The Use of the Typed Lambda Calculus for Guiding Naive Users in the Representation and Acquisition of Part-Whole Knowledge.

Mike Uschold

AIAI-TR-182

July 1995

Submitted to Data and Knowledge Engineering for a special issue on Parts and Wholes. An extended abstract of this paper appeared in the proceedings of a workshop on Parts and Wholes: Conceptual Part-Whole Relations and Formal Mereology at ECAI-94 in Amsterdam.

Artificial Intelligence Applications Institute
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
United Kingdom

© The University of Edinburgh, 1995.

Abstract

We address the task of enabling naive users in a practical context to define, comprehend and use knowledge bases for representing part-whole information. This work is part of a larger effort whose target users were ecologists who had little experience in mathematics, computing, and artificial intelligence, but who wished to build computer simulation models of ecological systems.

The ecological domain has a rich variety of part-whole information. This includes individuals, populations and sub-populations, as well as composite entities. We note the special requirements deriving from the need to satisfy naive users and show how various existing approaches are insufficient. We describe a novel representation, based on the typed lambda calculus which covers the above range of part-whole relationships in a flexible, uniform framework.

We emphasise the role of the typed lambda calculus in particular, and more generally, how a careful description of the ontology founding our representation can be used to guide users in creating accurate, transparent knowledge bases, which in turn facilitates reuse and sharing.

Keywords knowledge representation; part–whole; interaction between part–whole and specialisation; ontology; typed lambda calculus; ecological modelling; simulation

1 Introduction

In this paper, we show how the typed lambda calculus [13] can be used in novel way for the purpose of enabling naive users to build and use accurate, transparent knowledge bases containing a significant range of part–whole information. We stress two inter-dependent aspects of this work:

1. representing part-whole information;
2. the special requirements of naive users.

This work[1] was motivated by the goal of enabling naive users to define and use knowledge bases for representing arbitrary ecological systems [14, 17, 16]. Target users were ecologists who had little experience in mathematics, computing, and artificial intelligence, but who wished to build computer simulation models of ecological systems.

By concentrating on one application area, and identifying a range of requirements, and presenting solutions, we hope to contribute to the ultimate goal of having a complete and formal understanding of the nature of the part–whole relationship and how it may be used in a wide variety of applications. In the nearer term, it is necessary to clarify the sort of requirements that exist in different domains, and to develop a suite of techniques to address different requirements. While the requirements motivating this work were in the domain of ecological simulation modelling, most of the solutions described here have much more general application.

1.1 Requirements

1.1.1 Part–Whole Information

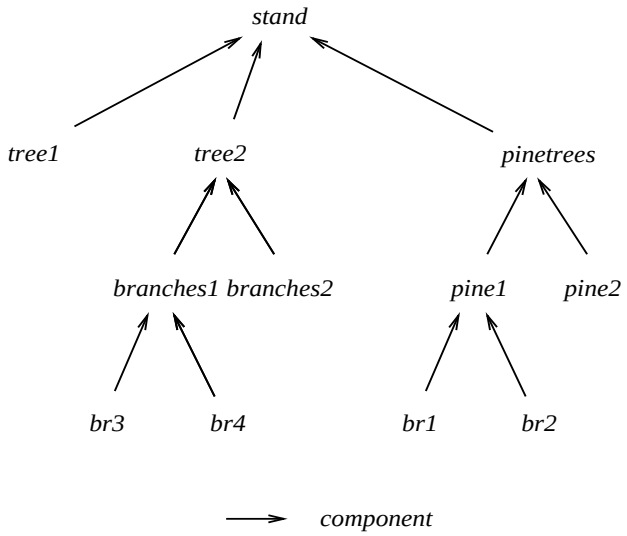
Sets and parts may be important in any application domain; they are especially so in ecological modelling. Examples include animal populations, forest stands, as well as branches and leaves as parts of trees. Different levels of detail are required in different circumstances, but all of this must be catered for. We note three basic kinds of part–whole relationships:

- member of a set
- subdivision, *i.e.* subset, of a set
- part of a composite

Example Suppose an ecologist is interested in modelling a forest stand with substructure. The stand consists of various individual trees as well as a substand consisting only of pines. Each tree may or may not have branches that are of interest. Different sets of branches might receive different amount of sunlight, and thus must be distinguished.

At one level of abstraction, there is a single part–whole relationship indicating the structure of this hypothetical stand, as depicted in figure 1. As such, users must not be forced to distinguish which kind of part–whole they are referring to when no purpose is served. Yet it must be possible to infer the distinctions when needed. From the names of the entities,

[1] This research was supported by SERC grant GR/E/00730 under the supervision of Alan Bundy and Bob Muetzelfeldt at Edinburgh University.



All three different kinds of substructure are represented in a uniform manner. The nature of a particular substructure relationship is inferrable from the types of the entities.

Figure 1: Substructure of a Forest Stand

it is easy to guess which are simple individuals, sets, or composite entities and hence to see which of the three types of relationship is associated with each link.

In this paper, we present a uniform framework which enables users to build up and display part–whole hierarchies without explicitly distinguishing the three kinds of relationship. We formalise the distinction between simple individuals, sets, and composite entities using a rich type structure. Inference is used to formalise the above-mentioned 'guessing' to determine the nature of the relationship when it is required.

A Note on Terminology Throughout this paper, we use the term 'substructure' to refer to the structuring of entities using the part–whole relationship. To avoid confusion with part–composite, we usually use the term 'component–whole' for this relationship; it should be regarded as a synonym.

Substructure is specified as a 'component hierarchy' (as in figure 1), which which includes set membership, subset of a set and incorporates partonomies which are restricted to the part–composite relationship. Note that the underlying specification of substructure may be viewed as grouping parts into a whole, subdividing a whole into its component parts, or both. Which view is most useful depends on the application, but is independent of the underlying substructure.

1.1.2 Catering for Naive Users

We had the following, partially conflicting requirements:

1. a very expressive formalism to capture a wide variety of complex ecological entities and relationships;
2. mechanisms to ensure accurate knowledge bases;
3. a 'user-friendly' knowledge acquisition system.

To achieve the first two, we developed Elklogic, based on the typed lambda calculus. However, this makes it difficult to achieve the third. The users would not understand the notation of such a language, nor, if they did, would they be able to know where to begin or how to proceed. Guidance is needed. To that end, we provided:

- a carefully described ontology which is
  - based on sound everyday intuitions,
  - rich in capturing part–whole information which is prevalent in ecological modelling
- an interactive computer program, ELK, for acquiring knowledge bases and whose interface embodies the ontology.
  - this gives the user a structure in which to work
  - the ontology is used to ensure consistency in the knowledge base

Of particular importance is that the constructs of the language be intuitively easy to grasp. Standard object-oriented programming systems are billed as being very natural, but they have representational shortcomings when it comes to part–whole information. We might have used a language from the KL-ONE family, but these are difficult to master, and come with no ontological guidelines as to how they are best used [4]. In various ways, they are inappropriate to our application, having features that we do not require (e.g. classification) and lacking the generality required for our greater concern of specifying simulation models.

1.2 The Basic Approach

In brief, our approach was to develop a formalism, Elklogic, based on the typed lambda calculus which embedded an object-oriented style of representation, thus taking advantage of its 'naturalness'. We gave it the extra expressive power required, and developed some ontological guidelines for using it. These guidelines help users avoid common errors, thus ensuring accurate knowledge bases.

Because of the nature of our inference requirements, the usually severe problems of computational efficiency associated with an expressive formalism do not arise.

1.3 Outline

We begin with an overview of Elklogic, including the ontology on which it is based. We then elaborate on motivation and development of the representation of sets and the part–composite relationship. We note our specific requirements, and indicate the shortcomings with existing representations. We present a novel representation based on the typed lambda calculus and show how it solves some of these problems. We show how providing ontological guidance helps ensure accurate knowledge bases which are therefore more reusable and sharable. Finally, we summarise our contributions and compare related work.

2 Overview of Elklogic

Elklogic is based on the typed lambda calculus [13] and consists of a kernel of generic primitives which are deemed to be useful to all applications. User's create their own additional primitives, dynamically changing the vocabulary of the language. Although end users are never aware of it, they are in fact defining the vocabulary of the logic formalism.

In this section, we first describe the ontology on which Elklogic is based and then give the details of the main features. In § 3 we concentrate on the details of representing substructure.

2.1 The Ontology

There are five main concepts in our ontology: Processes, Entities, Substructure, Attributes, and Values. We refer to this as the PESAV framework. We distinguish two fundamental levels of information:

Generic Level The role of information at this level is to characterise the range of *possible* systems in the domain of interest. Predications at this level express what is true in the domain in general. This includes:

existence of a kind of entity: there are such things as wildebeest

taxonomic information: a wildebeest is a kind of animal

existence of a certain kind of substructure relationship: A set of animals can be a subdivision of an aggregate animal population.

definition of an attribute: 'number' (set cardinality) is an attribute that applies to set entities; it can take on values from the set of natural numbers.

definition of a process: predation is a process that takes place between two kinds of animals, one is called the predator, the other the prey. The *effects* of predation include: the transfer of biomass from the prey to predator, to decrease the numbers of the prey population, etc.

System Level The role of information at this level is to describe some particular system in the domain of interest (*e.g.* the Serengeti). Predications at this level express what is true in this system. The key concepts at this level are:

- *existence of a specific entity:* there is a particular wildebeest population
- *a specific substructure relationship:* a particular lion population is a component of a particular aggregate predator population.
- *an occurrence of a process:* the predator population is preying on the wildebeest population.

There is a third major level which consists of the executable simulation program. It contains variables, equations etc, but does not concern us here (see [16] for details).

A complete specification at the generic level is called a generic model. It represents a *kind* of system, *e.g.* tropical rainforests. A complete specification at the system level is called a system model, *e.g.* a particular tropical rainforest. A system model is defined in terms of exactly one generic model. Collectively a system model and its corresponding generic model are referred to as a domain model. In ELK, the domain model is used to assist the construction and documentation of simulation software, but such knowledge bases could serve other purposes too.

Specifying a Domain Model

The way a user proceeds is to decide what *kinds* of things they need to represent; this entails using the five ontological concepts to create a generic model. Once any part of a generic model is created, a user may begin creating a system model.

This distinction between the generic and system levels facilitates:

Reuse, and Sharability: The PESAV framework can be used to create a wide variety of generic models. A single generic model can be reused to create many different system models for one or more users.

Controlling the Modelling Search Space: The PESAV framework assists a user in navigating the space of possible generic models by providing a structure in which to proceed which indicates what is possible to create. A generic model assists the user in navigating the space of possible system models in the same way. In addition, it helps ensure consistency by prohibiting senseless specifications such as grass eating cows, or a tree being part of a branch.

2.2 The Language

Elklogic, the foundation for ELK, is a wide-spectrum language based on the typed lambda calculus. The ontology described in § 2.1 is directly reflected in both the set of constructs in Elklogic (*e.g.* kinds of things are represented as types) and in the ELK user interface.

Our main concern here is the ability of Elklogic to represent ecological systems in general, and in particular the substructure relationships that arise. However it is also used to represent specifications for simulation programs, *and* links between the specifications and the corresponding domain models. The links automatically document the simulation software.

2.2.1 Requirements

The key requirements that motivated the design of Elklogic are:

- the ontological significance of each language construct must be intuitively clear to naive users;
- clearly defined intended semantics of each language construct so that complex expressions can translated into English, which in turn enables the user to check the accuracy of their knowledge base;
- minimum of effort on users part
  - ELK should be able to infer 'obviously' true facts;
  - avoid redundancy in creating the knowledge base;
- uniformity, *i.e.* similar things should be represented similarly
- user should be able to make distinctions when necessary without being forced to do so explicitly; nevertheless ELK must be able to infer distinctions when necessary.

By 'intended semantics' we do not mean a formal semantics in the traditional sense. The latter "...usually account for a set of models which is much larger than the models we are interested in ...[9]". Our intended semantics is informal but includes our ontological assumptions. Enforcing these helps ensure that sensible knowledge bases are constructed.

The main technique needed to facilitate the above requirements is *parsimony*. We keep the number of primitives small, and use a rich type structure and higher order functions. The generality of the primitives makes it possible to avoid specifying very similar things over and over again. One example of this is the use of a higher order function *average* which is used to induce the attribute 'average biomass' for a set of sheep from the fact that the attribute 'biomass' applies to the sort 'sheep' (details in § 3).

Having simple combinable primitives makes it easier for users to create complex expressions incrementally, and for the system to break them down for translation purposes (as in figure 4).

We draw special attention to the last point, which is a case of getting the balance right. We discussed one case of this in § 1.1.1 with respect to the different kinds of substructure relationship in the single component hierarchy. Another example is the situation whereby users rarely need to be aware of the internal substructure of a set entity. It usually suffices to specify and to know that an entity is *some* kind of set. Still, ELK must be able to determine the exact details to test applicability of certain functions; *e.g.* 'maximum average weight' is meaningless in the context of a simple flock. The rich type structure in Elklogic is instrumental in handling this.

2.2.2 Main Features of Elklogic

In this section, we introduce the basic components and features of Elklogic. We use the following terminology: a *type* is analogous to a concept in a KL-ONE style of language, or a class in an object-oriented representation; a *sort* is a primitive type; an *entity* is an instance of a type; *attributes* are analogous to roles or slots. The notation $E{:}T$ means that entity E is of type T. A summary of Elklogic notation is given in table 1.

1. a sort hierarchy
   e.g. *sheep* is the type of sheep, and a subsort of *animal*.

2. a type hierarchy induced from the sort hierarchy using a set type constructor[2].
   e.g. $set(sheep)$ is the type of sets of sheep, and a subtype of $set(animal)$
   (see figure 2)

3. entities are instances of types; they are either individuals or sets
   e.g. $shp{:}sheep$; $flock{:}set(sheep)$

4. attributes are represented as binary functions from entities and time to values
   e.g. $weight : phys_obj \times time \mapsto positive_real$

5. simple (not multiple) inheritance of attributes for subtypes from the supertype.
   e.g. weight for *animal* is inherited from *phys_obj*.

[2]By 'set' we mean any group of arbitrary entities; this is not intended to imply the use of the formal axioms of set theory.

Type Constructors	
set	set formation
#	collection;
Relations	
:	instance
$\sqsubset^s_\circ$	basic subsort relation; defines the sort hierarchy
$\sqsubset^s$	proper subsort relation; transitive closure of $\sqsubset^s_\circ$
$\sqsubseteq^s$	subsort relation; reflexive version of $\sqsubset^s$
$\sqsubset$	proper subtype relation; induced from $\sqsubset^s_\circ$ and *set*
$\sqsubseteq$	subtype; reflexive version of $\sqsubset$
$\prec^p_\circ$	basic possible part
$\prec^p$	possible part; transitive closure of $\prec^p_\circ$
$\subset^p_\circ$	basic possible component; induced from $\sqsubset^s_\circ$, #, and $\prec^p_\circ$
$\subset^p$	possible component; transitive closure of $\subset^p_\circ$
$\subset_\circ$	basic component
$\subset$	proper component; transitive closure of $\subset_\circ$
$\subseteq$	component; reflexive version of $\subset$

Table 1: Summary of Elklogic

6. values are ranges of attribute functions; they are a special type of entity
   e.g. real

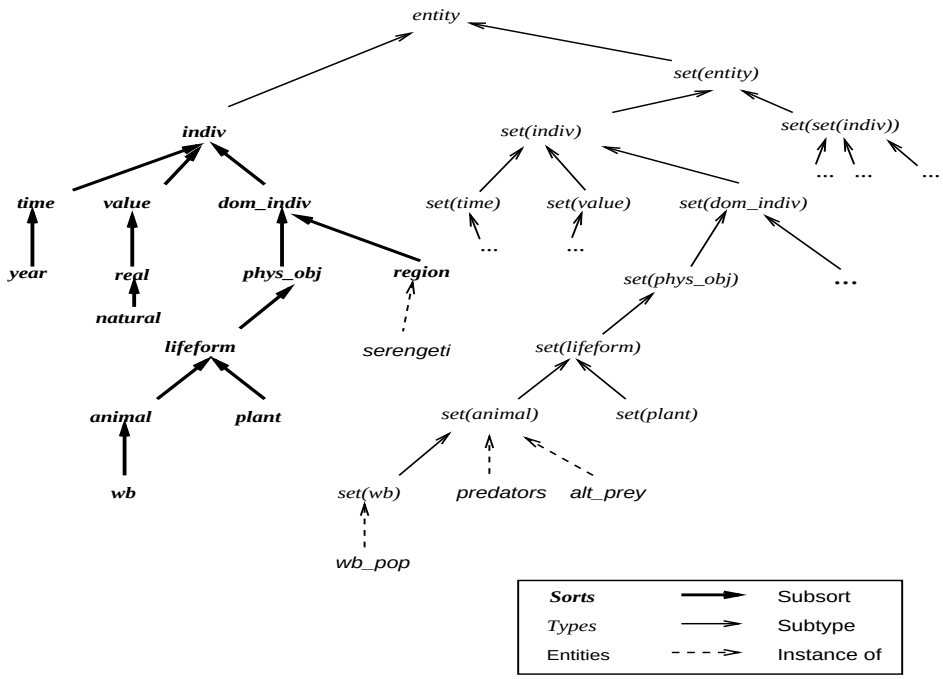
7. processes: defined to be something that changes the value of an attribute of one or more entities. Relations are used to represent what entities are involved in what processes, their effects are represented as functions.
   e.g. predation

8. a novel representation for sets and substructure;
   e.g. $flock{:}set(sheep)$ represents a flock of sheep; although $flock$ is a set, it is *not* a type, nor is it necessarily possible, even in principle to identify or represent any of its member sheep (see § 3 for further details).

9. induced attributes for set entities (*e.g.* average, maximum)
   These are represented as higher order functions (see § 3 for further details).

We do not support definition of subtypes that are specialisations of existing types on the basis of restricting values of attributes (*e.g.* to create a 'black sheep' from 'sheep'). Consequently, *we do not support classification.* The need for this has not arisen in our application,

We now turn our attention to a more detailed look at representing substructure.



entity	the most general type
indiv	the most general sort
set(*entity*)	the most general set type

Figure 2: Part of the Subtype Relation, and some Entities

3 Substructure

As noted above, we require to capture the similarity of the three different substructure relationships (member, subdivision, part) in a uniform framework so as not to force the user to distinguish them when it serves no purpose. Further, ELK must be able to infer the particular type on when required. We first consider sets, and then we turn our attention to incorporating parts into a uniform representation for substructure.

3.1 Sets

There is a fundamental distinction in Elklogic between individuals and sets as illustrated in figure 2. Very simply, an individual is anything that is *not* a set. The most general type of individual is *indiv*, the root of the sort hierarchy. The most general type of set is *set*(*entity*).

In attempting to represent ecological systems, we identified the need for:

- sets whose individual members need not, even in principle, be able to be identified;

- sets that are not types;
- the ability to ignore the internal substructure of a set by viewing it as equivalent to its flattened version while always retaining the ability to make the distinction if required;
- the ability to subdivide populations on the basis of different values of attributes (*e.g.* black and white sheep) *without* forcing the creation of these as new types.

Although we require the ability to represent the individuals of a population, this is usually not necessary. It is a hindrance to force this.

All typed logics that we are aware of [7, 1], require all sets to be types. However, this can force the the type mechanism to be used in an unintuitive, ontologically dubious manner. The ontological category for types is 'kinds of things'. While it will never be possible to say with certainty for every thing, whether it can reasonably be viewed as a kind, there are times when one can be fairly sure. A sheep is a natural kind but a particular flock may not be. If there was a specific property giving rise to the flock, *e.g.* its colour, then it may be useful to define subtypes for black and white sheep[3] However, the ecologist might wish to distinguish the flocks for reasons other than what kind of sheep they contain *e.g.* they are being fed different types of food. It is unnatural to force a user to define a type such as $sheep_that_is_fed_high_grade_food$. There are also sound ontological reasons for not doing so, (see the discussion of rigidity in [11]). Instead, one merely creates an instance of the set type for sheep: $hg_flock{:}set(sheep)$. All the members of this set entity are of the same type, $sheep$, but not all sheep are members of this set.

Elklogic allows this, and goes further by allowing meaningful expressions to be used as the names of the instances. These expressions use higher order functions referring to the attributes and values defining the sub-group. If $flock{:}set(sheep)$ represents the whole flock, then something like[4] $flock(grade_of_food, high){:}set(sheep)$ would represent the subflock in the above example.

The simple name hg_flock would be adequate for some purposes, but the meaning suggested by the name is only accessible to users. The structured name allows ELK to reason about the fact that all the members of this particular flock are fed high grade food (see [5, 16] for details). Of course, the complex expressions are completely hidden from the user; they are created via a graphical user interface depicted in figure 3.

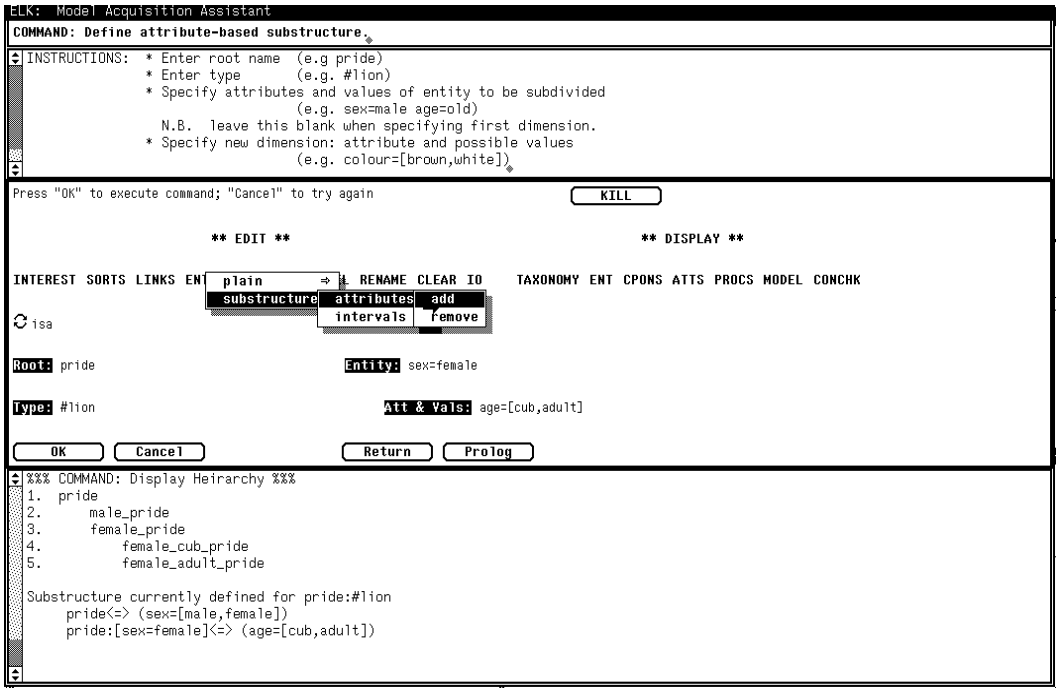
Collections

To allow users to ignore the details of a set entity's internal structure when it is irrelevant, we introduce the type $\#T$, a generalisation of $set(T)$. An entity of type $\#T$ is called a collection, and its members may be any of the types T, $set(T)$, $set(set(T))$, etc. ELK users always use this abstraction when creating instances of set entities.

When ELK needs to know the details, it is easily inferred by examining a collection's component members and sub-collections, and their components etc. This is required to determine the applicability of certain functions such as 'maximum average weight' which cannot be computed unless the collection is a subtype of $set(set(T))$ for some type T.

[3] Elklogic does not currently support this, but we have no reason to expect that doing so would present any significant difficulties.

[4] This is simplified for clarity; the actual representation is a syntactic variant of this.



The user specifies that the female pride, (not the male) is to be further subdivided by age. The substructure is displayed in the bottom window and consists of the portion of the component hierarchy rooted at pride. The entity names shown are meaningful abbreviations of the underlying complex logical expressions.

Figure 3: Attribute-Based Substructure

Inducing Attributes of Set Types

There are a variety of generally useful statistical attributes that only apply to set entities (*e.g.* 'average weight', 'maximum height'). We require a mechanism that enables us to infer the fact that, say 'average weight' applies to entities of type $set(sheep)$ from the fact that 'weight' applies to entities of type $sheep$. Furthermore, an average may be taken over a set of entities at a specific time, or for a single entity over a set of different times. We require a representation that can handle both cases, as well as support nesting to enable concepts like 'average daily maximum temperature for a three month period' to be represented. We sketch our solution below; see [5, 16] for full details.

Concepts such as 'average' and 'maximum' are naturally represented as second order functions with two arguments. The first argument is a unary function defined on some type of entity; the second is a set of entities required to be the same type over which the function is defined. For example, the type of $maximum$ is:

$$\forall T \sqsubseteq entity, \forall V \sqsubseteq value. \quad maximum\colon\ (T \mapsto V) \times set(T) \mapsto V$$

$set(T)$ is the type of the set of entities over which the maximum can be calculated. $T \mapsto V$ is the type of the unary function which would be applied to each member of this set. Note how the use of the type variable, T, and the type constructor, set, enables us to ensure that the members of the set are correctly typed to be arguments of the unary function.

We use this technique to induce attributes of entities of a set type from the attributes of the base type; *e.g.* we infer that the attribute 'maximum weight' applies to sets of trees from the fact that the attribute weight applies to trees. Suppose $weight{:}phys_obj \times time$ is a binary function representing the attribute weight. Below, we represent the maximum weight of a physical object over some time period as well as the maximum weight of a set of physical objects at a single time. Both are binary functions whose types are indicated.

$$\begin{array}{l} \lambda O{:}phys_obj\ .\ \lambda P{:}set(time)\ . \\ maximum(\lambda T{:}time.weight(O,T),P) \qquad : \quad phys_obj \times set(time) \ \mapsto\ value \end{array}$$

$$\begin{array}{l} \lambda Os{:}set(phys_obj)\ .\ \lambda T{:}time\ . \\ maximum(\lambda O{:}phys_obj.weight(O,T),Os) \quad : \quad set(phys_obj) \times time \ \mapsto\ value \end{array}$$

These complex expressions must, of course, be hidden from the user. Experienced users need only use short formally defined abbreviations; *e.g.* $max_T_biomass$, and $max_biomass$ to refer to the first and second cases respectively. The T denotes that the average is taken over a set of times.

The expression: $max_biomass(flock, now)$ thus represents the maximum biomass of $flock{:}set(sheep)$ at time $now{:}time$ and $max_T_biomass(tree1, 1980)$ represents the average biomass of $tree1{:}tree$ over the period $1980{:}set(month)$ where 1980 is required to be of type $set(time)$. For example, it could be a set of months having components: $jan80{:}month$, $feb80{:}month$, etc. The substructure of time is represented in the same way as any other substructure, using the component relationship.

For nesting more complex combinations as in the example given above, or for less experienced users, there is a specialised interface. This is depicted in figure 4.

This facility gives rise to a proliferation of induced attributes, several for each attribute of each type; indeed infinitely many since arbitrarily deep nesting is allowed. However, this

causes no problem because they are *implicit*; generated only on demand. Understanding how this works, requires an understanding of how the domain model is used in ELK.

Attributes are associated with the most general type to which they apply, and are inherited for more specific types. However, the role of the attributes is to document the meaning of simulation variables. For example, when a user wishes to create a simulation variable, they choose the entity which the variable will describe (*e.g.* $tree1$). ELK computes the set of attributes which apply to entities of type *tree* and presents these to the user in a menu. The user selects one, *e.g. height*; in doing so, they have formally associated a variable with what it simulates in the domain model.

Thus, when an instance is created of a type, it there is no data structure created which creates slots for each attribute; anyway it would be impossible for the infinitely many induced attributes for set entities.

Aside: Note that such attributes are useful, even when no explicit statistical computation will take place. Users always associate variables in the simulation model with attributes in the domain model. So, a variable which is to simulate the average weight of a particular flock at any given time would be formally associated with the attribute: avg_weight (actually, it would be a complex expression as above) of the entity $my_flock : set(sheep)$. This information enables the model to be self-documenting. This need for self-documenting simulation models, is not a major concern here. We mention it because it inspired many of the design decisions taken in this research.

3.2 Parts and Components

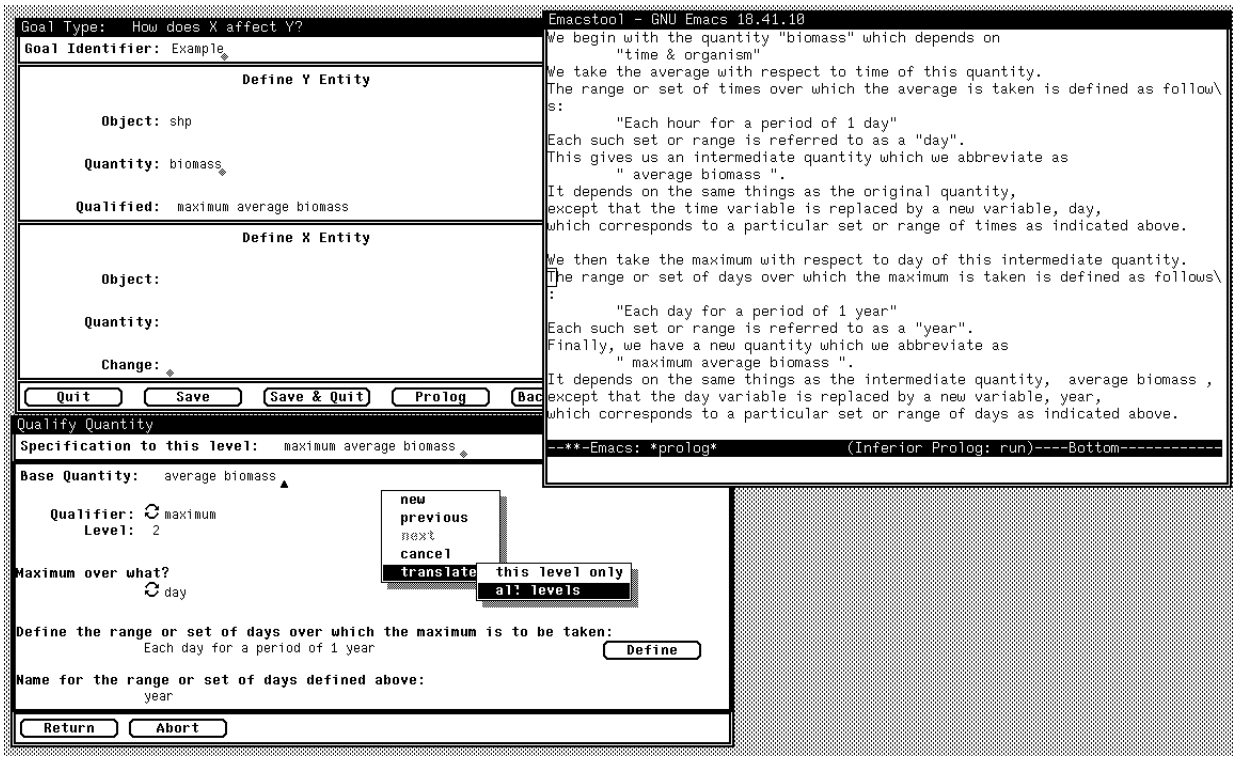
There have been some attempts to generalise the three different substructure relationships noted above into a single relationship which captures the idea of a structured whole [6]. However, we are aware of no implementations.

Our main reasons for creating a component relationship which incorporates these three cases, are to achieve parsimony and uniformity which in turn makes things easier for the user building up knowledge bases. There is no need to specify the similar properties more than once (*e.g.* transitivity). There is a uniform mechanism for specifying component relationships, independent of which of the three kinds they are. This enables a user to build up a hierarchy of components (as in figure 1) without explicitly specifying which of the three component-whole relationships holds in any particular case. ELK infers this information when needed.

Another important design constraint is the need to prevent users from making mistakes (*e.g. tree1* is part of *branch4*). We insist on having a specification at the generic level which says what kind of component–whole relations are possible and thus permitted in the system model.

Our notation for the system level relation for specifying components is adopted from [6]: '$E_c \subset E_w$' denotes that the entity E_c is a proper component of the [whole] entity E_w. $\subset$ is defined in terms of $\subset_o$, the base relation. $\subset$ is the transitive closure of $\subset_o$.

The following represents the fact that a particular paw is part of a particular lion which is in turn a member of a pride of lions which is a subdivision of a collection of prides which is in turn a subdivision of an aggregate predator population that includes other predator species:



The user is defining the dependent variable using the Define Y *window. They start by indicating what entity they are concerned with. They then select the attribute of interest from a menu constrained to those appropriate to the type of the entity. This is then qualified by average and maximum using a separate window. A detailed English translation of the state of the specification is automatically generated.*

Figure 4: Interface for Building up Complex Expressions

$$paw1 \subset_\circ lion1 \subset_\circ pride \subset_\circ ln_pop \subset_\circ predators$$

Where $paw1$:paw, $lion1$:$lion$, $pride$,ln_pop:$\#lion$ and $predators$:$\#animal$. Using transitivity we can infer $paw1 \subset ln_pop$, etc. We can also infer which of the three kinds of component-whole relationship holds by analysing the types of the component and whole entities. For example, because $lion1$:$lion$ and $pride$:$\#lion$, the $lion1$ is a member of $pride$, not a subdivision, or part of a composite.

Permitting Substructure in the System Model

It should never happen that a lion population be a component of a lion, nor for a lion to be a component of a paw. To prevent these mistakes, we require a mechanism for specifying what substructure relationships are permitted at the system level.

For this, we define the generic level relation, $\subset^p$, which we call 'possible component'. Its intended semantics is as follows: '$T_c \subset^p T_w$' denotes that it makes sense for an entity of type T_c to be a component of an entity of type T_w. Enforcing the following rule achieves our goal:

$$\forall T_c, T_w \sqsubseteq entity. \forall E_c{:}T_c, E_w{:}T_w \,.\, T_c \not\subset^p T_w \quad \rightarrow \quad E_c \not\subset E_w \tag{1}$$

For the above examples, we require:

$$\#lion \not\subset^p lion \qquad\qquad lion \not\subset^p paw$$
$$paw \subset^p lion \subset^p \#lion \subset^p \#lion \subset^p \#animal$$

Before we can define $\subset^p$, we require a mechanism for stating what kind of part–composite relationships can hold. For this we define another generic level relation, $\prec^p$, called 'possible part'. It is the transitive closure of $\prec^p_\circ$, which is a hierarchy specified directly by the knowledge base builder.

The intended semantics for this relation is analogous to that of $\subset^p$; '$S_p \prec^p S_w$' denotes that all entities of sort S_w are composite entities which have parts of sort S_p. For example, '$claw \prec^p lion$' says that all sorts of lions have claws. $claw \not\prec^p animal$ says that "it is not the case that all sorts of animals have claws" which is true because only some animals have claws. Given this interpretation, if we were building a comprehensive knowledge base, we would have $claw \not\prec^p_\circ lion$, but $claw \prec^p_\circ feline$ from which we would infer that $claw \subset^p lion$ because $lion \sqsubset feline$. This suggests a relationship between the sort and part hierarchies; it is formalised below.

Everything is now in place to define the possible component relation. Although $\subset^p_\circ$ and $\subset^p$ are similar in basic structure to the corresponding subtype, part and component hierarchies, $\subset^p_\circ$ is more complex than $\sqsubset^s_\circ$, $\prec^p_\circ$, or $\subset_\circ$. Rather than a single explicit relation which is directly defined, $\subset^p_\circ$ is defined in terms of 5 separate cases which are in turn based on the following three things:

1. the notion of a collection type, *i.e.* using #
2. the sort hierarchy, *i.e.* $\sqsubset^s$
3. the possible part hierarchy, *i.e.* $\prec^p$

The first is derived from set, and forms the backbone of the theory. The latter two are defined by the knowledge-base builder. The five basic cases are:

Possible member: it is possible for an entity of type $\#S$ to be subdivided into member entities of type S.
e.g. $tree \subset^p_\diamond \#tree$ says that a collection of trees can be subdivided into member trees.

Possible subdivision: is is possible for an entity of type $\#S$ to be a subdivided into entities of type $\#S$
e.g. $\#tree \subset^p_\diamond \#tree$ says that a collection of trees can be subdivided into (sub-)collections of trees.

Possible collection part: an entity of type S_w can be subdivided into collections of parts of type S_p, if $S_p \prec^p S_w$.
e.g. $\#branch \subset^p_\diamond tree$ says that a tree can be subdivided into collections of branches.

derived Possible subdivision: it is possible for an entity of type $\#S_2$ to be subdivided into entities of type $\#S_1$, if $S_1 \sqsubset^s S_2$.
e.g. $\#pine \subset^p_\diamond \#tree$ says that a collection of trees can be subdivided into collections of pines (because a pine is a tree).

derived Possible collection part: an entity of type S can subdivided into collections of parts of type $\#S_p$, if for some sort S_w, $S_p \prec^p S_w$ and $S \sqsubset^s S_w$
e.g. $\#branch \subset^p_\diamond pine$ says that a pine can be subdivided into collections of branches (because a pine is a tree, and a branch is a part of a tree).

The possible component relation is derived by starting with these basic cases and taking the transitive closure. Formally:

$$\forall S \sqsubset^s indiv. \quad S \subset^p_\diamond \#S \tag{2}$$

$$\forall S \sqsubset^s indiv. \quad \#S \subset^p_\diamond \#S \tag{3}$$

$$\forall S_p, S_w \sqsubset^s ecol_indiv. \quad (S_p \prec^p_\diamond S_w) \rightarrow \#S_p \subset^p_\diamond S_w \tag{4}$$

$$\forall S_1, S_2 \sqsubset^s indiv. \quad (S_1 \sqsubset^s S_2) \rightarrow \#S_1 \subset^p_\diamond \#S_2 \tag{5}$$

$$\forall S_p, S, S_w \sqsubset^s ecol_indiv. \quad S_p \prec^p_\diamond S_w \wedge S \sqsubset^s S_w \rightarrow \#S_p \subset^p_\diamond S \tag{6}$$

$$\begin{aligned} S_c \subset^p_\diamond S_w &\rightarrow S_c \subset^p S_w \\ S_c \subset^p_\diamond S_m \wedge S_m \subset^p S_w &\rightarrow S_c \subset^p S_w \end{aligned} \tag{7}$$

We use $\prec^p_\diamond$ rather than $\prec^p$ to avoid redundancy (and consequent computational difficulties) which would arise because both $\prec^p$ and $\subset^p$ are transitive. The required possible part and subsort hierarchies, and the induced possible component relation are depicted in figure 5. Although there are different kinds of component relationships, all are created and viewed with a single uniform mechanism, the same one used to handle other relations, *e.g.* subsort.

Thus, $\subset^p$ in conjunction with $\subset$ enables us to blur the distinction between the various kinds of substructure, however the information is not lost. To see this, consider again the example in figure 1. The types of the entities in the hierarchy are indicated below.

$stand:$	$\#tree$
$pinetrees:$	$\#pine$
$pine1, pine2:$	$pine$
$br1, br2, br3, br4:$	$branch$
$tree1, tree2:$	$tree$
$branches1, branches2:$	$\#branch$

It is a simple matter for ELK to examine the types of component and whole entities and match against the base cases that define $\subset^p$, applying transitivity, where required. For example, the manner in which $br1 : branch$ is a component of $pine1 : pine$ is clearly 'part-composite' because it would not be permitted unless $branch \prec^p tree$; whereas $tree1{:}\#tree$ is clearly a 'member' of $stand{:}\#tree$ because it is permitted by basic case 2 listed above.

The main function of the possible component relation is to ensure that only sensible substructure is defined; however it can also be used to suggest possible fixes, when errors are spotted. For example, if a user has not specified that branches are parts of trees, the system will not allow $br1 \subset tree2$; but it can suggest that the user specify $branch \prec^p_\diamond tree$ as a fix. In this way, users are supported in the creation of accurate knowledge bases.

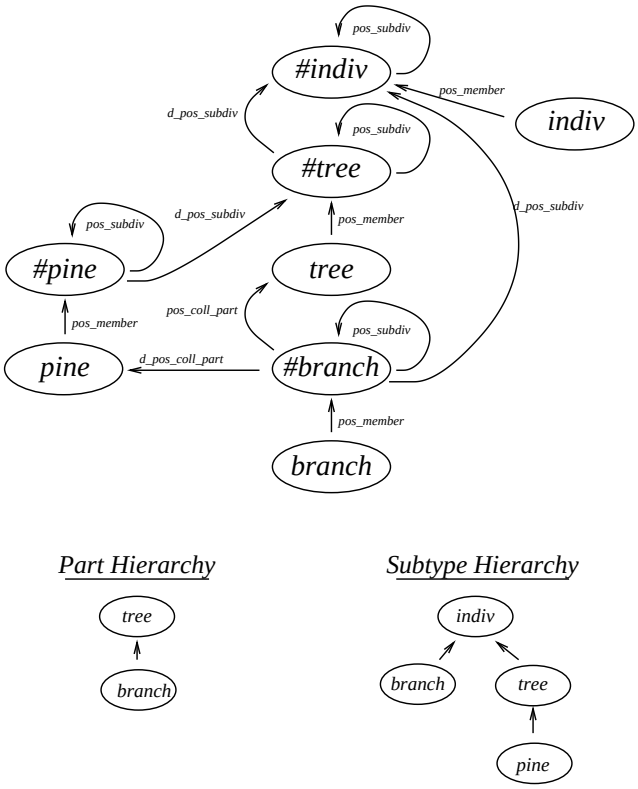
Wholes with no Components Just as set entities need not have members, composite entities – *i.e.* those whose types appear in the possible part hierarchy – need not have explicit components. Nevertheless, the fact that something is a composite type of entity may be used, for example, in determining the applicability of certain mathematical relationships for simulation modelling. If $branch \prec^p tree$ then ELK can infer that an equation which only applies to types of plants that have branches, will apply to an entity of type $tree$. Conversely, because $branch \not\prec^p potato_plant$, the equation would not apply to potato plants.

Homogeneity In Elklogic, the notion of homogeneity is easily defined. A type T is homogeneous iff $T \subset^p T$. All collection types are homogeneous by rules 3 and 7, but sorts in general, are not, *e.g.* $tree \not\subset^p tree$. We do require some sorts to be homogeneous, *e.g.* $region$. This may be accomplished by specifying $region \prec^p_\diamond region$. Rules 2, 4 and 7 imply that $region \subset^p region$ just as $branch \prec^p_\diamond tree$ allows a branch to be a component of a tree. Summarising,

- For all sorts S, the type $\#S$ is homogeneous.
- A sort S is homogeneous if and only if $S \prec^p S$.

The careful reader will see this as a hack, in conflict with the ontological role of the possible part relationship which is for the part–composite relationship. It may be possible to simply add a rider to the intended semantics of $\prec^p$, however the adequacy of this has not been explored. Distinguishing a whole new language construct for the ontologically distinct notion of a homogeneous sort would definitely complicate things, possibly unnecessarily.

Continuity Homogeneity is related to the continuous/discrete distinction. Most or all continuous entities are homogeneous, but the reverse it not true; *e.g.* a $region$ is continuous, but a finite set is not. However in Elklogic, because there is no need to specify members of sets, sets behave much as if they were continuously decomposable.



The possible component relation is wholly induced using the notion of a collection in conjunction with the sort and possible part hierarchies. It defines the space of possible components.

Figure 5: Induced Possible Component Relation

Elklogic provides no explicit mechanism for distinguishing between continuous and discrete homogeneous sorts. Insofar as users can specify components ad infinitum for homogeneous entities, they behave as if they were continuous. Entities like sand and sugar may be viewed both as continuous and discrete. The latter view can be made explicit by specifying $grain \prec^{p}_{\circ} sand$. Thus we can accommodate both the continuous and the discrete view of things like sand and sugar.

4 Benefits, Support and Inference

In this section, review and summarise the benefits of our approach, the nature of the support we provide to naive users, and indicate the range of inferences that ELK performs on the user's behalf.

Expressive Power: Elklogic, in conjunction with ELK gives users the ability to capture a wide range of important distinctions in a natural intuitive manner. The expressive power is mainly derived from the parsimonious use of few combinable primitives in conjunction with higher order functions and λ– abstraction.

Uniformity / Abstraction: The similarity of similar things is captured in the representation in a uniform manner and is reflected in the user interface. Users are not forced to make distinctions unnecessarily, yet the ability to make the distinctions if required is retained. Important examples are the abstraction of the three kinds of part–whole relationship in a single component hierarchy; and the ignoring of the internal structure of collections.

Consistency: The separation of the generic and system levels gives the capability to prevents users from making a range of errors. For example, a user may not specify that a flock of sheep is a component of a sheep nor that a tree is a component of a branch. Not only will ELK inform the user that they have made a likely mistake, it suggests ways to fix the problem.

Transparency: the ontological significance of each language construct is both intuitively clear to naive users and directly reflected in ELK's user interface. The main technique was the parsimonious use of relatively few combinable primitives, each having a clearly defined intended semantics. Complex expressions are easily broken down into their parts and translated into English giving clear access to the knowledge base.

From the user's perspective, uniformity also contributes to transparency.

Other Useful Services:

Induced set types: ELK induces parallel type hierarchies of set types from the user's specification of the sort hierarchy (see figure 2)

Induced Attributes of Set Types: Users need only specify attributes of individual types; the system infers the existence of attributes for corresponding set types.

Induced possible component hierarchy: users need only specify the sort hierarchy and the possible part hierarchy.

Guidance: The PESAV framework provides a structure in which to work thereby reducing the the space of possible generic models that can be specified. Once created, a generic model assists the user in navigating the space of possible system models in the same way. In important part of this is the ensuring of consistency as explained above.

Status of Elk

The status of ELK is a research prototype. The expressive capability of Elklogic was pushed fairly hard on one rich, diverse and open-ended example (the Serengeti). A wide range of difficult concepts, especially regarding substructure, were successfully captured.

ELK has been informally tested on a number of users. Before they started, we explained the fundamentals of the ontology, highlighting the differences with other tools they may have used previously, *e.g.* object-oriented programming languages, simulation languages. Initial results have been encouraging, but more thorough testing is required before any strong claims are made about the success of our ontological guidance in constructing knowledge bases by naive users.

5 Conclusions and Related Work

We have described an approach to enable naive users in a practical context to define, comprehend and use knowledge bases for representing part-whole information. Below we summarise our main achievements and consider related work.

5.1 Representing the Part–Whole Relationship

We described a novel representation for substructure based on the typed lambda calculus. Its key features are:

- a set type constructor (*set*) for inducing set types;
- a facility for representing sets:
  - that are *not* types (*e.g.* $flock{:}set(sheep)$);
  - whose individual members need not (even in principle) be able to be identified.
  - a mechanism for inducing the attributes of set types form the base sort;
- a component relation which incorporates in a uniform framework the three kinds of pert–whole relationship: member–set, subdivision–set and part–composite.

The inducing of the possible component relationship from the possible part and sort hierarchies, formally represents one type of interaction between part–whole and specialisation relationships.

There appears to be at least two major strands of research in this field: one is the extension of general purpose representation languages to include support for the part–whole relationship independently from considerations of a specific application. The other, including this research, is to identify the requirements for representation and inference of part–whole information in a particular domain and to develop solutions to meet those requirements. In order to further overall progress in the field, it is important that the solutions developed

for specific applications are nevertheless more generally applicable. This we have done; nothing in our solution is specifically limited to the domain of ecological modelling. The wider applicability of our solutions depend mainly on the extent to which the requirements for other domains are similar.

Our induced attributes for set types (e.g. *average*) is a contribution to to solution of the general problem of inferring what attributes apply to the whole(part) based on those applicable to the part(whole). In our case, this has been limited to sets.

There are many other kinds of part–whole relationships, as are described in [8] each with corresponding inference needs; many of these distinctions were not important for our domain.

Our requirements for inference are relatively minimal, from a computational standpoint. We have no need for classification inference; we have ignored the issues of overlapping, dependence of parts on corresponding composite wholes, and using existence of parts as defining properties of types (*e.g.* primate is 2-legged mammal ...).

As a matter of convenience, it could be useful to automate the creation of all component parts of composite entities, however, in the simulation modelling context, this must be the user's choice, it should *not* be imposed. In ELK, no entity, whether its type is is a set or a composite (by virtue of having possible parts) is required to have explicit components.

We are aware of no other approaches that allow a single representation which captures the similarity between the various part–whole relationships, yet allows the user to choose whether or not to make the finer distinctions.

An important emerging theme in part–whole inference, is the interaction between the part–whole, and specialisation (subtype) relationships[12, 2]. We have attempted no general theory of this, but have addressed one part of it.

Our work is similar in spirit to the part–whole extension to Classic described in [15]. Both efforts recognise that a small set of requirements should be identified and solved; both carefully separate types from entities and use this information to ensure consistency. There is an important difference between our approaches, however. They have a formal semantics and proven computational properties, with strict well defined limitations on expressive power. We, on the other hand, have much fewer restrictions on expressive power, but our inference requirements are rather less.

Our work is similar to that of CYC, in that we do not tradeoff expressive power and efficiency; instead, we lack formal semantics and ability to prove formal properties such as soundness and completeness.

The theoretical limitations of our work, reflect the fact that the requirements were driven by practical need. We have spent less effort addressing theoretical limitations which, while in principle may be beneficial, cause no obvious practical problems. Where others have emphasised theoretical aspects, many have ignored implementation, and none, to our knowledge, have attempted to cater for naive users.

5.2 Supporting Naive Users

The above representation, in conjunction with our distinction between the generic and system levels, facilitates guiding naive users in creating and using knowledge bases. Our use of the typed lambda calculus to control the modelling search space and facilitate transparency is itself novel. Of particular importance is the the use of few combinable primitives. A good

example of this is the use of higher-order functions and polymorphic typing which allows us to have general representations of concepts like maximum and average for any kind of attribute across any range of objects. Importantly, users can build up complex expressions in a gradual way, and because the meaning of each primitive is clear, translation into English is straightforward.

In the literature, there is plenty of discussion about various levels, including implementation, logical, epistemological, etc.[3]. [10] discusses an ontological level. Our approach is similar; where we differ is that our higher levels are formally represented, and furthermore, there are formal links connecting them that represent simplifications and assumptions that have been made. The cost of our approach is that the different levels are needed. The major benefit is the self-documenting models, and the consequent potential for reuse.

References

[1] Cohn. A. On the solution of shubert's steamroller in many sorted logic. In *Proceedings of IJCAI-85*. International Joint Conference on Artificial Intelligence, 1985.

[2] P. Bernauer. Modelling formal subsumption and part-whole relation for medical concepts description. In N. Guarino, S. Pribbenow, and L. Vieu, editors, *ECAI-94 Workshop W2 – Parts and Wholes: Conceptual Part-Whole Relations and Formal Merology*, pages 69–79, 1994.

[3] R.J. Brachman. On the epistemological status of semantic networks. In N.V. Findler, editor, *Associative Networks: Representation and Use of Knowledge by Computers*. Academic Press, 1979.

[4] R.J Brachman, D.L. McGuinnes, P.F. Patel-Schneider, L.A. Resnick, and A. Borgida. Living with classic: When and how to use a kl-one-like language. In J. Sowa, editor, *Principles of Semantic Networks*, pages 401–456. Morgan Kaufmann, 1990.

[5] A. Bundy and M. Uschold. The use of typed lambda calculus for requirements capture in the domain of ecological modelling. Research Paper RP-446, Dept. of Artificial Intelligence, Edinburgh, 1989. submitted to Artificial Intelligence Journal.

[6] Bunt. The formal representation of (quasi-) continuous concepts. In *Formal Theories of the Commonsense World*. Norwood Ablex, 1986.

[7] Luca Cardelli. The quest language and system. Tracking Draft, August 1989.

[8] P. Gerstl and S. Pribbenow. Midwinters, endgames and bodyparts: a classification of part-whole relations. In N. Guarino and R. Poli, editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation*. Kluwer, 1994. (to appear).

[9] N. Guarino. A concise presentation of itl. In M. Richter, editor, *Processing Declarative Knowledge*. Springer-Verlag, 1991. previously published in ACM SIGART Bulletin, Special Issue on Implemented Knowledge Representation and Reasoning Systems; Vol 2, No. 3, June 1991.

[10] N. Guarino and L. Boldrin. Concepts and relations. Technical Report LADSEB-CNR Int. Rep. 1/92, Institute for System Dynamics and Bioengineering, Padova Italy, 1992.

[11] N. Guarino, M. Carrara, and P. Giaretta. Formalising ontological commitments. In *Proceedings of National Conference on Artificial Intelligence (AAAI-94)*, pages 270–280, 1994.

[12] P. Hors. Description logics to specify the part-whole relation. In N. Guarino, S. Pribbenow, and L. Vieu, editors, *ECAI-94 Workshop W2 – Parts and Wholes: Conceptual Part-Whole Relations and Formal Merology*, pages 103–109, 1994.

[13] Barendregt. H.P. *The Lambda Calculus.* Elsevier, 1985.

[14] D. Robertson, A. Bundy, R. Muetzelfeldt, M. Uschold, and M. Haggith. *Eco-Logic: Logic-based approaches to Ecological Modelling.* MIT Press, 1991.

[15] P-H. Speel and P. Patel-Schneider. A whole-part extension for description logics. In N. Guarino, S. Pribbenow, and L. Vieu, editors, *ECAI-94 Workshop W2 – Parts and Wholes: Conceptual Part-Whole Relations and Formal Merology*, pages 111–121, 1994.

[16] M. Uschold. *The Use of Typed Lambda Calculus for Comprehension and Construction of Simulation Models in the Domain of Ecology.* PhD thesis, Dept. of Artificial Intelligence, Edinburgh, 1990.

[17] M. Uschold, D. Robertson, A. Bundy, and R. Muetzelfeldt. Helping inexperienced users to construct simulation programs: An overview of the ECO project. In S. Vadera, editor, *Expert System Applications*, pages 117–132. Sigma Press, 1989. This is a revised and extended version of a paper published in 'Research and Development in Expert Systems 4', BCSES 1987; Also available as D.A.I. Research paper 338.