Knowledge Representation, Computation, and Learning in Higher-order Logic*

J.W. Lloyd
Computer Sciences Laboratory
Research School of Information Sciences and Engineering
Australian National University

DRAFT, August 30, 2001

Abstract

This paper contains a systematic study of the foundations of knowledge representation, computation, and learning in higher-order logic. First, a polymorphically-typed higher-order logic, whose origins can be traced back to Church's simple theory of types, is presented. A model theory and proof theory for this logic are developed and basic theorems relating these two are given. A metric space of certain closed terms, which provides a rich language for representing individuals, is then studied. Also a method of systematically constructing predicates on such individuals is given. The technique of programming with abstractions is illustrated. Major applications of the logic to declarative programming languages and machine learning are indicated.

1 Introduction

This paper contains an account of a polymorphically typed, higher-order logic mainly intended for use as a basis for declarative programming languages and learning systems. Historically, this logic can be traced back to Church's simple theory of types [Chu40], which I refer to as *type theory* in the following. The appropriate model theory for type theory was given subsequently by Henkin in [Hen50]. More recent accounts of type theory appear in [And86] and [Wol93]. Accounts of intuitionistic versions of the logic are in [Bel88] and [LS86].

In fact, the logic presented here extends type theory in that it is polymorphic and admits product types. The polymorphism introduced is a simple form of parametric polymorphism. A declaration for a polymorphic constant is understood as standing for a collection of declarations for the (monomorphic) constants which can be obtained by instantiating all parameters in the polymorphic declaration with closed types. Similarly, a polymorphic term can be regarded as standing for a collection of (monomorphic) terms. Furthermore, the model-theoretic semantics of polymorphic type theory can easily be reduced to that of (monomorphic) type theory by adopting this view of the polymorphism. Another difference compared to the original formulation of type theory is that the proof theory developed by Church (and others) is modified here to give a more direct form of equational reasoning that is better suited to the application of the logic as a foundation for declarative programming languages. The logic also effectively admits subtypes by allowing the declaration of injections from one type to another.

A metric space of closed terms that is suitable for representing individuals in diverse applications is identified. The most interesting aspect of this space of 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.

*© J.W. Lloyd, 2001

A systematic method of constructing predicates on individuals is studied. For this purpose, particular kinds of functions, called transformations, are defined and predicates are incrementally constructed by composing transformations. This leads to a, generally large, set of predicates that must be searched to find a suitable predicate for the application at hand. Basic theoretical properties of this set of predicates are provided.

The logic is suitable as a foundation for declarative programming languages, for example, the Escher[1] programming language [Llo99]. The logic also provides a suitable framework for knowledge representation in machine learning. For example, the logic has been used for knowledge representation and predicate construction in inductive learning [BGCL00], [BGCL01].

This paper is organised as follows. The next section presents the syntax, semantics and proof theory of the logic. Section 3 provides a study of a metric space of certain closed terms that represent individuals. Section 4 contains a discussion of the systematic construction of a totally ordered set of predicates on individuals. Section 5 provides some illustrations of the technique of programming with abstractions. Section 6 contains a general discussion of the importance and use of higher-order logic in various applications. The last section contains some conclusions. An appendix contains the definitions for some basic functions.

2 The Logic

This section contains an account of the syntax, semantics and proof theory of the logic.

2.1 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 *1* and Ω both of arity 0. *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 $a, b, c, \ldots$. Each constant in $\mathfrak{C}$ has a signature (see below). The set $\mathfrak{V}$ is also denumerable. Variables are typically denoted by $x, y, z, \ldots$. 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 $\alpha_1 \times \cdots \times \alpha_n$ is a type. (For $n = 0$, this reduces to *1* being a type.)

[1] M.C.Escher® is a registered trademark of Cordon Art B.V., Baarn, Nederland. Used by permission. All rights reserved.

$\mathfrak{S}$ 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 $\alpha \rightarrow (\beta \rightarrow \gamma)$.

It is worthwhile being precise about the exact meaning of the phrase 'defined inductively' in the previous definition. First, a universe of expressions is needed. In this case, the set of all finite sequences of symbols drawn from the set $\mathfrak{T}$ of type constructors, the set $\mathfrak{P}$ of parameters, and '$\rightarrow$' and '$\times$' will serve. Then the definition states that $\mathfrak{S}$ is the intersection of all sets $\mathfrak{X}$ of expressions such that the following conditions are satisfied.

1. Each parameter in $\mathfrak{P}$ is in $\mathfrak{X}$.
2. If $\alpha_1, \ldots, \alpha_k \in \mathfrak{X}$ and T is a type constructor in $\mathfrak{T}$ of arity k, then $T\ \alpha_1 \ldots \alpha_k \in \mathfrak{X}$.
3. If $\alpha, \beta \in \mathfrak{X}$, then $\alpha \rightarrow \beta \in \mathfrak{X}$.
4. If $\alpha_1, \ldots, \alpha_n \in \mathfrak{X}$, then $\alpha_1 \times \cdots \times \alpha_n \in \mathfrak{X}$.

There is always at least one such set satisfying these conditions, namely the set of all expressions. Thus the intersection is well-defined and, furthermore, it satisfies Conditions 1 to 4, as can easily be checked. Hence $\mathfrak{S}$ is the smallest set of expressions satisfying Conditions 1 to 4 (where one set is 'smaller' than another if the former is a subset of the latter). Corresponding to this definition, there is also the following *principle of induction on the structure of types.*

Proposition 2.1. *Let $\mathfrak{X}$ be a subset of $\mathfrak{S}$ satisfying the following conditions.*

1. *Each parameter in $\mathfrak{P}$ is in $\mathfrak{X}$.*
2. *If $\alpha_1, \ldots, \alpha_k \in \mathfrak{X}$ and T is a type constructor in $\mathfrak{T}$ of arity k, then $T\ \alpha_1 \ldots \alpha_k \in \mathfrak{X}$.*
3. *If $\alpha, \beta \in \mathfrak{X}$, then $\alpha \rightarrow \beta \in \mathfrak{X}$.*
4. *If $\alpha_1, \ldots, \alpha_n \in \mathfrak{X}$, then $\alpha_1 \times \cdots \times \alpha_n \in \mathfrak{X}$.*

Then $\mathfrak{X} = \mathfrak{S}$.

Proof. Since $\mathfrak{X}$ satisfies Conditions 1 to 4 of the definition of a type and $\mathfrak{S}$ is the intersection of all such sets, it follows immediately that $\mathfrak{S} \subseteq \mathfrak{X}$. Thus $\mathfrak{X} = \mathfrak{S}$. □

The assumptions $\alpha_1, \ldots, \alpha_k \in \mathfrak{X}$ in Condition 2 of Proposition 2.1 are collectively known as the *induction hypothesis* (for that step of the proof). Similarly for Conditions 3 and 4. The majority of the proofs in this paper are induction arguments on the structure of various kinds of terms or types. The previous discussion, illustrated by the case of types, makes clear the precise basis of these arguments.

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 $\mathit{1}, \Omega \in \mathfrak{S}^c$. The next result provides a useful characterisation of $\mathfrak{S}^c$.

Proposition 2.2. *$\mathfrak{S}^c$ is the intersection of all sets $\mathfrak{Y}$ of types such that the following hold.*

1. *If $\alpha_1, \ldots, \alpha_k \in \mathfrak{Y}$ and T is a type constructor in $\mathfrak{T}$ of arity k, then $T\ \alpha_1 \ldots \alpha_k \in \mathfrak{Y}$.*
2. *If α, $\beta \in \mathfrak{Y}$, then $\alpha \rightarrow \beta \in \mathfrak{Y}$.*
3. *If $\alpha_1, \ldots, \alpha_n \in \mathfrak{Y}$, then $\alpha_1 \times \cdots \times \alpha_n \in \mathfrak{Y}$.*

Proof. First, $\mathfrak{S}^c$ is a set of types that satisfies Conditions 1, 2 and 3. Hence $\bigcap \mathfrak{Y} \subseteq \mathfrak{S}^c$.

On the other hand, let $\mathfrak{Y}$ be a set of types that satisfies Conditions 1, 2 and 3. I show that $\mathfrak{S}^c \subseteq \mathfrak{Y}$. Let $\mathfrak{X} = \{\alpha \in \mathfrak{S} \mid \alpha \text{ is closed implies that } \alpha \in \mathfrak{Y}\}$. It suffices to show that $\mathfrak{X} = \mathfrak{S}$. The proof is by induction on the structure of types.

If a is a parameter, then $a \in \mathfrak{X}$ (since a is not closed).

Suppose that $\alpha_1, \ldots, \alpha_k \in \mathfrak{X}$ and T is a type constructor in $\mathfrak{T}$ of arity k. If there exists $j \in \{1, \ldots, k\}$ such that α_j is not closed, then $T\ \alpha_1 \ldots \alpha_k$ is not closed and hence $T\ \alpha_1 \ldots \alpha_k \in \mathfrak{X}$. Otherwise, α_i is closed and so $\alpha_i \in \mathfrak{Y}$, for $i = 1, \ldots, k$. Then, according to Condition 1, $T\ \alpha_1 \ldots \alpha_k \in \mathfrak{Y}$. Thus $T\ \alpha_1 \ldots \alpha_k \in \mathfrak{X}$.

Suppose that $\alpha, \beta \in \mathfrak{X}$. If one or both of α or β is not closed, then $\alpha \to \beta$ is not closed and hence $\alpha \to \beta \in \mathfrak{X}$. Otherwise, α and β are closed and so $\alpha, \beta \in \mathfrak{Y}$. Then, according to Condition 2, $\alpha \to \beta \in \mathfrak{Y}$. Thus $\alpha \to \beta \in \mathfrak{X}$.

Suppose that $\alpha_1, \ldots, \alpha_n \in \mathfrak{X}$. If there exists $j \in \{1, \ldots, n\}$ such that α_j is not closed, then $\alpha_1 \times \cdots \times \alpha_n$ is not closed and hence $\alpha_1 \times \cdots \times \alpha_n \in \mathfrak{X}$. Otherwise, α_i is closed and so $\alpha_i \in \mathfrak{Y}$, for $i = 1, \ldots, n$. Then, according to Condition 3, $\alpha_1 \times \cdots \times \alpha_n \in \mathfrak{Y}$. Thus $\alpha_1 \times \cdots \times \alpha_n \in \mathfrak{X}$.

All four conditions in Proposition 2.1 have now been shown to hold, so that $\mathfrak{X} = \mathfrak{S}$, by the principle of induction on the structure of types. Thus $\mathfrak{S}^c \subseteq \mathfrak{Y}$. Since this is true for all such $\mathfrak{Y}$, it follows that $\mathfrak{S}^c \subseteq \bigcap \mathfrak{Y}$. Thus $\mathfrak{S}^c = \bigcap \mathfrak{Y}$. □

Using Proposition 2.2, the following *principle of induction on the structure of closed types* can be proved.

Proposition 2.3. *Let $\mathfrak{X}$ be a subset of $\mathfrak{S}^c$ satisfying the following conditions.*

1. *If $\alpha_1, \ldots, \alpha_k \in \mathfrak{X}$ and T is a type constructor in $\mathfrak{T}$ of arity k, then $T\ \alpha_1 \ldots \alpha_k \in \mathfrak{X}$.*
2. *If $\alpha, \beta \in \mathfrak{X}$, then $\alpha \to \beta \in \mathfrak{X}$.*
3. *If $\alpha_1, \ldots, \alpha_n \in \mathfrak{X}$, then $\alpha_1 \times \cdots \times \alpha_n \in \mathfrak{X}$.*

Then $\mathfrak{X} = \mathfrak{S}^c$.

Proof. Since $\mathfrak{X}$ satisfies Conditions 1 to 3 of Proposition 2.2 and $\mathfrak{S}^c$ is the intersection of all such sets, it follows immediately that $\mathfrak{S}^c \subseteq \mathfrak{X}$. Thus $\mathfrak{X} = \mathfrak{S}^c$. □

Proposition 2.2 provides a 'top-down' characterisation of $\mathfrak{S}^c$. There is also a 'bottom-up' characterisation. Let $\mathbb{N}$ denote the set of natural numbers, that is, non-negative integers.

Definition. Define $\{\mathfrak{S}^c_m\}_{m \in \mathbb{N}}$ inductively as follows:

$$\begin{aligned}
\mathfrak{S}^c_0 &= \{T \mid T \in \mathfrak{T} \text{ has arity } 0\}. \\
\mathfrak{S}^c_{m+1} &= \mathfrak{S}^c_m \cup \\
&\quad \{T\ \alpha_1 \ldots \alpha_k \mid T \in \mathfrak{T} \text{ has arity } k > 0 \text{ and } \alpha_i \in \mathfrak{S}^c_m, \text{ for } i = 1, \ldots, k\} \cup \\
&\quad \{\alpha \to \beta \mid \alpha, \beta \in \mathfrak{S}^c_m\} \cup \\
&\quad \{\alpha_1 \times \cdots \times \alpha_n \mid \alpha_i \in \mathfrak{S}^c_m, i = 1, \ldots, n, \text{ and } n \in \mathbb{N}\}.
\end{aligned}$$

Clearly, $\mathfrak{S}^c_m \subseteq \mathfrak{S}^c_{m+1}$, for $m \in \mathbb{N}$.

Proposition 2.4. $\mathfrak{S}^c = \bigcup_{m \in \mathbb{N}} \mathfrak{S}^c_m$.

Proof. First, I show that $\bigcup_{m \in \mathbb{N}} \mathfrak{S}^c_m \subseteq \mathfrak{S}^c$. To prove this, it suffices to show by induction that $\mathfrak{S}^c_m \subseteq \mathfrak{S}^c$, for $m \in \mathbb{N}$. Clearly, $\mathfrak{S}^c_0 \subseteq \mathfrak{S}^c$. Suppose next that $\mathfrak{S}^c_m \subseteq \mathfrak{S}^c$. It then follows from the definition of $\mathfrak{S}^c_{m+1}$ and the fact that $\mathfrak{S}^c$ satisfies Conditions 1, 2 and 3 in Proposition 2.2 that $\mathfrak{S}^c_{m+1} \subseteq \mathfrak{S}^c$.

Now I show that $\mathfrak{S}^c \subseteq \bigcup_{m \in \mathbb{N}} \mathfrak{S}^c_m$. Put $\bigcup_{m \in \mathbb{N}} \mathfrak{S}^c_m = \mathfrak{Y}$. It suffices to show that $\mathfrak{Y}$ satisfies Conditions 1, 2 and 3 of Proposition 2.2. Suppose that $\alpha_1, \ldots, \alpha_k \in \mathfrak{Y}$ and T is a type constructor in $\mathfrak{T}$ of arity k. Since the $\mathfrak{S}^c_m$ are increasing, there exists $p \in \mathbb{N}$ such that $\alpha_1, \ldots, \alpha_k \in \mathfrak{S}^c_p$. Hence $T\ \alpha_1 \ldots \alpha_k \in \mathfrak{S}^c_{p+1}$ and so $T\ \alpha_1 \ldots \alpha_k \in \mathfrak{Y}$. Similar arguments show that $\mathfrak{Y}$ satisfies Conditions 2 and 3. □

Proposition 2.5. *If $\mathfrak{T}$ is countable, then $\mathfrak{S}^c$ is countable.*

Proof. By Proposition 2.4, it suffices to show by induction that each $\mathfrak{S}^c_m$ is countable. Clearly, $\mathfrak{S}^c_0$ is countable. Suppose now that $\mathfrak{S}^c_m$ is countable. By the definition of $\mathfrak{S}^c_{m+1}$ and the fact that $\mathfrak{T}$ is countable, it follows easily that $\mathfrak{S}^c_{m+1}$ is also countable. □

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): *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.

Much use will be made of type substitutions.

Definition. A *type substitution* is a finite set of the form $\{a_1/\alpha_1, \ldots, a_n/\alpha_n\}$, where each a_i is a parameter, each α_i is a type, and $a_1, \ldots, a_n$ are distinct. Each element a_i/α_i is called a *binding*.

Definition. Let $\theta = \{a_1/\alpha_1, \ldots, a_n/\alpha_n\}$ be a type substitution and α a type. Then $\alpha\theta$, the *instance* of α by θ, is the type obtained from α by simultaneously replacing each occurrence of the parameter a_i in α by the type α_i $(i = 1, \ldots, n)$.

Definition. Let $\theta = \{a_1/\alpha_1, \ldots, a_m/\alpha_m\}$ and $\varphi = \{b_1/\beta_1, \ldots, b_n/\beta_n\}$ be type substitutions. Then the *composition* $\theta\varphi$ of θ and φ is the type substitution obtained from the set

$$\{a_1/\alpha_1\varphi, \ldots, a_m/\alpha_m\varphi, b_1/\beta_1, \ldots, b_n/\beta_n\}$$

by deleting any binding $a_i/\alpha_i\varphi$ for which $a_i = \alpha_i\varphi$ and deleting any binding b_j/β_j for which $b_j \in \{a_1, \ldots, a_m\}$.

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

2.2 Terms

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

Notation. The fact that a constant C has signature α is often 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 $\subseteq$, *split*, and *concatenate* introduced in examples below are all functions.

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

1. (), having signature *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 meanings of Σ and Π are given in Subsection 2.9, but informally Σ maps a predicate to $\top$ iff the predicate maps at least one element to $\top$ and Π maps a predicate to $\top$ iff the predicate maps all elements to $\top$.

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 $a_1, \ldots, a_k$ ($n \geq 0,\ k \geq 0$). The *arity* of the data constructor is n.

Example. The data constructors for constructing lists are [] having signature $List\ a$ and : having signature $a \to List\ a \to List\ a$, 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 $[4, 5, 6]$. In fact, $[t_1, \ldots, t_n]$ is often used as notational sugar for the term $t_1 : \cdots : t_n : []$.

I assume throughout that, for each type constructor T, there exists at least one data constructor having a signature of the form $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$. If C is a data constructor having a signature of the form $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$, then C is said to be *associated with* the type constructor T.

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[2] 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 $t_1, \ldots, t_n$ are terms of type $\alpha_1, \ldots, \alpha_n$[3], respectively, such that the set of equations of the form

   $$\rho_{i_1} = \rho_{i_2} = \cdots = \rho_{i_k},$$

   for each variable that is free with relative type ρ_{i_j} in the term t_{i_j} ($j = 1, \ldots, k$ and $k > 1$), have a most general unifier θ, then $(t_1, \ldots, t_n)$[4] is a term of type $\alpha_1\theta \times \cdots \times \alpha_n\theta$.

   A variable is free with relative type $\sigma\theta$ in $(t_1, \ldots, t_n)$ if the variable is free with relative type σ in t_j, for some $j \in \{1, \ldots, n\}$.

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.

Note. Since the previous definition is rather complicated, some remarks to clarify its exact meaning are in order. First, rather more than just a term is being defined. In fact, what shall be referred to as an *annotated term*, which is a triple, is actually being defined. The first component is the term itself, the second is its type, and the third is its set of free variables and their relative types. The inductive definition specifies that the set of annotated terms is the intersection of all sets of triples satisfying (appropriately reformulated versions of) Parts 1 to 5 of the definition (which are then called Conditions 1 to 5).

For this definition to make sense, one needs a suitable universe of triples. For a place in a triple where a type or relative type should appear, one can use a type as defined earlier. For the place where a term should appear, one can define a suitable set of expressions as follows. Given some alphabet, let an expression be a finite sequence of symbols drawn from the set of constants $\mathfrak{C}$, the set of variables $\mathfrak{V}$, and '(', ')', 'λ', '.', and ','. Then the universe is the set of triples for which the first component is an expression, the second is a type, and the third is a set of pairs consisting of a variable and a type. Thus the intersection of all sets of triples satisfying Conditions 1 to 5 is well-defined and, furthermore, it satisfies Conditions 1 to 5. Hence the set of annotated terms is the smallest set of triples satisfying Conditions 1 to 5.

Generally, when proving properties of terms, what are actually used are the corresponding annotated terms since there is usually some reference in the proof to the type of the term or to free variables and their relative types. When needed, this will be taken for granted in the following without any remark to this effect being made. In particular, where appropriate, $\mathfrak{L}$ will implicitly refer to the set of annotated terms.

[2] 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.

[3] 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.

[4] 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 *1*.

In Part 4 of the previous definition, a variable can be free in both s and t. To determine the relative type of the variable in $(s\ t)$, one can use either the relative type ρ in s or the relative type δ in t. The reason, of course, is that θ is a unifier of the set of equations for this part and so $\rho\theta = \delta\theta$. A similar remark applies to Part 5.

To prove properties of terms, one can employ the following *principle of induction on the structure of terms.*

Proposition 2.6. *Let $\mathfrak{X}$ be a subset of $\mathfrak{L}$ satisfying the following conditions.*

1. *Each variable in $\mathfrak{V}$ is in $\mathfrak{X}$.*
2. *Each constant in $\mathfrak{C}$ is in $\mathfrak{X}$.*
3. *If $t \in \mathfrak{X}$ and $x \in \mathfrak{V}$, then $\lambda x.t \in \mathfrak{X}$.*
4. *If $s, t \in \mathfrak{X}$ and $(s\ t) \in \mathfrak{L}$, then $(s\ t) \in \mathfrak{X}$.*
5. *If $t_1, \ldots, t_n \in \mathfrak{X}$ and $(t_1, \ldots, t_n) \in \mathfrak{L}$, then $(t_1, \ldots, t_n) \in \mathfrak{X}$.*

Then $\mathfrak{X} = \mathfrak{L}$.

Proof. The first step is to show that $\mathfrak{X}$ satisfies Conditions 1 to 5 of the definition of a term. For Conditions 1, 2 and 3, this is immediate from the first three conditions satisfied by $\mathfrak{X}$.

For Condition 4 of the definition of a term, suppose that $s, t \in \mathfrak{X}$ and the corresponding equations, that is, $\alpha = \gamma$ augmented with the equations of the form $\rho = \delta$, have a most general unifier. Since $\mathfrak{L}$ satisfies Condition 4, it follows that $(s\ t) \in \mathfrak{L}$. Thus $(s\ t) \in \mathfrak{X}$, by the fourth condition satisfied by $\mathfrak{X}$. Hence $\mathfrak{X}$ satisfies Condition 4 of the definition of a term.

For Condition 5 of the definition of a term, the proof is similar.

Now, since $\mathfrak{X}$ satisfies Conditions 1 to 5 of the definition of a term and $\mathfrak{L}$ is the intersection of all such sets, it follows immediately that $\mathfrak{L} \subseteq \mathfrak{X}$. Thus $\mathfrak{X} = \mathfrak{L}$. □

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 $List\ a \times List\ M$. 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 $(((\mathit{append}\ x)\ [])\ (\mathit{process}\ x))$ can be written more simply as $\mathit{append}\ x\ []\ (\mathit{process}\ x)$.

Proposition 2.7. *Let t be a term. Then exactly one of the following conditions holds.*

1. *t is a variable.*
2. *t is a constant.*
3. *t has the form $\lambda x.s$, where s is a term.*
4. *t has the form $(u\ v)$, where u and v are terms.*
5. *t has the form $(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms.*

Proof. It is clear that at most one of the conditions holds. Now suppose t is a term that is neither a variable, nor a constant, nor has the form $\lambda x.s$, $(u\ v)$, or $(t_1, \ldots, t_n)$. Then $\mathfrak{L} \setminus \{t\}$ satisfies Conditions 1 to 5 in the definition of a term, which contradicts the definition of $\mathfrak{L}$ as being the smallest set satisfying Conditions 1 to 5. Thus t is either a variable, a constant, or has the form $\lambda x.s$, $(u\ v)$, or $(t_1, \ldots, t_n)$.

Suppose that t has the form $\lambda x.s$, but that s is not a term. Then the set of terms $\mathfrak{L} \setminus \{t\}$ satisfies Conditions 1 to 5 in the definition of a term, which contradicts the definition of $\mathfrak{L}$ as being the smallest set satisfying Conditions 1 to 5. Thus s is a term.

The proofs for 4 and 5 are similar. □

The concept of a subterm of a term will play an important part in the applications of the logic especially those for programming languages. As preparation for the definition of a subterm, the concept of an occurrence in a term is defined. Let $\mathbb{Z}$ denote the set of integers, $\mathbb{Z}^+$ the set of positive integers, and $(\mathbb{Z}^+)^*$ the set of all strings over the alphabet of positive integers, with ε denoting the empty string.

Definition. The *occurrence set* of a term t, denoted $\mathcal{O}(t)$, is the set of strings in $(\mathbb{Z}^+)^*$ defined inductively as follows.

1. If t is a variable, then $\mathcal{O}(t) = \{\varepsilon\}$.
2. If t is a constant, then $\mathcal{O}(t) = \{\varepsilon\}$.
3. If t has the form $\lambda x.s$, then $\mathcal{O}(t) = \{\varepsilon\} \cup \{1p \mid p \in \mathcal{O}(s)\}$.
4. If t has the form $(u\ v)$, then $\mathcal{O}(t) = \{\varepsilon\} \cup \{1p \mid p \in \mathcal{O}(u)\} \cup \{2q \mid q \in \mathcal{O}(v)\}$.
5. If t has the form $(t_1, \ldots, t_n)$, then $\mathcal{O}(t) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{ip \mid p \in \mathcal{O}(t_i)\}$.

Each $p \in \mathcal{O}(t)$ is called an *occurrence* in t.

Note. More precisely, $\mathcal{O}$ is a function $\mathcal{O} : \mathfrak{L} \to 2^{(\mathbb{Z}^+)^*}$ from the set of terms into the powerset of the set of all strings of positive integers. It is easy to show by induction on the structure of terms that $\mathcal{O}$ is defined on the whole of $\mathfrak{L}$. Also the value of $\mathcal{O}(t)$ is unique for each t, by an application of Proposition 2.7.

Example. Consider the term $\mathit{append}\ x\ []\ (\mathit{process}\ x)$, illustrated in Figure 1 below. Its occurrence set is $\{\varepsilon, 1, 2, 11, 12, 21, 22, 111, 112\}$.

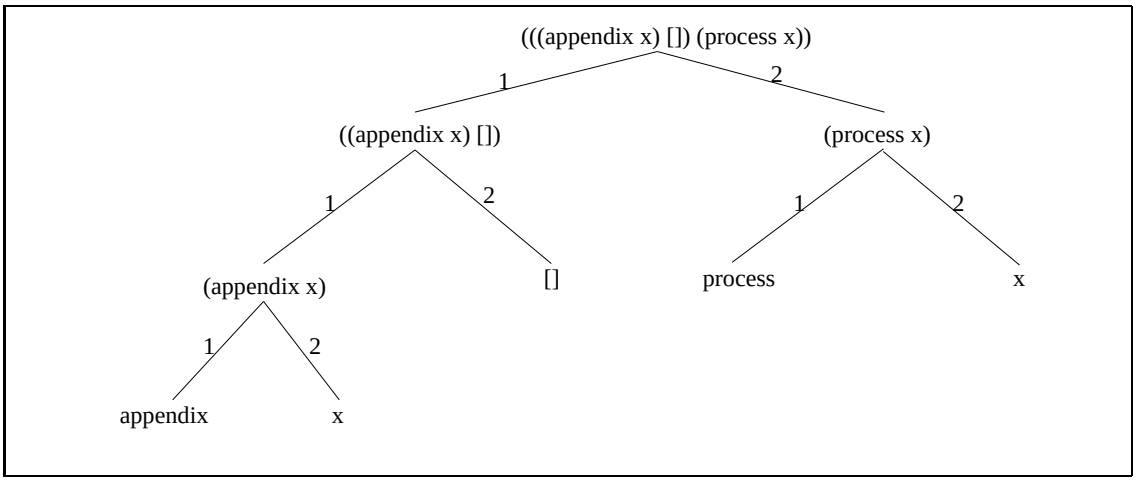

Figure 1: A term with occurrences

Definition. If t is a term and $p \in \mathcal{O}(t)$, then the *subterm of t at occurrence p*, denoted $t|_p$, is defined inductively on the length of p as follows.

1. If $p = \varepsilon$, then $t|_p = t$.
2. If $p = 1q$, for some q, and t has the form $\lambda x.s$, then $t|_p = s|_q$.
   If $p = 1q$, for some q, and t has the form $(u\ v)$, then $t|_p = u|_q$.
   If $p = 2q$, for some q, and t has the form $(u\ v)$, then $t|_p = v|_q$.
   If $p = iq$, for some q, and t has the form $(t_1, \ldots, t_n)$, then $t|_p = t_i|_q$, for $i = 1, \ldots, n$.

A *subterm* is a subterm of a term at some occurrence.

In connection with the previous definition, note that if t has the form $\lambda x.s$, then s is a term, by Proposition 2.7. Thus $s|_p$ is well defined. Similar comments apply to $(u\ v)$ and $(t_1, \ldots, t_n)$. Furthermore, if t is a variable or a constant, then $\mathcal{O}(t)$ must be $\{\varepsilon\}$. Also if t has the form $\lambda x.s$, then each $p \in \mathcal{O}(t)$ must be either ε or have the form $1q$, for some q. Similar comments apply to to $(u\ v)$ and $(t_1, \ldots, t_n)$.

Example. Consider again the term *append* x [] (*process* x). The subterm of *append* x [] (*process* x) at occurrence

ε is *append* x [] (*process* x),
1 is *append* x [],
2 is *process* x,
11 is *append* x,
12 is [],
21 is *process*,
22 is x,
111 is *append*, and
112 is x.

These subterms are illustrated in Figure 1.

Proposition 2.8. *Each subterm is a term.*

Proof. Let t be a term and $p \in \mathcal{O}(t)$. I show by induction on the length of p that $t|_p$ is a term.

If the length of p is 0, then $p = \varepsilon$. Thus $t|_p = t$, which is a term.

For the inductive step, suppose the length of p is $n + 1$ ($n \geq 0$). There are several cases to consider.

If $p = 1q$, for some q, and t has the form $\lambda x.s$, then $t|_p = s|_q$ which is a term by the induction hypothesis.

If $p = 1q$, for some q, and t has the form $(u\ v)$, then $t|_p = u|_q$ which is a term by the induction hypothesis.

If $p = 2q$, for some q, and t has the form $(u\ v)$, then $t|_p = v|_q$ which is a term by the induction hypothesis.

If $p = iq$, for some q, and t has the form $(t_1, \ldots, t_n)$, then $t|_p = t_i|_q$ which is a term by the induction hypothesis, for $i = 1, \ldots, n$. □

It is easy to show that if r is the subterm of s at occurrence q and s is the subterm of t at occurrence p, then r is the subterm of t at occurrence pq. Thus a subterm of a subterm of a term t is a subterm of t.

Proposition 2.9.

1. *Each constant appearing at a particular place in a term is a subterm of the term.*

2. *Each variable appearing at a particular place in a term (except a variable appearing immediately after a λ) is a subterm of the term.*

Proof. 1. The proof is by induction on the structure of terms. Let t be a term and C a constant appearing in t. It has to be shown that there exists $p \in \mathcal{O}(t)$ such that $t|_p = C$.

If t is a variable, then there can be no appearance of a constant and the result holds.

If t is the constant C, then the only appearance of a constant is t itself and $t|_\varepsilon = C$.

Let t have the form $\lambda x.s$ and C be a constant appearing in t. Thus C must appear in s. By the induction hypothesis, there exists $q \in \mathcal{O}(s)$ such that $s|_q = C$. Thus $t|_1 q = C$.

The argument when t has the form $(u\ v)$ or $(t_1, \ldots, t_n)$ is similar.

2. The proof is by induction on the structure of terms. Let t be a term and x a variable appearing (not immediately after a λ) in t. It has to be shown that there exists $p \in \mathcal{O}(t)$ such that $t|_p = x$.

If t is a variable x, then the only appearance of a variable is t itself and $t|_\varepsilon = x$.

If t is a constant, then there can be no appearance of a variable and the result holds.

Let t have the form $\lambda y.s$ and x be a variable appearing in t. Since x does not appear immediately after a λ, x must appear in s. By the induction hypothesis, there exists $q \in \mathcal{O}(s)$ such that $s|_q = x$. Thus $t|_1 q = x$.

The argument when t has the form $(u\ v)$ or $(t_1, \ldots, t_n)$ is similar. □

Note that a variable appearing immediately after a λ in a term is *not* a subterm since the variable appearing there is not at an occurrence of the term.

Definition. An occurrence of a variable x in a term is *bound* if it occurs within a subterm of the form $\lambda x.t$.

A variable in a term is *bound* if it has a bound occurrence.

An occurrence of a variable in a term is *free* if it is not a bound occurrence.

For a particular occurrence of a subterm $\lambda x.t$ in a term, the occurrence of t is called the *scope* of the λx.

In the definition of a term, the concept of a free variable is also defined. The following result establishes that free variables are exactly those variables which have free occurrences.

Proposition 2.10. *A variable is free in a term iff it has a free occurrence in the term.*

Proof. The proof is by induction on the structure of terms. Let the term be t.

If t is a variable or a constant, then the result is clear.

Let t have the form $\lambda x.s$. I claim that x is not a free variable in t. Thus suppose that x is a free variable in t. Let t' be the (annotated) term obtained from t by dropping the free variable x from the third component of t and let $\mathfrak{L}'$ be $\mathfrak{L}$ with t replaced by t'. Then $\mathfrak{L}'$ satisfies Conditions 1 to 5 in the definition of a term. Thus $\mathfrak{L} \cap \mathfrak{L}'$ also satisfies Conditions 1 to 5 and is strictly smaller than $\mathfrak{L}$ (since it does not contain t), which contradicts the definition of $\mathfrak{L}$ as being the smallest set satisfying Conditions 1 to 5. Thus x is not a free variable in t. Furthermore, any occurrence of x in t is bound, by definition. Thus the property holds for x. For another variable y occurring in s, by the induction hypothesis, y is free in s iff y has a free occurrence in s. Thus y is free in t iff y has a free occurrence in t.

Let t have the form $(u\ v)$. Then a variable x is free in $(u\ v)$ iff x is free in either u or v iff x has free occurrence in either u or v (by the induction hypothesis) iff x has a free occurrence in $(u\ v)$.

Let t have the form $(t_1, \ldots, t_n)$. Then a variable x is free in $(t_1, \ldots, t_n)$ iff x is free in t_j, for some $j \in \{1, \ldots, n\}$ iff x has a free occurrence in t_j, for some $j \in \{1, \ldots, n\}$ (by the induction hypothesis) iff x has a free occurrence in $(t_1, \ldots, t_n)$. □

A variable can be bound by more than one λ in a term.

Example. Let M and N be nullary types, and $f : M \to \Omega$ and $g : N \to \Omega$ be constants. Then $(\Sigma\ \lambda x.(f\ x)) \wedge (\Sigma\ \lambda x.(g\ x))$ is a term of type Ω, in which the variable x is bound by several λs.

A variable can be both bound and free in a term.

Example. Let L, M, N, and P be nullary type constructors, and $f : (L \to M) \to N \to P$, $g : L \to M$, and $h : L \to L \to N$ be constants. Consider the term $f\ \lambda x.(g\ x)\ (h\ x\ y)$. Then the first occurrence of x (that is, the x in $(g\ x)$) is bound and the second occurrence is free. Thus x is both bound and free in $f\ \lambda x.(g\ x)\ (h\ x\ y)$.

The previous two examples motivate the introduction of the following concept.

Definition. A term is *rectified* if each bound variable is bound by just one λ and no variable is both bound and free.

By a suitable change of bound variables any term can be converted to a rectified term. Since the names of the bound variables are of no consequence, the rectified term thus obtained is "equivalent" to the original term. Later, terms that differ only in the names of the bound variables are identified.

The concept of relative type of free variables is now extended to all subterms of a term.

Definition. The *relative type* of the subterm $t|_p$ of the term t at $p \in \mathcal{O}(t)$ is defined by induction on the length of p as follows.

If the length of p is 0, then the relative type of $t|_p$ is the same as the type of t.

For the inductive step, suppose the length of p is $n + 1$ ($n \geq 0$). There are several cases to consider.

If $p = 1q$, for some q, and t has the form $\lambda x.s$, then the relative type of $t|_p$ is the same as the relative type of $s|_q$ in s.

If $p = 1q$, for some q, and t has the form $(u\ v)$, then the relative type of $t|_p$ is $\sigma\theta$, where σ is the relative type of $u|_q$ in u and θ is the associated mgu.

If $p = 2q$, for some q, and t has the form $(u\ v)$, then the relative type of $t|_p$ is $\sigma\theta$, where σ is the relative type of $v|_q$ in v and θ is the associated mgu.

If $p = iq$, for some q, and t has the form $(t_1, \ldots, t_n)$, then the relative type of $t|_p$ is $\sigma\theta$, where σ is the relative type of $t_i|_q$ in t_i and θ is the associated mgu, for $i = 1, \ldots, n$.

Example. Let M be a nullary type, and $append : List\ a \to List\ a \to List\ a \to \Omega$, $process : List\ a \to List\ a$ and $A, B, C : M$ be constants. Consider the first occurrence of x in the term $append\ x\ []\ (process\ x)$. As a term in its own right, x has type a, for some parameter a. As a subterm of $append\ x\ []\ (process\ x)$, x has relative type $List\ a$.

The occurrence of x in the term $append\ x\ []\ (process\ [A, B, C])$ has relative type $List\ M$. Also the subterm $process$ of this term has relative type $List\ M \to List\ M$.

In the definition of a term, the concept of the relative type of a free variable is defined. The next proposition shows that this relative type is the same as the relative type of any of the free occurrences of the variable.

Proposition 2.11. *The relative type of each free variable x in a term t is the same as the relative type of each free occurrence of x in t.*

Proof. The proof is by induction on the structure of terms.

Let t be the variable x of type a. The relative type of x as a free variable is a and the relative type of the free occurrence of x in x is also a.

If t is a constant, there are no free variables and the result holds.

Let t have the form $\lambda y.s$ and x be a free variable in t. Then x cannot be y and x must be a free variable in s. By the induction hypothesis, the relative type of x in s is the same as the relative types of each of its free occurrences in s. Thus the relative type of x in t is the same as the relative types of each of its free occurrences in t.

Let t have the form $(u\ v)$ and x be a free variable in t. Thus x is a free variable in either u or v or both. Choose any free occurrence of x in u, say. By the induction hypothesis, the relative type of x in u is the same as the relative type of this free occurrence of x in u. Thus the relative type of x in $(u\ v)$ is the same as the relative type of this free occurrence of x in $(u\ v)$.

Let t have the form $(t_1, \ldots, t_n)$ and x be a free variable in t. Thus x is a free variable in t_j, for some $j \in \{1, \ldots, n\}$. Choose any free occurrence of x in t_j. By the induction hypothesis, the relative type of x in t_j is the same as the relative type of this free occurrence of x in t_j. Thus the relative type of x in $(t_1, \ldots, t_n)$ is the same as the relative type of this free occurrence of x in $(t_1, \ldots, t_n)$. □

Distinct bound occurrences of a variable can have distinct relative types.

Example. Let M and N be nullary types, and $f : M \to \Omega$ and $g : N \to \Omega$ be constants, and t the term $(\Sigma\ \lambda x.(f\ x)) \wedge (\Sigma\ \lambda x.(g\ x))$. The bound occurrence of x in $(\Sigma\ \lambda x.(f\ x))$ has relative type M in t, while the one in $(\Sigma\ \lambda x.(g\ x))$ has relative type N in t.

The next two results are concerned with replacing a subterm of a term by a new subterm.

Definition. Let t be a term, s a subterm of t at occurrence p, and r a term. Then the expression obtained by *replacing s in t by r*, denoted $t[s/r]_p$, is defined by induction on the length of p as follows.

If the length of p is 0, then $t[s/r]_p = r$.

For the inductive step, suppose the length of p is $n + 1$ ($n \geq 0$). There are several cases to consider.

If $p = 1q$, for some q, and t has the form $\lambda x.w$, then $(\lambda x.w)[s/r]_p = \lambda x.(w[s/r]_q)$.

If $p = 1q$, for some q, and t has the form $(u\ v)$, then $(u\ v)[s/r]_p = (u[s/r]_q\ v)$.

If $p = 2q$, for some q, and t has the form $(u\ v)$, then $(u\ v)[s/r]_p = (u\ v[s/r]_q)$.

If $p = iq$, for some q, and t has the form $(t_1, \ldots, t_n)$, then $(t_1, \ldots, t_n)[s/r]_p = (t_1, \ldots, t_i[s/r]_q, \ldots, t_n)$, for $i = 1, \ldots, n$.

Easy examples show that $t[s/r]_p$ does not have to be a term (because r may not have the correct type). However, under certain conditions that arise particularly in the application of the logic to declarative programming languages, it will be a term.

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.

Proposition 2.12. *Let t be a term, s a subterm of t at occurrence p, and r a term that is type-weaker than s. Then the following hold.*

1. *$t[s/r]_p$ is a term that is type-weaker than t.*
2. *$t = t[s/r]_p$ is a term.*

Proof. 1. The proof is by induction on the length of p.

If the length of p is 0, then $s = t$, $t[s/r]_p = r$, and $t[s/r]_p$ is type-weaker than t.

For the inductive step, suppose the length of p is $n + 1$ ($n \geq 0$). There are several cases to consider.

If $p = 1q$, for some q, and t has the form $\lambda x.w$, then $(\lambda x.w)[s/r]_p = \lambda x.(w[s/r]_q)$. By the induction hypothesis, $w[s/r]_q$ is a term that is type-weaker than w. Thus $\lambda x.(w[s/r]_q)$ is a term that is type-weaker than $\lambda x.w$. (If x is free in $w[s/r]_q$, this is obvious. If x is free in w, but not free in $w[s/r]_q$, then $\lambda x.(w[s/r]_q)$ has type of the form $a \to \beta$, for some parameter a, while $\lambda x.w$ has type of the form $\alpha \to \delta$, so the type substitution which shows that $\lambda x.(w[s/r]_q)$ is type-weaker than $\lambda x.w$ will include the binding a/α.) That is, $t[s/r]_p$ is a term that is type-weaker than t.

If $p = 1q$, for some q, and t has the form $(u\ v)$, then $(u\ v)[s/r]_p = (u[s/r]_q\ v)$. By the induction hypothesis, $u[s/r]_q$ is a term that is type-weaker than u. Hence $(u[s/r]_q\ v)$ is a term that is type-weaker than $(u\ v)$. That is, $t[s/r]_p$ is a term that is type-weaker than t.

If $p = 2q$, for some q, and t has the form $(u\ v)$, then the argument is similar to the previous part.

If $p = jq$, for some $j \in \{1, \ldots, n\}$ and q, and t has the form $(t_1, \ldots, t_n)$, then $(t_1, \ldots, t_n)[s/r]_p = (t_1, \ldots, t_j[s/r]_q, \ldots, t_n)$. By the induction hypothesis, $t_j[s/r]_q$ is a term that is type-weaker than t_j. Hence $(t_1, \ldots, t_j[s/r]_q, \ldots, t_n)$ is a term that is type-weaker than $(t_1, \ldots, t_n)$. That is, $t[s/r]_p$ is a term that is type-weaker than t.

2. This part follows easily from the facts that $=$ has signature $a \to a \to \Omega$ and that $t[s/r]_p$ is a term that is type-weaker than t. □

Example. Let $t = \lambda x.(f\ x\ y)$, where $f : M \to N \to N$. Thus t has type $M \to N$. Let $s = f\ x\ y$ (the subterm of t at occurrence 1), $r = y$, and $\gamma = \{b/N\}$, where b is the type of y. Then $b\gamma = N$ and r is type-weaker than s. Now $t[s/r]_1 = \lambda x.y$, which has type $a \to b$, for some new parameter a. Let $\xi = \{a/M, b/N\}$. Then $(a \to b)\xi = M \to N$ and $t[s/r]_1$ is type-weaker than t.

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).

Proposition 2.13. *Let t be a term, s a subterm of t at occurrence p, and r a term that is type-equivalent to s. Then $t[s/r]_p$ is a term that is type-equivalent to t.*

Proof. The proof follows immediately from Proposition 2.12 by considering $t[s/r]_p$ obtained from t by replacing s by r, and t obtained from $t[s/r]_p$ by replacing r by s. □

A common task, especially in the programming language applications of the logic, is to check that putative terms are correctly typed. A suitable class of putative terms is given by the set of well-formed expressions defined as follows.

Definition. A *well-formed expression* is defined inductively as follows.

1. Each variable in $\mathfrak{V}$ is a well-formed expression.
2. Each constant in $\mathfrak{C}$ is a well-formed expression.
3. If e is a well-formed expression and x is a variable, then $\lambda x.e$ is a well-formed expression.

4. If d and e are well-formed expression, then $(d\ e)$ is a well-formed expression.

5. If $e_1, \ldots, e_n$ are well-formed expressions, then $(e_1, \ldots, e_n)$ is a well-formed expression.

Thus the well-formed expressions are the terms of the untyped λ-calculus. Clearly the set of well-formed expressions (on some alphabet) is generally much larger than the set of terms (that is, $\mathfrak{L}$). The type checking problem is to determine whether or not some given well-formed expression t is in $\mathfrak{L}$ and, if so, to determine the type of t.

Proposition 2.14. *Type checking of terms is decidable and each well-formed expression can be well-typed in at most one way (up to variants).*

Proof. The proof proceeds by induction on the structure of well-formed expressions. □

2.3 Substitutions

In this subsection, the concept of instantiating a term by a substitution is studied.

Definition. A *substitution* is a finite set of the form $\{x_1/t_1, \ldots, x_n/t_n\}$, where each x_i is a variable, each t_i is a term, and $x_1, \ldots, x_n$ are distinct. Each element x_i/t_i is called a *binding*.

Intuitively, the concept of instantiating a term t by a substitution $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$, is simple – each free occurrence of a variable x_i in t is replaced by t_i. But there is a technical complication in that there may be a free variable y, say, in some t_i that is 'captured' in this process because, after instantiation, it occurs in the scope of a subterm of the form $\lambda y.s$ and therefore becomes bound in $t\theta$. Variable capture spoils the intended meaning of instantiation and hence it is necessary to avoid it. There are two approaches to this: one can disallow instantiation if variable capture would occur or one can rename bound variables in the term t to avoid variable capture altogether. The latter approach is adopted here.

Definition. Let t be a term and $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ a substitution. The *instance* $t\theta$ of t by θ is defined inductively on the structure of terms as follows.

1. If t is a variable x_i, for some $i \in \{1, \ldots, n\}$, then $x_i\theta = t_i$.
   If t is a variable y distinct from all the x_i, then $y\theta = y$.

2. If t is a constant C, then $C\theta = C$.

3. If t is an abstraction $\lambda x_i.s$, for some $i \in \{1, \ldots, n\}$, then

   $$(\lambda x_i.s)\theta = \lambda x_i.(s\{x_1/t_1, \ldots, x_{i-1}/t_{i-1}, x_{i+1}/t_{i+1}, \ldots, x_n/t_n\}).$$

   If t is an abstraction $\lambda y.s$, where y is distinct from all the x_i, and, for all $i \in \{1, \ldots, n\}$, either y is not free in t_i or x_i is not free in s, then

   $$(\lambda y.s)\theta = \lambda y.(s\theta).$$

   If t is an abstraction $\lambda y.s$, where y is distinct from all the x_i, and, for some $i \in \{1, \ldots, n\}$, y is free in t_i and x_i is free in s, then

   $$(\lambda y.s)\theta = \lambda z.(s\{y/z\}\theta).$$

   (Here z is chosen to be the first variable that is not free in s or any of the t_i.)

4. If t is an application $(u\ v)$, then $(u\ v)\theta = (u\theta\ v\theta)$.

5. If t is a tuple $(t_1, \ldots, t_n)$, then $(t_1, \ldots, t_n)\theta = (t_1\theta, \ldots, t_n\theta)$.

Note. In later proofs, it is assumed that the parameters in the type of t_i and relative types of its free variables are standardised apart from the parameters in the type of t_j and relative types of its free variables ($i, j = 1, \ldots, n$ and $i \neq j$), and also that the parameters that appear in the types of the t_i and the relative types of their free variables are standardised apart from those in the type of the term and the relative types of its free variables to which the substitution is applied.

Example. Suppose that t is the term *append* (u, v, w) and θ is the substitution $\{u/1 : x, v/[], w/[1, 2]\}$. Then $t\theta =$ *append* $(1 : x, [], [1, 2])$.

Example. Suppose that t is the term $\exists r.\exists x.\exists y.(u = r : x \ \wedge \ w = r : y \ \wedge \ \textit{append}\ (x, v, y))$ and θ is the substitution $\{u/1 : x, v/[], w/[1, 2]\}$. Then

$$t\theta = \exists r.\exists z.\exists y.(1 : x = r : z \ \wedge \ [1, 2] = r : y \ \wedge \ \textit{append}\ (z, [], y)).$$

Here the bound variable x in t is renamed to z to avoid capture of the x in the binding $u/1 : x$.

Note that $t\theta$ may not be a term.

Example. Let M and N be nullary type constructors and $F : M \to \Omega$ and $A : N$ be constants. Let t be $(F\ x)$ and θ be $\{x/A\}$. Then $t\theta = (F\ A)$ which is not a term.

For the purpose of establishing a necessary and sufficient condition for $t\theta$ to be a term, a technical result concerning solving a system of equations about types is required.

Definition. A collection $\{E_1, \ldots, E_n\}$ of sets of equations about types is *mutually disjoint* if the sets of parameters in each of the E_i are mutually disjoint.

Proposition 2.15. *Let $\{E_1, \ldots, E_n\}$ be a collection of sets of equations about types that is mutually disjoint and F a set of equations about types. Then $E_1 \cup \ldots \cup E_n \cup F$ has mgu φ iff there exist type substitutions $\varphi_1, \ldots, \varphi_n$, and η such that E_i has mgu φ_i, for $i = 1, \ldots, n$, and $F(\varphi_1 \cup \ldots \cup \varphi_n)$ has mgu η, where $(\varphi_1 \cup \ldots \cup \varphi_n)\eta = \varphi$.*

Proof. □

Now the result about a necessary and sufficient condition for $t\theta$ to be a term can be established.

Proposition 2.16. *Let t be a term having type σ and $x_1, \ldots, x_n$ be free variables in t, where the relative type of x_i in t is ρ_i $(i = 1, \ldots, n)$. Let $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ be a substitution, where t_i has type τ_i $(i = 1, \ldots, n)$. Then the following are equivalent.*

(a) $t\theta$ is a term of type $\sigma\varphi$ and, if y is a free variable in $t\theta$ and y has relative type δ in s or some t_i, then y has relative type $\delta\varphi$ in $t\theta$.

(b) The set of equations $U_{t,\theta} \cup V_{t,\theta}$ has mgu φ, where

$$\begin{aligned} U_{t,\theta} &= \{\rho_i = \tau_i \mid i = 1, \ldots, n\}, \ \textit{and} \\ V_{t,\theta} &= \{\delta_{i_1} = \cdots = \delta_{i_k}\ [= \delta] \mid \ \textit{there is a variable that is free with relative type } \delta_{i_j} \\ &\qquad \textit{in } t_{i_j} \textit{ and, possibly, the variable is free with relative type } \delta \textit{ in } t\}. \end{aligned}$$

Proof. The proof proceeds by induction on the structure of terms. □

2.4 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.

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

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 set of syntactical variables is denoted by $\mathfrak{M}$ ($\mathfrak{M}$ for 'meta'.) By reifying (that is, instantiating) the syntactical variables in a schema with terms, a term is obtained from the schema (under some restrictions given below).

I now turn to the definition of a schema. This definition can be obtained from the previous one for a term by also allowing syntactical variables to appear.

Definition. A *schema*, together with its type, its set of free variables and their relative types, and its set of syntactical variables and their relative types and bound sets, 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 syntactical variable $\boldsymbol{x}$ in $\mathfrak{M}$ is a schema of type a, where a is a parameter.

   The syntactical variable $\boldsymbol{x}$ has relative type a and bound set $\emptyset$ in $\boldsymbol{x}$.

3. Each constant C in $\mathfrak{C}$, where C has signature α, is a schema of type α.

4. (Abstraction) If s is a schema of type β and x a variable in $\mathfrak{V}$, then $\lambda x.s$ is a schema of type $\alpha \to \beta$, if x is free with relative type α in s, or type $a \to \beta$, where a is a new parameter, otherwise.

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

   A syntactical variable has relative type ω and bound set $\{x\} \cup B$ in $\lambda x.s$ if the syntactical variable has relative type ω and bound set B in s.

5. (Application) If s is a schema of type $\alpha \to \beta$ and t a schema of type γ such that each syntactical variable appearing in both s and t has the same bound set in s and t, and 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 free with relative type δ in t, and of the form

   $$\xi = \eta,$$

for each syntactical variable with relative type ξ in s and relative type η in t, have a most general unifier θ, then $(s\ t)$ is a schema 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.

A syntactical variable has relative type $\omega\theta$ and bound set B in $(s\ t)$ if the syntactical variable has relative type ω and bound set B in s or t.

6. (Tupling) If $s_1, \ldots, s_n$ are schemas of type $\alpha_1, \ldots, \alpha_n$ respectively, such that each syntactical variable appearing in the schemas $s_{j_1}, \ldots, s_{j_m}$ has the same bound set in each of $s_{j_1}, \ldots, s_{j_m}$ $(m > 1)$, and the set of equations of the form

$$\rho_{i_1} = \rho_{i_2} = \cdots = \rho_{i_k},$$

for each variable that is free with relative type ρ_{i_j} in the schema s_{i_j} $(j = 1, \ldots, k$ and $k > 1)$, and of the form

$$\omega_{p_1} = \omega_{p_2} = \cdots = \omega_{p_l},$$

for each syntactical variable with relative type ω_{p_r} in the schema s_{p_r} $(r = 1, \ldots, l$ and $l > 1)$, have a most general unifier θ, then $(s_1, \ldots, s_n)$ is a schema of type $\alpha_1\theta \times \cdots \times \alpha_n\theta$.

A variable is free with relative type $\sigma\theta$ in $(s_1, \ldots, s_n)$ if the variable is free with relative type σ in s_j, for some $j \in \{1, \ldots, n\}$.

A syntactical variable has relative type $\omega\theta$ and bound set B in $(s_1, \ldots, s_n)$ if the syntactical variable has relative type ω and bound set B in s_j, for some $j \in \{1, \ldots, n\}$.

The type substitution θ in Parts 5 and 6 of the definition is called the *associated* mgu.

In the previous definition, analogous assumptions about the parameters n the subschemas being brought together to form a schema are made as were made in the definition of a term. Note that, in an abstraction $\lambda x.s$, x must be an (ordinary) variable and not a syntactical variable. Clearly, each term is a schema that does not contain any syntactical variables, and conversely.

The bound set of a syntactical variable in a schema is the set of bound variables in the schema in whose scopes the syntactical variable lies. Later the concept of reifying a schema will be introduced in which the syntactical variables in a schema are replaced by terms. However, in contrast to the case of applying a substitution to a term, it will be possible, indeed desirable, for the term replacing some syntactical variable to contain free variables that are in the bound set of the syntactical variable thus leading to their capture in the instance of the schema. This extra flexibility is precisely the reason for introducing schemas.

Notation. $\overline{\mathfrak{L}}$ denotes the set of all schemas obtained from an alphabet.

Example. Here is a schema from the definition of the function *powerset* : $(a \to \Omega) \to (a \to \Omega) \to \Omega$.

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

In this schema, $\boldsymbol{u}$ and $\boldsymbol{w}$ are syntactical variables ranging over terms, while x and v are (ordinary) variables. Note that each occurrence of $\boldsymbol{u}$ is in the scope of some λx and the bound set of $\boldsymbol{u}$ is $\{x\}$. Similarly, for $\boldsymbol{w}$. If v is changed into a syntactical variable, the following expression results.

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

This expression is not a schema because the first occurrence of $\boldsymbol{v}$ has bound set $\{x\}$ in the left hand side, but the second occurrence has bound set $\emptyset$ in the right hand side.

To prove properties of schemas, one can employ the following *principle of induction on the structure of schemas.*

Proposition 2.17. *Let $\mathfrak{X}$ be a subset of $\overline{\mathfrak{L}}$ satisfying the following conditions.*

1. *Each variable in $\mathfrak{V}$ is in $\mathfrak{X}$.*
2. *Each syntactical variable in $\mathfrak{M}$ is in $\mathfrak{X}$.*
3. *Each constant in $\mathfrak{C}$ is in $\mathfrak{X}$.*
4. *If $s \in \mathfrak{X}$ and $x \in \mathfrak{V}$, then $\lambda x.s \in \mathfrak{X}$.*
5. *If $s, t \in \mathfrak{X}$ and $(s\ t) \in \overline{\mathfrak{L}}$, then $(s\ t) \in \mathfrak{X}$.*
6. *If $s_1, \ldots, s_n \in \mathfrak{X}$ and $(s_1, \ldots, s_n) \in \overline{\mathfrak{L}}$, then $(s_1, \ldots, s_n) \in \mathfrak{X}$.*

Then $\mathfrak{X} = \overline{\mathfrak{L}}$.

Proof. The first step is to show that $\mathfrak{X}$ satisfies Conditions 1 to 6 of the definition of a schema. For Conditions 1, 2, 3 and 4, this is immediate from the first four conditions satisfied by $\mathfrak{X}$.

For Condition 5 of the definition of a schema, suppose that $s, t \in \mathfrak{X}$ and the corresponding equations, that is, $\alpha = \gamma$ augmented with the equations of the form $\rho = \delta$ and $\xi = \eta$, have a most general unifier. Since $\overline{\mathfrak{L}}$ satisfies Condition 5, it follows that $(s\ t) \in \overline{\mathfrak{L}}$. Thus $(s\ t) \in \mathfrak{X}$, by the fifth condition satisfied by $\mathfrak{X}$. Hence $\mathfrak{X}$ satisfies Condition 5 of the definition of a schema.

For Condition 6 of the definition of a schema, the proof is similar.

Now, since $\mathfrak{X}$ satisfies Conditions 1 to 6 of the definition of a schema and $\overline{\mathfrak{L}}$ is the intersection of all such sets, it follows immediately that $\overline{\mathfrak{L}} \subseteq \mathfrak{X}$. Thus $\mathfrak{X} = \overline{\mathfrak{L}}$. □

The concepts of type-weaker and type-equivalence for schemas will play an important part.

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).

Proposition 2.18. *Let s and t be schemas such that s is type-weaker than t. Then $t = s$ is a schema.*

Proof. Recall that $=$ has signature $a \to a \to \Omega$ and $t = s$ means $((=\ t)\ s)$. Suppose that s has type σ and t has type τ. Then $(=\ t)$ is a schema of type $\tau \to \Omega$. Consequently, $((=\ t)\ s)$ is a schema of type Ω, since s is type-weaker than t. □

2.5 Reifiers

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 $i = 1, \ldots, n$.

Note. In later proofs, it is assumed that the parameters in the type of t_i and relative types of its free variables are standardised apart from the parameters in the type of t_j and relative types of its free variables ($i, j = 1, \ldots, n$ and $i \neq j$), and also that the parameters that appear in the types of the t_i and the relative types of their free variables are standardised apart from those in the type of the schema and the relative types of its free variables and syntactical variables to which the reifier is applied.

The next result provides a necessary and sufficient condition for $s\Psi$ to be a term.

Proposition 2.19. *Let s be a schema having type σ and containing the syntactical variables $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, where the relative type of $\boldsymbol{x}_i$ in s is ρ_i ($i = 1, \ldots, n$). Let $\Psi = \{\boldsymbol{x}_1/t_1, \ldots, \boldsymbol{x}_n/t_n\}$ be a reifier, where t_i has type τ_i ($i = 1, \ldots, n$). Then the following are equivalent.*

(a) $s\Psi$ is a term of type $\sigma\varphi$ and, if y is a free variable in $s\Psi$ and y has relative type δ in s or some t_i, then y has relative type $\delta\varphi$ in $s\Psi$.

(b) The set of equations $U_{s,\Psi} \cup V_{s,\Psi} \cup W_{s,\Psi}$ has mgu φ, where

$U_{s,\Psi} = \{\rho_i = \tau_i \mid i = 1, \ldots, n\}$,

$V_{s,\Psi} = \{\delta_{i_1} = \cdots = \delta_{i_k}[= \delta] \mid$ *there is a variable that is free with relative type δ_{i_j} in t_{i_j} and does not occur in the bound set of $\boldsymbol{x}_{i_j}$ ($j = 1, \ldots, k$) and, possibly, the variable is free with relative type δ in s*$\}$, *and*

$W_{s,\Psi} = \{\varepsilon_{j_1} = \cdots = \varepsilon_{j_m} = \varepsilon \mid$ *there is an occurrence of a subschema of the form $\lambda x.t$ with relative type $\epsilon \to \delta$ in s such that $\boldsymbol{x}_{j_p}$ occurs in t but does not occur in any subschema of the form $\lambda x.r$ in t, and x is free with relative type ε_{j_p} in t_{j_p} ($p = 1, \ldots, m$ and $m > 0$)*$\}$.

Proof. The proof proceeds by induction on the structure of schemas.

In case s is a variable, syntactical variable, or constant, the result is obvious.

Suppose s has the form $\lambda x.t$ with type $\alpha \to \beta$. Then

$U_{\lambda x.t,\Psi} \cup V_{\lambda x.t,\Psi} \cup W_{\lambda x.t,\Psi}$ has mgu φ

iff $U_{t,\Psi} \cup V_{t,\Psi} \cup W_{t,\Psi}$ has mgu φ

[since $U_{t,\Psi} = U_{\lambda x.t,\Psi}$, $V_{t,\Psi}$ is $V_{\lambda x.t,\Psi}$ plus an equation corresponding to free occurrences of x, and $W_{t,\Psi}$ is $W_{\lambda x.t,\Psi}$ minus the same equation, assuming x is free in $t\Psi$. Otherwise, $V_{t,\Psi} = V_{\lambda x.t,\Psi}$ and $W_{t,\Psi} = W_{\lambda x.t,\Psi}$]

iff $t\Psi$ is a term of type $\beta\varphi$ and, if y is a free variable in $t\Psi$ and y has relative type δ in t or some t_i, then y has relative type $\delta\varphi$ in $t\Psi$

[by the induction hypothesis]

iff $(\lambda x.t)\Psi$ is a term of type $(\alpha \to \beta)\varphi$ and, if y is a free variable in $(\lambda x.t)\Psi$ and y has relative type δ in t or some t_i, then y has relative type $\delta\varphi$ in $(\lambda x.t)\Psi$.

[$\Downarrow$: It suffices to show that $(\lambda x.t)\Psi$ is a term of type $(\alpha \to \beta)\varphi$. If x is not free in $t\Psi$, then α is a parameter a and $\lambda x.t$ has type $a \to \beta\varphi = (a \to \beta)\varphi$. If x is free in $t\Psi$ and free in t, then x has relative type α in t so that $(\lambda x.t)\Psi$ has type $(\alpha \to \beta)\varphi$. If x is free in $t\Psi$ but not free in t, then α is a parameter a and $(\lambda x.t)\Psi$ has type $(\delta \to \beta)\varphi$, where δ is the relative type of x in some t_i, and $\delta\varphi = a\varphi$.

$\Uparrow$: It suffices to show that, if x is a free variable in $t\Psi$ and x has relative type δ in t or some t_i, then x has relative type $\delta\varphi$ in $t\Psi$. Suppose that x is free in t and has relative type δ in t. Then $\alpha = \delta$ and the relative type of x in $(\lambda x.t)\Psi$ is $\alpha\varphi$. Thus the relative type of x in $(\lambda x.t)\Psi$ is $\delta\varphi$. If x is free in some t_i and has relative type δ in t_i, then the relative type of x in $(\lambda x.t)\Psi$ is $\alpha\varphi$ and $\alpha\varphi = \delta\varphi$]

Suppose s has the form $(s_1\ s_2)$ with type $\beta\theta$, where s_1 has type $\alpha \to \beta$, s_2 has type γ, and θ is the associated mgu for $(s_1\ s_2)$. Then

$U_{(s_1\ s_2),\Psi} \cup V_{(s_1\ s_2),\Psi} \cup W_{(s_1\ s_2),\Psi}$ has mgu φ

iff $U_{s_1,\Psi} \cup V_{s_1,\Psi} \cup W_{s_1,\Psi} \cup U_{s_2,\Psi} \cup V_{s_2,\Psi} \cup W_{s_2,\Psi} \cup X \cup \{\alpha = \gamma\}$ has mgu $\theta\varphi$, where

$X = \{\gamma_1 = \gamma_2 \mid$ there is a variable that is either free with relative type γ_i in some t_j, does not occur in the bound set of $\boldsymbol{x}_j$, and $\boldsymbol{x}_j$ occurs in s_i or else is free with relative type γ_i in s_i, for $i = 1, 2\}$

iff there exists type substitutions φ_1, φ_2 and η such that $U_{s_i,\Psi} \cup V_{s_i,\Psi} \cup W_{s_i,\Psi}$ has mgu φ_i, for $i = 1, 2$, and $(X \cup \{\alpha = \gamma\})(\varphi_1 \cup \varphi_2)$ has mgu η, where $(\varphi_1 \cup \varphi_2)\eta = \theta\varphi$

[by Proposition 2.15]

iff there exists type substitutions φ_1, φ_2 and η such that $s_1\Psi$ is a term of type $(\alpha \to \beta)\varphi_1$, $s_2\Psi$ is a term of type $\gamma\varphi_2$ and, if y is a free variable in $s_i\Psi$ and y has relative type δ in s_i or some t_j, then y has relative type $\delta\varphi_i$ in $s_i\Psi$, for $i = 1, 2$, and $(X \cup \{\alpha = \gamma\})(\varphi_1 \cup \varphi_2)$ has mgu η, where $(\varphi_1 \cup \varphi_2)\eta = \theta\varphi$

[by the induction hypothesis]

iff there exists type substitutions φ_1, φ_2 and η such that $(s_1\ s_2)\Psi$ is a term of type $\beta\varphi_1\eta$; if y is a free variable in $(s_1\ s_2)\Psi$ and y has relative type ρ in some $s_i\Psi$, then y has relative type $\rho\eta$ in $(s_1\ s_2)\Psi$; and, if y is a free variable in $s_i\Psi$ and y has relative type δ in s_i or some t_j, then y has relative type $\delta\varphi_i$ in $s_i\Psi$, for $i = 1, 2$, and $(X \cup \{\alpha = \gamma\})(\varphi_1 \cup \varphi_2)$ has mgu η; where $(\varphi_1 \cup \varphi_2)\eta = \theta\varphi$

[by the definition of an application, since $(X \cup \{\alpha = \gamma\})(\varphi_1 \cup \varphi_2)$ are the equations that have to be solved to form $(s_1\ s_2)\Psi$]

$\Uparrow$ correct?

iff $(s_1\ s_2)\Psi$ is a term of type $\beta\theta\varphi$ and, if y is a free variable in $(s_1\ s_2)\Psi$ and y has relative type δ in $(s_1\ s_2)$ or some t_i, then y has relative type $\delta\varphi$ in $(s_1\ s_2)\Psi$.

Suppose that s has the form $(s_1, \ldots, s_n)$ with type $(\sigma_1 \times \cdots \times \sigma_n)\theta$, where s_i has type σ_i, for $i = 1, \ldots, n$, and θ is the associated mgu for $(s_1, \ldots, s_n)$. Then

$U_{(s_1,\ldots,s_n),\Psi} \cup V_{(s_1,\ldots,s_n),\Psi} \cup W_{(s_1,\ldots,s_n),\Psi}$ has mgu φ

iff $U_{s_1,\Psi} \cup V_{s_1,\Psi} \cup W_{s_1,\Psi} \cup \cdots \cup U_{s_n,\Psi} \cup V_{s_n,\Psi} \cup W_{s_n,\Psi} \cup X$ has mgu $\theta\varphi$, where

$X = \{\gamma_{j_1} = \cdots = \gamma_{j_m} \mid$ there is a variable that is either free with relative type γ_{j_p} in some t_i, does not occur in the bound set of $\boldsymbol{x}_i$, and $\boldsymbol{x}_i$ occurs in s_{j_p} or else is free with relative type γ_{j_p} in s_{j_p} $(1 \leq p \leq m$ and $m > 1)\}$

iff there exists type substitutions $\varphi_1, \ldots, \varphi_n$ and η such that $U_{s_i,\Psi} \cup V_{s_i,\Psi} \cup W_{s_i,\Psi}$ has mgu φ_i, for $i = 1, \ldots, n$, and $X(\varphi_1 \cup \cdots \cup \varphi_n)$ has mgu η, where $(\varphi_1 \cup \cdots \cup \varphi_n)\eta = \theta\varphi$

[by Proposition 2.15]

iff there exists type substitutions $\varphi_1, \ldots, \varphi_n$ and η such that $s_i\Psi$ is a term of type $\sigma_i\varphi_i$ and, if y is a free variable in $s_i\Psi$ and y has relative type δ in s_i or some t_j, then y has relative type $\delta\varphi_i$ in $s_i\Psi$, for $i = 1, \ldots, n$, and $X(\varphi_1 \cup \cdots \cup \varphi_n)$ has mgu η, where $(\varphi_1 \cup \cdots \cup \varphi_n)\eta = \theta\varphi$

[by the induction hypothesis]

iff there exists type substitutions $\varphi_1, \ldots, \varphi_n$ and η such that $(s_1, \ldots, s_n)\Psi$ is a term of type $(\sigma_1\varphi_1 \times \cdots \times \sigma_n\varphi_n)\eta$; if y is a free variable in $(s_1, \ldots, s_n)\Psi$ and y has relative type ρ in some $s_i\Psi$, then y has relative type $\rho\eta$ in $(s_1, \ldots, s_n)\Psi$; and, if y is a free variable in $s_i\Psi$ and y has relative type δ in s_i or some t_j, then y has relative type $\delta\varphi_i$ in $s_i\Psi$, for $i = 1, \ldots, n$, and $X(\varphi_1 \cup \cdots \cup \varphi_n)$ has mgu η; where $(\varphi_1 \cup \cdots \cup \varphi_n)\eta = \theta\varphi$

[by the definition of a tuple, since $X(\varphi_1 \cup \cdots \cup \varphi_n)$ are the equations that have to be solved to form $(s_1, \ldots, s_n)\Psi$]

$\Uparrow$ correct?

iff $(s_1, \ldots, s_n)\Psi$ is a term of type $(\sigma_1 \times \cdots \times \sigma_n)\theta\varphi$ and, if y is a free variable in $(s_1, \ldots, s_n)\Psi$ and y has relative type δ in $(s_1, \ldots, s_n)$ or some t_i, then y has relative type $\delta\varphi$ in $(s_1, \ldots, s_n)\Psi$.

□

2.6 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.

Proposition 2.20. *Suppose that h is a term of the form $f\ t_1 \ldots t_n$, $n \geq 0$, for some function f, and b is a term that is type-weaker than h. Then $h = b$ is a statement.*

Proof. It is only necessary to show that $h = b$ is a term. Suppose that h has type σ. Then $(=\ h)$ is a term of type $\sigma \to \Omega$. Thus $((=\ h)\ b)$ is a term of type Ω since b is type-weaker than h. □

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

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

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 $\gamma = \{a'/List\ a\}$.

Proposition 2.21. *Let $h = b$ be a statement and θ a substitution such that $h\theta$ is a term. Then the following hold.*

1. *$b\theta$ is a term that is type-weaker than $h\theta$.*
2. *$h\theta = b\theta$ is a statement.*

Proof. 1. This part follows immediately from Proposition 2.16 and the fact that b is type-weaker than h.

2. This part follows immediately from Part 1 and Proposition 2.20. □

It will be important to also allow schemas 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.

Proposition 2.22. *Suppose that h is a schema of the form $f\ t_1 \ldots t_n$, $n \geq 0$, for some function f, and b is a schema that is type-weaker than h. Then $h = b$ is a statement schema.*

Proof. It is only necessary to show that $h = b$ is a schema. Suppose that h has type σ. Then $(=\ h)$ is a schema of type $\sigma \to \Omega$. Thus $((=\ h)\ b)$ is a schema of type Ω since b is type-weaker than h. □

Example. Consider the following equations for the function $\subseteq : (a \to \Omega) \to (a \to \Omega) \to \Omega$.

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

Clearly the first two equations 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. The following is a statement schema for the function *powerset* : $(a \to \Omega) \to (a \to \Omega) \to \Omega$ in which the head is type-equivalent to the body.

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

Proposition 2.23. *Let $h = b$ be a statement schema and Ψ a reifier for the syntactical variables in h such that $h\Psi$ is a term. Then the following hold.*

1. *$b\Psi$ is a term that is type-weaker than $h\Psi$.*
2. *$h\Psi = b\Psi$ is a statement.*

Proof. The reification of b by Ψ is well-defined since every syntactical variable in b appears in h.

1. This part follows immediately from Proposition 2.19 and the fact that b is type-weaker than h.
2. This part follows immediately from Part 1 and Proposition 2.22. □

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.

2.7 λ-conversion

The rules of λ-conversion can be incorporated into the logic in a straightforward way. The details of this are now given. But before giving these rules, since the logic is polymorphic, it is necessary to check that certain expressions are well-defined terms.

Proposition 2.24.

1. *If y is a variable that is not free in a term t and $\lambda x.t$ is a term, then $\lambda y.(t\{x/y\})$ is a term that is type-equivalent to $\lambda x.t$.*
2. *If $(\lambda x.s\ t)$ is a term, then $s\{x/t\}$ is a term that is type-weaker than $(\lambda x.s\ t)$.*
3. *If x is a variable that is not free in a term t and $\lambda x.(t\ x)$ is a term, then t is type-equivalent to $\lambda x.(t\ x)$.*

Proof. 1. This part is obvious.

2. Suppose that $(\lambda x.s\ t)$ is a term. It is now shown by induction on the structure of s that $s\{x/t\}$ is a term that is type-weaker than $(\lambda x.s\ t)$.

Suppose that s is a variable, say, y. If $y = x$, then $s\{x/t\} = t$. Thus $s\{x/t\}$ is a term and clearly $t \approx (\lambda x.x\ t)$. If $y \neq x$, then $s\{x/t\} = y$. Thus $s\{x/t\}$ is a term and clearly $y \precsim (\lambda x.y\ t)$.

Suppose that s is a constant, say, C. Hence $s\{x/t\} = C$. Thus $s\{x/t\}$ is a term and clearly $C \precsim (\lambda x.C\ t)$.

Suppose that s is an abstraction, say, $\lambda y.r$. Hence $s\{x/t\} = (\lambda y.r)\{x/t\} = \lambda y.(r\{x/t\})$. Clearly, $(\lambda x.r\ t)$ is a term, since t does not contain y as a free variable, and $\lambda y.(\lambda x.r\ t)$ is a term such that $\lambda y.(\lambda x.r\ t) \approx (\lambda x.(\lambda y.r)\ t)$. By the induction hypothesis, $r\{x/t\}$ is a term and $r\{x/t\} \precsim (\lambda x.r\ t)$. Hence $(\lambda y.r)\{x/t\} \precsim \lambda y.(\lambda x.r\ t) \approx (\lambda x.(\lambda y.r)\ t)$, and so $(\lambda y.r)\{x/t\} \precsim (\lambda x.(\lambda y.r)\ t)$, as required. (CHECK THIS)

Suppose that s is an application, say, $(u\ v)$. Hence $s\{x/t\} = (u\ v)\{x/t\} = (u\{x/t\}\ v\{x/t\})$. Clearly $(\lambda x.u\ t)$ and $(\lambda x.v\ t)$ are terms, and $((\lambda x.u\ t)\ (\lambda x.v\ t))$ is a term such that $((\lambda x.u\ t)\ (\lambda x.v\ t)) \approx (\lambda x.(u\ v)\ t)$. By the induction hypothesis, $u\{x/t\}$ and $v\{x/t\}$ are terms, and $u\{x/t\} \precsim (\lambda x.u\ t)$ and $v\{x/t\} \precsim (\lambda x.v\ t)$. Thus $(u\ v)\{x/t\} \precsim ((\lambda x.u\ t)\ (\lambda x.v\ t)) \approx (\lambda x.(u\ v)\ t)$, and so $(u\ v)\{x/t\} \precsim (\lambda x.(u\ v)\ t)$, as required.

Suppose that s is a tuple, say, $(s_1, \ldots, s_n)$. Hence $s\{x/t\} = (s_1, \ldots, s_n)\{x/t\} = (s_1\{x/t\}, \ldots, s_n\{x/t\})$. Clearly $(\lambda x.s_i\ t)$ is a term, for $i = 1, \ldots, n$, and $((\lambda x.s_1\ t), \ldots, (\lambda x.s_n\ t))$ is a term such that $((\lambda x.s_1\ t), \ldots, (\lambda x.s_n\ t)) \approx (\lambda x.(s_1, \ldots, s_n)\ t)$. By the induction hypothesis, $s_i\{x/t\}$ is a term, and $s_i\{x/t\} \precsim (\lambda x.s_i\ t)$, for $i = 1, \ldots, n$. Thus $(s_1, \ldots, s_n)\{x/t\} \precsim ((\lambda x.s_1\ t), \ldots, (\lambda x.s_n\ t)) \approx (\lambda x.(s_1, \ldots, s_n)\ t)$, and so $(s_1, \ldots, s_n)\{x/t\} \precsim (\lambda x.(s_1, \ldots, s_n)\ t)$, as required.

This completes the induction argument.

3. Suppose that t has type $\alpha \to \beta$. Then $(t\ x)$ has type β, since x is not free in t. Also x has relative type α in $(t\ x)$. Hence $\lambda x.(t\ x)$ has type $\alpha \to \beta$. Clearly t and $(t\ x)$ have the same set of free variables and each such free variable has the same relative type in t as it has in $\lambda x.(t\ x)$. □

Examination of the proof of Part 2 of Proposition 2.24 shows that $s\{x/t\}$ and $(\lambda x.s\ t)$ actually have the same type (up to variants). However, it is not generally true that $s\{x/t\} \approx (\lambda x.s\ t)$.

Example. Let M be a nullary type constructor and $C : M$ a constant. Let s be C and let t be the variable y. Then $s\{x/t\}$ is C, but $(\lambda x.s\ t)$ is $(\lambda x.C\ y)$ which contains the additional free variable y.

Proposition 2.25. *Let u be a term and v a subterm of u at occurrence p.*

1. *If v is $\lambda x.t$, where y is a variable that is not free in t, then $u[\lambda x.t/\lambda y.(t\{x/y\})]_p$ is a term that is type-equivalent to u.*
2. *If v is $(\lambda x.s\ t)$, then $u[(\lambda x.s\ t)/s\{x/t\}]_p$ is a term that is type-weaker than u.*
3. *If v is $\lambda x.(t\ x)$, where x is a variable that is not free in t, then $u[\lambda x.(t\ x)/t]_p$ is a term that is type-equivalent to u.*

Proof. Parts 1 and 3 follow immediately from Propositions 2.13 and 2.24, and Part 2 from Propositions 2.12 and 2.24. □

Note that it is not generally true that $u[(\lambda x.s\ t)/s\{x/t\}]_p \approx u$.

Example. Let M and N be nullary type constructors, and $C : M$ and $F : M \to N$ be constants. Let s be C, t be $(F\ y)$, where y is a variable, and u be $(y, (\lambda x.C\ (F\ y))$, which has type $M \times M$. But $u[(\lambda x.s\ t)/s\{x/t\}]_p$ is (y, C), which has type $a \times M$, for some parameter a.

I now define three relations $\succ_\alpha$, $\succ_\beta$, and $\succ_\eta$, corresponding to α-conversion, β-reduction, and η-reduction, respectively.

Definition. The rules of λ-conversion are as follows.

1. (α-conversion) $\lambda x.t \succ_\alpha \lambda y.(t\{x/y\})$, if y is not free in t.
2. (β-reduction) $(\lambda x.s\ t) \succ_\beta s\{x/t\}$.
3. (η-reduction) $\lambda x.(t\ x) \succ_\eta t$, if x is not free in t.

Definition. The relation $\longrightarrow_\alpha$ is defined by $u \longrightarrow_\alpha u[s/t]_p$ if s is a subterm of u at occurrence p and $s \succ_\alpha t$. Similarly, define $\longrightarrow_\beta$ and $\longrightarrow_\eta$. Let $\longrightarrow_{\beta\eta}$ be $\longrightarrow_\beta \cup \longrightarrow_\eta$.

Let $\xrightarrow{*}_{\beta\eta}$ be the reflexive, transitive closure of $\longrightarrow_{\beta\eta}$. Similarly, define $\xrightarrow{*}_\alpha$, $\xrightarrow{*}_\beta$ and $\xrightarrow{*}_\eta$.

Let $\xleftrightarrow{*}_{\beta\eta}$ be the reflexive, symmetric, and transitive closure of $\longrightarrow_{\beta\eta}$. Similarly, define $\xleftrightarrow{*}_\alpha$, $\xleftrightarrow{*}_\beta$ and $\xleftrightarrow{*}_\eta$.

Definition. If $s \xleftrightarrow{*}_\alpha t$ (resp., $s \xleftrightarrow{*}_\beta t$, $s \xleftrightarrow{*}_{\beta\eta} t$), then s and t are said to be *α-equivalent* (resp., *β-equivalent*, *$\beta\eta$-equivalent*).

Note that α-equivalent terms differ only in the names of their bound variables.

Proposition 2.26. *Let s and t be terms.*

1. *If $s \xrightarrow{*}_\alpha t$, then $s \approx t$.*
2. *If $s \xrightarrow{*}_\beta t$, then $s \precsim t$.*
3. *If $s \xrightarrow{*}_\eta t$, then $s \approx t$.*
4. *If $s \xrightarrow{*}_{\beta\eta} t$, then $s \precsim t$.*

Proof. These results follow immediately from Proposition 2.25. □

It is simple to incorporate the rules of λ-conversion into the logic. All that is necessary is to have an equality axiom for each rule. In fact, the axiom for α-conversion is essentially implicit in the logic, while the axiom for η-reduction is not used in this paper since the applications considered do not need it. The axiom for β-reduction is considered in the next subsection.

2.8 Axioms

In this subsection, I discuss a suitable set of axioms for the logic.

First consider the equality predicate $=$. In formulations of higher-order logics similar to the one proposed here, it is common for the axioms for equality to include the axiom of extensionality:

$$f = g \leftrightarrow \forall x.((f\ x) = (g\ x)).$$

However, this axiom is not used here for the reason that it is not computationally useful – showing that $\forall x.((f\ x) = (g\ x))$ is not generally possible as there can be infinitely many values of x to consider. Instead, several special cases of the axiom of extensionality are used. Here are the axioms for $=$.

$= :\ a \to a \to \Omega$

$(C\ x_1 \ldots x_n = C\ y_1 \ldots y_n) = (x_1 = y_1) \wedge \cdots \wedge (x_n = y_n)$

% where C is a data constructor of arity n. If $n = 0$, then the RHS is $\top$.

$(C\ x_1 \ldots x_n = D\ y_1 \ldots y_m) = \bot$

% where C is a data constructor of arity n, D is a data constructor of arity m, and $C \neq D$.

$(\lambda x.\boldsymbol{u} = \lambda y.\boldsymbol{v}) = (\mathit{less}\ \lambda x.\boldsymbol{u}\ \lambda y.\boldsymbol{v}) \wedge (\mathit{less}\ \lambda y.\boldsymbol{v}\ \lambda x.\boldsymbol{u})$

$((x_1, \ldots, x_n) = (y_1, \ldots, y_n)) = (x_1 = y_1) \wedge \cdots \wedge (x_n = y_n)$.

The first two of these axioms simply capture the intended meaning of data constructors, while the fourth axiom captures the major property of tuples. However, the third axiom is more subtle.

Its purpose is to provide a method of checking whether certain abstractions are equal. These abstractions represent finite sets, finite multisets and similar data types. In such cases, one can check for equality in a finite amount of time. The explanation of how the predicate *less* in this axiom is used to do this is given in Section 5.

The next axiom corresponds to β-reduction.

$$\lambda x.\boldsymbol{u} : a \to b$$
$$\lambda x.\boldsymbol{u}\ t = \boldsymbol{u}\{x/t\}.$$

The idea here is to regard $\lambda x.\boldsymbol{u}$ as the name of an anonymous function whose signature is $a \to b$. (This axiom is a special one since it is not a statement schema as its head does not have the form $f\ s_1 \ldots s_n$, for some function f.) Then the result of applying the function $\lambda x.\boldsymbol{u}$ to an argument t is simply given by the β-reduction rule.

The axioms for the connectives $\neg$, $\wedge$, and $\vee$, and the (generalised) quantifiers Σ and Π are given in Appendix A. Also, for each defined function, there is a set of axioms contained in the definition for the function. The definitions for some typical functions are also given in Appendix A.

2.9 Model Theory

Next I turn to the semantics of the logic, which is derived from the semantics for type theory originally given by Henkin [Hen50]. The main concept is that of an interpretation which provides a meaning for the symbols used to model an application.

Definition. A type substitution γ is *closed* if, for each binding a/α in γ, α is closed.

Definition. For each $\alpha \in \mathfrak{S}^c$, a *domain* $\mathcal{D}_\alpha$ corresponding to α is a non-empty set satisfying the following conditions.

1. If α has the form $T\ \alpha_1 \ldots \alpha_k$, then $\mathcal{D}_\alpha = \{C\ d_1 \ldots d_n \mid C$ is a data constructor having signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$, $a_1\gamma = \alpha_1, \ldots, a_k\gamma = \alpha_k$, for some closed type substitution γ, and $d_i \in \mathcal{D}_{\sigma_i\gamma}$, for $i = 1, \ldots, n\}$.

2. If α has the form $\beta \to \gamma$, then $\mathcal{D}_\alpha$ is the collection of all mappings from $\mathcal{D}_\beta$ to $\mathcal{D}_\gamma$.

3. If α has the form $\alpha_1 \times \cdots \times \alpha_n$, for some $n \geq 0$, then $\mathcal{D}_\alpha$ is the cartesian product $\mathcal{D}_{\alpha_1} \times \cdots \times \mathcal{D}_{\alpha_n}$. In particular, $\mathcal{D}_{\mathit{1}}$ is the distinguished singleton set.

For some (closed) types, these conditions uniquely specify the corresponding domain. However, for recursive data types, such as lists, they do not.

Example. For Ω, the only relevant data constructors are $\top$ and $\bot$. Consequently, $\mathcal{D}_\Omega = \{\top, \bot\}$.

Example. For *Int*, the only relevant data constructors are the integers. Thus $\mathcal{D}_{Int} = \mathbb{Z}$.

Example. For *Float*, the only relevant data constructors are the floating-point numbers. Let $\mathbb{F}$ be the set of floating-point numbers. Then $\mathcal{D}_{Float} = \mathbb{F}$.

Example. Consider an application involving geometrical shapes, say, circles and rectangles. To model this, one could introduce a type *Shape* and two data constructors
Circle : *Float* $\to$ *Shape*
Rectangle : *Float* $\to$ *Float* $\to$ *Shape*.
So, for example, a rectangle with length 23.5 and breadth 12.6 would be represented by the term (*Rectangle* 23.5 12.6). Then $\mathcal{D}_{Shape} = \{\mathit{Circle}\ f \mid f \in \mathbb{F}\} \cup \{\mathit{Rectangle}\ f\ g \mid f, g \in \mathbb{F}\}$.

Example. There are two possible domains corresponding to *List Int*. The smaller one is

$$\mathcal{D}_{List\ Int} = \{p_1 : p_2 : \cdots : p_n : [] \mid p_i \in \mathbb{Z},\ 1 \leq i \leq n, \text{ and } n \in \mathbb{N}\},$$

that is, the set of finite lists of integers.

The other domain is the set of finite or infinite lists of integers

$$\mathcal{D}'_{List\ Int} = \mathcal{D}_{List\ Int} \cup \{p_1 : p_2 : \cdots \mid p_i \in \mathbb{Z} \text{ and } i \geq 1\}.$$

Definition. An *interpretation* for an alphabet consists of a pair $\langle \{\mathcal{D}_\alpha\}_{\alpha \in \mathfrak{S}^c},\ V\rangle$, where $\{\mathcal{D}_\alpha\}_{\alpha \in \mathfrak{S}^c}$ is a collection of domains for the alphabet and V is a mapping which maps each pair consisting of a closed type substitution γ and a constant having signature α, say, to an element of $\mathcal{D}_{\alpha\gamma}$ (called the *denotation* of the constant wrt γ and the interpretation), such that the following conditions are satisfied.

1. If C is a data constructor having signature $\alpha = \sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$, then $V(\gamma, C)$ is the element of $\mathcal{D}_{\alpha\gamma}$ defined by $V(\gamma, C)\ d_1 \ldots d_n = C\ d_1 \ldots d_n$, where $d_i \in \mathcal{D}_{\sigma_i\gamma}$ $(i = 1, \ldots, n)$.

2. For $=$ with signature $a \to a \to \Omega$, $V(\gamma, =)$ is the mapping from $\mathcal{D}_{a\gamma}$ into $\mathcal{D}_{a\gamma\to\Omega}$ defined by $V(\gamma, =)\ x\ y = \begin{cases} \top & \text{if } x = y \\ \bot & \text{otherwise} \end{cases}$

3. $V(\gamma, \neg)$ is the mapping from $\mathcal{D}_\Omega$ into $\mathcal{D}_\Omega$ given by the following table.

x	$V(\gamma, \neg)\ x$
$\top$	$\bot$
$\bot$	$\top$

4. $V(\gamma, \wedge)$, $V(\gamma, \vee)$, $V(\gamma, \longrightarrow)$, $V(\gamma, \longleftarrow)$, and $V(\gamma, \longleftrightarrow)$ are the mappings from $\mathcal{D}_\Omega$ into $\mathcal{D}_{\Omega\to\Omega}$ given by the following table.

x	y	$V(\gamma, \wedge)\ x\ y$	$V(\gamma, \vee)\ x\ y$	$V(\gamma, \longrightarrow)\ x\ y$	$V(\gamma, \longleftarrow)\ x\ y$	$V(\gamma, \longleftrightarrow)\ x\ y$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\bot$
$\bot$	$\top$	$\bot$	$\top$	$\top$	$\bot$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$

5. For Σ with signature $(a \to \Omega) \to \Omega$, $V(\gamma, \Sigma)$ is the mapping from $\mathcal{D}_{a\gamma\to\Omega}$ to $\mathcal{D}_\Omega$ which maps an element f of $\mathcal{D}_{a\gamma\to\Omega}$ to $\top$ if f maps at least one element of $\mathcal{D}_{a\gamma}$ to $\top$; otherwise, it maps f to $\bot$.

6. For Π with signature $(a \to \Omega) \to \Omega$, $V(\gamma, \Pi)$ is the mapping from $\mathcal{D}_{a\gamma\to\Omega}$ to $\mathcal{D}_\Omega$ which maps an element f of $\mathcal{D}_{a\gamma\to\Omega}$ to $\top$ if f maps every element of $\mathcal{D}_{a\gamma}$ to $\top$; otherwise, it maps f to $\bot$.

Definition. A *variable assignment* wrt an interpretation $\langle \{\mathcal{D}_\alpha\}_{\alpha \in \mathfrak{S}^c},\ V\rangle$ is a mapping which maps each pair consisting of a closed type α and a variable to an element of $\mathcal{D}_\alpha$.

I can now define the meaning of a term wrt an interpretation, a closed type substitution, and a variable assignment.

Definition. Let $I = \langle \{\mathcal{D}_\alpha\}_{\alpha \in \mathfrak{S}^c},\ V\rangle$ be an interpretation, γ a closed type substitution, and φ a variable assignment wrt I. Then the *denotation* $\mathcal{V}(I, \gamma, \varphi, t)$ of a term t of type α wrt I, γ, and φ is an element of $\mathcal{D}_{\alpha\gamma}$ defined inductively as follows.

1. $\mathcal{V}(I, \gamma, \varphi, x) = \varphi(a\gamma, x)$, where x is a variable of type a.

2. $\mathcal{V}(I, \gamma, \varphi, C) = V(\gamma, C)$, where C is a constant.

3. $\mathcal{V}(I, \gamma, \varphi, \lambda x.t) =$ the function from $\mathcal{D}_{\alpha\gamma}$ into $\mathcal{D}_{\beta\gamma}$ whose value for each $d \in \mathcal{D}_{\alpha\gamma}$ is $\mathcal{V}(I, \gamma, \varphi', t)$, where $\lambda x.t$ has type $\alpha \to \beta$ and φ' is φ except $\varphi'(\alpha\gamma, x) = d$.

4. $\mathcal{V}(I, \gamma, \varphi, (s\ t)) = \mathcal{V}(I, \theta\gamma, \varphi, s)(\mathcal{V}(I, \theta\gamma, \varphi, t))$, where θ is the associated mgu for $(s\ t)$.

5. $\mathcal{V}(I, \gamma, \varphi, (t_1, \ldots, t_n)) = (\mathcal{V}(I, \theta\gamma, \varphi, t_1), \ldots, \mathcal{V}(I, \theta\gamma, \varphi, t_n))$, where θ is the associated mgu for $(t_1, \ldots, t_n)$.

If t is a closed term, then $\mathcal{V}(I, \gamma, \varphi, t)$ is independent of φ. If t is a monomorphic term, then $\mathcal{V}(I, \gamma, \varphi, t)$ is independent of γ.

Definition. A term of type Ω is called a *formula*. A *theory* is a collection of formulas.

Definition. Let t be a formula, I an interpretation, and φ a variable assignment wrt I.

1. φ *satisfies* t *in* I if $\mathcal{V}(I, \gamma, \varphi, t) = \top$, for all closed type substitutions γ.

2. t is *satisfiable in* I if there is a variable assignment which satisfies t in I.

3. t is *valid in* I if every variable assignment satisfies t in I.

4. t is *valid* if t is valid in every interpretation.

5. A *model* for a theory S is an interpretation in which each formula in S is valid.

Definition. A theory is *consistent* if it has a model.

Definition. Let S be a theory and t a formula. Then t is a *logical consequence* of S if t is valid in every model for S.

A semantics is also needed for schemas. The essential concept is that of a model for a set of schemas of type Ω.

Definition. Let s be a schema of type Ω in some alphabet and I an interpretation for the alphabet.

1. s is *valid in* I if $s\theta$ is valid in I, for all reifiers θ such that $s\theta$ is a formula.

2. s is *valid* if s is valid in every interpretation.

3. A *model* for a set S of schemas of type Ω is an interpretation in which each schema in S is valid.

Definition. Let S be a set of schemas and t a formula. Then t is a *logical consequence* of S if t is valid in every model for S.

2.10 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 subsection, I assume that each definition is given in the context of some program.

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) [And86, p. 164]. Also, for the purposes of the proof theory, the axiom for β-reduction is regarded as a statement.

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 [Wol93] 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*.

One interesting aspect of the definition of a computation step is that there are no checks to make sure that the new term obtained from a computation step really is a term, that is, really is type correct. Propositions 2.27 and 2.28 below show that a computation step respects the type requirements. This kind of result is known by the phrase 'run-time type checking is unnecessary'. A first-order version of this result was first obtained in [MO84] and later in [HT92].

Proposition 2.27. *Let t be a term and $h = b$ a statement. Suppose there is a subterm r of t at occurrence p and a substitution θ such that r is α-equivalent to $h\theta$. Then $t[r/b\theta]_p$ is a term that is type-weaker than t. Furthermore, $t = t[r/b\theta]_p$ is a term.*

Proof. The result follows directly from Propositions 2.12 and 2.16. □

The proof of the previous result crucially depends on the definition of the concept of a statement, in particular, on the requirement that the body of the statement be type-weaker than the head. Here are two examples to show that this requirement cannot be dropped.

Example. Consider the alphabet given by the nullary type constructors M and N, and the constants $p : a \to \Omega$, $q : M \to \Omega$, and $r : N \to \Omega$.

Then the expression $q(y) \wedge r(y)$ can be obtained by a computation step from the term $p(y) \wedge r(y)$ and the putative statement $p(x) = q(x)$ (using the redex $p(y)$). However, the expression $q(y) \wedge r(y)$ is not a term. The problem here is caused by the fact that $p(x) = q(x)$ is not a statement because $q(x)$ is not type-weaker than $p(x)$: the x in $p(x)$ has relative type a in $p(x)$, but the x in $q(x)$ has relative type M in $q(x)$.

Example. Consider the alphabet given by the nullary type constructors K, M, and N, and the constants $f : a \to K$, $g : K \times K \to \Omega$, $A : M$, and $B : N$.

Then the expression $A = B$ can be obtained by a computation step from the term $g(f(A), f(B))$ and the putative statement $g(f(x), f(y)) = (x = y)$. However, the expression $A = B$ is not a term. The problem here is caused by the fact that $g(f(x), f(y)) = (x = y)$ is not a statement because $x = y$ is not type-weaker than $g(f(x), f(y))$: the x and y have the same relative type a in $x = y$, but they have relative types a and b in $g(f(x), f(y))$.

The analogous result to Proposition 2.27 for statement schemas also holds.

Proposition 2.28. *Let t be a term and $h = b$ a statement schema. Suppose there is a subterm r of t at occurrence p and a reifier Ψ and a substitution θ such that r is α-equivalent to $h\Psi\theta$. Then $t[r/b\Psi\theta]_p$ is a term that is type-weaker than t. Furthermore, $t = t[r/b\Psi\theta]_p$ is a term.*

Proof. The result follows directly from Propositions 2.12, 2.16, and 2.19. □

The next result establishes the soundness of the proof procedure.

Proposition 2.29. *Let P be a program, s a goal and t an answer of a computation. Then $s = t$ is a logical consequence of P.*

Proof. The proof uses Propositions 2.27 and 2.28. □

The proof system can also be regarded as a rewrite system, in which each statement $h = b$ is regarded as a rewrite $h \to b$. In this terminology, a term is *irreducible* if it does not contain a redex.

3 Representation of Individuals

In this section, I study the application of the logic to knowledge representation. The main contribution is the identification of 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 5.

3.1 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 [Llo99] 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 $x \in \mathfrak{V}$, 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 $(t_1, \ldots, t_n) \in \mathfrak{D}$.

Note. To be precise, the meaning of the previous inductive definition is that $\mathfrak{D}$ is the intersection of all sets of terms each satisfying (appropriately reworded versions of) Conditions 1 to 3. A suitable universe for the construction is the set $\mathfrak{L}$ of all terms.

To prove properties of default terms, one can employ the following *principle of induction on the structure of default terms.*

Proposition 3.1. *Let $\mathfrak{X}$ be a subset of $\mathfrak{D}$ satisfying the following conditions.*

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

2. *If $t \in \mathfrak{X}$ and $x \in \mathfrak{V}$, then $\lambda x.t \in \mathfrak{X}$.*

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

Then $\mathfrak{X} = \mathfrak{D}$.

Proof. Since $\mathfrak{X}$ satisfies Conditions 1 to 3 of the definition of a default term, it follows that $\mathfrak{D} \subseteq \mathfrak{X}$. Thus $\mathfrak{X} = \mathfrak{D}$. □

Proposition 3.2. *Each default term is closed and irreducible.*

Proof. The proof is by induction on the structure of default terms.

Suppose first that the default term has the form $C\ t_1 \ldots t_n$. Hence each t_i is a default term. By the induction hypothesis, each t_i is closed and irreducible. Thus $C\ t_1 \ldots t_n$ is closed. It is also irreducible as there is no rewrite matching $C\ t_1 \ldots t_n$ and each t_i is irreducible.

Suppose next that the default term has the form $\lambda x.t$. Then t is a default term, and is closed and irreducible, by the induction hypothesis. Hence $\lambda x.t$ is closed. It is also irreducible as there is no rewrite matching $\lambda x.t$, and t is irreducible.

Finally, suppose that the default term has the form $(t_1, \ldots, t_n)$. Hence each t_i is a default term. By the induction hypothesis, each t_i is closed and irreducible. Thus $(t_1, \ldots, t_n)$ is closed. It is also irreducible as there is no rewrite matching $(t_1, \ldots, t_n)$ and each t_i is irreducible. □

Proposition 3.3. *Each subterm of a default term is a default term.*

Proof. □

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

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

Note that $\mathfrak{D} = \bigcup_{\alpha \in \mathfrak{S}^c} \mathfrak{D}_\alpha$. However, the $\mathfrak{D}_\alpha$ may not be disjoint. For example, if the alphabet includes the type constructor *List* and [] is the default data constructor for *List*, then $[] \in \mathfrak{D}_{List\ \alpha}$, for each closed type α. Furthermore, $\mathfrak{D}_\alpha$ may be empty, for some α.

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 $\mathfrak{D}_M$ is empty.

The next definition will provide a necessary and sufficient condition for each $\mathfrak{D}_\alpha$ to be non-empty.

Definition. A data constructor C having signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$ is *full* if, for all $\alpha_1, \ldots, \alpha_k \in \mathfrak{S}^c$, it is true that $\mathfrak{D}_{\sigma_i \xi} \neq \emptyset$, for $i = 1, \ldots, n$, where $\xi = \{a_1/\alpha_1, \ldots, a_k/\alpha_k\}$.

In particular, if $n = 0$, then C is full.

Proposition 3.4. *$\mathfrak{D}_\alpha \neq \emptyset$, for each $\alpha \in \mathfrak{S}^c$, iff each default data constructor is full.*

Proof. Suppose first that each default data constructor is full. The proof that $\mathfrak{D}_\alpha \neq \emptyset$, for each $\alpha \in \mathfrak{S}^c$, proceeds by induction on the structure of closed types.

Let $\alpha \in \mathfrak{S}^c$ have the form $T\ \alpha_1 \ldots \alpha_k$ and suppose C is the default data constructor associated with T. Let $\xi = \{a_1/\alpha_1, \ldots, a_k/\alpha_k\}$. Since C is full, there exist $t_i \in \mathfrak{D}_{\sigma_i \xi}$, for $i = 1, \ldots, n$. Then $C\ t_1 \ldots t_n \in \mathfrak{D}_{T\ \alpha_1 \ldots \alpha_k}$. (Note that this part does not need the induction hypothesis.)

Let $\alpha = \beta \to \gamma$. By the induction hypothesis, there exists $t \in \mathfrak{D}_\gamma$. Thus $\lambda x.t \in \mathfrak{D}_\alpha$.

Let $\alpha = \alpha_1 \times \cdots \times \alpha_n$. By the induction hypothesis, there exists $t_i \in \mathfrak{D}_{\alpha_i}$, for $i = 1, \ldots, n$. Thus $(t_1, \ldots, t_n) \in \mathfrak{D}_\alpha$.

Conversely, suppose that there exists a default data constructor C having signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$ that is not full. Hence there exist $\alpha_1, \ldots, \alpha_k \in \mathfrak{S}^c$ such that $\mathfrak{D}_{\sigma_{i_0} \xi} = \emptyset$, for some $i_0 \in \{1, \ldots, n\}$, where $\xi = \{a_1/\alpha_1, \ldots, a_k/\alpha_k\}$. Consequently, $\mathfrak{D}_{T\ \alpha_1 \ldots \alpha_k} = \emptyset$. □

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

Proposition 3.5. *For each $\alpha \in \mathfrak{S}^c$, there exists at most one default term in $\mathfrak{D}_\alpha$.*

Proof. The proof is by induction on the structure of default terms. Suppose $s, t \in \mathfrak{D}$ and s and t each have a type more general than some $\alpha \in \mathfrak{S}^c$. There are three cases depending on the form of α.

Suppose that $\alpha = T\ \alpha_1 \ldots \alpha_k$, for some $T, \alpha_1, \ldots, \alpha_k$. Let C be the unique default data constructor for T, having signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$, say, and $\xi = \{a_1/\alpha_1, \ldots, a_k/\alpha_k\}$. Thus $s = C\ s_1 \ldots s_n$, for some $s_1, \ldots, s_n$ and $t = C\ t_1 \ldots t_n$, for some $t_1, \ldots, t_n$, where $s_i, t_i \in \mathfrak{D}$, for $i = 1, \ldots, n$. I claim that s_i has type more general than $\sigma_i \xi$, for $i = 1, \ldots, n$. Let τ_i be the type of s_i and θ the mgu of $\sigma_1, \ldots, \sigma_n$ and $\tau_1, \ldots, \tau_n$. Then $\sigma_i \theta = \tau_i \theta$ and so τ_i is more general than $\sigma_i \theta$, for $i = 1, \ldots, n$. But $(T\ a_1 \ldots a_n)\theta$ is more general than $(T\ a_1 \ldots a_n)\xi$. Thus $\sigma_i \theta$ is more general than $\sigma_i \xi$ and so τ_i is more general than $\sigma_i \xi$. Similarly, t_i has type more general than $\sigma_i \xi$, for $i = 1, \ldots, n$. It follows by the induction hypothesis that $s_i = t_i$, for $i = 1, \ldots, n$. Thus $s = t$.

Suppose that $\alpha = \beta \to \gamma$, for some β, γ. Then $s = \lambda x.u$ and $t = \lambda x.v$, for some $u, v \in \mathfrak{D}$. Thus u and v both have type more general than γ. By the induction hypothesis, $u = v$ and so $s = t$.

Suppose that $\alpha = \alpha_1 \times \cdots \times \alpha_n$, for some $\alpha_1, \ldots, \alpha_n$. Then $s = (s_1, \ldots, s_n)$, for some $s_1, \ldots, s_n \in \mathfrak{D}$, and $t = (t_1, \ldots, t_n)$, for some $t_1, \ldots, t_n \in \mathfrak{D}$. Thus s_i and t_i have type more general than α_i, for $i = 1, \ldots, n$. By the induction hypothesis, $s_i = t_i$, for $i = 1, \ldots, n$. Thus $s = t$. □

In the second part of the previous proof it may happen that s and t have different bound variables. This is an unimportant difference, that is, I regard identity of terms as being "identity up to α-conversion". See the note in Subsection 3.3 below.

It follows from Propositions 3.3 and 3.4 that if each default data constructor is full, then $\mathfrak{D}_\alpha$ is a singleton set, for each $\alpha \in \mathfrak{S}^c$.

A bottom-up characterisation of default terms will be needed.

Definition. Define $\{\mathfrak{D}_m\}_{m \in \mathbb{N}}$ inductively as follows:
$\mathfrak{D}_0 = \{C \mid C \text{ is a default data constructor of arity } 0\}$
$\mathfrak{D}_{m+1} = \mathfrak{D}_m \cup$
$\{C\ t_1 \ldots t_n \in \mathfrak{L} \mid C \text{ is a default data constructor and } t_1, \ldots, t_n \in \mathfrak{D}_m (n > 0)\} \cup$
$\{\lambda x.t \in \mathfrak{L} \mid t \in \mathfrak{D}_m\} \cup$
$\{(t_1, \ldots, t_n) \in \mathfrak{L} \mid t_1, \ldots, t_n \in \mathfrak{D}_m\}$.

Clearly, $\mathfrak{D}_m \subseteq \mathfrak{D}_{m+1}$, for $m \in \mathbb{N}$.

Proposition 3.6. $\mathfrak{D} = \bigcup_{m \in \mathbb{N}} \mathfrak{D}_m$.

Proof. First, I show that $\bigcup_{m\in\mathbb{N}} \mathfrak{D}_m \subseteq \mathfrak{D}$. To prove this, it suffices to show by induction that $\mathfrak{D}_m \subseteq \mathfrak{D}$, for $m \in \mathbb{N}$. Clearly $\mathfrak{D}_0 \subseteq \mathfrak{D}$. Suppose next that $\mathfrak{D}_m \subseteq \mathfrak{D}$. It then follows from the definitions of $\mathfrak{D}_{m+1}$ and $\mathfrak{D}$ that $\mathfrak{D}_{m+1} \subseteq \mathfrak{D}$.

Now I show that $\mathfrak{D} \subseteq \bigcup_{m\in\mathbb{N}} \mathfrak{D}_m$. For this, it suffices to show that $\bigcup_{m\in\mathbb{N}} \mathfrak{D}_m$ satisfies Conditions 1, 2 and 3 in the definition of $\mathfrak{D}$ (since $\mathfrak{D}$ is the smallest such set). Suppose that C is a default data constructor, $t_1, \ldots, t_n \in \bigcup_{m\in\mathbb{N}} \mathfrak{D}_m$, and $C\ t_1 \ldots t_n \in \mathfrak{L}$. Since the $\mathfrak{D}_m$ are increasing, there exists $p \in \mathbb{N}$ such that $t_1, \ldots, t_n \in \mathfrak{D}_p$. Hence $C\ t_1 \ldots t_n \in \mathfrak{D}_{p+1}$ and so $C\ t_1 \ldots t_n \in \bigcup_{m\in\mathbb{N}} \mathfrak{D}_m$. Similar arguments show that $\bigcup_{m\in\mathbb{N}} \mathfrak{D}_m$ satisfies Conditions 2 and 3. □

Proposition 3.7. *If the set $\mathfrak{T}$ of type constructors is countable, then $\mathfrak{D}$ is countable.*

Proof. Since $\mathfrak{T}$ is countable and, for every type constructor, there is associated a unique default data constructor, the set of default data constructors is also countable. Since $\mathfrak{D} = \bigcup_{m\in\mathbb{N}} \mathfrak{D}_m$, it suffices to show by induction that $\mathfrak{D}_m$ is countable, for $m \in \mathbb{N}$. However, this follows easily from the definition of $\{\mathfrak{D}_m\}_{m\in\mathbb{N}}$. □

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 (*Circle* 7.5) and (*Rectangle* 42.0 21.3) are normal terms of type *Shape*. However, (*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. The term s_0 is called the *default value* for the abstraction. 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 $\alpha_1 \times \cdots \times \alpha_n$.

The next result is the *principle of induction on the structure of normal terms.*

Proposition 3.8. *Let $\mathfrak{X}$ be a subset of $\mathfrak{N}$ satisfying the following conditions.*

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

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

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

Then $\mathfrak{X} = \mathfrak{N}$.

Proof. Since $\mathfrak{X}$ satisfies Conditions 1 to 3 of the definition of a normal term, it follows that $\mathfrak{N} \subseteq \mathfrak{X}$. Thus $\mathfrak{X} = \mathfrak{N}$. □

Proposition 3.9. $\mathfrak{D} \subseteq \mathfrak{N}$.

Proof. This is a straightforward induction argument on the structure of default terms. □

Proposition 3.10. *Each normal term is closed and irreducible.*

Proof. The proof is by induction on the structure of normal terms.

Suppose first that the normal term has the form $C\ t_1 \ldots t_n$. Hence each t_i is a normal term. By the induction hypothesis, each t_i is closed and irreducible. Thus $C\ t_1 \ldots t_n$ is closed. It is also irreducible as there is no rewrite matching $C\ t_1 \ldots t_n$ and each t_i is irreducible.

Suppose next that t is the normal term $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$. Then each t_i and s_i is a normal term, and is closed and irreducible, by the induction hypothesis. Also the default term s_0 is closed and irreducible, by Proposition 3.2. Hence t is closed. It is also irreducible as there is no rewrite matching t, and s_0 and each t_i and s_i is irreducible.

Finally, suppose that the normal term has the form $(t_1, \ldots, t_n)$. Hence each t_i is a normal term. By the induction hypothesis, each t_i is closed and irreducible. Thus $(t_1, \ldots, t_n)$ is closed. It is also irreducible as there is no rewrite matching $(t_1, \ldots, t_n)$ and each t_i is irreducible. □

Proposition 3.11. *A subterm of a normal term is normal iff it is closed.*

Proof. □

One can 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 $\mathfrak{N}_\alpha$ may not be disjoint. For example, if the alphabet includes *List*, then $[] \in \mathfrak{N}_{List\ \alpha}$, for each closed type α. Furthermore, $\mathfrak{N}_\alpha$ may be empty, for some α.

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 $G : N \to M$. Then there are no *closed* terms of type M and hence $\mathfrak{N}_M$ is empty.

Since $\mathfrak{D} \subseteq \mathfrak{N}$, it follows that $\mathfrak{D}_\alpha \subseteq \mathfrak{N}_\alpha$, for each $\alpha \in \mathfrak{S}^c$.

Proposition 3.12. *If each default data constructor is full, then $\mathfrak{N}_\alpha \neq \emptyset$, for each $\alpha \in \mathfrak{S}^c$.*

Proof. By Proposition 3.4, since each default data constructor is full, $\mathfrak{D}_\alpha \neq \emptyset$, for each $\alpha \in \mathfrak{S}^c$. But $\mathfrak{D}_\alpha \subseteq \mathfrak{N}_\alpha$, for each $\alpha \in \mathfrak{S}^c$. Hence the result. □

It will be useful to gather together the types of all subterms of terms in $\mathfrak{N}_\alpha$.

Definition. For each $\alpha \in \mathfrak{S}^c$, the set *embed*$(\alpha)$ of *embedded types* in α is defined by *embed*$(\alpha) = \{\beta \mid$ there exists $t \in \mathfrak{N}_\alpha$ and a subterm s of t such that $s \in \mathfrak{N}_\beta$ $\}$.

Example. Let T_1, and T_2 be type constructors of arity 0. Suppose there is a data constructor having signature $T_1 \times T_1 \to T_1 \times T_1 \to T_2$, and there are data constructors having signature T_1. Consider the type $\alpha = T_2 \times \{Int\} \times List\ Float$. Then

$$embed(\alpha) = \{T_2 \times \{Int\} \times List\ Float, T_2, \{Int\}, List\ Float, T_1 \times T_1, Int, \Omega, Float, T_1\}.$$

Note carefully that *embed*(α) is not the same as the set of subtypes of α. In the previous example, $T_1 \times T_1 \in embed(\alpha)$, but it is not a subtype of α.

Proposition 3.13. *Let $s, t \in \mathfrak{N}$. Then exactly one of the following conditions holds.*

1. *There exists $\alpha \in \mathfrak{S}^c$ such that $s, t \in \mathfrak{N}_\alpha$, and s and t are both normal structures.*
2. *There exists $\alpha \in \mathfrak{S}^c$ such that $s, t \in \mathfrak{N}_\alpha$, and s and t are both normal abstractions.*
3. *There exists $\alpha \in \mathfrak{S}^c$ such that $s, t \in \mathfrak{N}_\alpha$, and s and t are both normal tuples.*
4. *There does not exist $\alpha \in \mathfrak{S}^c$ such that $s, t \in \mathfrak{N}_\alpha$.*

Proof. Suppose there exists $\alpha \in \mathfrak{S}^c$ such that $s, t \in \mathfrak{N}_\alpha$. Then each of s and t have type more general than α. Consequently, either both are normal structures or both are normal abstractions or both are normal tuples. □

Proposition 3.14.

1. *If $C\ t_1 \ldots t_n \in \mathfrak{N}_{T\ \alpha_1 \ldots \alpha_k}$, where C has signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$ and $\theta = \{a_1/\alpha_1, \ldots, a_k/\alpha_k\}$, then $t_i \in \mathfrak{N}_{\sigma_i\theta}$, for $i = 1, \ldots, n$.*
2. *If $\lambda x.if\ x = t_1\ then\ s_1\ else\ \ldots\ if\ x = t_n\ then\ s_n\ else\ s_0 \in \mathfrak{N}_{\beta\to\gamma}$, then $t_i \in \mathfrak{N}_\beta$, for $i = 1, \ldots, n$, and $s_i \in \mathfrak{N}_\gamma$, for $i = 0, \ldots, n$.*
3. *If $(t_1, \ldots, t_n) \in \mathfrak{N}_{\alpha_1 \times \cdots \times \alpha_n}$, then $t_i \in \mathfrak{N}_{\alpha_i}$, for $i = 1, \ldots, n$.*

Proof. The proof is by induction on the structure of normal terms.

Suppose first that $C\ t_1 \ldots t_n \in \mathfrak{N}_{T\ \alpha_1 \ldots \alpha_k}$. Then $C\ t_1 \ldots t_n \in \mathfrak{N}$ and so $t_1, \ldots, t_n \in \mathfrak{N}$ (since $\mathfrak{N}$ is the *smallest* set of terms satisfying the conditions in the definition of a normal term). Furthermore, t_i has type ρ_i, where $\sigma_i\theta = \rho_i\gamma$, for some γ, for $i = 1, \ldots, n$. Thus $t_i \in \mathfrak{N}_{\sigma_i\theta}$, for $i = 1, \ldots, n$.

Next suppose that $\lambda x.if\ x = t_1\ then\ s_1\ else\ \ldots\ if\ x = t_n\ then\ s_n\ else\ s_0 \in \mathfrak{N}_{\beta\to\gamma}$. Hence $\lambda x.if\ x = t_1\ then\ s_1\ else\ \ldots\ if\ x = t_n\ then\ s_n\ else\ s_0 \in \mathfrak{N}$ and so $t_i \in \mathfrak{N}$ and $s_i \in \mathfrak{N}$, for $i = 1, \ldots, n$. Now each t_i has type more general than β and each s_i has type more general than γ. Hence $t_i \in \mathfrak{N}_\beta$, for $i = 1, \ldots, n$, and $s_i \in \mathfrak{N}_\gamma$, for $i = 0, \ldots, n$.

Finally, suppose that $(t_1, \ldots, t_n) \in \mathfrak{N}_{\alpha_1 \times \ldots \times \alpha_n}$. Then $(t_1, \ldots, t_n) \in \mathfrak{N}$ and so $t_i \in \mathfrak{N}$, for $i = 1, \ldots, n$. Furthermore, each t_i has type more general than α_i. Thus $t_i \in \mathfrak{N}_{\alpha_i}$, for $i = 1, \ldots, n$. □

Proposition 3.15. *If $s, t \in \mathfrak{N}_{\beta\to\gamma}$, for some $\beta, \gamma \in \mathfrak{S}^c$, then s and t have the same default value.*

Proof. Let s_0 be the default value for s and t_0 the default value for t. By Proposition 3.6, $s_0, t_0 \in \mathfrak{N}_\gamma$. Hence, by Proposition 3.3, $s_0 = t_0$. □

Next, a bottom-up characterisation of $\mathfrak{N}$ is provided.

Definition. Define $\{\mathfrak{N}_m\}_{m\in\mathbb{N}}$ inductively as follows:
$\mathfrak{N}_0 = \{C \mid C \text{ is a data constructor of arity } 0\}$
$\mathfrak{N}_{m+1} = \mathfrak{N}_m \cup$
$\{C\ t_1 \ldots t_n \in \mathfrak{L} \mid C \text{ is a data constructor and } t_1, \ldots, t_n \in \mathfrak{N}_m (n > 0)\} \cup$
$\{\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{L} \mid t_1, \ldots, t_n \in \mathfrak{N}_m,$
$s_1, \ldots, s_n \in \mathfrak{N}_m, \text{ and } s_0 \in \mathfrak{D}\} \cup$
$\{(t_1, \ldots, t_n) \in \mathfrak{L} \mid t_1, \ldots, t_n \in \mathfrak{N}_m\}.$

Clearly, $\mathfrak{N}_m \subseteq \mathfrak{N}_{m+1}$, for $m \in \mathbb{N}$.

Proposition 3.16. $\mathfrak{N} = \bigcup_{m\in\mathbb{N}} \mathfrak{N}_m$.

Proof. First, I show that $\bigcup_{m\in\mathbb{N}} \mathfrak{N}_m \subseteq \mathfrak{N}$. To prove this, it suffices to show by induction that $\mathfrak{N}_m \subseteq \mathfrak{N}$, for $m \in \mathbb{N}$. Clearly $\mathfrak{N}_0 \subseteq \mathfrak{N}$. Suppose next that $\mathfrak{N}_m \subseteq \mathfrak{N}$. It then follows from the definitions of $\mathfrak{N}_{m+1}$ and $\mathfrak{N}$ that $\mathfrak{N}_{m+1} \subseteq \mathfrak{N}$.

Now I show that $\mathfrak{N} \subseteq \bigcup_{m\in\mathbb{N}} \mathfrak{N}_m$. For this, it suffices to show that $\bigcup_{m\in\mathbb{N}} \mathfrak{N}_m$ satisfies Conditions 1, 2 and 3 in the definition of $\mathfrak{N}$ (since $\mathfrak{N}$ is the smallest such set). Suppose that C is a data constructor, $t_1, \ldots, t_n \in \bigcup_{m\in\mathbb{N}} \mathfrak{N}_m$, and $C\ t_1 \ldots t_n \in \mathfrak{L}$. Since the $\mathfrak{N}_m$ are increasing, there exists $p \in \mathbb{N}$ such that $t_1, \ldots, t_n \in \mathfrak{N}_p$. Hence $C\ t_1 \ldots t_n \in \mathfrak{N}_{p+1}$ and so $C\ t_1 \ldots t_n \in \bigcup_{m\in\mathbb{N}} \mathfrak{N}_m$. Similar arguments show that $\bigcup_{m\in\mathbb{N}} \mathfrak{N}_m$ satisfies Conditions 2 and 3. □

Proposition 3.17. *If the set of data constructors is countable, then $\mathfrak{N}$ is countable.*

Proof. Since the set of data constructors is countable, the set $\mathfrak{T}$ of type constructors is also countable. Thus $\mathfrak{D}$ is countable, by Proposition 3.7. Now, since $\mathfrak{N} = \bigcup_{m\in\mathbb{N}} \mathfrak{N}_m$, it suffices to show by induction that $\mathfrak{N}_m$ is countable, for $m \in \mathbb{N}$. However, this follows easily from the definition of $\{\mathfrak{N}_m\}_{m\in\mathbb{N}}$. □

3.2 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. $\alpha = \alpha_1 \times \cdots \times \alpha_n$, 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 $i = 1, \ldots, n$.

One might conjecture that $\equiv$ is an equivalence relation on $\mathfrak{N}$, but this fails.

Example. Let $A_1 : T_1$ and $A_2 : T_2$ be nullary data constructors, r be $\lambda x.\mathit{if}\ x = A_1\ \mathit{then}\ \bot\ \mathit{else}\ \bot$, s be $\lambda x.\bot$, and t be $\lambda x.\mathit{if}\ x = A_2\ \mathit{then}\ \bot\ \mathit{else}\ \bot$. Then $r, s \in \mathfrak{N}_{T_1 \to \Omega}$ and $r \equiv s$. Furthermore, $s, t \in \mathfrak{N}_{T_2 \to \Omega}$ and $s \equiv t$. But $r \not\equiv t$, since there does not exist $\alpha \in \mathfrak{S}^c$ such that $r, t \in \mathfrak{N}_\alpha$. Thus $\equiv$ is not transitive on $\mathfrak{N}$.

However, all that is really needed is that $\equiv$ be an equivalence relation on each $\mathfrak{N}_\alpha$, and this is indeed true.

Proposition 3.18. *For each $\alpha \in \mathfrak{S}^c$, $\equiv|_{\mathfrak{N}_\alpha}$ is an equivalence relation on $\mathfrak{N}_\alpha$.*

Proof. It is clear that $\equiv$ is reflexive and symmetric on each $\mathfrak{N}_\alpha$. For transitivity, the proof is by induction on the structure of terms in $\mathfrak{N}$. Suppose $r, s, t \in \mathfrak{N}_\alpha$, $r \equiv s$ and $s \equiv t$. Then there are three cases to consider.

Suppose that $\alpha = T\ \alpha_1 \ldots \alpha_k$. Then r is $C\ r_1 \ldots r_n$, s is $C\ s_1 \ldots s_n$, and t is $C\ t_1 \ldots t_n$. Also $r_i \equiv s_i$ and $s_i \equiv t_i$, for $i = 1, \ldots, n$. By Proposition 3.13 and the induction hypothesis, $r_i \equiv t_i$, for $i = 1, \ldots, n$. Hence $r \equiv t$.

Next suppose that $\alpha = \beta \to \gamma$. Then
r is $\lambda z.\mathit{if}\ z = h_1\ \mathit{then}\ k_1\ \mathit{else}\ \ldots\ \mathit{if}\ z = h_l\ \mathit{then}\ k_l\ \mathit{else}\ s_0$,
s is $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$,
and t is $\lambda y.\mathit{if}\ y = u_1\ \mathit{then}\ v_1\ \mathit{else}\ \ldots\ \mathit{if}\ y = u_m\ \mathit{then}\ v_m\ \mathit{else}\ s_0$.
Furthermore, $\forall b \in \mathfrak{N}_\beta$,
$(\exists i, j.\ b \equiv h_i\ \wedge\ b \not\equiv h_k(\forall k < i)\ \wedge\ b \equiv t_j\ \wedge\ b \not\equiv t_m(\forall m < j)\ \wedge\ k_i \equiv s_j)\ \vee$
$(\exists i.\ b \equiv h_i\ \wedge\ b \not\equiv h_k(\forall k < i)\ \wedge\ b \not\equiv t_j(\forall j)\ \wedge\ k_i \equiv s_0)\ \vee$
$(\exists j.\ b \not\equiv h_i(\forall i)\ \wedge\ b \equiv t_j\ \wedge\ b \not\equiv t_m(\forall m < j)\ \wedge\ s_0 \equiv s_j)\ \vee$
$(b \not\equiv h_i(\forall i)\ \wedge\ b \not\equiv t_j(\forall j))$
and
$(\exists i, j.\ b \equiv t_i\ \wedge\ b \not\equiv t_k(\forall k < i)\ \wedge\ b \equiv u_j\ \wedge\ b \not\equiv u_m(\forall m < j)\ \wedge\ s_i \equiv v_j)\ \vee$
$(\exists i.\ b \equiv t_i\ \wedge\ b \not\equiv t_k(\forall k < i)\ \wedge\ b \not\equiv u_j(\forall j)\ \wedge\ s_i \equiv s_0)\ \vee$
$(\exists j.\ b \not\equiv t_i(\forall i)\ \wedge\ b \equiv u_j\ \wedge\ b \not\equiv u_m(\forall m < j)\ \wedge\ s_0 \equiv v_j)\ \vee$
$(b \not\equiv t_i(\forall i)\ \wedge\ b \not\equiv u_j(\forall j))$.
Thus, $\forall b \in \mathfrak{N}_\beta$,
$(\exists i, j.\ b \equiv h_i\ \wedge\ b \not\equiv h_k(\forall k < i)\ \wedge\ b \equiv u_j\ \wedge\ b \not\equiv u_m(\forall m < j)\ \wedge\ k_i \equiv v_j)\ \vee$
$(\exists i.\ b \equiv h_i\ \wedge\ b \not\equiv h_k(\forall k < i)\ \wedge\ b \not\equiv u_j(\forall j)\ \wedge\ k_i \equiv s_0)\ \vee$
$(\exists j.\ b \not\equiv h_i(\forall i)\ \wedge\ b \equiv u_j\ \wedge\ b \not\equiv u_m(\forall m < j)\ \wedge\ s_0 \equiv v_j)\ \vee$
$(b \not\equiv h_i(\forall i)\ \wedge\ b \not\equiv u_j(\forall j))$,
by Proposition 3.13 and the induction hypothesis. Hence $r \equiv t$.

Finally, suppose that $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Then r is $(r_1, \ldots, r_n)$, s is $(s_1, \ldots, s_n)$ and t is $(t_1, \ldots, t_n)$. Also $r_i \equiv s_i$ and $s_i \equiv t_i$, for $i = 1, \ldots, n$. By Proposition 3.13 and the induction hypothesis, $r_i \equiv t_i$, for $i = 1, \ldots, n$. Hence $r \equiv t$. □

Proposition 3.19. *For each $\alpha \in \mathfrak{S}^c$, if $s, t \in \mathfrak{N}_\alpha$ and $s \equiv t$, then $s = t$ is a logical consequence of the equality theory.*

Proof. The proof is by induction on the structure of terms in $\mathfrak{N}$.

Suppose first that s is $C\ s_1 \ldots s_n$ and $\alpha = T\ \alpha_1 \ldots \alpha_k$, where C has signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$ and $\theta = \{a_1/\alpha_1, \ldots, a_k/\alpha_k\}$. Then t is $C\ t_1 \ldots t_n$ and $s_i \equiv t_i$, for $i = 1, \ldots, n$. Also, by Proposition 3.13, $s_i, t_i \in \mathfrak{N}_{\sigma_i\theta}$, for $i = 1, \ldots, n$. By the induction hypothesis, $s_i = t_i$ is a logical consequence of the equality theory, for $i = 1, \ldots, n$. Hence $C\ s_1 \ldots s_n = C\ t_1 \ldots t_n$ is a logical consequence of the equality theory.

Next suppose that s is $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$
and $\alpha = \beta \to \gamma$. Then 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, $\forall r \in \mathfrak{N}_\beta$,
$(\exists i, j.\ r \equiv t_i\ \wedge\ r \not\equiv t_k(\forall k < i)\ \wedge\ r \equiv u_j\ \wedge\ r \not\equiv u_m(\forall m < j)\ \wedge\ s_i \equiv v_j)\ \vee$
$(\exists i.\ r \equiv t_i\ \wedge\ r \not\equiv t_k(\forall k < i)\ \wedge\ r \not\equiv u_j(\forall j)\ \wedge\ s_i \equiv s_0)\ \vee$

$(\exists j.\ r \not\equiv t_i(\forall i)\ \wedge\ r \equiv u_j\ \wedge\ r \not\equiv u_m(\forall m < j)\ \wedge\ s_0 \equiv v_j) \vee$
$(r \not\equiv t_i(\forall i)\ \wedge\ r \not\equiv u_j(\forall j))$.
Also, by Proposition 3.13, $t_i \in \mathfrak{N}_\beta$, $s_i \in \mathfrak{N}_\gamma$, for $i = 1, \ldots n$, and $u_j \in \mathfrak{N}_\beta$, $v_j \in \mathfrak{N}_\gamma$, for $j = 1, \ldots m$. It follows that $s = t$ is a logical consequence of the equality theory.

Finally, suppose that s is $(s_1, \ldots, s_n)$ and $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Then t is $(t_1, \ldots, t_n)$ and $s_i \equiv t_i$, for $i = 1, \ldots, n$. Also, by Proposition 3.13, $s_i, t_i \in \mathfrak{N}_{\alpha_i}$, for $i = 1, \ldots, n$. By the induction hypothesis, $s_i = t_i$ is a logical consequence of the equality theory, for $i = 1, \ldots, n$. Hence $(s_1, \ldots, s_n) = (t_1, \ldots, t_n)$ is a logical consequence of the equality theory. □

Next I provide a neater formulation of the definition of $\equiv$. This uses 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.

Proposition 3.20. *Let $t \in \mathfrak{N}_{\beta\to\gamma}$ and $r \in \mathfrak{N}_\beta$. Then $V(t\ r) \in \mathfrak{N}_\gamma$.*

Proof. Let t be $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$. Then $V(t\ r)$ is some s_i, where $i = 0, \ldots, n$. Each s_i has type more general than β. Hence each $s_i \in \mathfrak{N}_\beta$ and so $V(t\ r) \in \mathfrak{N}_\beta$. □

Proposition 3.21. *For each $s, t \in \mathfrak{N}$, $s \equiv t$ iff 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, for all $r \in \mathfrak{N}_\beta$, $V(s\ r) \equiv V(t\ r)$.*
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 $i = 1, \ldots, n$.*

Proof. The result follows immediately from the definition of $\equiv$ and $V(t\ r)$. □

3.3 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 $1 \le j \le n$, $a_1 = b_1, \ldots, a_{j-1} = b_{j-1}$ 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 $s_j < t_j$.

Note. Before continuing, a closer examination of what it means for two normal terms to be identical is needed. Consider the two terms

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

The only difference between these two terms is that the bound variables in each of them have distinct names. In fact, this difference is quite inessential (and could be removed altogether with a suitable notation) and hence I prefer to regard the terms as being identical. Generally, identity of normal terms is regarded as meaning α-equivalent, that is, the names of bound variables are not important. This convention is used implicitly in the remainder of the paper.

Proposition 3.22. *For each $\alpha \in \mathfrak{S}^c$, $<|_{\mathfrak{N}_\alpha}$ is a (strict) total order on $\mathfrak{N}_\alpha$.*

Proof. The proof is by induction on the structure of terms in $\mathfrak{N}$.

First, I show that $<$ is irreflexive. Let $t \in \mathfrak{N}_\alpha$.

1. Let $\alpha = T\ \alpha_1 \ldots \alpha_k$. Then t is $C\ t_1 \ldots t_n$. Thus $t \not< t$, since $t_i \not< t_i$, for $1 \leq i \leq n$, by the induction hypothesis.

2. Let $\alpha = \beta \to \gamma$. Then t is $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$. Thus $t \not< t$, since $t_i \not< t_i$ and $s_i \not< s_i$, for $1 \leq i \leq n$, by the induction hypothesis.

3. Let $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Then t is $(t_1, \ldots, t_n)$. Thus $t \not< t$, since $t_i \not< t_i$, for $1 \leq i \leq n$, by the induction hypothesis.

Next, I show that $<$ is asymmetric. Let $s, t \in \mathfrak{N}_\alpha$ and suppose that $s < t$.

1. Let $\alpha = T\ \alpha_1 \ldots \alpha_k$. Then s is $C\ s_1 \ldots s_n$ and t is $D\ t_1 \ldots t_m$. If $C \prec_T D$, then $D \not\prec_T C$ and $D \neq C$ and so $t \not< s$. Otherwise, $C = D$, in which case there exists j such that $s_1 = t_1, \ldots, s_{j-1} = t_{j-1}$, $s_j < t_j$ and $1 \leq j \leq n$. Hence $t_1 \not< s_1, \ldots, t_{j-1} \not< s_{j-1}$, and $t_j \neq s_j$, since $<$ is irreflexive. By the induction hypothesis, $t_j \not< s_j$. Thus $t \not< s$.

2. Let $\alpha = \beta \to \gamma$. Then $\mathit{trace}(s) < \mathit{trace}(t)$. By examining each of the two cases in the definition of the induced lexicographic ordering and using the induction hypothesis, one can see that $\mathit{trace}(t) \not< \mathit{trace}(s)$. Thus $t \not< s$.

3. Let $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Then s is $(s_1, \ldots, s_n)$, t is $(t_1, \ldots, t_n)$ and there exists j such that $s_1 = t_1, \ldots, s_{j-1} = t_{j-1}$, $s_j < t_j$ and $1 \leq j \leq n$. Hence $t_1 \not< s_1, \ldots, t_{j-1} \not< s_{j-1}$, and $t_j \neq s_j$ since $<$ is irreflexive. By the induction hypothesis, $t_j \not< s_j$. Thus $t \not< s$.

Next, I show that $<$ is transitive. Let $r, s, t \in \mathfrak{N}_\alpha$ and suppose that $r < s$ and $s < t$.

1. Let $\alpha = T\ \alpha_1 \ldots \alpha_k$. Then r is $B\ r_1 \ldots r_p$, s is $C\ s_1 \ldots s_n$, t is $D\ t_1 \ldots t_m$, either $B \prec_T C$ or $B = C$ and there exists j such that $r_1 = s_1, \ldots, r_{j-1} = s_{j-1}$, $r_j < s_j$ and $1 \leq j \leq n$, and either $C \prec_T D$ or $C = D$ and there exists k such that $s_1 = t_1, \ldots, s_{k-1} = t_{k-1}$, $s_k < t_k$ and $1 \leq k \leq n$. If $B = C = D$, by the induction hypothesis, there exists l such that $r_1 = t_1, \ldots, r_{l-1} = t_{l-1}$, $r_l < t_l$ and $1 \leq l \leq n$. Hence $r < t$. If either $B \prec_T C$ or $C \prec_T D$, then $B \prec_T D$. Thus $r < t$.

2. Let $\alpha = \beta \rightarrow \gamma$. Then $\mathit{trace}(r) < \mathit{trace}(s)$ and $\mathit{trace}(s) < \mathit{trace}(t)$. By examining each of the two cases in the definition of the induced lexicographic ordering and using the induction hypothesis, one can see that $\mathit{trace}(r) < \mathit{trace}(t)$. Thus $r < t$.

3. Let $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Then r is $(r_1, \ldots, r_n)$, s is $(s_1, \ldots, s_n)$ t is $(t_1, \ldots, t_n)$, there exists j such that $r_1 = s_1, \ldots, r_{j-1} = s_{j-1}$, $r_j < s_j$ and $1 \leq j \leq n$, and there exists k such that $s_1 = t_1, \ldots, s_{k-1} = t_{k-1}$, $s_k < t_k$ and $1 \leq k \leq n$. Then, by the induction hypothesis, there exists p such that $r_1 = t_1, \ldots, r_{p-1} = t_{p-1}$, $r_p < t_p$ and $1 \leq p \leq n$. Thus $r < t$.

Finally, I show that $<$ is total. Let $s, t \in \mathfrak{N}_\alpha$ and suppose $s \neq t$.

1. Let $\alpha = T\ \alpha_1 \ldots \alpha_k$. Then s is $C\ s_1 \ldots s_n$ and t is $D\ t_1 \ldots t_m$. If $C \neq D$, then either $C \prec_T D$ or $D \prec_T C$, since $\prec_T$ is total. Thus either $s < t$ or $t < s$. Otherwise, $C = D$, in which case there exists j such that $s_1 = t_1, \ldots, s_{j-1} = t_{j-1}$, $s_j \neq t_j$ and $1 \leq j \leq n$. By the induction hypothesis, either $s_j < t_j$ or $t_j < s_j$. Thus either $s < t$ or $t < s$.

2. Let $\alpha = \beta \rightarrow \gamma$. Then $\mathit{trace}(s) \neq \mathit{trace}(t)$. If $\mathit{trace}(s)$ is a (strict) prefix of $\mathit{trace}(t)$, then $s < t$. If $\mathit{trace}(t)$ is a (strict) prefix of $\mathit{trace}(s)$, then $t < s$. Otherwise, there exists an index at which $\mathit{trace}(s)$ and $\mathit{trace}(t)$ differ. Using the induction hypothesis, one can see that either $\mathit{trace}(s) < \mathit{trace}(t)$ or $\mathit{trace}(t) < \mathit{trace}(s)$. Thus either $s < t$ or $t < s$.

3. Let $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Then s is $(s_1, \ldots, s_n)$, t is $(t_1, \ldots, t_n)$ and there exists j such that $s_1 = t_1, \ldots, s_{j-1} = t_{j-1}$, $s_j \neq t_j$ and $1 \leq j \leq n$. By the induction hypothesis, either $s_j < t_j$ or $t_j < s_j$. Thus either $s < t$ or $t < s$.

□

3.4 Basic Terms

The definition of the key concept of a basic term can now be given.

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

1. If C is a data constructor having signature $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow (T\ a_1 \ldots a_k)$ 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 $t_1, \ldots, t_n \in \mathfrak{B}$, $s_1, \ldots, s_n \in \mathfrak{B}$, $t_1 < \cdots < t_n$, $s_i \notin \mathfrak{D}$, 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*.

Note. A suitable universe for the construction of the previous definition is the set $\mathfrak{N}$ of all normal terms. Thus the relation assumed on $t_1, \ldots, t_n$ in Condition 2 is well-defined.

The next result is the *principle of induction on the structure of basic terms.*

Proposition 3.23. *Let $\mathfrak{X}$ be a subset of $\mathfrak{B}$ satisfying the following conditions.*

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

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

   then

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

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

Then $\mathfrak{X} = \mathfrak{B}$.

Proof. Since $\mathfrak{X}$ satisfies Conditions 1 to 3 of the definition of a basic term, it follows that $\mathfrak{B} \subseteq \mathfrak{X}$. Thus $\mathfrak{X} = \mathfrak{B}$. □

Proposition 3.24. $\mathfrak{D} \subseteq \mathfrak{B} \subseteq \mathfrak{N}$.

Proof. The first of these inclusions is an easy induction argument on the structure of default terms, while the second is a similar induction argument on the structure of basic terms. □

Proposition 3.25. *If the set of data constructors is countable, then $\mathfrak{B}$ is countable.*

Proof. By Proposition 3.17, $\mathfrak{N}$ is countable. By Proposition 3.24, $\mathfrak{B} \subseteq \mathfrak{N}$. □

Proposition 3.26. *A subterm of a basic term is basic iff it is closed.*

Proof. □

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

Proof. The result follows directly from the fact that $\mathfrak{D} \subseteq \mathfrak{B} \subseteq \mathfrak{N}$. □

Proposition 3.28. *If each default data constructor is full, then $\mathfrak{B}_\alpha \neq \emptyset$, for all $\alpha \in \mathfrak{S}^c$.*

Proof. By Proposition 3.4, since each default data constructor is full, $\mathfrak{D}_\alpha \neq \emptyset$, for each $\alpha \in \mathfrak{S}^c$. But $\mathfrak{D}_\alpha \subseteq \mathfrak{B}_\alpha$, for each $\alpha \in \mathfrak{S}^c$. Hence the result. □

Next, a bottom-up characterisation of $\mathfrak{B}$ is provided.

Definition. Define $\{\mathfrak{B}_m\}_{m \in \mathbb{N}}$ inductively as follows:
$\mathfrak{B}_0 = \{C \mid C \text{ is a data constructor of arity } 0\}$
$\mathfrak{B}_{m+1} = \mathfrak{B}_m \cup$
$\{C\ t_1 \ldots t_n \in \mathfrak{L} \mid C \text{ is a data constructor and } t_1, \ldots, t_n \in \mathfrak{B}_m (n > 0)\} \cup$
$\{\lambda x.\textit{if } x = t_1 \textit{ then } s_1 \textit{ else } \ldots \textit{ if } x = t_n \textit{ then } s_n \textit{ else } s_0 \in \mathfrak{L} \mid t_1, \ldots, t_n \in \mathfrak{B}_m,$
$s_1, \ldots, s_n \in \mathfrak{B}_m,\ t_1 < \cdots < t_n,\ s_i \notin \mathfrak{D}, \text{ for } 1 \leq i \leq n\ (n \geq 0), \text{ and } s_0 \in \mathfrak{D}\} \cup$
$\{(t_1, \ldots, t_n) \in \mathfrak{L} \mid t_1, \ldots, t_n \in \mathfrak{B}_m\}$.

Clearly, $\mathfrak{B}_m \subseteq \mathfrak{B}_{m+1}$, for $m \in \mathbb{N}$.

Proposition 3.29. $\mathfrak{B} = \bigcup_{m \in \mathbb{N}} \mathfrak{B}_m$.

Proof. First, I show that $\bigcup_{m\in\mathbb{N}} \mathfrak{B}_m \subseteq \mathfrak{B}$. To prove this, it suffices to show by induction that $\mathfrak{B}_m \subseteq \mathfrak{B}$, for $m \in \mathbb{N}$. Clearly $\mathfrak{B}_0 \subseteq \mathfrak{B}$. Suppose next that $\mathfrak{B}_m \subseteq \mathfrak{B}$. It then follows from the definitions of $\mathfrak{B}_{m+1}$ and $\mathfrak{B}$ that $\mathfrak{B}_{m+1} \subseteq \mathfrak{B}$.

Now I show that $\mathfrak{B} \subseteq \bigcup_{m\in\mathbb{N}} \mathfrak{B}_m$. For this, it suffices to show that $\bigcup_{m\in\mathbb{N}} \mathfrak{B}_m$ satisfies Conditions 1, 2 and 3 in the definition of $\mathfrak{B}$ (since $\mathfrak{B}$ is the smallest such set). Suppose that C is a data constructor, $t_1, \ldots, t_n \in \bigcup_{m\in\mathbb{N}} \mathfrak{B}_m$, and $C\ t_1 \ldots t_n \in \mathfrak{L}$. Since the $\mathfrak{B}_m$ are increasing, there exists $p \in \mathbb{N}$ such that $t_1, \ldots, t_n \in \mathfrak{B}_p$. Hence $C\ t_1 \ldots t_n \in \mathfrak{B}_{p+1}$ and so $C\ t_1 \ldots t_n \in \bigcup_{m\in\mathbb{N}} \mathfrak{B}_m$. Similar arguments show that $\bigcup_{m\in\mathbb{N}} \mathfrak{B}_m$ satisfies Conditions 2 and 3. □

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

Proposition 3.30. *Let $s, t \in \mathfrak{B}$. Then $s \equiv t$ iff $s = t$.*

Proof. If $s = t$, then it is clear that $s \equiv t$. Suppose now that $s \equiv t$. By Proposition 3.21, there exists $\alpha \in \mathfrak{S}^c$ such that $s, t \in \mathfrak{B}_\alpha$ with three cases to consider.

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 $i = 1, \ldots, n$. By the induction hypothesis, $s_i = t_i$, for $i = 1, \ldots, n$. Thus $s = t$.

2. $\alpha = \beta \to \gamma$, for some β, γ, and, for all $r \in \mathfrak{B}_\beta$, $V(s\ r) \equiv V(t\ r)$. Suppose that
   s is $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$, and
   t is $\lambda y.\mathit{if}\ y = u_1\ \mathit{then}\ v_1\ \mathit{else}\ \ldots\ \mathit{if}\ y = u_m\ \mathit{then}\ v_m\ \mathit{else}\ s_0$.
   Then $V(s\ t_i) \equiv V(t\ t_i)$, for $i = 1, \ldots, n$, and $V(s\ u_j) \equiv V(t\ u_j)$, for $j = 1, \ldots, m$. By the induction hypothesis, $V(s\ t_i) = V(t\ t_i)$, for $i = 1, \ldots, n$, and $V(s\ u_j) = V(t\ u_j)$, for $j = 1, \ldots, m$. That is, $V(t\ t_i) = s_i$, for $i = 1, \ldots, n$, and $V(s\ u_j) = v_j$, for $j = 1, \ldots, m$. It follows that $\{t_1, \ldots, t_n\} = \{u_1, \ldots, u_m\}$. Since each of s and t is regular, one can see that $s = t$.

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 $i = 1, \ldots, n$. By the induction hypothesis, $s_i = t_i$, for $i = 1, \ldots, n$. Thus $s = t$.

□

Definition. Let $u \in \mathfrak{B}_{\beta\to\gamma}$, for some $\beta, \gamma \in \mathfrak{S}^c$. The *support* of u, denoted $\mathit{supp}(u)$, is the set $\{v \in \mathfrak{B}_\beta \mid V(u\ v) \notin \mathfrak{D}\}$.

Proposition 3.31. *Let $u \in \mathfrak{B}_{\beta\to\gamma}$, for some $\beta, \gamma \in \mathfrak{S}^c$. Then $\mathit{supp}(u)$ is a finite set.*

Proof. Let u be the term $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$. Now proposition 3.24 shows that, if $s, t \in \mathfrak{B}$, then $s \equiv t$ iff $s = t$. Thus $V(u\ v) \notin \mathfrak{D}$ iff $v \in \{t_1, \ldots, t_n\}$. Hence $\mathit{supp}(u)$ is finite. □

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

Proposition 3.32. *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 $s \equiv t$.*

Proof. Uniqueness follows immediately from Proposition 3.30, thus only existence has to be shown. The proof of existence is by induction on the structure of s. There are three cases to consider.

1. $\alpha = T\ \alpha_1 \ldots \alpha_k$, for some $T, \alpha_1, \ldots, \alpha_k$. Suppose that s is $C\ s_1 \ldots s_n$, where C has signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$. Let $\theta = \{a_1/\alpha_1, \ldots, a_k/\alpha_k\}$. By Proposition 3.14, $s_i \in \mathfrak{N}_{\sigma_i\theta}$, for $i = 1, \ldots, n$. By the induction hypothesis, there exist $t_1, \ldots, t_n \in \mathfrak{B}_{\sigma_i\theta}$ such that $s_i \equiv t_i$, for $i = 1, \ldots, n$. Then $C\ t_1 \ldots t_n \in \mathfrak{B}_{T\ \alpha_1 \ldots \alpha_k}$ and, by Proposition 3.18, $s \equiv C\ t_1 \ldots t_n$.

2. $\alpha = \beta \to \gamma$, for some β, γ. Suppose that s is

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

By Proposition 3.14, $t_i \in \mathfrak{N}_\beta$ and $s_i \in \mathfrak{N}_\gamma$, for $i = 1, \ldots, n$. By the induction hypothesis, there exist $t'_i \in \mathfrak{B}_\beta$ and $s'_i \in \mathfrak{B}_\gamma$ such that $t_i \equiv t'_i$ and $s_i \equiv s'_i$, for $i = 1, \ldots, n$. Now let s' be

$$\lambda x.\mathit{if}\ x = t'_1\ \mathit{then}\ s'_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t'_n\ \mathit{then}\ s'_n\ \mathit{else}\ s_0.$$

By Proposition 3.21, $s \equiv s'$. For any $\{i_1, \ldots, i_p\}$ $(p \geq 2)$ such that $i_1 < \cdots < i_p$ and $t'_{i_1} = \cdots = t'_{i_p}$, drop from s' the components of the if-then-else containing $x = t'_{i_2}, \ldots, x = t'_{i_p}$ to obtain s'' of the form

$$\lambda x.\mathit{if}\ x = t''_1\ \mathit{then}\ s''_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t''_m\ \mathit{then}\ s''_m\ \mathit{else}\ s_0.$$

By Proposition 3.21, $s \equiv s''$. Also drop any components of the if-then-else in s'' for which the value s''_i is in $\mathfrak{D}$ to obtain s''' of the form

$$\lambda x.\mathit{if}\ x = t'''_1\ \mathit{then}\ s'''_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t'''_k\ \mathit{then}\ s'''_k\ \mathit{else}\ s_0.$$

By Proposition 3.21, $s \equiv s'''$. At this stage, all the t'''_i are pairwise distinct and all the s'''_i are not in $\mathfrak{D}$. Finally, let $t''''_1 \ldots t''''_k$ be the result of ordering the sequence $t'''_1 \ldots t'''_k$ according to the total order on $\mathfrak{N}_\beta$ induced by the $\prec_T$. Let t be

$$\lambda x.\mathit{if}\ x = t''''_1\ \mathit{then}\ s''''_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t''''_k\ \mathit{then}\ s''''_k\ \mathit{else}\ s_0,$$

the result of the corresponding reordering of the components of the if-then-else in s'''. Then $t \in \mathfrak{B}_{\beta\to\gamma}$ and, by Proposition 3.21, $s \equiv t$.

3. $\alpha = \alpha_1 \times \cdots \times \alpha_n$, for some $\alpha_1, \ldots, \alpha_n$. Suppose s is $(s_1, \ldots, s_n)$. By Proposition 3.14, $s_i \in \mathfrak{N}_{\alpha_i}$, for $i = 1, \ldots, n$. By the induction hypothesis, there exist $t_i \in \mathfrak{B}_{\alpha_i}$ such that $s_i \equiv t_i$, for $i = 1, \ldots, n$. Then $(t_1, \ldots, t_n) \in \mathfrak{B}_{\alpha_1\times\cdots\times\alpha_n}$ and, by Proposition 3.21, $s \equiv (t_1, \ldots, t_n)$.

□

Proposition 3.32 justifies the next definition.

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.

The next result provides an alternative characterisation of the equivalence relation $\equiv$.

Proposition 3.33. *Let $s, t \in \mathfrak{N}_\alpha$, for some $\alpha \in \mathfrak{S}^c$. Then $s \equiv t$ iff s and t have the same basic form.*

Proof. Suppose that $s \equiv t$. Let s' be the basic form of s and t' the basic form of t. Then, by definition, $s \equiv s'$ and $t \equiv t'$. Since $\equiv$ is an equivalence relation, $s' \equiv t'$. By Proposition 3.30, $s' = t'$.

Conversely, suppose that s and t have the same basic form, say, r. Then $s \equiv r$ and $t \equiv r$, and so $s \equiv t$. □

Based on the proof of Proposition 3.32, one can give an algorithm that computes the basic form of a normal term. This algorithm, for the function *Reduce*, is given in Figure 2 below.

Here is an example to show how this algorithm works.

Example. Let s be the normal term

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

function *Reduce*$(s, \{\prec_T\}_{T\in\mathfrak{T}})$ **returns** the basic form of s;
input: s, a normal term;
$\{\prec_T\}_{T\in\mathfrak{T}}$, a set of total orders, one for each type constructor T;

case s **of**
$C\ s_1 \ldots s_n$:
  for $i = 1, \ldots, n$ **do**
    $t_i := Reduce(s_i, \{\prec_T\}_{T\in\mathfrak{T}})$;
  return $C\ t_1 \ldots t_n$;

$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$:
  for $i = 1, \ldots, n$ **do**
    $t'_i := Reduce(t_i, \{\prec_T\}_{T\in\mathfrak{T}})$;
    $s'_i := Reduce(s_i, \{\prec_T\}_{T\in\mathfrak{T}})$;
  $s' := \lambda x.\ \mathit{if}\ x = t'_1\ \mathit{then}\ s'_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t'_n\ \mathit{then}\ s'_n\ \mathit{else}\ s_0$;
  $s'' := s'$ modified by dropping occurrences of components in s' containing duplicate occurrences of some t'_i;
  $s''' := s''$ modified by dropping occurrences of components in s'' whose value is a default term;
  $t := s'''$ modified by reordering components in s''' according to the total ordering induced by $\{\prec_T\}_{T\in\mathfrak{T}}$;
  return t;

$(s_1, \ldots, s_n)$:
  for $i = 1, \ldots, n$ **do**
    $t_i := Reduce(s_i, \{\prec_T\}_{T\in\mathfrak{T}})$;
  return $(t_1, \ldots, t_n)$;

Figure 2: Algorithm for computing the basic form of a normal term

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.

3.5 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 [Mit97, 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 Subsection 3.1, 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 $\alpha = \alpha_1 \times \cdots \times \alpha_n$, 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 $\alpha_1 \times \cdots \times \alpha_n$, 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 3.) Then

$$\begin{aligned}
d(s,t) &= d([A,B,C],[A,D]) \\
&= \frac{1}{2}\max\{\varphi(d(A,A)), \varphi(d([B,C],[D]))\} \\
&= \frac{1}{2}\varphi(d([B,C],[D])) \\
&= \frac{1}{2}\varphi(\frac{1}{2}\max\{\varphi(d(B,D)), \varphi(d([C],[]))\}) \\
&= \frac{1}{2}\varphi(\frac{1}{2}\max\{\varphi(1), \varphi(1)\}) \\
&= \frac{1}{2}\varphi(\frac{1}{2}\cdot\frac{1}{2}) \\
&= \frac{1}{2}\cdot\frac{\frac{1}{4}}{1+\frac{1}{4}} \\
&= \frac{1}{10}.
\end{aligned}$$

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*).

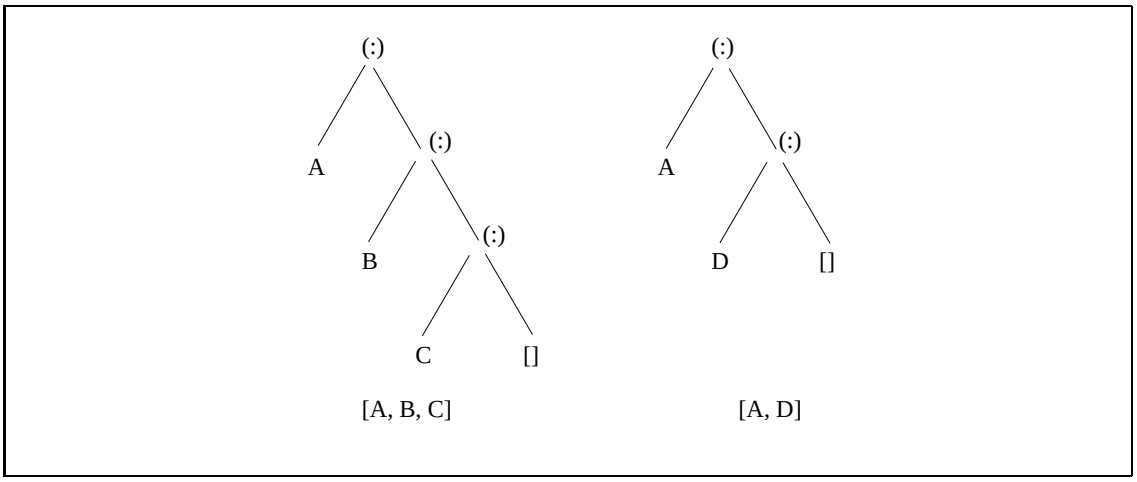

Figure 3: Two lists

(See Figure 4.) Then

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

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 3.34. *Let $\alpha \in \mathfrak{S}^c$. Then, for each $s, t \in \mathfrak{B}_\alpha$, $d(s,t) = 0$ iff $s = t$.*

Proof. If $s = t$, then $d(s,t) = 0$, using an induction argument and the definition of d. Conversely, suppose that $d(s,t) = 0$. Then, using an induction argument, there are three cases to consider.

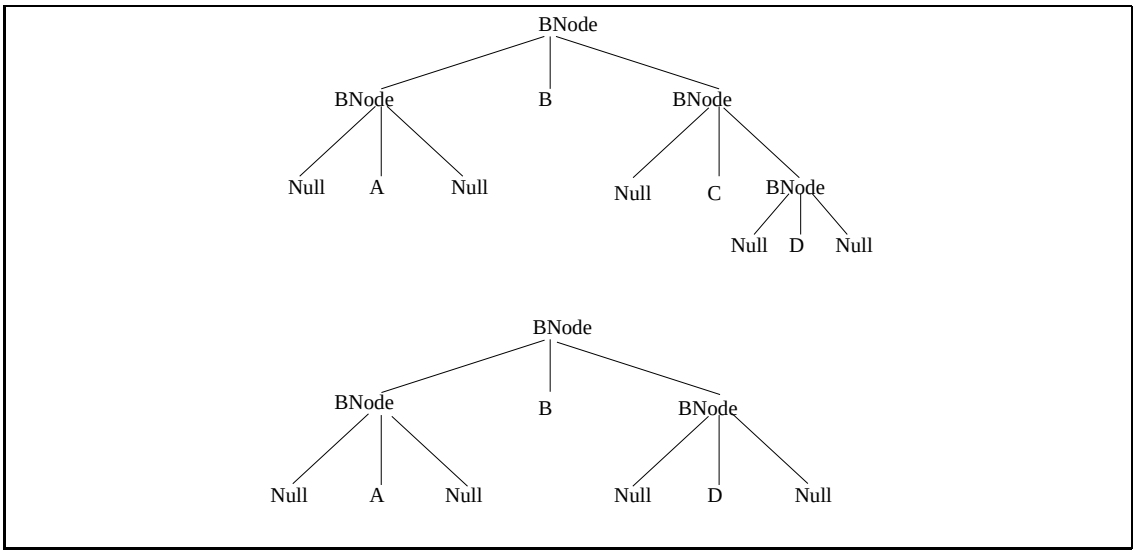

Figure 4: Two binary trees

1. Let $\alpha = T\ \alpha_1 \ldots \alpha_k$. Then s is $C\ s_1 \ldots s_n$, t is $C\ t_1 \ldots t_n$, and $d(s_i, t_i) = 0$, for $i = 1, \ldots, n$. By the induction hypothesis, $s_i = t_i$, for $i = 1, \ldots, n$. Thus $s = t$.

2. Let $\alpha = \beta \to \gamma$. Then, for all $r \in \mathfrak{B}_\beta$, $d(V(s\ r), V(t\ r)) = 0$. By the induction hypothesis, for all $r \in \mathfrak{B}_\beta$, $V(s\ r) = V(t\ r)$. Thus $s = t$, by Propositions 3.21 and 3.30.

3. Let $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Then s is $(s_1, \ldots, s_n)$, t is $(t_1, \ldots, t_n)$ and $d(s_i, t_i) = 0$, for $i = 1, \ldots, n$. By the induction hypothesis, $s_i = t_i$, for $i = 1, \ldots, n$. Thus $s = t$.

□

Proposition 3.35. *For each $\alpha \in \mathfrak{S}^c$, $(\mathfrak{B}_\alpha, d)$ is a metric space.*

Proof. It is easy to show by induction that for each $s, t \in \mathfrak{B}_\alpha$, $d(s,t) \geq 0$ and $d(s,t) = d(t,s)$. Furthermore, by Proposition 3.34, $d(s,t) = 0$ iff $s = t$. Thus it is only necessary to prove the triangle inequality. The proof is by induction on the structure of terms in $\mathfrak{B}$.

Suppose that $r, s, t \in \mathfrak{B}_\alpha$. It has to be shown that $d(r,t) \leq d(r,s) + d(s,t)$. There are three cases to consider.

1. Let $\alpha = T\ \alpha_1 \ldots \alpha_k$. Then r is $B\ r_1 \ldots r_p$, s is $C\ s_1 \ldots s_n$, and t is $D\ t_1 \ldots t_m$. If there is no data constructor associated with T of arity > 0, then $p = n = m = 0$ and the inequality holds because ρ is a metric. Otherwise, there is a data constructor of arity > 0 and ρ is the discrete metric. Suppose first that B, C, and D are not all the same. Thus at least one of $d(r,s)$ or $d(s,t)$ is 1, and the inequality holds since $d(r,t) \leq 1$. Now suppose that $B = C = D$. Then $p = n = m$. By the induction hypothesis, $d(r_i, t_i) \leq d(r_i, s_i) + d(s_i, t_i)$, for $i = 1, \ldots, n$. Hence $\max_{i=1,\ldots,n} \varphi(d(r_i, t_i)) \leq \max_{i=1,\ldots,n} \varphi(d(r_i, s_i)) + \max_{i=1,\ldots,n} \varphi(d(s_i, t_i))$, by the properties of φ, and so $d(r,t) \leq d(r,s) + d(s,t)$.

2. Let $\alpha = \beta \to \gamma$. By the induction hypothesis, $d(V(r\ b), V(t\ b)) \leq d(V(r\ b), V(s\ b)) + d(V(s\ b), V(t\ b))$, for all $b \in \mathfrak{B}_\beta$. Hence $\sum_{b \in \mathfrak{B}_\beta} d(V(r\ b),\ V(t, b)) \leq \sum_{b \in \mathfrak{B}_\beta} d(V(r\ b),\ V(s\ b))$ $+ \sum_{b \in \mathfrak{B}_\beta} d(V(s\ b),\ V(t\ b))$, and so $d(r,t) \leq d(r,s) + d(s,t)$.

3. Let $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Then r is $(r_1, \ldots, r_n)$, s is $(s_1, \ldots, s_n)$, and t is $(t_1, \ldots, t_n)$. By the induction hypothesis, $d(r_i, t_i) \leq d(r_i, s_i) + d(s_i, t_i)$, for $i = 1, \ldots, n$. Hence $d(r,t) = \sum_{i=1}^n d(r_i, t_i) \leq \sum_{i=1}^n d(r_i, s_i) + \sum_{i=1}^n d(s_i, t_i) = d(r,s) + d(s,t)$.

□

Notation. Let $\alpha \in \mathfrak{S}^c$ and $\{t_n\}_{n\in\mathbb{N}}$ be a sequence in $\mathfrak{B}_\alpha$. Then convergence in the metric d of the sequence $\{t_n\}_{n\in\mathbb{N}}$ to a term $t \in \mathfrak{B}_\alpha$ is denoted by $t_n \xrightarrow{d} t$.

The function d is not generally a metric on the *whole* of $\mathfrak{B}$.

Example. Suppose that ρ is the metric for *Float* given by $\rho(n,m) = |n-m|$ and that 0.0 is the default data constructor for *Float*. Let T_1 and T_2 be nullary type constructors, $A_1 : T_1$ and $A_2 : T_2$ be data constructors, r be $\lambda x.\mathit{if}\ x = A_1\ \mathit{then}\ 0.1\ \mathit{else}\ 0.0$, s be $\lambda x.\ 0.0$, and t be $\lambda x.\mathit{if}\ x = A_2\ \mathit{then}\ 0.1\ \mathit{else}\ 0.0$. Then $r, s \in \mathfrak{B}_{T_1\to \mathit{Float}}$ and $d(r,s) = 0.1$. Furthermore, $s, t \in \mathfrak{B}_{T_2\to \mathit{Float}}$ and $d(s,t) = 0.1$. But $d(r,t) = 1$, since there does not exist $\alpha \in \mathfrak{S}^c$ such that $r, t \in \mathfrak{B}_\alpha$. Thus d on $\mathfrak{B}$ does not satisfy the triangle inequality, since $d(r,t) \not\leq d(r,s) + d(s,t)$.

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:

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

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 $\mathit{card}(t) = 3$. That is, $\mathit{card}(t)$ 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 $\mathit{card}(t) = 2 + 2 + 1 = 5$. That is, $\mathit{card}(t)$ is the sum of the multiplicities of the elements of the multiset t.

It will be of interest to know whether or not $(\mathfrak{B}_\alpha, d)$ is separable.

Proposition 3.36. *Suppose that the set of data constructors is countable. Then, for each $\alpha \in \mathfrak{S}^c$, the metric space $(\mathfrak{B}_\alpha, d)$ is separable.*

Proof. If the set of data constructors is countable, then $\mathfrak{B}$ is countable, by Proposition 3.25. Thus $\mathfrak{B}_\alpha$ is also countable and so $(\mathfrak{B}_\alpha, d)$ is separable. □

Proposition 3.37. *If $\mathfrak{B}_\alpha$ is uncountable, for some $\alpha \in \mathfrak{S}^c$, then $(\mathfrak{B}_{\alpha\to\Omega}, d)$ is not separable.*

Proof. For each $t \in \mathfrak{B}_\alpha$, define $t' \in \mathfrak{B}_{\alpha\to\Omega}$ by t' is $\lambda x.\mathit{if}\ x = t\ \mathit{then}\ \top\ \mathit{else}\ \bot$. (Thus $t' = \{t\}$.) Let $\delta = \rho(\top, \bot)$. Then $\delta > 0$. If s and t are distinct terms in $\mathfrak{B}_\alpha$, then $d(s', t') = 2\delta$. Thus $(\mathfrak{B}_{\alpha\to\Omega}, d)$ contains an uncountable discrete subset and hence is not separable. □

3.6 A Partial Order on Basic Terms

Other interesting orders on basic terms can be defined. In the following, I define a (strict) *partial* order $\lhd$ on basic terms which corresponds to (strict) set inclusion for basic abstractions that are sets.

Definition. The binary relation $\lhd$ on $\mathfrak{N}$ is defined inductively as follows. Let $s, t \in \mathfrak{N}$. Then $s \lhd t$ 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 $s_1 \equiv t_1, \ldots, s_{j-1} \equiv t_{j-1}$, $s_j \lhd t_j$ and $1 \leq j \leq n$.

2. $\alpha = \beta \to \gamma$, for some β, γ, and either $V(s\ r) \equiv V(t\ r)$ or $V(s\ r) \lhd V(t\ r)$, for all $r \in \mathfrak{N}_\beta$, and $V(s\ b) \lhd V(t\ b)$, for some $b \in \mathfrak{N}_\beta$.

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 $s_1 \equiv t_1, \ldots, s_{j-1} \equiv t_{j-1}$, $s_j \lhd t_j$ and $1 \leq j \leq n$.

In preparation for the proof that $\lhd$ is a (strict) partial order, two properties of $\lhd$ are established.

Proposition 3.38. *For each $\alpha \in \mathfrak{S}^c$, if $s, t \in \mathfrak{N}_\alpha$ and $s \equiv t$, then $s \ntriangleleft t$.*

Proof. The proof is by induction on the structure of terms in $\mathfrak{N}$.

1. Let $\alpha = T\ \alpha_1 \ldots \alpha_k$. Then s is $C\ s_1 \ldots s_n$, t is $C\ t_1 \ldots t_n$ and $s_i \equiv t_i$, for $i = 1, \ldots, n$. By the induction hypothesis, $s_i \ntriangleleft t_i$, for $i = 1, \ldots, n$, and so $s \ntriangleleft t$.

2. Let $\alpha = \beta \rightarrow \gamma$. Then, for all $r \in \mathfrak{N}_\beta$, $V(s\ r) \equiv V(t\ r)$. By the induction hypothesis, $V(s\ r) \ntriangleleft V(t\ r)$, for all $r \in \mathfrak{N}_\beta$, and so $s \ntriangleleft t$.

3. Let $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Then s is $(s_1, \ldots, s_n)$, t is $(t_1, \ldots, t_n)$ and $s_i \equiv t_i$, for $i = 1, \ldots, n$. By the induction hypothesis, $s_i \ntriangleleft t_i$, for $i = 1, \ldots, n$, and so $s \ntriangleleft t$.

□

Proposition 3.39. *For each $\alpha \in \mathfrak{S}^c$, if $r, s, t \in \mathfrak{N}_\alpha$, $r \equiv s$ and $s \lhd t$, then $r \lhd t$.*

Proof. The proof is by induction on the structure of terms in $\mathfrak{N}$.

1. Let $\alpha = T\ \alpha_1 \ldots \alpha_k$. Then r is $C\ r_1 \ldots r_n$, s is $C\ s_1 \ldots s_n$, t is $D\ t_1 \ldots t_m$, $r_i \equiv s_i$, for $i = 1, \ldots, n$, and either $C \prec_T D$ or $C = D$ and there exists j such that $s_1 \equiv t_1, \ldots, s_{j-1} \equiv t_{j-1}$, $s_j \lhd t_j$ and $1 \leq j \leq n$. If $C \prec_T D$, then $r \lhd t$. If $C = D$, then $r \lhd t$, using the induction hypothesis and the fact that $\equiv$ is an equivalence relation.

2. Let $\alpha = \beta \rightarrow \gamma$. Then, for all $b \in \mathfrak{N}_\beta$, $V(r\ b) \equiv V(s\ b)$ and either $V(s\ b) \equiv V(t\ b)$ or $V(s\ b) \lhd V(t\ b)$, for all $b \in \mathfrak{N}_\beta$, and $V(s\ c) \lhd V(t\ c)$, for some $c \in \mathfrak{N}_\beta$. Then $r \lhd t$, using the induction hypothesis and the fact that $\equiv$ is an equivalence relation.

3. Let $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Then r is $(r_1, \ldots, r_n)$, s is $(s_1, \ldots, s_n)$, t is $(t_1, \ldots, t_n)$, $r_i \equiv s_i$, for $i = 1, \ldots, n$, and there exists j such that $s_1 \equiv t_1, \ldots, s_{j-1} \equiv t_{j-1}$, $s_j \lhd t_j$ and $1 \leq j \leq n$. Then $r \lhd t$, using the induction hypothesis and the fact that $\equiv$ is an equivalence relation.

□

Proposition 3.40. *For each $\alpha \in \mathfrak{S}^c$, if $r, s, t \in \mathfrak{N}_\alpha$, $r \lhd s$ and $s \equiv t$, then $r \lhd t$.*

Proof. The proof is similar to the proof of Proposition 3.39. □

Now I can establish that $\lhd$ is a (strict) partial order.

Proposition 3.41. *For each $\alpha \in \mathfrak{S}^c$, $\lhd|_{\mathfrak{N}_\alpha}$ is a (strict) partial order on $\mathfrak{N}_\alpha$.*

Proof. The proof is by induction on the structure of terms in $\mathfrak{N}$. First, I show that $\lhd$ is irreflexive. Let $t \in \mathfrak{N}_\alpha$.

1. Let $\alpha = T\ \alpha_1 \ldots \alpha_k$. Then t is $C\ t_1 \ldots t_n$. Thus $t \ntriangleleft t$, since $t_i \ntriangleleft t_i$, for $1 \leq i \leq n$, by the induction hypothesis.

2. Let $\alpha = \beta \rightarrow \gamma$. Thus $t \ntriangleleft t$, since $V(t\ r) \ntriangleleft V(t\ r)$, for all $r \in \mathfrak{N}_\beta$, by the induction hypothesis.

3. Let $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Then t is $(t_1, \ldots, t_n)$. Thus $t \ntriangleleft t$, since $t_i \ntriangleleft t_i$, for $1 \leq i \leq n$, by the induction hypothesis.

Next, I show that $\lhd$ is asymmetric. Let $s, t \in \mathfrak{N}_\alpha$ and suppose that $s \lhd t$.

1. Let $\alpha = T\ \alpha_1 \ldots \alpha_k$. Then s is $C\ s_1 \ldots s_n$ and t is $D\ t_1 \ldots t_m$. If $C \prec_T D$, then $D \not\prec_T C$ and so $t \not\lhd s$. Otherwise, $C = D$, in which case there exists j such that $s_1 \equiv t_1, \ldots, s_{j-1} \equiv t_{j-1}$, $s_j \lhd t_j$ and $1 \leq j \leq n$. By Proposition 3.38, $t_1 \not\lhd s_1, \ldots, t_{j-1} \not\lhd s_{j-1}$, and $t_j \not\equiv s_j$. By the induction hypothesis, $t_j \not\lhd s_j$. Thus $t \not\lhd s$.

2. Let $\alpha = \beta \to \gamma$. Then either $V(s\ r) \equiv V(t\ r)$ or $V(s\ r) \lhd V(t\ r)$, for all $r \in \mathfrak{N}_\beta$, and $V(s\ b) \lhd V(t\ b)$, for some $b \in \mathfrak{N}_\beta$. By the induction hypothesis and Proposition 3.38, $V(t\ r) \not\lhd V(s\ r)$, for all $r \in \mathfrak{N}_\beta$. Thus $t \not\lhd s$.

3. Let $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Then s is $(s_1, \ldots, s_n)$, t is $(t_1, \ldots, t_n)$ and there exists j such that $s_1 \equiv t_1, \ldots, s_{j-1} \equiv t_{j-1}$, $s_j \lhd t_j$ and $1 \leq j \leq n$. By Proposition 3.38, $t_1 \not\lhd s_1, \ldots, t_{j-1} \not\lhd s_{j-1}$, and $t_j \not\equiv s_j$. By the induction hypothesis, $t_j \not\lhd s_j$. Thus $t \not\lhd s$.

Finally, I show that $\lhd$ is transitive. Let $r, s, t \in \mathfrak{N}_\alpha$ and suppose that $r \lhd s$ and $s \lhd t$.

1. Let $\alpha = T\ \alpha_1 \ldots \alpha_k$. Then r is $B\ r_1 \ldots r_p$, s is $C\ s_1 \ldots s_n$, t is $D\ t_1 \ldots t_m$, either $B \prec_T C$ or $B = C$ and there exists j such that $r_1 \equiv s_1, \ldots, r_{j-1} \equiv s_{j-1}$, $r_j \lhd s_j$ and $1 \leq j \leq n$, and either $C \prec_T D$ or $C = D$ and there exists k such that $s_1 \equiv t_1, \ldots, s_{k-1} \equiv t_{k-1}$, $s_k \lhd t_k$ and $1 \leq k \leq n$. If $B = C = D$, by Proposition 3.39 and the induction hypothesis, there exists l such that $r_1 \equiv t_1, \ldots, r_{l-1} \equiv t_{l-1}$, $r_l \lhd t_l$ and $1 \leq l \leq n$. Hence $r \lhd t$. If either $B \prec_T C$ or $C \prec_T D$, then $B \prec_T D$. Thus $r \lhd t$.

2. Let $\alpha = \beta \to \gamma$. Then either $V(r\ b) \equiv V(s\ b)$ or $V(r\ b) \lhd V(s\ b)$, for all $b \in \mathfrak{N}_\beta$, and $V(r\ c) \lhd V(s\ c)$, for some $c \in \mathfrak{N}_\beta$. Also either $V(s\ b) \equiv V(t\ b)$ or $V(s\ b) \lhd V(t\ b)$, for all $b \in \mathfrak{N}_\beta$, and $V(s\ d) \lhd V(t\ d)$, for some $d \in \mathfrak{N}_\beta$. By Proposition 3.39 and the induction hypothesis, either $V(r\ b) \equiv V(t\ b)$ or $V(r\ b) \lhd V(t\ b)$, for all $b \in \mathfrak{N}_\beta$, and $V(r\ e) \lhd V(t\ e)$, for some $e \in \mathfrak{N}_\beta$. Thus $r \lhd t$.

3. Let $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Then r is $(r_1, \ldots, r_n)$, s is $(s_1, \ldots, s_n)$ t is $(t_1, \ldots, t_n)$, there exists j such that $r_1 \equiv s_1, \ldots, r_{j-1} \equiv s_{j-1}$, $r_j \lhd s_j$ and $1 \leq j \leq n$, and there exists k such that $s_1 \equiv t_1, \ldots, s_{k-1} \equiv t_{k-1}$, $s_k \lhd t_k$ and $1 \leq k \leq n$. Then, by Proposition 3.39 and the induction hypothesis, there exists p such that $r_1 \equiv t_1, \ldots, r_{p-1} \equiv t_{p-1}$, $r_p \lhd t_p$ and $1 \leq j \leq n$. Thus $r \lhd t$.

□

Since $\mathfrak{B}_\alpha \subseteq \mathfrak{N}_\alpha$, for each $\alpha \in \mathfrak{S}^c$, it follows that $\lhd|_{\mathfrak{N}_\alpha}$ is also a (strict) partial order on $\mathfrak{B}_\alpha$.

Assuming that $\bot \prec_\Omega \top$, the partial order $\lhd$ on sets corresponds to (strict) set inclusion, that is, $s \lhd t$ iff $s \subset t$. For example, $\{A, B, C\} \lhd \{A, B, C, D\}$. Similarly, assuming that $\prec_{Int}$ is the usual $<$ on the integers, the partial order $\lhd$ on multisets corresponds to (strict) multiset inclusion, that is, $s \lhd t$ iff $s \sqsubset t$. For example, $\langle A, B, A, C, C\rangle \lhd \langle A, B, A, A, C, C, C\rangle$. For a product type, $\lhd$ corresponds to the (strict) lexicographic ordering.

Recall that a preorder on a set A is a binary relation $\leq$ on A such that, for each $a, b, c \in A$, $a \leq a$ (reflexivity), and $a \leq b$ and $b \leq c$ implies $a \leq c$ (transitivity). A preorder is a partial order if, in addition, for each $a, b \in A$, $a \leq b$ and $b \leq a$ implies $a = b$ (antisymmetry). A partial order is a total order if, in addition, for each $a, b \in A$, either $a = b$ or $a \leq b$ or $b \leq a$.

Definition. The binary relation $\unlhd$ on $\mathfrak{N}$ is defined by $s \unlhd t$ if $s \lhd t$ or $s \equiv t$.

Proposition 3.42. *For each $\alpha \in \mathfrak{S}^c$, $\unlhd|_{\mathfrak{N}_\alpha}$ is a preorder on $\mathfrak{N}_\alpha$.*

Proof. First, it is clear that $\unlhd$ is reflexive. To show that $\unlhd$ is transitive, consider $r, s, t \in \mathfrak{N}_\alpha$ such that $r \unlhd s$ and $s \unlhd t$. If $r \lhd s$ and $s \lhd t$, then $r \lhd t$, by Proposition 3.41, and so $r \unlhd t$. If $r \equiv s$ and $s \lhd t$, then $r \lhd t$, by Proposition 3.39, and so $r \unlhd t$. If $r \lhd s$ and $s \equiv t$, then $r \lhd t$, by Proposition 3.40, and so $r \unlhd t$. If $r \equiv s$ and $s \equiv t$, then $r \equiv t$, and so $r \unlhd t$. □

Note that $\unlhd$ is not a partial order on $\mathfrak{N}_\alpha$, since it is not antisymmetric.

Example. Let s be $\lambda x.\mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \bot$ and t be $\lambda x.\mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \bot$. Then $s \trianglelefteq t$ and $t \trianglelefteq s$. (In fact, $s \equiv t$.) But $s \neq t$.

However, if $\trianglelefteq$ is restricted to basic terms, it is a partial order. To prove this, yet another characterisation of $\equiv$ on normal terms is needed.

Proposition 3.43. *For each $\alpha \in \mathfrak{S}^c$, if $s, t \in \mathfrak{N}_\alpha$, then $s \equiv t$ iff $s \trianglelefteq t$ and $t \trianglelefteq s$.*

Proof. $s \equiv t$ implies $s \trianglelefteq t$ and $t \trianglelefteq s$, by the definition of $\trianglelefteq$. Conversely, let $s \trianglelefteq t$ and $t \trianglelefteq s$. Suppose that $s \not\equiv t$. Then $s \lhd t$ and $t \lhd s$, which contradicts $\lhd$ being asymmetric. □

Now I can show that $\trianglelefteq$ is a partial order on basic terms.

Proposition 3.44. *For each $\alpha \in \mathfrak{S}^c$, $\trianglelefteq|_{\mathfrak{B}_\alpha}$ is a partial order on $\mathfrak{B}_\alpha$.*

Proof. By Proposition 3.42, $\trianglelefteq|_{\mathfrak{B}_\alpha}$ is a preorder on $\mathfrak{B}_\alpha$. Thus it is only necessary to show that $\trianglelefteq$ is antisymmetric on basic terms. Thus suppose that $s, t \in \mathfrak{B}_\alpha$ and that $s \trianglelefteq t$ and $t \trianglelefteq s$. By Proposition 3.43, $s \equiv t$. Thus, by Proposition 3.30, $s = t$. □

The preorder $\trianglelefteq$ on sets corresponds to set inclusion, that is, $s \trianglelefteq t$ iff $s \subseteq t$. For example, $\{A, B, C\} \trianglelefteq \{A, B, C, D\}$. The preorder $\trianglelefteq$ on multisets corresponds to multiset inclusion, that is, $s \trianglelefteq t$ iff $s \sqsubseteq t$. For example, $\langle A, B, C\rangle \trianglelefteq \langle A, B, C, D, A\rangle$.

3.7 Some Examples of Representation

In this subsection, 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 [Mit97]. 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.

Example. 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 [DLLP97], 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.

Let *Make*, *Length*, and *Width* be nullary types. Then the following declarations of data constructors are made.

$$Abloy, Chubb, Rubo, Yale : Make$$
$$Short, Medium, Long : Length$$
$$Narrow, Normal, Broad : Width$$

The following type synonyms are helpful.

$$NumProngs = Nat$$
$$Key = Make \times NumProngs \times Length \times Width$$
$$Bunch = \{Key\}$$

(Recall that $\{Key\}$ is the same as $Key \to \Omega$.) Thus the individuals in this case are sets whose elements are 4-tuples. The function to be learned is

$$opens : Bunch \to \Omega.$$

For further development of this example, see Subsection 4.7.

Example. 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.

$$AtomType = Nat$$
$$Charge = Float$$
$$Atom = Element \times AtomType \times Charge$$
$$Bond = Nat.$$

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

$$Label = Nat$$
$$Graph\ \nu\ \varepsilon = \{Label \times \nu\} \times \{(Label \to Nat) \times \varepsilon\}.$$

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 $Graph\ \nu\ \varepsilon$ 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 [Yor]. 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 in an important way to structure this search space. More details can be found in Section 4.

3.8 First-order versus Higher-order Representation

This section concludes with a brief discussion of how one can represent individuals in first-order logic, when this logic, or the restricted subset provided by Prolog, is used for knowledge representation instead of higher-order logic. In the first-order case, abstractions are not available so other techniques have to be used. A common way of introducing sets and multisets is to have certain data constructors (analogous to *cons* for lists) which have special equality theories to obtain the right semantics [DPR96]. In the case of Prolog, this technique is not available and so it becomes necessary to "flatten" terms in the following way. (This description is given rather informally.)

Definition. Let t be a normal term and s a subterm of t 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 avoid some technical difficulties, I assume, as I may, that $i \neq j$ implies $t_i \not\equiv t_j$.) Let t' be the term obtained from t by replacing s by a constant f_s, say, and adding the equations

$$
\begin{aligned}
&f_s(t_1) = s_1\\
&f_s(t_2) = s_2\\
&\quad\vdots\\
&f_s(t_n) = s_n\\
&f_s(x) = s_0 \leftarrow x \neq t_1 \wedge \cdots \wedge x \neq t_n.
\end{aligned}
$$

The set of terms consisting of t' and these equations is said to have been obtained from t by a *flattening step*.

Definition. Given a normal term t, the *flattened form* of t is the set of terms obtained from t by iteratively applying flattening steps until the resulting set of terms contains no abstractions.

Example. Let t be the term $([A, B, C], \{32, 45, 67, 78\}, \langle \mathit{Bill}, \mathit{Fred}, \mathit{Bill}, \mathit{Fred}, \mathit{Joe} \rangle)$. Then the flattened form of t is the following set of terms.

$$
\begin{aligned}
&([A, B, C], t_1, t_2)\\
&t_1(32) = \top\\
&t_1(45) = \top\\
&t_1(67) = \top\\
&t_1(78) = \top\\
&t_1(x) = \bot \leftarrow x \neq 32 \wedge x \neq 45 \wedge x \neq 67 \wedge x \neq 78\\
&t_2(\mathit{Bill}) = 2\\
&t_2(\mathit{Fred}) = 2\\
&t_2(\mathit{Joe}) = 1\\
&t_2(x) = 0 \leftarrow x \neq \mathit{Bill} \wedge x \neq \mathit{Fred} \wedge x \neq \mathit{Joe}.
\end{aligned}
$$

One can also write the equations relationally, as would be necessary if using a language such as Prolog.

Example. Continuing the previous example, the equations can be written relationally as follows (where now t_2 has a new meaning).

$$
\begin{aligned}
&t_1(32)\\
&t_1(45)\\
&t_1(67)\\
&t_1(78)\\
&t_2(\mathit{Bill}, 2)\\
&t_2(\mathit{Fred}, 2)\\
&t_2(\mathit{Joe}, 1)\\
&t_2(x, 0) \leftarrow x \neq \mathit{Bill} \wedge x \neq \mathit{Fred} \wedge x \neq \mathit{Joe}.
\end{aligned}
$$

Note how the default equation disappears in the case of sets.

Returning to the case of Prolog, some final tidying up is needed. The term t itself is usually named by some constant, and inserted into a fact. The higher-order constants (t_1 and t_2, in the above example) that have been introduced can be removed by replacing them by first-order constants and introducing new relations with one extra argument to contain the first-order constants. Finally, embedded tuples can be removed by simply removing the parentheses around them, effectively creating more arguments in the surrounding relation. (Subterms made up of data constructors can be left untouched since Prolog admits these.) The final result is a database of facts about the original term which is "equivalent" to the original term (assuming the obvious intended meaning for the introduced relations).

Example. Continuing the previous example, the database corresponding to t has the form.

$$
\begin{aligned}
&p(T, [A, B, C], T_1, T_2)\\
&s(T_1, 32)\\
&s(T_1, 45)\\
&s(T_1, 67)\\
&s(T_1, 78)
\end{aligned}
$$

$$m(T_2, \mathit{Bill}, 2)$$
$$m(T_2, \mathit{Fred}, 2)$$
$$m(T_2, \mathit{Joe}, 1)$$
$$m(T_2, x, 0) \leftarrow x \neq \mathit{Bill} \wedge x \neq \mathit{Fred} \wedge x \neq \mathit{Joe}.$$

Here T is the name of t.

Other approaches to the use of logic in machine learning, for example, inductive logic programming [NCdW97], generally use first-order logic. The method outlined in this section can be used to obtain the first-order representation of individuals from the higher-order representation used here.

4 Predicate Construction

Having established the use of the logic for representing individuals, I now turn to the problem of constructing predicates that individuals may or may not satisfy. Essentially, all that is required is the definition of a suitable collection of predicates on the type of an individual. However, these predicates are usually built up incrementally by composition and it is this incremental construction that I study here. A major application of these ideas occurs in machine learning, as in [BGCL00] and [BGCL01], for example.

4.1 Transformations

Composition is handled by the (reverse) composition function

$$\cdot : (a \to b) \to (b \to c) \to (a \to c)$$

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 which 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 $\mu \to \Omega$, some 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 *Int* 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

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

defined by

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

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

Example. The transformation

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

defined by

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

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

$$\sim : (\mu \to \Omega) \to \mu \to \Omega$$

defined by

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

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

$$\neg : \Omega \to \Omega$$

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

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

defined by

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

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

Example. For each type μ, there are two fundamental transformations $top : \mu \to \Omega$ and $bottom : \mu \to \Omega$ 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

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

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* which has various order relations defined on it. Then, for any integer N, one can define the transformation

$$(< N) : Int \to \Omega$$

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

$$ite : (\mu \to \Omega) \to (\mu \to \Omega) \to (\mu \to \Omega) \to \mu \to \Omega$$

is defined by

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

As an illustration, one could define the predicate

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

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 $Graph\ \nu\ \varepsilon$ is the type of (undirected) graphs whose vertices have type $Vertex\ \nu\ \varepsilon$ and edges have type $Edge\ \nu\ \varepsilon$, 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.

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

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

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

on individuals of type $Graph\ \nu\ \varepsilon$ 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

$$(subgraphs\ 3) \cdot (domCard\ (edges \cdot (setExists_2\ (edge \cdot b)\ (edge \cdot c)))) \cdot (> 3),$$

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

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

The function $isCircle : (Float \to \Omega) \to Shape \to \Omega$ defined by

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

is a transformation. Similarly, the function $isRectangle : (Float \to \Omega) \to (Float \to \Omega) \to Shape \to \Omega$ defined by

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

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.

4.2 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, $(setExists_2\ p\ q)$ and $(setExists_2\ q\ p)$ are semantically equivalent. Less obviously, $(domCard\ p) \cdot (> 0)$ and $(setExists_1\ p)$ 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 that 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 $setExists_n$ 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 standard 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 4.1. *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}.*

Proof. The proof is by induction on the maximum of the number of transformations in p and the number in q.

Suppose first that f_1 is distinct from g_1. Then the second alternative holds. Otherwise, f_1 and g_1 are identical, which gives rise to two cases: either the arguments to f_1 in p and g_1 in q are pairwise identical or they are not. In the first case, consider the suffixes of p and q obtained by removing the common prefix $(f_1\ b_{1,1} \ldots b_{1,k_1})$. If one of the suffixes is empty, then the first alternative holds; otherwise, one can apply the inductive hypothesis to the suffixes to obtain the result. In the second case, the leftmost pair of b_{1,j_1} and c_{1,j_1} that are distinct gives the third alternative. □

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

Proof. Let $p, q, r \in \mathbf{S}_\alpha$, for some $\alpha \in \mathfrak{S}^c$, where p is $(f_1\ b_{1,1} \ldots b_{1,k_1}) \cdot \ldots \cdot (f_n\ b_{n,1} \ldots b_{n,k_n})$, q is $(g_1\ c_{1,1} \ldots c_{1,s_1}) \cdot \ldots \cdot (g_r\ c_{r,1} \ldots c_{r,s_r})$ and r is $(h_1\ d_{1,1} \ldots d_{1,t_1}) \cdot \ldots \cdot (h_r\ d_{v,1} \ldots d_{v,t_v})$. First, I show by induction on the number of transformations in p that $p \not\prec p$. For this, note that p cannot be a (strict) prefix of itself and there cannot exist i such that $f_i < f_i$, since $<$ is a strict total order. Also there cannot exist i and j_i such that $b_{i,j_i} \prec b_{i,j_i}$, by the induction hypothesis.

Next I show by induction on the maximum of the number of transformations in p and the number in q that $p \prec q$ implies $q \not\prec p$. Thus suppose that $p \prec q$. If p is a (strict) prefix of q, then it is clear that $q \not\prec p$. If there exists i such that $f_i < g_i$, and p and q agree to the left of f_i and g_i, then it is clear that $q \not\prec p$, since $<$ is a strict total order. Finally, if 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}, then $q \not\prec p$, since by the induction hypothesis, we have that $c_{i,j_i} \not\prec b_{i,j_i}$.

Now I show by induction on the maximum of the number of transformations in p, the number in q, and the number in r that $p \prec q$ and $q \prec r$ imply that $p \prec r$. There are three cases to consider, corresponding to the three cases in the definition of $p \prec q$.

Suppose that p is a (strict) prefix of q. If q is a (strict) prefix of r, then p is a (strict) prefix of r and so $p \prec r$. If there exists i such that $g_i < h_i$, and q and r agree to the left of g_i and h_i, then either p is a (strict) prefix of r or $f_i < h_i$ and p and r agree to the left of f_i and h_i, and so $p \prec r$. If there exist i and j_i such that $c_{i,j_i} \prec d_{i,j_i}$, and q and r agree to the left of c_{i,j_i} and d_{i,j_i}, then either p is a (strict) prefix of r or $b_{i,j_i} \prec d_{i,j_i}$ and p and r agree to the left of b_{i,j_i} and d_{i,j_i}, and so $p \prec r$.

For the second case, suppose there exists i such that $f_i < g_i$, and p and q agree to the left of f_i and g_i, If q is a strict prefix of r, then $f_i < h_i$, and p and r agree to the left of f_i and h_i, and so $p \prec r$. If there exists i' such that $g_{i'} < h_{i'}$, and q and r agree to the left of $g_{i'}$ and $h_{i'}$, then, for $i'' = min(i, i')$, $f_{i''} < h_{i''}$ and p and r agree to the left of $f_{i''}$ and $h_{i''}$, and so $p \prec r$. Finally, suppose that there exist i' and $j_{i'}$ such that $c_{i',j_{i'}} \prec d_{i',j_{i'}}$, and q and r agree to the left of $c_{i',j_{i'}}$ and $d_{i',j_{i'}}$. If $i \leq i'$, then $f_i < h_i$ and p and r agree to the left of f_i and h_i. If $i' < i$, then $b_{i',j_{i'}} \prec d_{i',j_{i'}}$ and p and r agree to the left of $b_{i',j_{i'}}$ and $d_{i',j_{i'}}$. In either case, $p \prec r$.

For the third case, suppose 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}. If q is a strict prefix of r, then $b_{i,j_i} \prec d_{i,j_i}$, and p and r agree to the left of b_{i,j_i} and d_{i,j_i}, and so $p \prec r$. Suppose that there exists i' such that $g_{i'} < h_{i'}$, and q and r agree to the left of $g_{i'}$ and $h_{i'}$. If $i' \leq i$, then $f_{i'} < h_{i'}$ and p and r agree to the left of $f_{i'}$ and $h_{i'}$. If $i < i'$, then $b_{i,j_i} \prec d_{i,j_i}$ and p and r agree to the left of b_{i,j_i} and d_{i,j_i}. In either case, $p \prec r$. Finally, suppose there exist i' and $j_{i'}$ such that $c_{i',j_{i'}} \prec d_{i',j_{i'}}$, and q and r agree to the left of $c_{i',j_{i'}}$ and $d_{i',j_{i'}}$. Put $i'' = min(i, i')$ and let $j_{i''}$ be j_i if $i < i'$, or $j_{i'}$ if $i' < i$, or $min(j_{i'}, j_{i''})$, otherwise. Then $b_{i'',j_{i''}} \prec d_{i'',j_{i''}}$ and p and r agree to the left of $b_{i'',j_{i''}}$ and $d_{i'',j_{i''}}$, and so $p \prec r$. (Here the induction hypothesis is used if $i = i'$ and $j_i = j_{i'}$.)

Thus $\prec$ is a strict partial order.

Finally, I show that $\prec$ is total, that is, for any standard predicates p and q, either $p = q$ or $p \prec q$ or $q \prec p$. The proof proceeds by induction on the maximum of the number of predicates in p and the number in q. Suppose that p and q are distinct. By Proposition 4.1, one of p or q is a (strict) prefix of the other; or 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; or 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}. In the first case, either $p \prec q$ or $q \prec p$. In the second case, since $<$ is a total order, either $f_i < g_i$ or $g_i < f_i$, and hence either $p \prec q$ or $q \prec p$. In the third case, by the induction hypothesis, either $b_{i,j_i} \prec c_{i,j_i}$ or $c_{i,j_i} \prec b_{i,j_i}$, and so once again either $p \prec q$ or $q \prec p$. □

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 $\mathbf{S}_\alpha$.

4.3 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,

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

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

Proof. The existence of the regular predicate q is shown by induction on the number of transformations in p. Let p be $(f_1\ b_{1,1} \ldots b_{1,k_1}) \cdot \ldots \cdot (f_n\ b_{n,1} \ldots b_{n,k_n})$. By the induction hypothesis, for each $i = 1, \ldots, n$ and $j_i = 1, \ldots, k_i$, there is a regular predicate b'_{i,j_i} such that b'_{i,j_i} and b_{i,j_i} have the same type and are semantically equivalent. Let r be $(f_1\ b'_{1,1} \ldots b'_{1,k_1}) \cdot \ldots \cdot (f_n\ b'_{n,1} \ldots b'_{n,k_n})$. Then p and r have the same type and are semantically equivalent. Now, since $\preceq$ is a total order on standard predicates, for each symmetric transformation f_i $(i = 1, \ldots, n)$ in r, one can sort the arguments $b'_{i,1}, \ldots, b'_{i,k_i}$ according to $\preceq$ to obtain the desired regular predicate q such that p and q are semantically equivalent. □

One cannot expect the regular predicate q in Proposition 4.3 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 4.3 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 5 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 proof of Proposition 4.3 shows that 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 *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 5: 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 $\mathit{leftTree} : \mathit{BTree}\ \mu \to \mathit{BTree}\ \mu$. Here $(\mathit{BTree}\ \mu)$ 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 4.4. *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.*

Proof. The product of the number of transformations and the maximum number of times any transformation can appear gives an upper bound on the number of transformations that can appear in any standard predicate. Hence the number of standard predicates is finite. □

4.4 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 [BGCL01]. 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 4.5. *For each $\alpha \in \mathfrak{S}^c$, $\Leftarrow|_{\mathbf{S}_\alpha}$ is a preorder on $\mathbf{S}_\alpha$.*

Proof. Let $p, q, r \in \mathbf{S}_\alpha$. Clearly, $p \Leftarrow p$. Also if $p \Leftarrow q$ and $q \Leftarrow r$, then $p \Leftarrow r$, since $\longleftarrow$ is transitive. Thus $\Leftarrow$ is a preorder. □

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

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

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

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

if, for any standard predicate p of the form

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

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 occurrence of $b_{r,s}$ in p by c. Here and in the following the subscript used before to indicate the occurrence of the subterm is omitted.

Example. For the skeletons $(\mathit{setExists}_n\ x_1 \ldots x_n)$ and $(\wedge_n\ x_1 \ldots x_n)$, each x_i is a monotone argument, for $i = 1, \ldots, n$.

Example. The variable x is a monotone argument in the skeleton $(\mathit{domCard}\ x) \cdot (> N)$, 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 4.6. *Let $(f_1\ x_{1,1} \ldots x_{1,k_1}) \cdot \ldots \cdot (f_n\ x_{n,1} \ldots x_{n,k_n})$ be a skeleton, f_r a symmetric transformation, and $x_{r,s}$ a monotone argument of the skeleton, for some $r \in \{1, \ldots, n\}$ and some $s \in \{1, \ldots, k_r\}$. Then x_{r,j_r} is a monotone argument of the skeleton, for $j_r = 1, \ldots, k_r$.*

Proof. The result follows immediately from the definitions of a monotone argument and a symmetric transformation. □

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 $(f_1\ b_{1,1} \ldots b_{1,k_1}) \cdot \ldots \cdot (f_n\ b_{n,1} \ldots b_{n,k_n})$ and q a standard predicate that has an occurrence as a subterm of p. Then q is in a *monotone position* in p if either

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

In the case when a skeleton $(f_t\ x_{t,1} \ldots x_{t,k_t}) \cdot \ldots \cdot (f_n\ x_{n,1} \ldots x_{n,k_n})$ has the same sequence of transformations $[f_t, \ldots, f_n]$ as a suffix $(f_t\ b_{t,1} \ldots b_{t,k_t}) \cdot \ldots \cdot (f_n\ b_{n,1} \ldots b_{n,k_n})$ 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 4.7. *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.*

Proof. 1. This part is obvious.

2. This part follows straightforwardly from the definition of a monotone position.

3. The proof proceeds by induction on the number of transformations in p. Suppose first that q is a suffix of p. Since r is in a monotone position in q, the result now follows from Part 2. Otherwise, there is a skeleton $(f_t\ x_{t,1} \ldots x_{t,k_t}) \cdot \ldots \cdot (f_n\ x_{n,1} \ldots x_{n,k_n})$, for some $t \geq 1$, matching a suffix of p and having a monotone argument $x_{v,w}$, where $v \in \{t, \ldots, n\}$ and $w \in \{1, \ldots, k_v\}$, such that q is in a monotone position in $b_{v,w}$. By the induction hypothesis, r is in a monotone position in $b_{v,w}$. Hence r is in a monotone position in p (by Part 2 of the definition of monotone position). □

Example. The predicate b is in a monotone position in $\mathit{vertices} \cdot (\mathit{setExists}_2\ (\wedge_2\ b\ c)\ d)$. This can be proved using Proposition 4.7 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 $(\mathit{setExists}_2\ x\ y)$, b is in a monotone position in $(\mathit{setExists}_2\ (\wedge_2\ b\ c)\ d)$. Finally, by Part 2 of Proposition 4.7, it follows that b is in a monotone position in $\mathit{vertices} \cdot (\mathit{setExists}_2\ (\wedge_2\ b\ c)\ d)$.

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

Proposition 4.8. *Assume that there is given a set of skeletons together with their monotone arguments. Let $p, q \in \mathbf{S}$, where q has an 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 $p \Leftarrow p[q/q']$.*

Proof. The proof is by induction on the number of transformations in p. Let p be the predicate $(f_1\ b_{1,1} \ldots b_{1,k_1}) \cdot \ldots \cdot (f_n\ b_{n,1} \ldots b_{n,k_n})$. There are two cases to consider. First, q may be a suffix of p, in which case the result follows immediately. Second, there may be a skeleton $(f_t\ x_{t,1} \ldots x_{t,k_t}) \cdot \ldots \cdot (f_n\ x_{n,1} \ldots x_{n,k_n})$, for some $t \geq 1$, matching a suffix of p and having a monotone argument $x_{r,s}$, where $r \in \{t, \ldots, n\}$ and $s \in \{1, \ldots, k_r\}$, such that q is in a monotone position in $b_{r,s}$. By the induction hypothesis, $b_{r,s} \Leftarrow b_{r,s}[q/q']$. Since $x_{r,s}$ is a monotone argument, it follows that

$$(f_t\ b_{t,1} \ldots b_{t,k_t}) \cdot \ldots \cdot (f_n\ b_{n,1} \ldots b_{n,k_n}) \Leftarrow (f_t\ b_{t,1} \ldots b_{t,k_t}) \cdot \ldots \cdot (f_n\ b_{n,1} \ldots b_{n,k_n})[q/q'].$$

Hence $p \Leftarrow p[q/q']$. □

Given below, in Figure 6, 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.

Example. Consider the predicate p defined by

$$\mathit{vertices} \cdot (\mathit{setExists}_2\ (\wedge_2\ (\mathit{proj}_1 \cdot (= A))\ (\mathit{proj}_2 \cdot (= 3)))\ (\mathit{proj}_1 \cdot (= B))).$$

Suppose the skeletons are $(\mathit{setExists}_2\ x\ y)$ 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

$$\begin{aligned}
\{&\mathit{vertices} \cdot (\mathit{setExists}_2\ (\wedge_2\ (\mathit{proj}_1 \cdot (= A))\ (\mathit{proj}_2 \cdot (= 3)))\ (\mathit{proj}_1 \cdot (= B))),\\
&\mathit{setExists}_2\ (\wedge_2\ (\mathit{proj}_1 \cdot (= A))\ (\mathit{proj}_2 \cdot (= 3)))\ (\mathit{proj}_1 \cdot (= B)),\\
&\wedge_2\ (\mathit{proj}_1 \cdot (= A))\ (\mathit{proj}_2 \cdot (= 3)),\\
&\mathit{proj}_1 \cdot (= A),\\
&(= A),\\
&\mathit{proj}_2 \cdot (= 3),\\
&(= 3),\\
&\mathit{proj}_1 \cdot (= B),\\
&(= B)\}.
\end{aligned}$$

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

pos := {};
for each suffix s of p do
    pos := pos ∪ {s};
for each skeleton in 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 ∪ Positions(b_{r,s}, S);
return pos;
```

Figure 6: Algorithm for finding the monotone positions in a standard predicate

For later use, I establish two properties of monotone positions and regularisation. Recall that $\overline{p}$ denotes the regularisation of a standard predicate p.

Proposition 4.9. *Assume that there is given a set of skeletons together with their monotone arguments. Let p be a standard predicate and r a regular predicate that is a subterm of p in a monotone position in p. Then r is a subterm of $\overline{p}$ in a monotone position in $\overline{p}$.*

Proof. As usual, the proof is by induction on the number of transformations in p. There are two cases to consider. Suppose first that r is a suffix of p. Since r is regular, r will be a suffix of the regularisation $\overline{p}$ of p. Hence r is a subterm of $\overline{p}$ and is also in a monotone position in $\overline{p}$.

For the second case, there is a skeleton $(f_t\ x_{t,1} \ldots x_{t,k_t}) \cdot \ldots \cdot (f_n\ x_{n,1} \ldots x_{n,k_n})$, for some $t \geq 1$, matching a suffix of p and having a monotone argument $x_{v,w}$, where $v \in \{t, \ldots, n\}$ and $w \in \{1, \ldots, k_v\}$, such that r is in a monotone position in $b_{v,w}$. By the induction hypothesis, r is a subterm of $\overline{b_{v,w}}$ and is in a monotone position in $\overline{b_{v,w}}$. Inspection of the regularisation algorithm shows that r is therefore a subterm of $\overline{p}$. Furthermore, the second case in the definition of monotone position holds for $\overline{p}$ with r in a monotone position in $\overline{b_{v,w}}$ and $\overline{b_{v,w}}$ corresponding to a monotone argument in a skeleton. Hence r is in a monotone position in $\overline{p}$. □

Proposition 4.10. *Assume that there is given a set of skeletons together with their monotone arguments. Let p be a standard predicate, r a regular predicate that is a subterm of p in a monotone position in p, and s a regular predicate such that $p[r/s]$ is a term. Then $\overline{p[r/s]} = \overline{p}[r/s]$.*

Proof. By Proposition 4.9, r is a subterm of $\overline{p}$ in a monotone position in $\overline{p}$. Hence $\overline{p}[r/s]$ is well-defined. That $\overline{p[r/s]}$ is the same as $\overline{p}[r/s]$ then follows by inspection of the algorithm for regularising a standard predicate. □

4.5 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 [BGCL01], the implication-consistent relation $\rightarrowtail$ includes the following relationships. (Recall that *top* is the predicate defined by $top\ x = \top$, for all x.).

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

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

$$
\begin{aligned}
&(subgraphs\ 3) \cdot (domCard\ (edges \cdot (setExists_2\ (edge \cdot b)\ (edge \cdot c)))) \cdot (> 1) \\
&\ll (subgraphs\ 3) \cdot (domCard\ (edges \cdot (setExists_2\ (edge \cdot b)\ (edge \cdot c)))) \cdot (> 2)
\end{aligned}
$$

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

Also the relation

$$
\begin{aligned}
&(subgraphs\ 3) \cdot (domCard\ (edges \cdot (setExists_2\ (edge \cdot b)\ (edge \cdot c)))) \cdot (< 2) \\
&\ll (subgraphs\ 3) \cdot (domCard\ (edges \cdot (setExists_2\ (edge \cdot b)\ (edge \cdot c)))) \cdot (< 1)
\end{aligned}
$$

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

Finally, the relation

$$
\begin{aligned}
&(subgraphs\ 3) \cdot (domCard\ (edges \cdot (setExists_2\ (edge \cdot b)\ (edge \cdot c)))) \cdot (> 1) \\
&\ll (subgraphs\ 3) \cdot (domCard\ (edges \cdot (setExists_2\ (edge \cdot b')\ (edge \cdot c)))) \cdot (> 1)
\end{aligned}
$$

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

Proof. First I establish that $p \Leftarrow p[r/s]$, where r is a standard predicate in a monotone position in p and $r \rightarrowtail s$. Since $r \rightarrowtail s$, it follows that $r \Leftarrow s$ by the definition of an implication-consistent relation. Thus, by Proposition 4.8, $p \Leftarrow p[r/s]$. Hence $p \ll q$ implies that $p \Leftarrow q$.

Now as easy induction argument on the length of the path from p to q, where $p \ll^* q$, establishes that $p \Leftarrow q$. □

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 $\ll^* = \Leftarrow$.

4.6 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 7 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.

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 4.12. *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.*

Proof. The proof is by induction on the length n of the path from *top* to q in $top \ll \cdots \ll q$.

If $n = 1$, then $top \ll q$ and $p = \overline{q}$. Thus $top \rightarrowtail q$ and so p is returned by the enumeration algorithm.

For the induction step, suppose the result holds for paths of length $n - 1$. Suppose p is a regular predicate for which there is a standard predicate q such that $top \ll^* q$ by a path of length n and $p = \overline{q}$. Then there exists a standard predicate u such that $top \ll^* u$ by a path of length $n - 1$ and $u \ll q$. By the induction hypothesis, $\overline{u}$ is returned by the enumeration algorithm. Since $u \ll q$, there are regular predicates r and s with r in a monotone position in u such that $r \rightarrowtail s$ and $q = u[r/s]$. By Propositions 4.9 and 4.10, r is in a monotone position in $\overline{u}$ and $\overline{u[r/s]} = \overline{u}[r/s]$. Thus $p = \overline{q} = \overline{u}[r/s]$, and so p is returned by the enumeration algorithm. □

```
function Enumerate(⇀, S) returns a set of regular predicates;
input: ⇀, a regular, implication-consistent relation;
       S, a set of skeletons together with their monotone arguments;

regular := {};
openList := [top];
seenSet := {top};
while openList ≠ [] do
    p := head(openList);
    openList := tail(openList);
    regular := regular ∪ {p};
    for each r such that r ∈ Positions(p, S) and r ⇀ s, for some s, do
        q := Regularise(p[r/s]);
        if q ∉ seenSet
        then
            seenSet := seenSet ∪ {q};
            openList := openList ++ [q];
return regular;
```

Figure 7: Algorithm for enumerating regular predicates

In fact, the set of predicates returned by the enumeration algorithm is generally strictly larger than the set of regularisations of standard predicates q such that $top \ll^* q$. The reason is that the step in the algorithm where $p[r/s]$ is regularised creates additional redexes.

Example. Consider the implication-consistent relation defined by

$$top \rightharpoonup f\ b\ c$$
$$c \rightharpoonup a$$
$$f\ a\ b \rightharpoonup d,$$

where f is a symmetric transformation and a, b, c, and d are regular predicates such that $a \prec b$ and $b \preceq c$. Then the predicate d can be obtained by the following sequence of rewrite steps.

$$top$$
$$f\ b\ c$$
$$f\ a\ b$$
$$d.$$

Note that a regularisation has been applied in the step from $(f\ b\ c)$ to $(f\ a\ b)$. Thus d is returned by the enumeration algorithm. But there does not exist a standard predicate e such that $top \ll^* e$ and d is $\overline{e}$, since the only path from top using $\ll$ (without regularisation) ends at $(f\ b\ a)$.

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).

4.7 An Example of Predicate Construction

The predicate construction process just described has applications throughout machine learning. Perhaps the most obvious of these is to decision-tree learning [Mit97, Chapter 3]. 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 4, one can enumerate a large class of regular predicates. This subsection 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 [DLLP97]. From a knowledge representation point of view, a multiple-instance problem is characterised by having a term which is a set at the top level representing an individual.

Example. Consider the problem discussed earlier of determining whether a key in a bunch of keys can open a door. A characteristic property of this kind of "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.

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

The following type synonyms are helpful.

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

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

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

Here are two typical examples.

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

The hypothesis language contains the following transformations.

$$
\begin{aligned}
&(= \mathit{Abloy}) : \mathit{Make} \to \Omega \\
&(= \mathit{Chubb}) : \mathit{Make} \to \Omega \\
&(= \mathit{Rubo}) : \mathit{Make} \to \Omega \\
&(= \mathit{Yale}) : \mathit{Make} \to \Omega \\
&(= 2) : \mathit{NumProngs} \to \Omega \\
&(= 3) : \mathit{NumProngs} \to \Omega \\
&(= 4) : \mathit{NumProngs} \to \Omega \\
&(= 5) : \mathit{NumProngs} \to \Omega \\
&(= 6) : \mathit{NumProngs} \to \Omega \\
&(= \mathit{Short}) : \mathit{Length} \to \Omega \\
&(= \mathit{Medium}) : \mathit{Length} \to \Omega \\
&(= \mathit{Long}) : \mathit{Length} \to \Omega \\
&(= \mathit{Narrow}) : \mathit{Width} \to \Omega \\
&(= \mathit{Normal}) : \mathit{Width} \to \Omega \\
&(= \mathit{Broad}) : \mathit{Width} \to \Omega \\
&\mathit{prMake} : \mathit{Key} \to \mathit{Make} \\
&\mathit{prNumProngs} : \mathit{Key} \to \mathit{NumProngs} \\
&\mathit{prLength} : \mathit{Key} \to \mathit{Length} \\
&\mathit{prWidth} : \mathit{Key} \to \mathit{Width} \\
&\mathit{setExists}_1 : (\mathit{Key} \to \Omega) \to \mathit{Bunch} \to \Omega \\
&\wedge_2 : (\mathit{Key} \to \Omega) \to (\mathit{Key} \to \Omega) \to \mathit{Key} \to \Omega \\
&\wedge_3 : (\mathit{Key} \to \Omega) \to (\mathit{Key} \to \Omega) \to (\mathit{Key} \to \Omega) \to \mathit{Key} \to \Omega \\
&\wedge_4 : (\mathit{Key} \to \Omega) \to (\mathit{Key} \to \Omega) \to (\mathit{Key} \to \Omega) \to (\mathit{Key} \to \Omega) \to \mathit{Key} \to \Omega.
\end{aligned}
$$

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.

$$
\begin{aligned}
&\mathit{top} \rightarrowtail \mathit{setExists}_1\ (\wedge_4\ (\mathit{prMake} \cdot \mathit{top})\ (\mathit{prNumProngs} \cdot \mathit{top})\ (\mathit{prLength} \cdot \mathit{top})\ (\mathit{prWidth} \cdot \mathit{top})) \\
&\mathit{top} \rightarrowtail (= \mathit{Abloy}) \\
&\mathit{top} \rightarrowtail (= \mathit{Chubb}) \\
&\mathit{top} \rightarrowtail (= \mathit{Rubo}) \\
&\mathit{top} \rightarrowtail (= \mathit{Yale}) \\
&\mathit{top} \rightarrowtail (= 2) \\
&\mathit{top} \rightarrowtail (= 3) \\
&\mathit{top} \rightarrowtail (= 4) \\
&\mathit{top} \rightarrowtail (= 5) \\
&\mathit{top} \rightarrowtail (= 6) \\
&\mathit{top} \rightarrowtail (= \mathit{Short}) \\
&\mathit{top} \rightarrowtail (= \mathit{Medium}) \\
&\mathit{top} \rightarrowtail (= \mathit{Long}) \\
&\mathit{top} \rightarrowtail (= \mathit{Narrow}) \\
&\mathit{top} \rightarrowtail (= \mathit{Normal})
\end{aligned}
$$

$$top \rightarrowtail (= Broad)$$

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,

$$top \rightarrowtail setExists_1\ (\wedge_4\ (prMake \cdot top)\ (prNumProngs \cdot top)\ (prLength \cdot top)\ (prWidth \cdot top)),$$

which gives the new predicate

$$setExists_1\ (\wedge_4\ (prMake \cdot top)\ (prNumProngs \cdot top)\ (prLength \cdot top)\ (prWidth \cdot top)).$$

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.

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

Taking the first of these children,

$$setExists_1\ (\wedge_4\ (prMake \cdot (= Abloy))\ (prNumProngs \cdot top)\ (prLength \cdot top)\ (prWidth \cdot top)),$$

and applying the rewrites at each available redex, one obtains

$$\begin{array}{l}
setExists_1\ (\wedge_4\ (prMake \cdot (= Abloy))\ (prNumProngs \cdot (= 2))\ (prLength \cdot top)\ (prWidth \cdot top)) \\
setExists_1\ (\wedge_4\ (prMake \cdot (= Abloy))\ (prNumProngs \cdot (= 3))\ (prLength \cdot top)\ (prWidth \cdot top)) \\
\qquad\vdots \\
setExists_1\ (\wedge_4\ (prMake \cdot (= Abloy))\ (prNumProngs \cdot top)\ (prLength \cdot top)\ (prWidth \cdot (= Narrow))) \\
setExists_1\ (\wedge_4\ (prMake \cdot (= Abloy))\ (prNumProngs \cdot top)\ (prLength \cdot top)\ (prWidth \cdot (= Normal))) \\
setExists_1\ (\wedge_4\ (prMake \cdot (= Abloy))\ (prNumProngs \cdot top)\ (prLength \cdot top)\ (prWidth \cdot (= Broad))).
\end{array}$$

This process continues, producing successively stronger predicates.

For this application, the learning system in [BGCL00] and [BGCL01] found the following definition for the function *opens*.

$$\begin{array}{l}
opens\ b = \\
\quad if\ setExists_1\ (\wedge_2\ (projMake \cdot (= Abloy))\ (projLength \cdot (= Medium)))\ b \\
\quad then\ \top \\
\quad else\ \bot.
\end{array}$$

"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:

$$\wedge_4\ (prMake \cdot (= Abloy))\ (prNumProngs \cdot top)\ (prLength \cdot (= Medium))\ (prWidth \cdot top)$$

can be simplified to

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

This completes the example.

5 Programming with Abstractions

In this section, the paradigm of programming with abstractions 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 [Je] 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 8, 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.

$$append\ :\ List\ a \times List\ a \times List\ a \rightarrow \Omega$$

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

$$permute\ :\ List\ a \times List\ a \rightarrow \Omega$$

$$\begin{aligned} & permute\ ([], x)\ =\ x = [] \\ & permute\ (x : y, w)\ =\ \exists u.\exists v.\exists z.(w = u : v\ \wedge\ delete\ (u, x : y, z) \wedge permute\ (z, v)) \end{aligned}$$

$$delete\ :\ a \times List\ a \times List\ a \rightarrow \Omega$$

$$\begin{aligned} & delete\ (x, [], y)\ =\ \bot \\ & delete\ (x, y : z, w)\ =\ (x = y \wedge w = z)\ \vee\ \exists v.(w = y : v\ \wedge\ delete\ (x, z, v)) \end{aligned}$$

$$sorted\ :\ List\ Int \rightarrow \Omega$$

$$\begin{aligned} & sorted\ []\ =\ \top \\ & sorted\ x : y\ =\ if\ y = []\ then\ \top\ else\ \exists u.\exists v.(y = u : v\ \wedge\ x \leq u\ \wedge\ sorted\ y) \end{aligned}$$

Figure 8: List-processing functions

What extra machinery needs to added to Haskell to enable it to run the definitions in Figure 8? 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 8 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 $\mathit{append}\ ([1],[2],x)$. 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 $\Sigma : (a \to \Omega) \to \Omega$ in the appendix and has λ-abstractions in its head.

The above ideas, plus some carefully chosen definitions in the appendix, 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 [Llo87] 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 9. 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

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

Using the statement

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

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

Figure 9: A database to illustrate set processing

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

$$\{x \mid (\mathit{if}\ x = \mathit{Mary}\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = \mathit{Bill}\ \mathit{then}\ \top\ \mathit{else}\ \bot) \wedge \\ (\mathit{if}\ x = \mathit{Joe}\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = \mathit{Bill}\ \mathit{then}\ \top\ \mathit{else}\ \bot)\}$$

by β-reduction. After several uses of the statements

$$(\mathit{if}\ u\ \mathit{then}\ v\ \mathit{else}\ w) \wedge t = \mathit{if}\ u \wedge t\ \mathit{then}\ v\ \mathit{else}\ w \wedge t, \\ t \wedge (\mathit{if}\ u\ \mathit{then}\ v\ \mathit{else}\ w) = \mathit{if}\ t \wedge u\ \mathit{then}\ v\ \mathit{else}\ t \wedge w, \ \text{and} \\ \boldsymbol{v} \wedge (\boldsymbol{x} = \boldsymbol{u}) \wedge \boldsymbol{w} = \boldsymbol{v}\{\boldsymbol{x}/\boldsymbol{u}\} \wedge (\boldsymbol{x} = \boldsymbol{u}) \wedge \boldsymbol{w}\{\boldsymbol{x}/\boldsymbol{u}\}$$

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

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

and so on, to the answer

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

During this computation, use is made of the statements

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

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. 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.

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

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 $\{x \mid if\ \boldsymbol{u}\ then\ v\ else\ \boldsymbol{w}\}$ can represent an empty set if both v and $\boldsymbol{w}$ are $\perp$, 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 $\perp$, $if\ \boldsymbol{u}\ then\ v\ else\ \boldsymbol{w}$, $x = t$, 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

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

reduces to the answer

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

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 $(s \sqcap t)\ x = min\ (s\ x)\ (t\ x)$. Now

recall from Section 3 that a multiset is represented by a basic term, that is, an abstraction of the form

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

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.

$$\sqcap : (a \to Nat) \to (a \to Nat) \to (a \to Nat)$$

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

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

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

reduces to the answer

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

6 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 two examples. In this development, the higher-order nature of the logic was essential: as a semantics for the functional component of declarative programming languages, functions need the ability to take functions as arguments; in the use of the logic for knowledge representation, sets and similar abstractions are needed; and for programming with abstractions, the ability to allow abstractions as arguments in function definitions is needed.

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 [Je], 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 [NM98]. 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 [Han94]. Probably the best developed of these functional logic languages is the Curry language [He], which is the result of an international collaboration over the last 5 or so years. To quote from [He]: "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)."

Closely related to Curry is the Escher language [Llo99]. Escher is a general-purpose, declarative programming language which integrates the best features of both functional and logic programming languages. It has types and modules, higher-order and meta-programming facilities, concurrency, and declarative input/output. Escher goes beyond Haskell in its ability to run function calls containing variables, its more flexible handling of the connectives and quantifiers, and its ability to allow programming with abstractions, as was explained in Section 5.

There are many other outstanding examples of systems that exploit the power of higher-order logic. For example, the HOL system [GM93] 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 underexploited 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.

7 Conclusion

< To be added >

Acknowledgements

I am indebted to Antony Bowers, Peter Flach, and Christophe Giraud-Carrier for many interesting and helpful discussions on the subject matter of this paper. In particular, the theoretical foundations presented here grew out of joint work with Antony Bowers and Christophe Giraud-Carrier on the development of an inductive learning system.

References

[And86] P.B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, 1986.

[Bel88] J.L. Bell. *Toposes and Local Set Theories*. Oxford Science Publications, 1988.

[BGCL00] 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.

[BGCL01] 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.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[DLLP97] 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.

[DPR96] A. Dovier, A. Policriti, and G. Rossi. Integrating lists, multisets, and sets in a logic programming framework. In F. Baader and K. Schulz, editors, *Proc. of FROCOS'96*, pages 213–229. Kluwer, 1996.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[Han94] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[He] M. Hanus (ed.). Curry: An integrated functional logic language. Available at `http://www.informatik.uni-kiel.de/~curry`.

[Hen50] L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15(2):81–91, 1950.

[HT92] P.M. Hill and R.W. Topor. A semantics for typed logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 1–62. MIT Press, 1992.

[Je] S. Peyton Jones and J. Hughes (editors). Haskell98: A non-strict purely functional language. Available at `http://haskell.org/`.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.

[Llo99] J.W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1999(3), March 1999.

[LS86] J. Lambek and P.J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986.

[Mit97] T.M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[MO84] A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[NCdW97] S.H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. Lecture Notes in Artificial Intelligence, 1228. Springer-Verlag, 1997.

[NM98] 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.

[Wol93] D.A. Wolfram. *The Clausal Theory of Types*. Cambridge University Press, 1993.

[Yor] Home page of Machine Learning Group, The University of York. `http://www.cs.york.ac.uk/mlg/`.

A Definitions of Some Basic Functions

$= : \ a \to a \to \Omega$

$(C\ x_1 \ldots x_n = C\ y_1 \ldots y_n) = (x_1 = y_1) \wedge \cdots \wedge (x_n = y_n)$

% where C is a data constructor of arity n. If $n = 0$, then the RHS is $\top$.

$(C\ x_1 \ldots x_n = D\ y_1 \ldots y_m) = \bot$

% where C is a data constructor of arity n, D is a data constructor of arity m, and $C \neq D$.

$(\lambda x.\boldsymbol{u} = \lambda y.\boldsymbol{v}) = (\mathit{less}\ \lambda x.\boldsymbol{u}\ \lambda y.\boldsymbol{v}) \wedge (\mathit{less}\ \lambda y.\boldsymbol{v}\ \lambda x.\boldsymbol{u})$

$((x_1, \ldots, x_n) = (y_1, \ldots, y_n)) = (x_1 = y_1) \wedge \cdots \wedge (x_n = y_n)$

$\mathit{less}\ :\ (a \to b) \to (a \to b) \to \Omega$

$\mathit{less}\ \lambda x.d\ z = \top$

% where d is a default term and z has default value d.

$\mathit{less}\ (\lambda x.\mathit{if}\ \boldsymbol{u}\ \mathit{then}\ v\ \mathit{else}\ \boldsymbol{w})\ z = (\forall x.(\boldsymbol{u} \longrightarrow v = (z\ x))) \wedge (\mathit{less}\ (\mathit{remove}\ \{x \mid \boldsymbol{u}\}\ \lambda x.\boldsymbol{w})\ z)$

$\mathit{remove}\ :\ (a \to \Omega) \to (a \to b) \to (a \to b)$

$\mathit{remove}\ s\ \lambda x.d = \lambda x.d$

% where d is a default term.

$\mathit{remove}\ s\ \lambda x.\mathit{if}\ \boldsymbol{u}\ \mathit{then}\ v\ \mathit{else}\ \boldsymbol{w} = \lambda x.\mathit{if}\ \boldsymbol{u} \wedge (x \notin s)\ \mathit{then}\ v\ \mathit{else}\ ((\mathit{remove}\ s\ \lambda x.\boldsymbol{w})\ x)$

$\mathit{card}\ :\ (a \to b) \to \mathit{Float}$

$\mathit{card}\ \lambda x.d = 0$

% where d is a default term.

$\mathit{card}\ \lambda x.\mathit{if}\ x = t\ \mathit{then}\ v\ \mathit{else}\ \boldsymbol{w} = (\mathit{distance}\ v\ d) + \mathit{card}\ (\mathit{remove}\ \{t\}\ \lambda x.\ \boldsymbol{w})$

% where $\lambda x.\mathit{if}\ x = t\ \mathit{then}\ v\ \mathit{else}\ \boldsymbol{w}$ is a normal abstraction with default value d.

$\mathit{distance}\ :\ a \to a \to \mathit{Float}$

$\mathit{distance}\ (C\ x_1 \ldots x_n)\ (D\ y_1 \ldots y_m) = \rho\ C\ D$

% where C is a data constructor of arity n, D is a data constructor of arity m, and $C \neq D$.

$\mathit{distance}\ (C\ x_1 \ldots x_n)\ (C\ y_1 \ldots y_n) = 0.5 * \mathit{maximum}\ [(\varphi\ (\mathit{distance}\ x_1\ y_1)), \ldots, (\varphi\ (\mathit{distance}\ x_n\ x_n))]$

% where C is a data constructor of arity n.

$\mathit{distance}\ \lambda x.d\ s = \mathit{card}\ s$

% where s is a normal abstraction with default value d.

$\mathit{distance}\ (\lambda x.\mathit{if}\ x = t\ \mathit{then}\ v\ \mathit{else}\ \boldsymbol{w})\ s =$
$(\mathit{distance}\ v\ (s\ t)) + (\mathit{distance}\ (\mathit{remove}\ \{t\}\ \lambda x.\boldsymbol{w})\ (\mathit{remove}\ \{t\}\ s))$

% where $\lambda x.\mathit{if}\ x = t\ \mathit{then}\ v\ \mathit{else}\ \boldsymbol{w}$ is a normal abstraction with default value d.

$\mathit{distance}\ (x_1, \ldots, x_n)\ (y_1, \ldots, y_n) = (\mathit{distance}\ x_1\ y_1) + \cdots + (\mathit{distance}\ x_n\ y_n)$

$\rho\ :\ a \to a \to \mathit{Float}$

% ρ is a metric on the set of data constructors.

$\varphi\ :\ \mathit{Float} \to \mathit{Float}$

% φ is 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.

$maximum \ : \ List\ Float \rightarrow Float$

$maximum\ [] = 0$
$maximum\ [x] = x$
$maximum\ x : y : z = if\ x \leq y\ then\ maximum\ y : z\ else\ maximum\ x : z$

$\neq : \ a \rightarrow a \rightarrow \Omega$

$x \neq y = \neg\ (x = y)$

$\wedge \ : \ \Omega \rightarrow \Omega \rightarrow \Omega$

$\top \wedge x = x$
$x \wedge \top = x$
$\bot \wedge x = \bot$
$x \wedge \bot = \bot$
$(x \vee y) \wedge z = (x \wedge z) \vee (y \wedge z)$
$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
$x \wedge (\exists x_1. \cdots \exists x_n.\boldsymbol{v}) \wedge y = \exists x_1. \cdots \exists x_n.(x \wedge \boldsymbol{v} \wedge y)$
$\boldsymbol{v} \wedge (\boldsymbol{x} = \boldsymbol{u}) \wedge \boldsymbol{w} = \boldsymbol{v}\{\boldsymbol{x}/\boldsymbol{u}\} \wedge (\boldsymbol{x} = \boldsymbol{u}) \wedge \boldsymbol{w}\{\boldsymbol{x}/\boldsymbol{u}\}$
% $\boldsymbol{x}$ is a variable free in $\boldsymbol{v}$ or $\boldsymbol{w}$ or both, but not free in $\boldsymbol{u}$.
$(if\ u\ then\ v\ else\ w) \wedge t = if\ u \wedge t\ then\ v\ else\ w \wedge t$
$t \wedge (if\ u\ then\ v\ else\ w) = if\ t \wedge u\ then\ v\ else\ t \wedge w$

$\vee \ : \ \Omega \rightarrow \Omega \rightarrow \Omega$

$\top \vee x = \top$
$x \vee \top = \top$
$\bot \vee x = x$
$x \vee \bot = x$
$(if\ u\ then\ v\ else\ w) \vee t = if\ v\ then\ (if\ u\ then\ \top\ else\ w \vee t)\ else\ (\neg\ u \wedge w) \vee t$
$t \vee (if\ u\ then\ v\ else\ w) = if\ v\ then\ (if\ u\ then\ \top\ else\ t \vee w)\ else\ t \vee (\neg\ u \wedge w)$

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

$\neg \bot = \top$
$\neg \top = \bot$
$\neg (\neg\ x) = x$
$\neg (x \wedge y) = (\neg\ x) \vee (\neg\ y)$
$\neg (x \vee y) = (\neg\ x) \wedge (\neg\ y)$
$\neg (if\ u\ then\ v\ else\ w) = if\ u\ then\ \neg\ v\ else\ \neg\ w$

$\Sigma \ : \ (a \rightarrow \Omega) \rightarrow \Omega$

$\exists x_1.\cdots\exists x_n.\top = \top$

$\exists x_1.\cdots\exists x_n.\bot = \bot$

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

% If both $\boldsymbol{x}$ and $\boldsymbol{y}$ are absent, then the RHS is $\top$.

$\exists x_1.\cdots\exists x_n.(\boldsymbol{u} \vee \boldsymbol{v}) = (\exists x_1.\cdots\exists x_n.\boldsymbol{u}) \vee (\exists x_1.\cdots\exists x_n.\boldsymbol{v})$

$\exists x_1.\cdots\exists x_n.(\mathit{if}\ \boldsymbol{u}\ \mathit{then}\ v\ \mathit{else}\ \boldsymbol{w}) = \mathit{if}\ v\ \mathit{then}\ \exists x_1.\cdots\exists x_n.(\boldsymbol{u} \vee \boldsymbol{w})\ \mathit{else}\ \exists x_1.\cdots\exists x_n.(\neg\ \boldsymbol{u} \wedge \boldsymbol{w})$

$\Pi\ :\ (a \to \Omega) \to \Omega$

$\forall x_1.\cdots\forall x_n.(\bot \longrightarrow \boldsymbol{u}) = \top$

$\forall x_1.\cdots\forall x_n.(\boldsymbol{x} \wedge (x_1 = u) \wedge \boldsymbol{y} \longrightarrow \boldsymbol{v}) = \forall x_2.\cdots\forall x_n.(\boldsymbol{x}\{x_1/u\} \wedge \boldsymbol{y}\{x_1/u\} \longrightarrow \boldsymbol{v}\{x_1/u\})$

% If both $\boldsymbol{x}$ and $\boldsymbol{y}$ are absent, then the RHS is $\forall x_2.\cdots\forall x_n.(\top \longrightarrow \boldsymbol{v}\{x_1/u\})$.

$\forall x_1.\cdots\forall x_n.(\boldsymbol{u} \vee \boldsymbol{v} \longrightarrow \boldsymbol{t}) = (\forall x_1.\cdots\forall x_n.(\boldsymbol{u} \longrightarrow \boldsymbol{t})) \wedge (\forall x_1.\cdots\forall x_n.(\boldsymbol{v} \longrightarrow \boldsymbol{t}))$

$\forall x_1.\cdots\forall x_n.((\mathit{if}\ \boldsymbol{u}\ \mathit{then}\ v\ \mathit{else}\ \boldsymbol{w}) \longrightarrow \boldsymbol{t}) =$
$\mathit{if}\ v\ \mathit{then}\ \forall x_1.\cdots\forall x_n.(\boldsymbol{u} \vee \boldsymbol{w} \longrightarrow \boldsymbol{t})\ \mathit{else}\ \forall x_1.\cdots\forall x_n.(\neg\ \boldsymbol{u} \wedge \boldsymbol{w} \longrightarrow \boldsymbol{t})$

$\cup\ :\ (a \to \Omega) \to (a \to \Omega) \to (a \to \Omega)$

$s \cup t = \{x \mid (x \in s) \vee (x \in t)\}$

$\cap\ :\ (a \to \Omega) \to (a \to \Omega) \to (a \to \Omega)$

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

$\setminus\ :\ (a \to \Omega) \to (a \to \Omega) \to (a \to \Omega)$

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

$\subseteq :\ (a \to \Omega) \to (a \to \Omega) \to \Omega$

$s \subseteq t = \forall x.(x \in s \ \longrightarrow\ x \in t)$

$\supseteq :\ (a \to \Omega) \to (a \to \Omega) \to \Omega$

$s \supseteq t = t \subseteq s$

$\mathit{mapset}\ :\ (a \to b) \to (a \to \Omega) \to (b \to \Omega)$

$\mathit{mapset}\ f\ s = \{y \mid \exists x.((x \in s) \wedge (y = (f\ x)))\}$

$\mathit{powerset}\ :\ (a \to \Omega) \to (a \to \Omega) \to \Omega$

$\mathit{powerset}\ \{\} = \{\{\}\}$

$\mathit{powerset}\ \{x \mid \mathit{if}\ \boldsymbol{u}\ \mathit{then}\ v\ \mathit{else}\ \boldsymbol{w}\} = \mathit{if}\ v\ \mathit{then}\ \mathit{powerset}\ \{x \mid \boldsymbol{u} \vee \boldsymbol{w}\}\ \mathit{else}\ \mathit{powerset}\ \{x \mid \neg\ \boldsymbol{u} \wedge \boldsymbol{w}\}$

$\mathit{powerset}\ \{x \mid x = t\} = \{\{\}, \{t\}\}$

$\mathit{powerset}\ \{x \mid \boldsymbol{u} \vee \boldsymbol{v}\} = \{s \mid \exists l.\exists r.(l \in (\mathit{powerset}\ \{x \mid \boldsymbol{u}\})) \wedge (r \in (\mathit{powerset}\ \{x \mid \boldsymbol{v}\}) \wedge (s = l \cup r))\}$

$linearise \ : \ (a \to \Omega) \to (a \to \Omega)$

$linearise \ \{\} = \{\}$

$linearise \ \{x \mid if \ \boldsymbol{u} \ then \ v \ else \ \boldsymbol{w}\} = if \ v \ then \ linearise \ \{x \mid \boldsymbol{u} \vee \boldsymbol{w}\} \ else \ linearise \ \{x \mid \neg \ \boldsymbol{u} \wedge \boldsymbol{w}\}$

$linearise \ \{x \mid x = t\} = \{x \mid if \ x = t \ then \ \top \ else \ \bot\}$

$linearise \ \{x \mid \boldsymbol{u} \vee \boldsymbol{v}\} = (linearise \ \{x \mid \boldsymbol{u}\}) \cup (linearise \ \{x \mid \boldsymbol{v}\})$

$\uplus : (a \to Nat) \to (a \to Nat) \to (a \to Nat)$

$\lambda x.0 \ \uplus \ m = m$

$(\lambda x.if \ x = t \ then \ v \ else \ \boldsymbol{w}) \ \uplus \ m = \lambda x.if \ x = t \ then \ v + (m \ t) \ else \ (\lambda x.\boldsymbol{w} \ \uplus \ (remove \ \{t\} \ m)) \ x$

$\ominus : (a \to Nat) \to (a \to Nat) \to (a \to Nat)$

$\lambda x.0 \ \ominus m = \lambda x.0$

$(\lambda x.if \ x = t \ then \ v \ else \ \boldsymbol{w}) \ \ominus m = \lambda x.if \ x = t \ then \ v \dot{-} (m \ t) \ else \ (\lambda x.\boldsymbol{w} \ \ominus m) \ x$

$\sqcup : (a \to Nat) \to (a \to Nat) \to (a \to Nat)$

$\lambda x.0 \ \sqcup \ m = m$

$(\lambda x.if \ x = t \ then \ v \ else \ \boldsymbol{w}) \ \sqcup \ m = \lambda x.if \ x = t \ then \ max \ v \ (m \ t) \ else \ (\lambda x.\boldsymbol{w} \ \sqcup \ (remove \ \{t\} \ m)) \ x$

$\sqcap : (a \to Nat) \to (a \to Nat) \to (a \to Nat)$

$\lambda x.0 \ \sqcap \ m = \lambda x.0$

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

$\sqsubseteq : (a \to Nat) \to (a \to Nat) \to \Omega$

$\lambda x.0 \sqsubseteq m = \top$

$(\lambda x.if \ \boldsymbol{u} \ then \ v \ else \ \boldsymbol{w}) \sqsubseteq m = (\forall x.(\boldsymbol{u} \longrightarrow v \leq (m \ x))) \wedge ((remove \ \{x \mid \boldsymbol{u}\} \ \lambda x.\boldsymbol{w}) \sqsubseteq m)$

$\sqsubset : a \to (a \to Nat) \to \Omega$

$x \ \sqsubset m = (m \ x) > 0$

$if_then_else \ : \ \Omega \times a \times a \to a$

$if \ \top \ then \ u \ else \ v = u$

$if \ \bot \ then \ u \ else \ v = v$

$ifsome_then_else \ : \ \Omega \times \Omega \times \Omega \to \Omega$

$ifsome \ \exists x_1. \cdots \exists x_n.\top \ then \ \boldsymbol{x} \ else \ y = \exists x_1. \cdots \exists x_n.\boldsymbol{x}$

$ifsome \ \exists x_1. \cdots \exists x_n.\bot \ then \ \boldsymbol{x} \ else \ y = y$

$ifsome \ \exists x_1. \cdots \exists x_n.(\boldsymbol{x} \wedge (x_i = u) \wedge \boldsymbol{y}) \ then \ \boldsymbol{z} \ else \ v =$
$\quad ifsome \ \exists x_1. \cdots \exists x_{i-1}.\exists x_{i+1}. \cdots \exists x_n.(\boldsymbol{x}\{x_i/u\} \wedge \boldsymbol{y}\{x_i/u\}) \ then \ \boldsymbol{z}\{x_i/u\} \ else \ v$

$ifsome \ \exists x_1. \cdots \exists x_n.(\boldsymbol{x} \vee (x_i = u) \vee \boldsymbol{y}) \ then \ \boldsymbol{z} \ else \ v = \exists x_1. \cdots \exists x_n.((\boldsymbol{x} \vee (x_i = u) \vee \boldsymbol{y}) \wedge \boldsymbol{z})$

$\in : \ a \to (a \to \Omega) \to \Omega$

$t \in \{x \mid \boldsymbol{u}\} = \boldsymbol{u}\{x/t\}$

$\notin : \ a \to (a \to \Omega) \to \Omega$

$x \notin s = \neg\,(x \in s)$

$\lambda x.\boldsymbol{u} : a \to b$

$\lambda x.\boldsymbol{u}\ t = \boldsymbol{u}\{x/t\}$