without loss of generality, one can suppose that the parameters in $\alpha \to \beta$, taken together with the parameters in the relative types of the free variables in s, and the parameters in γ, taken together with the parameters in the relative types of the free variables in t, are standardised apart.

[2] Without loss of generality, one can suppose that the parameters of each α_i, taken together with the parameters in the relative types of the free variables of t_i, are standardised apart.

[3] If $n = 1$, (t_1) is defined to be t_1. If $n = 0$, the term obtained is the empty tuple, $()$, which is a term of type $\mathit{1}$.

$List\ a \times List\ M$. Finally, by Part 4 again, $(concatenate\ ([], [A]))$ is a term of type $List\ M$, where the equation to be solved is $List\ a \times List\ a = List\ a \times List\ M$ whose mgu is $\{a/M\}$.

Example. Consider the constants $append : List\ a \to List\ a \to List\ a \to \Omega$ and $process : List\ a \to List\ a$. I will show that $(((append\ x)\ [])\ (process\ x))$ is a term. First, the variable x is a term of type b, where the parameter is chosen to avoid a clash in the next step. Then $(append\ x)$ is a term of type $List\ a \to List\ a \to \Omega$, for which the equation solved is $List\ a = b$. Next $((append\ x)\ [])$ is a term of type $List\ a \to \Omega$ and x has relative type $List\ a$ in $((append\ x)\ [])$. Now consider $(process\ x)$, for which the constituent parts are $process$ of type $List\ c \to List\ c$ and the variable x of type d. Thus $(process\ x)$ is a term of type $List\ c$ and x has relative type $List\ c$ in $(process\ x)$. Finally, we have to apply $((append\ x)\ [])$ to the term $(process\ x)$. For this, by Part 4, there are two equations. These are $List\ a = List\ c$, coming from the top-level types, and $List\ a = List\ c$, coming from the free variable x in each of the components. These equations have the mgu $\{c/a\}$. Thus $(((append\ x)\ [])\ (process\ x))$ is a term of type Ω.

Notation. 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 accord with the intended meaning of Σ and Π). In a higher-order logic, one may identify sets and predicates – the actual identification is between a set and its characteristic function which is a predicate. Thus, if t is of type Ω, the abstraction $\lambda x.t$ may be written as $\{x \mid t\}$ if it is intended to emphasise that its intended meaning is a set. The notation $\{\}$ means $\{x \mid \bot\}$. The notation $s \in t$ means $(t\ s)$, where t has type $\alpha \to \Omega$ and s has type α, for some α. Furthermore, notwithstanding the fact that sets are mathematically identified with predicates, it is sometimes convenient to maintain an informal distinction between sets (as “collections of objects”) and predicates. For this reason, the notation $\{\alpha\}$ is introduced as a synonym for the type $\alpha \to \Omega$. The term $(s\ t)$ is often written as simply $s\ t$, using juxtaposition to denote application. Juxtaposition is left associative, so that $r\ s\ t$ means $((r\ s)\ t)$. Thus $(((append\ x)\ [])\ (process\ x))$ can be written more simply as $append\ x\ []\ (process\ x)$.

For the later definition of a statement, the concept of one term being type-weaker than another will be needed.

Definition. Let s be a term of type σ and t a term of type τ. Then s is *type-weaker* than t, denoted $s \precsim t$, if there exists a type substitution γ such that $\tau = \sigma\gamma$, every free variable in s is a free variable in t, and, if the relative type of a free variable in s is δ, then the relative type of this free variable in t is $\delta\gamma$.

Example. Let $s = y$ and $t = f\ x\ y$, where $f : M \to N \to N$. Let $\gamma = \{b/N\}$, where b is the type of y. Then $b\gamma = N$ and s is type-weaker than t.

Example. Let $s = y$ and $t = f\ y\ x$, where $f : M \to N \to N$. Then s is not type-weaker than t since no suitable γ exists.

Definition. Two terms s and t are *type-equivalent*, denoted $s \approx t$, if they have the same types, the same set of free variables, and, for every free variable x in s and t, x has the same relative type in s as it has in t (up to variants of types).

4 Schemas

The above definition of terms is sufficient for many knowledge representation tasks. However, it turns out that a more flexible notation on top of the term syntax is needed in some applications, the prime example of which is programming languages. Here is an example to motivate the ideas.

Example. Consider the programming problem of writing some code to implement the subset relation between sets. Here is a possible definition of the function $\subseteq : (a \to \Omega) \to (a \to \Omega) \to \Omega$, which is written infix.

$$\{\} \subseteq s = \top$$
$$\{x \mid x = u\} \subseteq s = u \in s$$
$$\{x \mid \boldsymbol{u} \vee \boldsymbol{v}\} \subseteq s = (\{x \mid \boldsymbol{u}\} \subseteq s) \wedge (\{x \mid \boldsymbol{v}\} \subseteq s).$$

At first sight, all these equations look like terms. However, closer inspection of the third equation reveals that $\boldsymbol{u}$ and $\boldsymbol{v}$ there are not ordinary variables. Intuitively, these are intended to stand for expressions (possibly) containing x as a free variable. Technically, they are syntactical variables in the meta-language that range over object-level terms. Syntactical variables are distinguished from (ordinary) variables by writing them in bold font.

This use of syntactical variables is common in the presentation of axioms for logics. The third equation in the above example is thus a schema rather than a term in which the syntactical variables $\boldsymbol{u}$ and $\boldsymbol{v}$ range over object-level terms.

The definition of a schema can be obtained from the previous one for a term by also allowing syntactical variables to appear. The details are given in [13].

A schema can be reified to obtain a term.

Definition. A *reifier* is a finite set of the form $\{\boldsymbol{x}_1/t_1, \ldots, \boldsymbol{x}_n/t_n\}$, where each $\boldsymbol{x}_i$ is a syntactical variable, each t_i is a term, and $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ are distinct.

Definition. Let s be a schema whose syntactical variables are included in $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ and $\Psi = \{\boldsymbol{x}_1/t_1, \ldots, \boldsymbol{x}_n/t_n\}$ a reifier. Then $s\Psi$, the *reification* of s by Ψ, is the expression obtained from s by replacing each occurrence of a syntactical variable $\boldsymbol{x}_i$ in s by the term t_i, for $i = 1, \ldots, n$.

Under certain natural conditions discussed in [13], $s\Psi$ is a term. The concepts of type-weaker and type-equivalence for schemas will be needed.

Definition. Let s be a schema of type σ and t a schema of type τ. Then s is *type-weaker* than t, denoted $s \precsim t$, if there exists a type substitution γ such that $\tau = \sigma\gamma$, every free variable in s is a free variable in t and, if the relative type of a free variable in s is δ, then the relative type of this free variable in t is $\delta\gamma$, and every syntactical variable in s is a syntactical variable in t and, if the relative type of a syntactical variable in s is ρ, then the relative type of this syntactical variable in t is $\rho\gamma$.

Definition. Two schemas s and t are *type-equivalent*, denoted $s \approx t$, if they have the same types, the same set of free variables and, for every free variable x in s and t, x has the same relative type in s as it has in t, and the same set of syntactical variables and, for every syntactical variable $\boldsymbol{x}$ in s and t, $\boldsymbol{x}$ has the same relative type in s as it has in t (up to variants of types).

5 Statements and Statement Schemas

Next the definition of a class of terms (and schemas) that can serve as statements in declarative programming languages whose programs are equational theories is presented.

Definition. A *statement* is a term of the form $h = b$, where h has the form $f\ t_1 \ldots t_n$, $n \geq 0$, for some function f, and b is type-weaker than h.

The term h is called the *head* and the term b is called the *body* of the statement. The statement is said to be *about* f.

Example. Consider the function $append : List\ a \times List\ a \times List\ a \to \Omega$. Then the following term is a statement about $append$.

$$\begin{aligned} append\ (u, v, w) \;&=\; (u = [] \;\wedge\; v = w) \vee \\ &\qquad \exists r.\exists x.\exists y.(u = r : x \;\wedge\; w = r : y \;\wedge\; append\ (x, v, y)). \end{aligned}$$

The head has type Ω and the free variables u, v, and w have relative type $List\ a$ in the head. The body also has type Ω and its free variables u, v, and w have relative type $List\ a$ in the body. Thus the body is type-weaker than (in fact, type-equivalent to) the head.

Usually, the head and the body of a statement are type-equivalent, but this is not always the case.

Example. Consider the statement

$$concatenate\ ([], x) \;=\; x$$

about the function $concatenate : List\ a \times List\ a \to List\ a$. Here h is $concatenate\ ([], x)$ and b is x. Then h has type $List\ a$ and b has type a', for some parameters a and a'. Thus b is type-weaker than h with $\gamma = \{a'/List\ a\}$.

Schemas should also be allowed to appear in programs.

Definition. A *statement schema* is a schema of the form $h = b$, where h has the form $f\ s_1 \ldots s_n$, $n \geq 0$, for some function f, and b is type-weaker than h.

The schema h is called the *head* and the schema b is called the *body* of the statement schema. The statement schema is said to be *about* f.

Each statement is a statement schema.

Example. Consider the function $\subseteq$ studied earlier.

$$\begin{aligned}&\{\} \subseteq s = \top\\ &\{x \mid x = u\} \subseteq s = u \in s\\ &\{x \mid \boldsymbol{u} \vee \boldsymbol{v}\} \subseteq s = (\{x \mid \boldsymbol{u}\} \subseteq s) \wedge (\{x \mid \boldsymbol{v}\} \subseteq s).\end{aligned}$$

The first two are statements, while the third is a statement schema. In the first statement, the body is type-weaker than the head, but not type-equivalent to the head because of the extra variable s. For the second statement and the third statement schema, the body is type-equivalent to the head.

Example. Consider the function $powerset : (a \to \Omega) \to (a \to \Omega) \to \Omega$. The following is a statement schema in which the head is type-equivalent to the body.

$$\begin{aligned}&powerset\ \{x \mid if\ \boldsymbol{u}\ then\ v\ else\ \boldsymbol{w}\} =\\ &\qquad if\ v\ then\ powerset\ \{x \mid \boldsymbol{u} \vee \boldsymbol{w}\}\ else\ powerset\ \{x \mid \neg\boldsymbol{u} \wedge \boldsymbol{w}\}.\end{aligned}$$

The condition that the body of a statement (or statement schema) be type-weaker than the head is needed to prove the important property that run-time type checking is unnecessary in the computational model [13].

Definition. The *definition* of a function f is the collection of all statement schemas about f, together with the signature for f.

Definition. A *program* is a collection of definitions.

6 Proof Theory

I now turn to the proof-theoretic aspects of the logic. The main goal here is to define a suitable operational behaviour for programs in declarative programming languages whose programs are equational theories. Throughout this section, I assume that each definition is given in the context of some program. Two terms are said to be α-equivalent if they differ only in the names of their bound variables.

Definition. A *redex* of a term t is an occurrence of a subterm of t that is α-equivalent to either an instance of the head of a statement or an instance of a reification of the head of a statement schema.

Example. Consider the function $\subseteq$ again.

$$\begin{aligned}&\{\} \subseteq s = \top\\ &\{x \mid x = u\} \subseteq s = u \in s\\ &\{x \mid \boldsymbol{u} \vee \boldsymbol{v}\} \subseteq s = (\{x \mid \boldsymbol{u}\} \subseteq s) \wedge (\{x \mid \boldsymbol{v}\} \subseteq s).\end{aligned}$$

The term

$$(\{\} \subseteq \{D\}) \wedge (\{y \mid (y = A) \vee (y = B)\} \subseteq \{A, C, D\})$$

has two redexes. The first is

$$\{\} \subseteq \{D\},$$

which is an instance of the head of the first statement by the substitution $\{s/\{D\}\}$. The second is

$$\{y \mid (y = A) \vee (y = B)\} \subseteq \{A, C, D\},$$

which is α-equivalent to an instance of a reification of the head of the third statement schema. The reifier is $\{\boldsymbol{u}/(x = A), \boldsymbol{v}/(x = B)\}$ and the substitution is $\{s/\{A, C, D\}$.

Definition. Let $\mathfrak{L}$ be the set of terms constructed from the alphabet of a program and $\mathfrak{DS}_{\mathfrak{L}}$ the set of subterms of terms in $\mathfrak{L}$ (distinguished by their occurrence). A *selection rule* S is a function from $\mathfrak{L}$ to the power set of $\mathfrak{DS}_{\mathfrak{L}}$ satisfying the following condition: if t is a term in $\mathfrak{L}$, then $S(t)$ is a subset of the set of outermost redexes in t.

A redex is *outermost* if it is not a (proper) subterm of another redex. Typical selection rules are the parallel-outermost selection rule for which all outermost redexes are selected and the leftmost selection rule in which the leftmost outermost redex is selected. The choice of using outermost redexes is motivated by the desire for evaluation strategy to be lazy.

Definition. A term s is obtained from a term t by a *computation step* using the selection rule S if the following conditions are satisfied:

1. $S(t)$ is a non-empty set, $\{r_i\}$, say.
2. For each i, the redex r_i is α-equivalent to either an instance $h_i\theta$ of the head of a statement $h_i = b_i$ or to an instance of a reification $h'_i\Psi\theta$ of the head of a statement schema $h'_i = b'_i$.
3. s is the term obtained from t by replacing, for each i, the redex r_i by $b_i\theta$ or $b'_i\Psi\theta$, respectively.

Note that a computation step is essentially a (multiple) application of the inference rule of type theory (Rule R) [1, p. 164]. Each computation step is decidable in the sense that there is an algorithm that can decide for a given subterm whether or not there is an instance of the head of a statement or an instance of a reification of the head of a statement schema that is α-equivalent to the subterm. This algorithm is similar to the (first-order) unification algorithm. In particular, the undecidability of higher-order unification [16] is not relevant here because α-equivalence is demanded rather than $\beta\eta$-equivalence (or β-equivalence) for higher-order unification.

Definition. A *computation* from a term t is a sequence $\{t_i\}_{i=1}^n$ of terms such that the following conditions are satisfied.

1. $t = t_1$.
2. t_{i+1} is obtained from t_i by a computation step, for $i = 1, \ldots, n-1$.

The term t_1 is called the *goal* of the computation and t_n is called the *answer*.

7 Normal Terms

Next 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. From a (higher-order) programming language perspective, basic terms are data values. 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* as shown in Section 13.

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 [12] 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 $\perp$ 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 $\perp$, 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 $\lambda x.t \in \mathfrak{L}$, 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.

Example. Assume the alphabet contains just the nullary type constructors M and N (in addition to 1 and Ω) and the data constructors $F : M \to N$ and $G : N \to M$. (Recall that each type constructor must have an associated data constructor.) Let G be the default data constructor for M. Then there are no *closed* terms of type M and hence there is no default term of type M.

However, if it exists, one can show that the default term for each closed type is unique.

Proposition 7.1. *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*.

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

Then $(\mathit{Circle}\ 7.5)$ 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 $\bot$ 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 $\alpha_1 \times \ldots \times \alpha_n$.

Proposition 7.2. $\mathfrak{D} \subseteq \mathfrak{N}$.

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 $\mathfrak{N}_\alpha = \{t \in \mathfrak{N} \mid t$ 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 necessarily disjoint. For example, if the alphabet includes *List*, then $[] \in \mathfrak{N}_{\mathit{List}\ \alpha}$, for each closed type α.

8 An Equivalence Relation on Normal Terms

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

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

and

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

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. $\alpha = \alpha_1 \times \ldots \times \alpha_n$, 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 8.1. *For each $\alpha \in \mathfrak{S}^c$, $\equiv|_{\mathfrak{N}_\alpha}$ is an equivalence relation on $\mathfrak{N}_\alpha$.*

Later I will need the following concept.

Definition. Let t be $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else} \ldots \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{N}_{\beta \to \gamma}$ and $r \in \mathfrak{N}_\beta$. Then $V(t\ r)$ is defined by

$$V(t\ r) = \begin{cases} s_i \text{ if } r \equiv t_i \text{ and } r \not\equiv t_k (\forall k < i) \\ s_0 \text{ if } r \not\equiv t_i (\forall i) \end{cases}$$

Intuitively, $V(t\ r)$ is the "value" returned when t is applied to r.

9 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. $\alpha = \alpha_1 \times \ldots \times \alpha_n$, 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 9.1. *For each $\alpha \in \mathfrak{S}^c$, $<|_{\mathfrak{N}_\alpha}$ is a (strict) total order on $\mathfrak{N}_\alpha$.*

10 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 10.1. $\mathfrak{D} \subseteq \mathfrak{B} \subseteq \mathfrak{N}$.

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

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

Proposition 10.2. *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 10.3. *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 10.4. *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.

11 A Metric on Basic Terms

For a number of reasons, it is important to have a metric defined on basic terms. For example, in instance-based learning, such a metric is needed to determine those terms that are "nearby" some given term [14, Ch.8]. Thus I give now the definition of a suitable function d from $\mathfrak{B} \times \mathfrak{B}$ into $\mathbb{R}$, where $\mathbb{R}$ denotes the set of real numbers.

The definition of this function depends upon some given functions ρ and φ. The real-valued function ρ is defined on the product of the set of data constructors with itself and is assumed to satisfy the following conditions:

1. For each type constructor $T \in \mathfrak{T}$, ρ is a metric on the set of data constructors associated with T.
2. For the set of data constructors associated with a type constructor for which there is at least one data constructor of arity > 0, ρ is the discrete metric.

For example, the type constructor *List* has two data constructors $:$ (of arity > 0) and $[]$, and so $\rho([], :) = 1$. In contrast, *Nat* has only nullary data constructors and hence the second condition does not apply. As will become apparent in the definition below of the function d, it is only necessary to be concerned about the value of $\rho(C, D)$ for those data constructors C and D that are associated with the *same* type constructor. The next example gives typical choices for ρ.

Example. For the types *1* and Ω, ρ could be the discrete metric. For the types *Nat*, *Int*, and *Float*, one could use $\rho(n, m) = |n - m|$. For a type constructor like *Shape* in Section 7, it is natural to employ the discrete metric on the set of data constructors $\{\mathit{Circle}, \mathit{Rectangle}\}$.

The second function φ must be a non-decreasing function from the non-negative reals into the closed interval $[0, 1]$ such that $\varphi(0) = 0$, $\varphi(x) > 0$ if $x > 0$, and $\varphi(x + y) \leq \varphi(x) + \varphi(y)$, for each x and y.

Example. Typical choices for φ could be $\varphi(x) = \frac{x}{1+x}$ or $\varphi(x) = \min\{1, x\}$.

Definition. The function $d : \mathfrak{B} \times \mathfrak{B} \to \mathbb{R}$ is defined inductively on the structure of terms in $\mathfrak{B}$ as follows. Let $s, t \in \mathfrak{B}$.

1. If $s, t \in \mathfrak{B}_\alpha$, where $\alpha = T\ \alpha_1 \ldots \alpha_k$, for some $T, \alpha_1, \ldots, \alpha_k$, then

$$d(s, t) = \begin{cases} \rho(C, D) & \text{if } C \neq D \\ (1/2) \max\limits_{i=1,\ldots,n} \varphi(d(s_i, t_i)) & \text{otherwise} \end{cases}$$

   where s is $C\ s_1 \ldots s_n$ and t is $D\ t_1 \ldots t_m$.
2. If $s, t \in \mathfrak{B}_\alpha$, where $\alpha = \beta \to \gamma$, for some β, γ, then

$$d(s, t) = \sum_{r \in \mathfrak{B}_\beta} d(V(s\ r), V(t\ r)).$$

3. If $s, t \in \mathfrak{B}_\alpha$, where $\alpha = \alpha_1 \times \ldots \times \alpha_n$, for some $\alpha_1, \ldots, \alpha_n$, then
$$d(s,t) = \sum_{i=1}^{n} d(s_i, t_i),$$
where s is $(s_1, \ldots, s_n)$ and t is $(t_1, \ldots, t_n)$.
4. If there does not exist $\alpha \in \mathfrak{S}^c$ such that $s, t \in \mathfrak{B}_\alpha$, then $d(s,t) = 1$.

In Part 1 of the definition, if $n = 0$, then $\max_{i=1,\ldots,n} \varphi(d(s_i, t_i)) = 0$. The purpose of the function φ is to scale the values of the $d(s_i, t_i)$ so that they lie in the interval $[0, 1]$. Thus $\max_{i=1,\ldots,n} \varphi(d(s_i, t_i)) \leq 1$. The factor of $1/2$ means that the greater the "depth" to which s and t agree, the smaller will be their distance apart. So for lists, for example, the longer the prefix on which two lists agree, the smaller will be their distance apart.

In Part 2 of the definition, the sum $\sum_{r \in \mathfrak{B}_\beta} d(V(s\ r), V(t\ r))$ is finite since s and t differ on at most finitely many r in $\mathfrak{B}_\beta$. In the case of sets, $\sum_{r \in \mathfrak{B}_\beta} d(V(s\ r), V(t\ r))$ is the cardinality of the symmetric difference of the sets s and t (assuming that ρ is the discrete metric for Ω).

It should be clear that the definition of d does not depend on the choice of α such that $s, t \in \mathfrak{B}_\alpha$. (There may be more than one such α.) What is important is only whether α has the form $T\ \alpha_1 \ldots \alpha_k$, $\beta \to \gamma$, or $\alpha_1 \times \ldots \times \alpha_n$, and this is invariant.

The definition given above for d is, of course, only one of a number of possibilities. For example, one could use instead the Euclidean form of the metric (with the square root of the sum of the squares) in Part 3 or a more specialised metric for lists in Part 1. For a particular instance-based learning application, such fine tuning would be almost certainly needed. These variant definitions for d are likely to share the following properties of d; in any case, the proofs of these properties for d show the way for proving similar properties for the variants.

Example. Suppose that ρ is the metric given by $\rho(n, m) = |n - m|$ for *Int* and *Float*. Then $d(42, 42) = 0$, $d(21, 42) = 21$ and $d(42.1, 42.2) = 0.1$.

Example. Suppose that ρ is the discrete metric for *List* and also for the set of data constructors $\{A, B, C, D\}$ (of some unnamed type) and that $\varphi(x) = \frac{x}{1+x}$. Let s be the list $[A, B, C]$ and t the list $[A, D]$. (See Figure 1.) Then
$$\begin{aligned}
d(s,t) &= d([A,B,C],[A,D]) \\
&= \frac{1}{2}\max\{\varphi(d(A,A)), \varphi(d([B,C],[D]))\} \\
&= \frac{1}{2}\varphi(d([B,C],[D])) \\
&= \frac{1}{2}\varphi(\frac{1}{2}\max\{\varphi(d(B,D)), \varphi(d([C],[]))\}) \\
&= \frac{1}{2}\varphi(\frac{1}{2}\max\{\varphi(1), \varphi(1)\}) \\
&= \frac{1}{2}\varphi(\frac{1}{2}\cdot\frac{1}{2})
\end{aligned}$$

$$= \frac{1}{2} \cdot \frac{\frac{1}{4}}{1 + \frac{1}{4}}$$
$$= \frac{1}{10}.$$

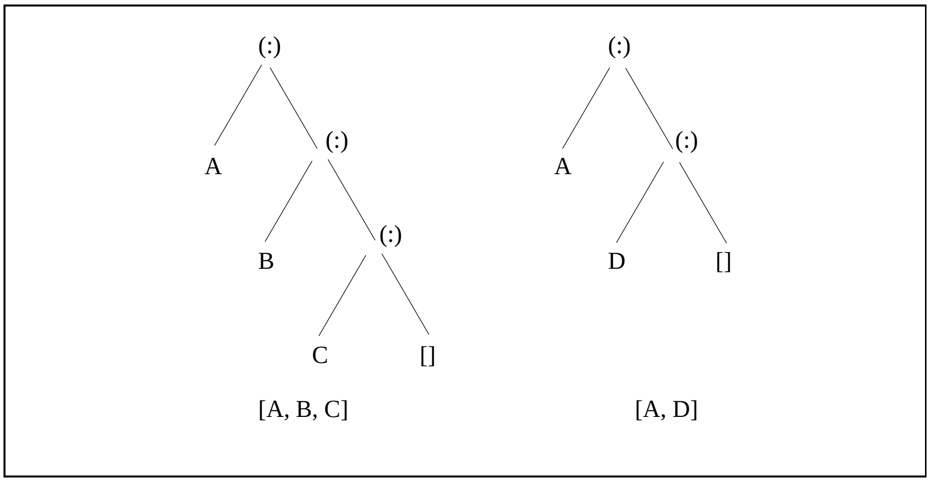

Fig. 1. Two lists

Example. Let $BTree$ be a unary type constructor, and $Null : BTree\ a$ and $BNode : BTree\ a \to a \to BTree\ a \to BTree\ a$ be data constructors. Here $BTree\ a$ is the type of binary trees, $Null$ represents the empty binary tree, and $BNode$ is used to represent non-empty binary trees. Let $A, B, C, D : M$ be data constructors. Suppose that ρ is the discrete metric on M and $\varphi(x) = \frac{x}{1+x}$. Let s be

$$BNode\ (BNode\ Null\ A\ Null)\ B\ (BNode\ Null\ C\ (BNode\ Null\ D\ Null)),$$

a binary tree of type $BTree\ M$, and t be

$$BNode\ (BNode\ Null\ A\ Null)\ B\ (BNode\ Null\ D\ Null).$$

(See Figure 2.) Then

$$\begin{aligned} d(s,t) &= \frac{1}{2} \max\{\varphi(d(BNode\ Null\ A\ Null, BNode\ Null\ A\ Null)), \varphi(d(B,B)), \\ &\qquad \varphi(d(BNode\ Null\ C\ (BNode\ Null\ D\ Null), BNode\ Null\ D\ Null))\} \\ &= \frac{1}{2}\varphi(d(BNode\ Null\ C\ (BNode\ Null\ D\ Null), BNode\ Null\ D\ Null)) \end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{2}\varphi(\frac{1}{2}\max\{\varphi(d(\mathit{Null}, \mathit{Null})), \varphi(d(C, D)), \\
&\qquad\qquad\qquad\qquad\qquad\qquad \varphi(d(\mathit{BNode\ Null\ D\ Null}, \mathit{Null}))\}) \\
&= \frac{1}{2}\varphi(\frac{1}{2}\max\{\varphi(1), \varphi(1)\}) \\
&= \frac{1}{2}\cdot\varphi(\frac{1}{4}) \\
&= \frac{1}{2}\cdot\frac{\frac{1}{4}}{1+\frac{1}{4}} \\
&= \frac{1}{10}.
\end{aligned}
$$

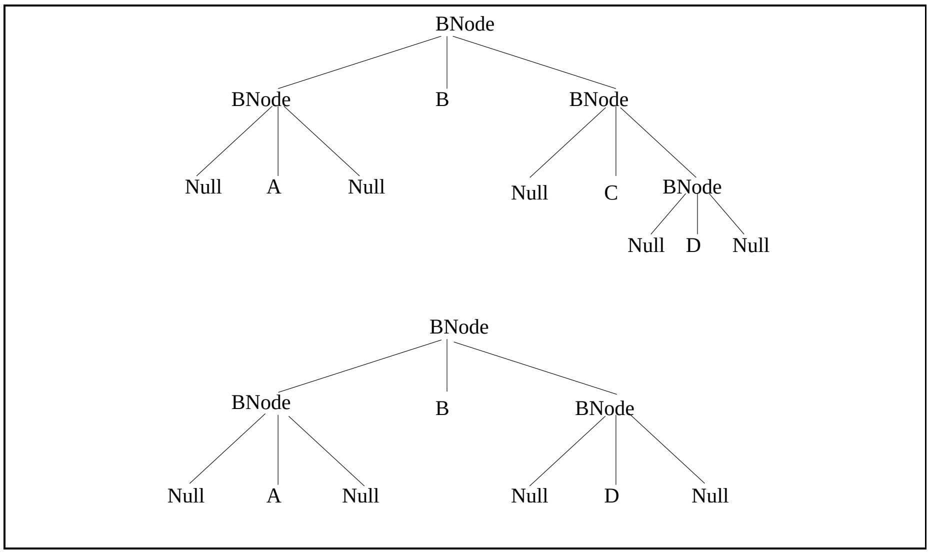

Fig. 2. Two binary trees

Notation. The basic abstraction $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ \top\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ \top\ \mathit{else}\ \bot \in \mathfrak{B}_{\beta\to\Omega}$ is a set whose elements have type more general than β and is denoted by $\{t_1, \ldots, t_n\}$.

Example. Suppose that ρ is the discrete metric for Ω. If s is the set $\{A, B, C\} \in \mathfrak{B}_{\beta\to\Omega}$ and t is the set $\{A, D\} \in \mathfrak{B}_{\beta\to\Omega}$, then $d(s,t) = \sum_{r\in\mathfrak{B}_\beta} d(V(s\ r), V(t\ r)) = 1+1+1 = 3$.

Notation. The basic abstraction $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ m_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ m_n\ \mathit{else}\ 0 \in \mathfrak{B}_{\beta\to Nat}$ is a multiset whose elements have type more general than β and is denoted by $\langle t_1, \ldots, t_1, \ldots, t_n, \ldots, t_n \rangle$, where there are m_i occurrences of t_i, for $i = 1, \ldots, n$. (That is, the number of times an element appears in the expression is its multiplicity in the multiset.) Obviously, this notation is only useful for "small" multisets.

Example. Suppose that ρ is the metric for *Nat* given by $\rho(n, m) = |n - m|$. If s is $\langle A, A, B, C, C, C \rangle \in \mathfrak{B}_{\beta \to Nat}$ and t is $\langle B, C, C, D \rangle \in \mathfrak{B}_{\beta \to Nat}$, then $d(s, t) = \sum_{r \in \mathfrak{B}_\beta} d(V(s\ r), V(t\ r)) = 2 + 1 + 1 = 4$.

Proposition 11.1. *For each $\alpha \in \mathfrak{S}^c$, $(\mathfrak{B}_\alpha, d)$ is a metric space.*

I now give a generalised definition of cardinality for basic abstractions.

Definition. Let t be $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{B}_{\beta \to \gamma}$. Then the (generalised) cardinality of t is defined as follows:

$$card(t) = \sum_{r \in \mathfrak{B}_\beta} d(V(t\ r),\ s_0).$$

The function *card* measures how much a basic abstraction deviates from being constant.

Example. Suppose that ρ is the discrete metric for Ω. If t is the set $\{A, B, C\}$, then $card(t) = 3$. That is, $card(t)$ is the cardinality of the set t.

Example. Suppose that ρ is the metric for *Nat* given by $\rho(n, m) = |n - m|$. If t is the multiset $\langle A, B, A, B, C \rangle$, then $card(t) = 2 + 2 + 1 = 5$. That is, $card(t)$ is the sum of the multiplicities of the elements of the multiset t.

12 Representation of Individuals

In this section, some practical issues concerning the representation of individuals are discussed and the ideas are illustrated with two examples.

To make the ideas more concrete, consider an inductive learning problem, in which there is some collection of individuals for which a general classification is required [14]. Training examples are available that state the class of certain individuals. The classification is given by a function from the domain of the individuals to some small finite set corresponding to the classes.

I adopt a standard approach to knowledge representation. The basic principle is that *an individual should be represented by a (closed) term*; this is referred to as the 'individuals-as-terms' approach. Thus the individuals are represented by basic terms. For a complex individual, the term will be correspondingly complex. Nevertheless, this approach has significant advantages: the representation is compact, all information about an individual is contained in one place, and the structure of the term provides strong guidance on the search for a suitable induced definition.

What types are needed to represent individuals? Typically, one needs the following: integers, floats, characters, strings, and booleans; data constructors;

tuples; sets; multisets; lists; trees; and graphs. The first group are the basic types, such as *Int*, *Float*, and Ω. Also needed are data constructors for user-defined types. For example, see the data constructors *Abloy* and *Chubb* for the nullary type constructor *Make* below. Tuples are essentially the basis of the attribute-value representation of individuals, so their utility is clear. Less commonly used elsewhere for representing individuals are sets and multisets. However, sets, especially, and multisets are basic and extremely useful data types. Other constructs needed for representing individuals include the standard data types, lists, trees, and graphs. This catalogue of data types is a rich one, and intentionally so. I advocate making a careful selection of the type which best models the application being studied.

Consider now the problem of determining whether a key in a bunch of keys can open a door. More precisely, suppose there are some 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 drug activity prediction [5], as a bunch corresponds to a molecule and a key corresponds to a conformation of a molecule, and a molecule has a certain behaviour if some conformation of it does. I make the following declarations.

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

I also make the following type synonyms.

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

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

$$opens : Bunch \rightarrow \Omega.$$

Here is a typical example.

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

For further details on this example, see [2] or [3].

As another example of knowledge representation, consider the problem of modelling a chemical molecule. The first issue is to choose a suitable type to represent a molecule. I use an undirected graph to model a molecule – an atom is a vertex in the graph and a bond is an edge. Having made this choice, suitable types are then set up for the atoms and bonds. For this, the nullary type constructor *Element*, which is the type of the (relevant) chemical elements, is first introduced. Here are the constants of type *Element*.

$$Br, C, Cl, F, H, I, N, O, S : Element.$$

I also make the following type synonyms.

$$\begin{aligned}&AtomType = Nat\\&Charge = Float\\&Atom = Element \times AtomType \times Charge\\&Bond = Nat.\end{aligned}$$

For (undirected) graphs, there is a "type constructor" *Graph* such that the type of a graph is $Graph\ v\ e$, where v is the type of information in the vertices and e is the type of information in the edges. *Graph* is defined as follows.

$$\begin{aligned}&Label = Nat\\&Graph\ v\ e = \{Label \times v\} \times \{(Label \to Nat) \times e\}.\end{aligned}$$

Here the multisets of type $Label \to Nat$ are intended to all have cardinality 2, that is, they are intended to be regarded as *unordered* pairs. Note that this definition corresponds closely to the mathematical definition of a graph: each vertex is labelled by a unique integer and each edge is uniquely labelled by the unordered pair of labels of the vertices it connects. Also it should be clear by now that *Graph* is not actually a type constructor at all; instead $Graph\ v\ e$ is simply notational sugar for the expression on the right hand side of its definition.

The type of a molecule is now obtained as an (undirected) graph whose vertices have type *Atom* and whose edges have type *Bond*. This leads to the following definition.

$$Molecule = Graph\ Atom\ Bond.$$

Here is an example molecule, called `d1`, from the mutagenesis dataset available at [17]. The notation $\langle s, t\rangle$ is used as a shorthand for the multiset that takes the value 1 on each of s and t, and is 0 elsewhere. Thus $\langle s, t\rangle$ is essentially an unordered pair.

$$
\begin{aligned}
(\{&(1,(C,22,-0.117)),(2,(C,22,-0.117)),(3,(C,22,-0.117)),\\
&(4,(C,195,-0.087)),(5,(C,195,0.013)),(6,(C,22,-0.117)),\\
&(7,(H,3,0.142)),(8,(H,3,0.143)),(9,(H,3,0.142)),\\
&(10,(H,3,0.142)),(11,(C,27,-0.087)),(12,(C,27,0.013)),\\
&(13,(C,22,-0.117)),(14,(C,22,-0.117)),(15,(H,3,0.143)),\\
&(16,(H,3,0.143)),(17,(C,22,-0.117)),(18,(C,22,-0.117)),\\
&(19,(C,22,-0.117)),(20,(C,22,-0.117)),(21,(H,3,0.142)),\\
&(22,(H,3,0.143)),(23,(H,3,0.142)),(24,(N,38,0.812)),\\
&(25,(O,40,-0.388)),(26,(O,40,-0.388))\},\\
\{&(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 1,7\rangle,1),(\langle 2,3\rangle,7),(\langle 2,8\rangle,1),\\
&(\langle 3,4\rangle,7),(\langle 3,9\rangle,1),(\langle 4,5\rangle,7),(\langle 4,11\rangle,7),(\langle 5,6\rangle,7),\\
&(\langle 5,14\rangle,7),(\langle 6,10\rangle,1),(\langle 11,12\rangle,7),(\langle 11,17\rangle,7),\\
&(\langle 12,13\rangle,7),(\langle 12,20\rangle,7),(\langle 13,14\rangle,7),(\langle 13,15\rangle,1),\\
&(\langle 14,16\rangle,1),(\langle 17,18\rangle,7),(\langle 17,21\rangle,1),(\langle 18,19\rangle,7),\\
&(\langle 18,22\rangle,1),(\langle 19,20\rangle,7),(\langle 19,24\rangle,1),(\langle 20,23\rangle,1),\\
&(\langle 24,25\rangle,2),(\langle 24,26\rangle,2)\}).
\end{aligned}
$$

Having represented the molecules, the next task is to learn a function that provides a classification of the carcinogenicity of the molecules. One way of doing this is to build, using a set of training examples, a decision tree from which the definition of the classification function can be extracted. The most important aspect of building this tree is to find suitable predicates to split the training examples. The search space of predicates is determined by the type of the individuals and the constants that appear in the corresponding alphabet. The higher-order facilities of the logic are used to structure this search space. More details can be found in [2] and [3].

13 Programming with Abstractions

In this section, the paradigm of programming with abstractions, introduced in [12], is illustrated with some examples.

But first some motivation. One approach to the problem of designing and implementing a declarative programming language that integrates the functional programming and logic programming styles is based on the observation that the functional programming language Haskell [9] is a highly successful, modern declarative programming language that can serve as the basis for such an integration. Haskell provides types, modules, higher-order programming and declarative input/output, amongst other features. With Haskell as a basis, the problem then reduces to identifying the extensions that are needed to provide the usual logic programming idioms. In this section, I illustrate how the paradigm of programming with abstractions can provide these extensions.

Consider the definitions of the functions *append*, *permute*, *delete*, and *sorted* given in Figure 3, which have been written in the relational style of logic programming. The intended meaning of *append* is that it is true iff its third argument is the concatenation of its first two arguments. The intended meaning of *permute* is that it is true iff its second argument is a permutation of its first argument. The intended meaning of *delete* is that it is true iff its third argument is the result of deleting its first argument from its second argument. The intended meaning of *sorted* is that it is true iff its argument is an increasingly ordered list of integers. As can be seen, the definition of each function has a declarative reading that respects the intended meaning.

$$\begin{aligned}
&\mathit{append} : \mathit{List}\ a \times \mathit{List}\ a \times \mathit{List}\ a \to \Omega\\
&\mathit{append}\ (u, v, w) \;=\; (u = [\,] \;\wedge\; v = w) \vee\\
&\qquad\qquad \exists r.\exists x.\exists y.(u = r : x \;\wedge\; w = r : y \;\wedge\; \mathit{append}\ (x, v, y))\\[1em]
&\mathit{permute} : \mathit{List}\ a \times \mathit{List}\ a \to \Omega\\
&\mathit{permute}\ ([\,], x) \;=\; x = [\,]\\
&\mathit{permute}\ (x : y, w) \;=\exists u.\exists v.\exists z.(w = u : v \;\wedge\; \mathit{delete}\ (u, x : y, z) \wedge \mathit{permute}\ (z, v))\\[1em]
&\mathit{delete} : a \times \mathit{List}\ a \times \mathit{List}\ a \to \Omega\\
&\mathit{delete}\ (x, [\,], y) \;=\; \bot\\
&\mathit{delete}\ (x, y : z, w) \;=\; (x = y \wedge w = z) \;\vee\; \exists v.(w = y : v \;\wedge\; \mathit{delete}\ (x, z, v))\\[1em]
&\mathit{sorted} : \mathit{List}\ \mathit{Int} \to \Omega\\
&\mathit{sorted}\ [\,] \;=\; \top\\
&\mathit{sorted}\ x : y \;=\; \mathit{if}\ \ y = [\,]\ \mathit{then}\ \top\ \mathit{else}\ \exists u.\exists v.(y = u : v \;\wedge\; x \leq u \;\wedge\; \mathit{sorted}\ y)
\end{aligned}$$

Fig. 3. List-processing functions

What extra machinery needs to added to Haskell to enable it to run the definitions in Figure 3? First, the constructor-based assumption of Haskell has to be relaxed to allow λ-abstractions and functions to appear in arguments in the heads of statements. Also, (free and bound) variables have to be allowed to appear in redexes. The most crucial idea behind these extensions is the introduction of λ-abstractions in arguments in heads and so this programming style is termed 'programming with abstractions'.

The notable feature of the definitions in Figure 3 is the presence of existential quantifiers in the bodies of the statements, so not surprisingly the key statement that makes all this work is concerned with the existential quantifier. To motivate this, consider the computation that results from the goal *append* $([1], [2], x)$. At one point in the computation, the following term is reached:

$$\exists r'.\exists x'.\exists y'.(r' = 1 \ \wedge \ x' = [] \ \wedge \ x = r' : y' \ \wedge \ \mathit{append}\ (x', [2], y').$$

An obviously desirable simplification that can be made to this term is to eliminate the local variable r' since we have a "value" (that is, 1) for it. This leads to the term

$$\exists x'.\exists y'.(x' = [] \ \wedge \ x = 1 : y' \ \wedge \ \mathit{append}\ (x', [2], y').$$

Similarly, one can eliminate x' to obtain

$$\exists y'.(x = 1 : y' \ \wedge \ \mathit{append}\ ([], [2], y').$$

After some more computation, the answer $x = [1, 2]$ results. Now the statement that makes all this possible is

$$\exists x_1.\cdots\exists x_n.(\boldsymbol{x} \wedge (x_1 = u) \wedge \boldsymbol{y}) = \exists x_2.\cdots\exists x_n.(\boldsymbol{x}\{x_1/u\} \wedge \boldsymbol{y}\{x_1/u\}),$$

which comes from the definition of $\Sigma : (a \to \Omega) \to \Omega$ and has λ-abstractions in its head.

The above ideas, plus some carefully chosen definitions, are all that are needed to allow Haskell thus extended to encompass the relational style of logic programming. The definitions of predicates look a little different to the way one would write them in, for example, Prolog. A mechanical translation of a Prolog definition into one that runs in this extended version of Haskell simply involves using the completion [11] of the Prolog definition. The definition here of *append* is essentially the completion of the Prolog version of *append*. Alternatively, one can specialise the completion to the [] and (:) cases, as has been done here for the definitions of *permute*, *delete*, and *sorted*. One procedural difference of note is that Prolog's method of returning answers one at a time via backtracking is replaced here by returning all answers together as a disjunction (or a set). Thus the goal

$$\mathit{append}\ (x, y, [1])$$

reduces to the answer

$$(x = [] \wedge y = [1]) \vee (x = [1] \wedge y = []).$$

However, the idea of programming with abstractions can be pushed much further to enable direct programming with sets, multisets and other abstractions, a facility not provided by either Haskell or Prolog. First, I deal with sets. Consider the definition of the function *likes* in Figure 4. This definition is essentially a database of facts about certain people and the sports they like.

Here are some examples of set processing. Consider first the goal

$$\{\mathit{Mary}, \mathit{Bill}\} \cap \{\mathit{Joe}, \mathit{Bill}\}.$$

$$\begin{array}{l}
\mathit{Mary}, \mathit{Bill}, \mathit{Joe}, \mathit{Fred} : \mathit{Person} \\[1ex]
\mathit{Cricket}, \mathit{Football}, \mathit{Tennis} : \mathit{Sport} \\[2ex]
\begin{aligned}
\mathit{likes} &: \mathit{Person} \times \mathit{Sport} \to \Omega \\
\mathit{likes} &= \{(\mathit{Mary}, \mathit{Cricket}), (\mathit{Mary}, \mathit{Tennis}), (\mathit{Bill}, \mathit{Cricket}), (\mathit{Bill}, \mathit{Tennis}), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Joe}, \mathit{Tennis}), (\mathit{Joe}, \mathit{Football})\}
\end{aligned}
\end{array}$$

Fig. 4. A database to illustrate set processing

Using the statement

$$s \cap t \ = \ \{x \mid (x \in s) \wedge (x \in t)\}$$

in the definition of $\cap \ : \ (a \to \Omega) \to (a \to \Omega) \to (a \to \Omega)$, one obtains

$$\{x \mid (x \in \{\mathit{Mary}, \mathit{Bill}\}) \wedge (x \in \{\mathit{Joe}, \mathit{Bill}\})\},$$

and then

$$\begin{aligned}
&\{x \mid (\mathit{if}\ x = \mathit{Mary}\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = \mathit{Bill}\ \mathit{then}\ \top\ \mathit{else}\ \bot) \wedge \\
&\qquad\qquad\qquad\qquad (\mathit{if}\ x = \mathit{Joe}\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = \mathit{Bill}\ \mathit{then}\ \top\ \mathit{else}\ \bot)\},
\end{aligned}$$

by β-reduction. After several uses of the statements

$$\begin{aligned}
(\mathit{if}\ u\ \mathit{then}\ v\ \mathit{else}\ w) \wedge t \ &= \ \mathit{if}\ u \wedge t\ \mathit{then}\ v\ \mathit{else}\ w \wedge t \\
t \wedge (\mathit{if}\ u\ \mathit{then}\ v\ \mathit{else}\ w) \ &= \ \mathit{if}\ t \wedge u\ \mathit{then}\ v\ \mathit{else}\ t \wedge w \\
\boldsymbol{v} \wedge (\boldsymbol{x} = \boldsymbol{u}) \wedge \boldsymbol{w} \ &= \ \boldsymbol{v}\{\boldsymbol{x}/\boldsymbol{u}\} \wedge (\boldsymbol{x} = \boldsymbol{u}) \wedge \boldsymbol{w}\{\boldsymbol{x}/\boldsymbol{u}\}
\end{aligned}$$

from the definition of $\wedge$, the answer $\{\mathit{Bill}\}$ is obtained. In this example, the main novel aspect compared to Haskell is the simplification that has taken place *inside* the body of a λ-abstraction.

For a second example, consider the goal

$$\{x \mid \forall y.(y \in \{\mathit{Cricket}, \mathit{Tennis}\} \longrightarrow \mathit{likes}\ (x, y))\},$$

which reduces via the steps

$$\begin{aligned}
&\{x \mid \forall y.((\mathit{if}\ y = \mathit{Cricket}\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ y = \mathit{Tennis}\ \mathit{then}\ \top\ \mathit{else}\ \bot) \longrightarrow \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{likes}\ (x, y))\} \\
&\{x \mid \forall y.(((y = \mathit{Cricket}) \vee (\mathit{if}\ y = \mathit{Tennis}\ \mathit{then}\ \top\ \mathit{else}\ \bot)) \longrightarrow \mathit{likes}\ (x, y))\} \\
&\{x \mid \forall y.((y = \mathit{Cricket}) \longrightarrow \mathit{likes}\ (x, y)) \wedge \\
&\qquad\qquad\qquad\qquad \forall y.((\mathit{if}\ y = \mathit{Tennis}\ \mathit{then}\ \top\ \mathit{else}\ \bot) \longrightarrow \mathit{likes}\ (x, y))\} \\
&\{x \mid \mathit{likes}\ (x, \mathit{Cricket}) \wedge \forall y.((\mathit{if}\ y = \mathit{Tennis}\ \mathit{then}\ \top\ \mathit{else}\ \bot) \longrightarrow \mathit{likes}\ (x, y))\},
\end{aligned}$$

and so on, to the answer

$$\{Mary, Bill\}.$$

During this computation, use is made of the statements

$$\begin{aligned}
&\forall x_1.\cdots\forall x_n.(\boldsymbol{x} \wedge (x_1 = u) \wedge \boldsymbol{y} \longrightarrow \boldsymbol{v}) \ = \\
&\qquad\qquad\qquad \forall x_2.\cdots\forall x_n.(\boldsymbol{x}\{x_1/u\} \wedge \boldsymbol{y}\{x_1/u\} \longrightarrow \boldsymbol{v}\{x_1/u\}) \\
&\forall x_1.\cdots\forall x_n.(\boldsymbol{u} \vee \boldsymbol{v} \longrightarrow \boldsymbol{t}) \ = \\
&\qquad\qquad\qquad (\forall x_1.\cdots\forall x_n.(\boldsymbol{u} \longrightarrow \boldsymbol{t})) \wedge (\forall x_1.\cdots\forall x_n.(\boldsymbol{v} \longrightarrow \boldsymbol{t})) \\
&\forall x_1.\cdots\forall x_n.((\mathit{if}\ \boldsymbol{u}\ \mathit{then}\ v\ \mathit{else}\ \boldsymbol{w}) \longrightarrow \boldsymbol{t}) \ = \\
&\qquad \mathit{if}\ v\ \mathit{then}\ \forall x_1.\cdots\forall x_n.(\boldsymbol{u} \vee \boldsymbol{w} \longrightarrow \boldsymbol{t})\ \mathit{else}\ \forall x_1.\cdots\forall x_n.(\neg\, \boldsymbol{u} \wedge \boldsymbol{w} \longrightarrow \boldsymbol{t})
\end{aligned}$$

from the definition of $\Pi : (a \to \Omega) \to \Omega$.

The example in the previous paragraph is reminiscent of list comprehension in Haskell. In fact, one could set the database up as a list of facts and then give Haskell a goal which would be a list comprehension analogous to the set goal above and obtain a list, say $[Mary, Bill]$, as the answer. Substituting lists for sets in knowledge representation is a standard device to get around the fact that few programming languages support set processing in a sophisticated way. However, sets and lists are actually significantly different types and this shows up, for example, in the different sets of transformations that each type naturally supports ([2], [3]). Consequently, I advocate a careful analysis for any particular knowledge representation task to see what types are most appropriate and also that programming languages support a full range of types, including sets and multisets.

Another point to make about the previous example is that it is an illustration of *intensional* set processing. Extensional set processing in which the descriptions of the sets manipulated are explicit representations of the collection of elements in the sets is commonly provided in programming languages. For example, it is straightforward in Haskell to set up an abstract data type for (extensional) sets using lists as the underlying representation. A language such as Java also provides various ways of implementing extensional sets. But the example above is different in that the goal is an intensional representation of a set (in fact, the set $\{Mary, Bill\}$) and the computation is able to reveal this. The ability to process intensional sets and the smooth transition between intensional and extensional set processing are major advantages of the approach to sets advocated here. Similar comments apply to programming with other kinds of abstractions such as multisets.

Consider next the problem of giving a definition for the powerset function that computes the set of all subsets of a given set. Here is the definition.

$$\begin{aligned}
&\mathit{powerset}\ :\ (a \to \Omega) \to (a \to \Omega) \to \Omega \\
&\mathit{powerset}\ \{\}\ =\ \{\{\}\}
\end{aligned}$$

$$\begin{aligned}
&\textit{powerset}\ \{x \mid \textit{if}\ \boldsymbol{u}\ \textit{then}\ v\ \textit{else}\ \boldsymbol{w}\} \ = \\
&\qquad\qquad \textit{if}\ v\ \textit{then powerset}\ \{x \mid \boldsymbol{u} \vee \boldsymbol{w}\}\ \textit{else powerset}\ \{x \mid \neg\, \boldsymbol{u} \wedge \boldsymbol{w}\} \\
&\textit{powerset}\ \{x \mid x = t\} \ = \ \{\{\}, \{t\}\} \\
&\textit{powerset}\ \{x \mid \boldsymbol{u} \vee \boldsymbol{v}\} \ = \ \{s \mid \exists l.\exists r.(l \in (\textit{powerset}\ \{x \mid \boldsymbol{u}\})) \wedge \\
&\qquad\qquad\qquad\qquad (r \in (\textit{powerset}\ \{x \mid \boldsymbol{v}\}) \wedge (s = l \cup r))\}.
\end{aligned}$$

The first and second statements cover the cases of an empty set and a 'non-empty' one, where the set is represented by a basic term. (Non-empty is quoted since $\{x \mid \textit{if}\ \boldsymbol{u}\ \textit{then}\ v\ \textit{else}\ \boldsymbol{w}\}$ can represent an empty set if both v and $\boldsymbol{w}$ are $\bot$, for example.) The third and fourth statements are needed to handle calls which arise in the second statement. They could also be used if the representation of the set is not a basic term, but has an equality or disjunction at the top level in the body. Of course, if the representation of the set does not match any of the statements, then it will have to be reduced (by using the definitions of other functions) until it does. One can see immediately that each statement in the definition is declaratively correct.

Note the analogy between set processing as illustrated by *powerset* and list processing in which the definition of a list-processing function is broken up into two statements – one for the empty list and one for a non-empty list. In the case of sets, it is convenient to have four cases corresponding to where the body of the set abstraction has the form $\bot$, $\textit{if}\ \boldsymbol{u}\ \textit{then}\ v\ \textit{else}\ \boldsymbol{w}$, $x = t$, or $\boldsymbol{u} \vee \boldsymbol{v}$. The third and fourth cases arise because of the richness of the set of functions on the booleans. For other kinds of abstractions typically only the first two cases arise, as is illustrated below for multisets.

As an illustration of the use of *powerset*, the goal

$$\textit{powerset}\ \{\textit{Mary}, \textit{Bill}\}$$

reduces to the answer

$$\{\{\}, \{\textit{Mary}\}, \{\textit{Bill}\}, \{\textit{Mary}, \textit{Bill}\}\}.$$

This section concludes with an illustration of multiset processing. Suppose one wants to compute the pairwise minimum $s \sqcap t$ of two multisets s and t, where $(s \sqcap t)\ x = \textit{min}\ (s\ x)\ (t\ x)$. Now recall that a multiset is represented by a basic term, that is, an abstraction of the form

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

where the type of each s_i is *Nat*. In particular, the empty multiset is represented by the abstraction $\lambda x.0$.

How can one compute the function $\sqcap$? The idea is to consider two cases: one in which the first argument to $\sqcap$ is the empty multiset and one in which it is not. This leads to the following definition.

$$\sqcap : (a \to Nat) \to (a \to Nat) \to (a \to Nat)$$
$$\lambda x.0 \ \sqcap \ m \ = \ \lambda x.0$$
$$(\lambda x.if \ x = t \ then \ v \ else \ \boldsymbol{w}) \ \sqcap \ m \ =$$
$$\lambda x.if \ x = t \ then \ min \ v \ (m \ t) \ else \ (\lambda x.\boldsymbol{w} \ \sqcap \ m) \ x.$$

The first statement just states that the pairwise minimum of the empty multiset and any multiset is the empty multiset. The second statement is the recursive case in which the minimum of the multiplicity of the first item in the first argument and its multiplicity in the second argument is computed and then the rest of the items in the first argument are considered. Note once again the similarity with the definitions of many list-processing functions that consist of a statement for the empty list and one for a non-empty list.

As an illustration of the use of $\sqcap$, the goal

$$(\lambda x.if \ x = A \ then \ 42 \ else \ if \ x = B \ then \ 21 \ else \ 0) \sqcap$$
$$(\lambda x.if \ x = A \ then \ 16 \ else \ if \ x = C \ then \ 4 \ else \ 0)$$

reduces to the answer

$$\lambda x.if \ x = A \ then \ 16 \ else \ 0.$$

14 Discussion

At this point, I summarise what has been achieved and put the developments into a wider context.

I started from the position that higher-order logic provides a suitable foundation for computational logic. A particular higher-order logic, based on the simple theory of types, was then presented. This logic is suitable for use as a basis for declarative programming languages. Next the issue of knowledge representation was discussed and a suitable class of terms, the basic terms, was identified as appropriate for representing individuals. The set of basic terms is a metric space in a natural way and hence provides a context in which metric-based machine learning can take place. The approach to representation was then illustrated with a couple of applications that arise in machine learning. Finally, the technique of programming with abstractions was illustrated with some examples. In this development, the higher-order nature of the logic was essential: as a foundation for the functional component of declarative programming languages, since functions can take other functions as arguments; in the use of the logic for knowledge representation, since sets and similar abstractions are needed; and for programming with abstractions, since abstractions can appear as arguments in function definitions.

However, the advantages of using higher-order logic for computational logic have been advocated by others for at least the last 30 years. Here I remark on some of this work that is most relevant to the present paper.

First, the functional programming community has used higher-order functions from the very beginning. The latest versions of functional languages, such as Haskell98 [9], show the power and elegance of higher-order functions, as well as related features such as strong type systems. Of course, the traditional foundation for functional programming languages has been the λ-calculus, rather than a higher-order *logic*. However, it is possible to regard functional programs as equational theories in a logic such as the one I have introduced here and this also provides a satisfactory semantics.

In the 1980's, higher-order programming in the logic programming community was introduced through the language λProlog [15]. The logical foundations of λProlog are provided by almost exactly the logic introduced earlier in this paper. However, a different sublogic is used for λProlog programs than the equational theories proposed here. In λProlog, program statements are higher-order hereditary Harrop formulas, a generalisation of the definite clauses used by Prolog. The language provides an elegant use of λ-terms as data structures, meta-programming facilities, universal quantification and implications in goals, amongst other features.

A long-term interest amongst researchers in declarative programming has been the goal of building integrated functional logic programming languages. A survey of progress on this problem up to 1994 can be found in [7]. Probably the best developed of these functional logic languages is the Curry language [8], which is the result of an international collaboration over the last 5 or so years. To quote from [8]: "Curry is a universal programming language aiming to amalgamate the most important declarative programming paradigms, namely functional programming and logic programming. Moreover, it also covers the most important operational principles developed in the area of integrated functional logic languages: 'residuation' and 'narrowing'. Curry combines in a seamless way features from functional programming (nested expressions, higher-order functions, lazy evaluation), logic programming (logical variables, partial data structures, built-in search), and concurrent programming (concurrent evaluation of expressions with synchronisation on logical variables). Moreover, Curry provides additional features in comparison to the pure languages (compared to functional programming: search, computing with partial information; compared to logic programming: more efficient evaluation due to the deterministic and demand-driven evaluation of functions)."

There are many other outstanding examples of systems that exploit the power of higher-order logic. For example, the HOL system [6] is an environment for interactive theorem proving in higher-order logic. Its most outstanding feature is its high degree of programmability through the meta-language ML. The system has a wide variety of uses from formalising pure mathematics to verification of industrial hardware. In addition, there are at least a dozen other systems related to HOL. On the theoretical side, much of the research in theoretical Computer Science, especially semantics, is based on the λ-calculus and hence is intrinsically higher-order in nature.

I finish with some remarks about open research issues. I believe the most important open research problem, and one that would have a major practical impact, is that of producing a widely-used functional logic programming language. Unfortunately, there is a gulf between the functional programming and logic programming communities that is holding up progress. A common programming language would do a great deal to bridge that gulf. More specifically, the logic programming community needs to make much greater use of the power of higher-order features and the related type systems. Furthermore, higher-order logic has generally been under-exploited as a knowledge representation language, with sets and related data types rarely being used simply because they are not so easily provided by first-order logic. Having sets directly available (as predicates) in higher-order logic is a big advantage.

Just over 25 years ago, Robert Kowalski started a major stream of research in Computer Science with the Horn clause subset of first-order logic. The natural and ultimate setting for that research stream is clearly higher-order logic – every contribution towards that goal would be valuable.

References

1. P.B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, 1986.
2. 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.
3. 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.
4. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
5. 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.
6. M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.
7. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
8. M. Hanus (ed.). Curry: An integrated functional logic language. Available at `http://www.informatik.uni-kiel.de/~curry`.
9. S. Peyton Jones and J. Hughes (editors). Haskell98: A non-strict purely functional language. Available at `http://haskell.org/`.
10. R. A. Kowalski. Predicate logic as a programming language. In *Information Processing 74*, pages 569–574, Stockholm, 1974. North Holland.
11. J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.
12. J.W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1999(3), March 1999.
13. J.W. Lloyd. Knowledge representation, computation, and learning in higher-order logic. Available at `http://csl.anu.edu.au/~jwl`, 2001.

14. T.M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
15. G. Nadathur and D.A. Miller. Higher-order logic programming. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *The Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 499–590. Oxford University Press, 1998.
16. D.A. Wolfram. *The Clausal Theory of Types*. Cambridge University Press, 1993.
17. Home page of Machine Learning Group, The University of York. `http://www.cs.york.ac.uk/mlg/`.