Theoretical Computer Science 45 (1986) 293-351
North-Holland

AN ALGEBRAIC SEMANTICS APPROACH TO THE EFFECTIVE RESOLUTION OF TYPE EQUATIONS

Hassan AÏT-KACI*
Microelectronics and Computer Technology Corporation, Austin, TX 78759, U.S.A.

Communicated by M. Nivat
Received October 1985
Revised April 1986

Abstract. This article presents a syntactic calculus of partially-ordered data type structures and its application to computation. A syntax of record-like terms and a type subsumption ordering are defined and shown to form a lattice structure. A simple 'type-as-set' interpretation of these term structures extends this lattice to a distributive one, and in the case of finitary terms, to a complete Brouwerian lattice. As a result, a method for solving systems of type equations by iterated rewriting of type symbols is proposed which defines an operational semantics for KBL—a Knowledge Base Language. It is shown that a KBL program can be seen as a system of equations. Thanks to the lattice properties of finite structures, systems of simultaneous type equations are shown to admit least fixed-point solutions. An operational semantics for KBL is described as term rewriting. *Fan-out rewriting,* a particular rewriting computation order related to the conventional outermost term rewriting which rewrites symbols closer to the root first, is defined and shown to be maximal. Correctness with respect to least fixed-point semantics of KBL's operational semantics, as defined by fan-out rewriting, is discussed. Finally, extensions and further research directions are sketched.

Key Words: Partially-ordered types, type inheritance, type equations, algebraic semantics, graph-rewriting systems.

Contents

1. Introduction 294
 1.1. Disclaimer 294
 1.2. Organization of contents 295
2. Motivation and background 296
 2.1. Semantic networks 296
 2.2. The first-order logic approach 297
 2.3. The initial algebra approach 299
 2.4. The denotational approach 299
3. Concrete data type structures 302
 3.1. Desideratum 302
 3.2. A dialectic approach 303
  3.2.1. Thesis 303

* Most of the research reported in this article was done while the author was at the University of Pennsylvania, and it completes results previously published in [1].

0304-3975/86/$3.50 © 1986, Elsevier Science Publishers B.V. (North-Holland)

3.2.2. Antithesis ... 303
3.2.3. Synthesis ... 304
4. A calculus of type subsumption ... 305
4.1. A syntax of structured types ... 306
4.2. The subsumption ordering ... 311
4.3. A distributive lattice of types ... 316
5. Solving equational type specifications ... 319
5.1. A KBL interpreter ... 321
5.2. Graph rewriting ... 324
5.2.1. Wft substitution ... 324
5.2.2. Symbol-rewriting systems ... 330
5.2.3. Correctness ... 334
6. Extension of research ... 336
6.1. Negative information ... 336
6.1.1. Capturing inequalities ... 336
6.1.2. Complemented types ... 337
6.2. Polymorphic types ... 337
6.3. Further research ... 338
6.3.1. Psi-expansion considered unnecessary ... 338
6.3.2. Epsilon-types as a type system for a programming language ... 339
6.3.3. Higher-order types ... 339
6.3.4. Integrating logic ... 340
7. Conclusion ... 340
Appendix A. Lattice ideals ... 341
Appendix B. Brouwerian lattices ... 342
Appendix C. Powerlattice constructions ... 342
C.1. A semi-lattice construction ... 343
C.2. A distributive lattice construction ... 344
Appendix D. A semantics of type inheritance ... 346
Acknowledgment ... 349
References ... 349

Ce cours qu'Irène Guessarian professe depuis quelques années à l'Université Paris VII va beaucoup plus loin qu'elle ne le dit elle-même: il intéresse certainement toute l'informatique dont les progrès passeront nécessairement par une étude approfondie des structures mathématiques qui la sous-tendent, il doit intéresser le mathématicien et le logicien sensible à la réalité et l'existence qu'on peut presque dire physique des objets manipulés.

Maurice Nivat, *Preface to Irène Gessarian's lecture notes on algebraic semantics* [30].

1. Introduction

1.1. Disclaimer

It has yet again become clear to the author that ideas from virtually mutually exclusive branches of computer science may contribute constructively to one another. On one hand, universal algebra has been rather successful at formalizing, explicating, and thus predicting the behavior of computational concepts. On the other hand, artificial intelligence (AI) has made undeniably intriguing contributions towards comprehending rather esoteric—and thus inherently *very hard* and *very controversial*—phenomena related to human intelligence seen as a computational process.

Thus, we feel compelled to begin this article with a disclaimer pertaining to a certain scientific attitude.

Amongst the varied fauna and flora thriving in the circles of research in computer science, there is a particular breed whose members, self-proclaimed or not, are commonly known as 'AI researchers'. Whatever life-forms that breed really encompasses, where they live, what criterion they indeed fill to adhere to the species, is the subject of a rather uninteresting and fruitless on-going debate. An interesting observation, however, is the fact that a large number of them practice their trade favoring rather empirical, as opposed to formal, methods. A potential explanation for this, in our opinion, is that their quest has an almost alchemical nature. By that token, much of their contribution suffers from a certain despite by the formalists, especially of the 'pure type'. Among many possible reasons, a likely one is their choice of words which are perhaps a bit too pretentious for the sterner classicists of theoretical computer science (e.g., artificial intelligence, epistemological engineering, knowledge representation, knowledge base, expert systems, etc.). Be that as it may, the result of this state of affairs is mutual exclusion of ideas.

Now, being not so clever (brave?) as to be in either camp, this author wishes to voice the clear and loud disclaimer that it is clearly a loss of scientific opportunity to isolate such disciplines—especially for so trivial and vain reasons. Research is mining a gold field, and a formal scientific method leading to replication of empirical results through a systematic and minute study of *even the wildest idea* makes as obvious sense as an ore refinement process.

The nature and direction of this reported research is precisely such an effort of cross-fertilization of ideas: on the one hand, the ideas and insights evolved out of 'knowledge representation' in AI; and on the other hand, the rigor and formal tools crafted in the study of algebraic semantics of recursive program schemes. The outcome of the exercise is a better understanding (at least in the author's mind) of great intuitions put forth by some members of the AI community, as well as an impressive show of power of the abstract tools developed mostly by the French school of algebra for a clear formal explanation of new challenges in computer science. Further, the exercise has even led to new practical enhancements (read, 'running software') of some computation processes and architectures [2].

1.2. Organization of contents

Following a survey of related approaches to set this work's motivation, the first part of this article focuses on syntactic properties of record-like data type structures. A syntax of structured types is introduced as labelled infinite trees, which may be seen as extrapolated from the syntax of first-order terms as used in algebraic semantics [14, 30, 31]. However, since the terms defined here are not to be interpreted as operations, the similarity is purely syntactic. A calculus of partially ordered record structures is presented. It is then extended to variant record structures through a powerlattice construction. The second part deals with solving recursive type

equations in a lattice of variant records. An operational semantics of type structure rewriting is first informally described. Then, a fixed-point semantics is discussed. Finally, a discussion of the correctness of the former with respect to the latter concludes the main point. Extensions and further research are ultimately sketched as a conclusion.

2. Motivation and background

The area of semantics of data types has experienced growing interest in the last decade. A recent, albeit already dating, compendium of technical approaches covering the latest research in this topic can be found in [34]. The notion of *partially ordered* data types has just begun being the focus of intense research. Although it would be quite presumptuous here to attempt a complete survey of what has been and is being done, this section gives a rapid overview of a few major approaches.

2.1. Semantic networks

It has often been the case in artificial intelligence (AI) research that naïve experimentation with intuitive ideas has led to some interesting concepts. Such experimentations dealing with ways of representing commonsense knowledge in a computer brought about the notion of *semantic network*. In 1968, Quillian [54] proposed a computer representation of associative memory. What he was suggesting was a graph-theoretic denotation of '*concepts*' as vertices and '*conceptual associations*' as edges. Soon after, Minsky [46] introduced a theory of *frames*, essentially record-like data structures, related together by a so-called '*Is-A*' link meant to import or '*inherit*' information from one frame to another. A set of so-related frames was called a *conceptual taxonomy* or *knowledge base*. In the years that followed, many AI and database researchers proposed a flurry of semantic network models and languages along the lines of these early ideas [21].

Figure 1 shows an example of a piece of 'knowledge base', expressed using KL-ONE [8], one among the most popular semantic network formalisms. The ellipses are called *concept* nodes, and denote generic entities. The squares are called *role* nodes, and denote attributes of the concept to which they are attached. The diamonds are called *role/value maps* and capture equality constraints among roles. The double arrows are called *inheritance* links and denote the 'Is-A' relation among concepts and roles. The single arrows are *value restriction* links pointing to the concept restricting a role instance. The dashed arrows are called *focus/subfocus chains* and denote sequences of attributes linked by the English '*of*' connective, and constituting two arms of a role/value map. Among other things, the network in Fig. 1 expresses that an employee is a person, that a self-employed person is an employee who is her own boss, and that a locally-employed person works in the city where she lives.

Although formally naïve, the notion of semantic network has had great practical appeal as it is intended to capture some sort of '*static inference*', using frames as

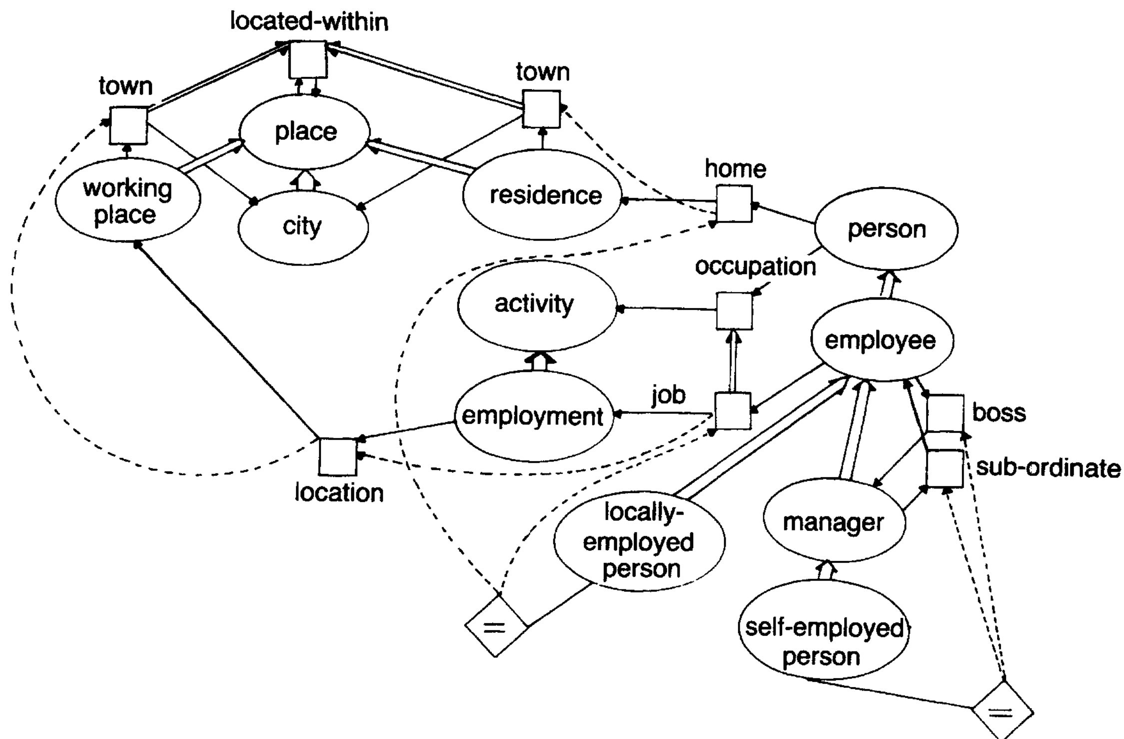

Fig. 1. Example of a KL-ONE semantic network.

generic types specifying templates or patterns whose instances inherit properties by 'pointer chasing', as opposed to '*dynamic inference*', carried out by a theorem prover or a database query language. Unfortunately, as can easily be understood by looking at Fig. 1, semantics of semantic networks is at best approximate, and it becomes quickly intractable to maintain, let alone debug, even relatively small databases of this nature. For this reason, the notion of semantic network has been subject to a controversy, even among AI researchers, where logic programming proponents have decried the very concept as being an unnecessary and semantically unclean notational variant of the first-order predicate calculus.[1] We next briefly examine some of these proposals.

2.2. *The first-order logic approach*

The glaring lack of formal semantics for semantic networks has induced certain AI and database researchers [5, 16, 36] to use the well-established first-order predicate calculus to formalize conceptual taxonomies.

There are essentially two ideas. One [5] proposes to use first-order logic as a metalanguage to describe a semantic network. The subsumption relation is denoted by a SUBTYPE binary predicate. Of course, to reflect the fact that this is an ordering, axioms for transitivity must be added. Another predicate ROLE, is a ternary one: ROLE(t_1, r, t_2) asserts that a t_1 has a role r which is a t_2. Ground objects are declared to be of some type of a TYPE predicate, as in TYPE(o, t), denoting that the object

[1] But see [32] for a vehement, as well as entertaining, counterpoint.

o is of type t. Thus, the authors of [5] make their case by showing that a semantic network can be expressed as first-order sentences involving these predicates.

The other proposal [16, 36] may roughly be summed up as follows. Since links in a network can be viewed as binary relations, and since a predicate denotes a relation, then a semantic network is nothing but a set of logical formulae involving only binary predicates, constants, and logical variables. A binary predicate ISA is used to assert that an object is of a certain type, as in ISA(socrates, human). The subsumption relation is defined as logical implications, as in

$$\mathrm{ISA}(x, \mathrm{human}) \Rightarrow \mathrm{ISA}(x, \mathrm{animal}).$$

Attributes are expressed by assertions of the form $A(t_1, t_2)$ meaning that t_1 has an attribute A of type t_2, as in FRIEND(socrates, plato).

In addition, to achieve a better abstraction power (in particular incomplete information, and limited temporal reasoning), a first-order *event calculus* is necessary. Thus, all predicative actions (verbs) and assertions are made in relation to some existential event. That is, rather than GIVES(john, mary, book) they propose to write

$$\mathrm{EVENT}(e_1, \mathrm{giving})\ \&\ \mathrm{AGENT}(e_1, \mathrm{john})\ \&\ \mathrm{RECIPIENT}(e_1, \mathrm{mary})\ \&\ \mathrm{OBJECT}(e_1, \mathrm{book}).$$

In both of the foregoing proposals, it is argued that first-order logic theorem provers are thus sufficient to implement sophisticated knowledge bases in a semantically pure fashion. The idea is indeed difficult to challenge.

Nevertheless, the 'static' inference motivation is definitely lost. Further, doing everything in first-order logic may even introduce inefficiency of computation. This is made clear by taking an example. Let us suppose that PROLOG is given a type system along the above lines, and consider the typed tplus operation defining addition on integers:

$$\mathrm{tplus}(X : \mathrm{integer},\ Y : \mathrm{integer},\ Z : \mathrm{integer})$$

becomes

$$\mathrm{tplus}(X, Y, Z) :\!-\ \mathrm{isa}(X, \mathrm{integer}), \mathrm{isa}(Y, \mathrm{integer}), \mathrm{plus}(X, Y, Z), \mathrm{isa}(Z, \mathrm{integer}).$$

Obviously, nesting this operation many times over in a sequence of calls to 'tplus' so that, for instance, the result of one becomes argument of the next, leads to grossly redundant (run-time) type checking. It is however semantically immaculate.

A work strongly related to ours is one that this author has just recently learned about [58]. Rounds and Kasper present a Kripke-style semantics for a logic of record structures for linguistic information. Their syntactic calculus is a strict subcalculus of ours, and theirs is a model-theoretic semantics. Their contribution is a completeness theorem for their calculus. However, the same comment that we made earlier on the use of logic for type structure semantics applies in this case as

well. Namely, the motivation for such work and concepts as inheritance among type structures and logical deduction is precisely one which aims at taking some computational burden off the back of a logic by using a *semantically* equivalent, albeit *pragmatically* more practical, order-sorted logic. In addition, it encourages such subtle misconceptions as the belief that any logic is semantically equivalent to its unsorted version. A blatant counterexample is shown in [2] where an order-sorted PROLOG is shown to be of a strictly larger semantic model variety since sorts may accommodate limited forms of disjunctions and negations—which are disallowed in Horn logic.

2.3. *The initial algebra approach*

An *abstract data type* is defined to represent the essential properties of a data type in complete abstraction of how the data type may be implemented. For example, whether a set is implemented as a linear list, as an array, or as a tree, is not relevant for the abstract meaning of the set data type. Hence, universal algebra provides sufficient tools for a mathematically elegant and powerful characterization of types. Indeed, *initial algebra semantics* [24, 25, 44] provides precisely this kind of characterization by defining a data type as *the* initial algebra in the variety induced by a set of equations. The approach actually uses the concept of *multi-sorted* algebra which is a straightforward generalization of a Σ-algebra endowed with a set of *sorts*, and whose operation symbols are indexed by strings of such sorts. Thus, a function symbol f of arity n has a sort $s_1 \dots s_n, s_{n+1}$ where s_i, $i = 1, \dots, n$ is the sort of the ith argument of the operation corresponding to f, and s_{n+1} is the sort of the value returned by this operation.

Many programming languages have been implemented which are based on this idea: the better known are OBJ [26], AFFIRM [47], and CLU [38]. Among these, only OBJ has also integrated the notion of partially ordered types based on the lattice-theoretic properties of algebras and extension of sorts to be partially ordered [22]. Figures 2 and 3 suggest the way subtyping is achieved in OBJ. Figure 2 defines an abstract data type specification[2] of a ring, and Fig. 3 defines a distributive ring (i.e., a ring whose additive law is distributive on the multiplicative law) as subtype of a ring. The **uses** primitive imports the definition of the previously defined type.

The initial algebra approach is surely one of the most mathematically solid theories of data types proposed today. It however addresses issues which are concerned with abstract notions of types, and neglects to consider implementation issues.

2.4. *The denotational approach*

The denotational semantics of programming languages is essentially based on the work of Scott [59, 62]. Programs are interpreted as continuous functions between

[2] Actually, of a family of such ADTs: a variety of finitely-presented equational algebras.

```
theory RING
  sorts ring
  fns
    _+_: ring, ring → ring (assoc comm id: 1)
    _*_: ring, ring → ring (assoc comm id: 0)
    _−_: ring → ring
  vars
    X, Y, Z: ring
  axioms
    X + (−X) = 0.
    X * (Y + Z) = (X * Y) + (X * Z).
endth
```

Fig. 2. An OBJ specification of a ring structure.

```
theory DRING uses RING
  sorts dring
  subsorts dring < ring
  vars
    X, Y, Z: dring
  axioms
    X + (Y * Z) = (X + Y) * (X + Z).
endth
```

Fig. 3. An OBJ specification of a distributive ring structure.

complete lattices, also called *domains* [60].[3] Thus, the meaning of a program is defined as the least fixed point of the continuous function denoted by the program.[4] Defining the interpretation domains as complete lattices offers the possibility to define so-called *reflexive domains* which are domains isomorphic to lattice constructs of themselves and other domains (generally involving product, sum, and continuous function operations). Recursive domain equations are thus guaranteed to have well-defined solutions.

Followers of Scott have essentially adopted two ways of defining the meaning of data types. The first is the original definition proposed by Scott himself in [59] and identifies a data type to a *retract*, which is an involutive continuous function. Scott proposes to model *everything* in a *universal domain* $P\omega$, defined as the powerset of the natural numbers ordered by inclusion. Thus, a function on $P\omega$ is represented by its graph; that is, the set of pairs of antecedents and images, which may hence be recursively enumerated as sets of integers. By a bijective number coding of sets (*Gödel numbering*), Scott thus shows that $P\omega$ is isomorphic to the lattice of continuous functions on $P\omega$, and by this token explains why such apparent paradoxes

[3] Complete partial orders (cpo's) are often more appropriate than lattices to model fixed-point computation as explained in [53, 66].

[4] '*Fixed-point semantics*' is the other name for this approach to programming language semantics.

as application of a function to itself are not at all paradoxical and can be very well interpreted in $P\omega$. Furthermore, any domain can be obtained as the image of $P\omega$ by a suitable retract, thus justifying the definition of a data type as a retract. Since, by definition, a retract is a fixed point of the evidently continuous function which maps a function f on $P\omega$ to the function $f \circ f$, it follows that the retracts of $P\omega$ ordered by set inclusion form a complete sublattice. However, this ordering clearly does not correspond to containment of image-domains by retracts. A more appropriate ordering on retracts of $P\omega$ denoting subtyping is defined as:

$$r \leqslant r' \text{ iff } r = r \circ r' = r' \circ r,$$

that is, r is a subtype of r' if and only if r is a retract of $r'(P\omega)$. Unfortunately, the set of retracts of $P\omega$ does not have a lattice structure for this ordering. At best, if two retracts commute, their glb is given by their composition. Application of the semantics of data types as retracts can be found in [17, 41].

Another denotational approach to the formal semantics of data types is one which defines them as *ideals* of a semantic domain and is due to MacQueen and Sethi [39, 40]. For example, let us suppose that **Bool** is a (flat) domain of truth values, i.e.,

$$\mathbf{Bool} = \{\bot, \mathbf{true}, \mathbf{false}, \top\}$$

and that **Int** is a (flat) domain of integers. Then, we can define a domain of values **Val** as the reflexive domain solution of the following domain equation:

$$\mathbf{Val} = \mathbf{Bool} + \mathbf{Int} + \mathbf{Val} \times \mathbf{Val} + [\mathbf{Val} \to \mathbf{Val}] + \{\mathbf{wrong}\},$$

where **wrong** is the value of inconsistent objects. Hence, a type in this domain is formally defined as an ideal of **Val** which does not contain the **wrong** value.

This is, in our opinion, one of the most adequate approaches to partially ordered types since it defines in a clear way a complete lattice of types ordered by inclusion. It is however essentially aimed at expressing the meaning of higher-order functional types. Nevertheless, as shown by Cardelli [12], it offers a powerful and elegant model to define a semantics of partially ordered record-like type structures with inheritance of attributes. In the context of the MacQueen–Sethi type model, Cardelli defines what he calls *flexible record* types. Essentially, a flexible record is an *finite indexing of types*, that is something which may be represented as an *association list* of label indices and types. One can inductively define Cardelli's flexible record types as follows. Given a countably infinite set of *label symbols* and atomic types like *integer, string, boolean,* etc., as primitive types, then a construct of the form $(l_1 : t_1, \ldots, l_n : t_n)$, $n \geqslant 0$, is a flexible record if the l_i's are indexing labels and the t_i's are atomic or flexible record types, for $n \geqslant 0$. The type ( ) is the top element, and is the least informative type. Such indexings may be viewed as functions from labels to types which map all labels to $\top$ except for a finite set of labels (finite partial functions). Thus, flexible records are partially ordered using the function ordering. As a result, a complete lattice of record types is obtained, with easy and

Bibliotheek
Centrum voor Wiskunde en Informatica

practical rules to compute meets and joins. As will be seen, Cardelli's model of partially ordered record structures is close to the one we shall introduce. However, it fails to offer equality among parts of a record, and is given a different semantics than ours.

3. Concrete data type structures

Sections 2.1 and 2.3 presented two extreme approaches to the problem of object representation. On the one hand, the semantic network approach is the intuitive, perhaps naïve, attempt to provide physical data structures for a programmer to capture one's concepts into concrete records. On the other hand, the algebraic approach is concerned with abstract foundations and properties of the meaning of data types, regardless of how they may be physically implemented. Thus, it favors a systematic study, perhaps to the detriment of presenting a simple and straightforward motivation to the layperson.

Section 2.2 described some attempts to fill the gap between intuitive representations and clear semantics through the use of first-order logic. Granting that logic both shares semantic clarity with the algebraic approach, and possesses simple expressive elegance to please intuition, my main criticism of this trend of research is that it goes against the primary motivation for the use of data type in programming. As illustrated then, we argued that if information about domains of objects is handled by the same logic formalism used for computation, the very notion of data type as *static* information to be factored out of *dynamic* computation is lost.

In yet another attempt to fill the same gap, we propose in this section that naïve data structures like records can be of great power if formalized in a way to have clear meaning and to make them amenable to manipulations which are congruent with that meaning. Thus, a calculus of *concrete data types* can be developed which offers the simple representational capability inherent to record-like structures and semantic networks, and which bears a clear denotation of data types as domains of elements. In this section, an approach is motivated by means of a close look at the practical use of first-order terms as a potential data model.

3.1. Desideratum

Almost all programming languages provide for some notion of structured data type, even if only as an after-thought. Examples are record types in ADA [37], PASCAL [33], ALGOL-W [61], structures in C [35], flavors in ZETALISP [65] and INTERLISP [64], first-order terms in PROLOG [13], frames in FRL [56], concepts in KL-ONE [8], etc. These are meant as a facility to encapsulate attributive information, and their syntactic appearance is characterized as variations on sets of attribute/value pairs.

3.2. *A dialectic approach*

3.2.1. *Thesis*

Subtyping is concerned with capturing the notion of *subsumption*[5] among *concrete* objects. Thus, we would like to define a notational system for representing *approximations* of objects of which one conceives in one's mind. Moreover, we want this system to contain some mechanism which could automatically classify thus represented objects in a fashion which is congruent with their interpretation as approximations.

An example of such a system is provided by first-order *terms* or *trees* in universal algebra and logic. In PROLOG [13], the underlying logic model means first-order terms as *functions*. However, operationally, term structures are *uninterpreted* constructors. Hence, one finds it very practical to use them as *record structures*, completely forgetting their functional semantics. For example, we would like to express the fact that a person has a name, a birth date, and a sex. Representing a thus specified generic person as a term could be person(x, y, z). Then, by a convention remembered at interpretation, the symbol person at the root of a term denotes a person object, and the variables x, y, z as *place markers* for a person's name, date of birth, and sex, respectively. The classification mechanism in this model is *term instantiation*. The meaning of variables is that they stand for incomplete information and may be substituted for by terms. Thus, person('Hassan', y, z) denotes any person named 'Hassan', and person('Hassan', date(14, june, y), z) designates any person named 'Hassan' and born the 14th of June. The term appearing as the date of birth in the latter 'person' illustrates the substitution process. If we choose to define a type to be a first-order term as shown, and the type classification ordering to be term instantiation, then we have at hand a type system as wished. Indeed, the types thus defined form a *lattice* whose *meet* operation (i.e., greatest lower bound) is first-order unification [57], and whose *join* operation (i.e., least upper bound) is first-order *anti-unification*, or *generalization* [55]. PROLOG programmers are well familiar with this model which is unlike any other available in conventional programming languages and turns out to be very handy in practice.

3.2.2. *Antithesis*

There is however a certain amount of inflexibility inherent to the definition of types as terms. Firstly, a term is a finitely branching tree. In particular, it has a *fixed number of arguments*. If we want to extend the definition of a person to have also a marital status, we must entirely redefine the type 'person' to take one more argument, and hence revise all previously used instances of a person. Secondly, a term has a *fixed order of arguments*. This is very convenient to interpret consistently position within a term as having a fixed meaning. For example, in a 'person'-term, the first argument is once and for all meant to denote the person's name. Indeed,

[5] We are borrowing this term from Plotkin [52]. Although his definition is different from what will be presented here, it inspired its approach.

this is also taken advantage of by the unification process; i.e., in order to match, two terms are expected to have their corresponding subterms in the same order. This is the same principle used in most programming languages to pass procedure parameters. As a result, one must constantly keep in mind the original intended interpretation of the order of arguments. Thirdly, type subsumption as one-way pattern-matching is forcing a *common syntactical pattern* for all terms in a chain in the lattice. For example, if we define a type student(x, y, z), then we cannot express that we also intend a student to be a person since a type is identified by its constant root symbol and 'student' is distinct from 'person'. Finally, there is no provision in the definition of a term for specifying any *restriction on the pattern* of subterms. For example, restricting the name of a person to terms whose root symbols belong to, or better yet do *not* belong to a given set, is not syntactically possible.

The foregoing shortcomings of the first-order term model of types make it look rather limited. However, it has appeal because of its solid formal grounds and its simplicity. It would be of great advantage if this model of types could be enhanced so that it may keep its elegance and sound formal basis, lend itself to a powerful interpretation scheme, and yet overcome the limitations explicated above.

3.2.3. Synthesis

We propose to modify the notion of a type by extrapolating on the classical definition of a term. Let us first relax the fixed-arity constraint, i.e., a term may have an unbounded number of arguments. Next, let us relax the fixed-position constraint by *explicitly* indexing or labelling the arguments. The reader familiar with ADA [37] will note that this language allows a procedure call's actual parameters to be specified either by position, or possibly out of order by explicit labelling. However, in ADA *all* actual parameters *must* be present at run-time, possibly by default. In our case, since a type can now have a *potentially infinite* number of attributes, all that is ever needed is to specify only those which are relevant at any given time. For example, person(name⇒'Hassan') denotes the type of persons named 'Hassan', and person(sex⇒male) stands for the type of male persons. Furthermore, let us assume some partial ordering on the root symbols. This can easily be extended to an ordering on terms in a way very similar to a homomorphic extension. For example, if the symbols 'person' and 'student' are such that student<person, then we can consistently say that student(name⇒'Hassan', sex⇒male) is a subtype of person(name⇒'Hassan', sex⇒male).

The idea behind this kind of extension of a term is based on the concept of *multi-sorted* terms with the very peculiar difference that the sorts are implicitly denoted by terms themselves. This is quite a new formal window through which to look at data and program structures that makes them syntactically undistinguishable, and it forces rethinking of many related familiar notions. The concepts of *variable* and *symbol* which are central in programming as well as formal languages are to be construed in a completely different yet more general way. A variable in a first-order term term has *two* distinct purposes: it is a *wild card* and a *tag*. As a wild card, it

specifies that any term may be substituted for it; and as a tag, it constrains all positions in the term where it appears to be substituted for by the *same* term. We contend that these two roles ought to be explicitly separated. In fact, we shall try to explain that if symbols are partially ordered, the familiar notion of variable has but the restrictive designation of a term which is a *maximal element.* Symbols, and extended terms for that matter, may be specified as *upper bound constraints* within other terms. We shall try and show how a natural extension of a partial ordering on the symbols may be consistently defined on extended terms. Such classical operations as *variable substitution, term unification*, etc. also take on a radically new interpretation, of which the familiar well-known notions are but special cases.

4. A calculus of type subsumption

The notion of subtyping has recently been integrated as a feature in some programming languages, although in a limited fashion. For example, in PASCAL it is provided only for so-called *simple* types like enumeration or range types. For more complex types, in general, subtyping is not *implicitly* inferred. For example, in ADA, one must declare *explicitly* most subtyping relationships. This is true even in those formalisms like KL-ONE [8] or OBJ [26] where subtyping is a central feature. The only formalism which may be used for implicit subtyping is provided by first-order terms in PROLOG as first-order term instantiation. However, even this representation is limited as a model for partially ordered type structures. Nevertheless, it is of great inspiration for what is desired, which is a practical system of type structures which must have at least as much expressive power as offered by, say, classical record structures, as well as the capability of efficiently automating subtyping inference, and the construction of new structures from old ones.

A specific *desideratum* can be informally sketched as follows. A structured data type must have:

- a *head symbol* which denotes a class of objects being restricted;
- *attributes* (or *fields*, or *slots*, etc.) possessed by this type, which are typed by structured types themselves;
- *coreference constraints* between attributes, and compositions thereof, denoting the fact that the *same* substructure is to be shared by different compositions of attributes.

Then, a type structure t_1 is a *subtype* of a type structure t_2 if and only if:

- the class denoted by the head of t_1 is contained in the class denoted by the head of t_2; and,
- *all* the attributes of t_2 are present in t_1 and have types which are subtypes of their counterparts in t_2; and,
- *all* the coreference constraints binding in t_2 are also binding in t_1.

For example, if the symbols 'student', 'person', 'austin', 'cityname' denote sets of objects, and if student < person and austin < cityname denote set inclusion, then

the type

$$\begin{aligned}&\text{student}(\text{id}\Rightarrow\text{name}(\text{last}\Rightarrow X:\text{string});\\&\qquad\text{domicile}\Rightarrow Y:\text{address}(\text{city}\Rightarrow\text{austin});\\&\qquad\text{father}\Rightarrow\text{person}(\text{id}\Rightarrow\text{name}(\text{last}\Rightarrow X);\\&\qquad\qquad\qquad\text{domicile}\Rightarrow Y))\end{aligned}$$

should be a subtype of

$$\begin{aligned}&\text{person}(\text{id}\Rightarrow\text{name};\\&\qquad\text{domicile}\Rightarrow\text{address}(\text{city}\Rightarrow\text{cityname});\\&\qquad\text{father}\Rightarrow\text{person}).\end{aligned}$$

The letters X and Y in this example denote coreference constraints as will be explained. Formalizing the above informal wish is what this section attempts to achieve.

4.1. *A syntax of structured types*

Let Σ be a partially ordered *signature* of *type symbols* with a *top* element $\top$, and a *bottom* element $\bot$. Let $\mathscr{L}$ be a set of *label symbols*, and let $\mathscr{T}$ be a set of *tag symbols*, both nonempty and countably infinite. We shall represent type symbols and labels by strings of characters starting with a *lower-case* letter, and tags by strings of characters starting with an *upper-case* letter.

A simple 'type-as-set' semantics for these objects is elaborated in [4] and summarized in Appendix D. It will suffice to mention that type symbols in Σ denote sets of objects, and label symbols in $\mathscr{L}$ denote the *intension* of functions. This semantics takes the partial ordering on type symbols into set inclusion, and label concatenation into function composition. Thus, the syntax of terms introduced next can be interpreted as describing commutative composition diagrams of attributes.

In a manner akin to tree-addressing as defined in [14, 27, 28], we define a *term domain on* $\mathscr{L}$ to be the *skeleton* built from label symbols of such a commutative diagram. This is nothing other than the graph of arrows that one draws to picture functional maps. Formally, we have the following definition.

4.1. Definition. A *term* (or *tree*) *domain* Δ on $\mathscr{L}$ is a set of finite strings of labels of $\mathscr{L}$ such that (1) Δ is prefix-closed:

$$\forall u\in\mathscr{L}^*,\ \forall v\in\mathscr{L}^*,\ \text{if } u.v\in\Delta,\ \text{then } u\in\Delta;$$

and (2) Δ is finitely branching:

$$\text{if } u\in\Delta,\ \text{then } \{u.a\in\Delta \mid a\in\mathscr{L}\} \text{ is finite.}$$

It follows from this definition that the empty string ε must belong to all term domains. Elements of a term domain are called (*term*) *addresses*. Addresses in a domain which are not the prefix of any other address in the domain are called

leaves. The empty string is called the *root* address. For example, if $\mathscr{L} = \{a, b, c, d, e, f, g, h\}$, a term-domain on $\mathscr{L}$ may be

$$\Delta_1 = \{\varepsilon, b, b.c, b.d, b.e, a, a.g, h, h.a, h.a.f\}.$$

A term domain need not be finite; for instance, the regular expression $\Delta_2 = a(ba)^* + (ab)^*$, where $a, b \in \mathscr{L}$, denotes a regular set which is closed under prefixes and finitely branching; thus, it is a term domain and it is infinite.

Given a term domain Δ and an address w in Δ, we define the *subdomain of Δ at address w* to be the term domain $\Delta \backslash w = \{w' \mid w.w' \in \Delta\}$. In the last example, the subdomain at address b of Δ_1 is the set $\{\varepsilon, c, d, e\}$ and the subdomain of Δ_2 at address $a.b$ is Δ_2 itself.

4.2. Definition. A term domain Δ is a *regular* term domain if the set of all subdomains of Δ defined as $\mathbf{Subdom}(\Delta) = \{\Delta \backslash w \mid w \in \Delta\}$ is finite.

In the previous examples, the term domain Δ_1 is a finite (regular) term domain, and Δ_2 is a regular infinite term domain since $\mathbf{Subdom}(\Delta_2) = \{\Delta_2, b.\Delta_2\}$. In this article, we shall consider only regular term domains.

The 'flesh' that goes on the skeleton defined by a term domain consists of signature symbols labelling the nodes which are arrow extremities. Keeping the 'arrow graph' picture in mind, this stands for information about the origin and destination sets of the arrow representation of functions. As for notation, we proceed to introduce a specific syntax of terms as record-like structures. Thus, a term has a *head* which is a type symbol, and a *body* which is a (possibly empty) list of pairs associating labels with terms in a unique fashion—a (partial) function. An example of such an object is shown in Fig. 4.

```
person(id⇒name;
       born⇒date(day⇒integer;
                 month⇒monthname;
                 year⇒integer);
       father⇒person)
```

Fig. 4. An example of a term structure.

The domain of a term is the set of addresses which explicitly appear in the expression of the term. For example, the domain of the term in Fig. 4 is the set of addresses

$$\{\varepsilon, \text{id}, \text{born}, \text{born.day}, \text{born.month}, \text{born.year}, \text{father}\}.$$

The example in Fig. 4 shows an expression which one may intend to use as a data structure for a person. The terms associated with the labels are to *restrict* the types of possible values that may be used under each label. However, there is no explicit constraint, in this particular structure, *among* the substructures appearing

```
person(id⇒name(last⇒X : string);
       born⇒date(day⇒integer;
                 month⇒monthname;
                 year⇒integer);
       father⇒person(id⇒name(last⇒X : string)))
```

Fig. 5. An example of tagging in a term structure.

under distinct labels. For instance, a person bearing a last name which is not the same as his father's would be a legal instance of this structure. In order to capture this sort of constraints, one can *tag* the addresses in a term structure, and *enforce* identically tagged addresses to be identically instantiated. For example, if, in the above example, one is to express that a person's father's last name must be the same as that person's last name, a better representation may be the term in Fig. 5.

4.3. Definition. A *term* is a triple $\langle \Delta, \psi, \tau \rangle$ where Δ is a term domain on $\mathscr{L}$, ψ is a *symbol* function from $\mathscr{L}^*$ to Σ such that $\psi(\mathscr{L}^* - \Delta) = \{\top\}$, and τ is a *tag* function from Δ to $\mathscr{T}$. A term is finite (respectively regular) if its domain is finite (respectively regular).

Such a definition illustrated for the term in Fig. 5 is captured in Table 1. Note the syntactic sugar implicitly used in Fig. 5. Namely, we shall omit writing explicitly tags for addresses which are not sharing theirs. In the sequel, by 'term' will be meant 'regular term'.

4.4. Definition. Given a term $t = \langle \Delta, \psi, \tau \rangle$ and an address w in Δ, the *subterm of t at address w* is the term $t \backslash w = \langle \Delta \backslash w, \psi \backslash w, \tau \backslash w \rangle$ where $\psi \backslash w : \mathscr{L}^* \to \Sigma$ and $\tau \backslash w : \Delta \backslash w \to \mathscr{T}$ are defined by

$$\forall w' \in \mathscr{L}^*, \quad \psi \backslash w(w') = \psi(w.w'),$$

$$\forall w' \in \Delta \backslash w, \quad \tau \backslash w(w') = \tau(w.w').$$

Table 1
$\langle \Delta, \psi, \tau \rangle$-Definition of the term in Fig. 5.

Addresses (Δ)	Symbols (ψ)	Tags (τ)
ε	person	X_0
id	name	X_1
id.last	string	X
born	date	X_2
born.day	integer	X_3
born.month	monthname	X_4
born.year	integer	X_5
father	person	X_6
father.id	name	X_7
father.id.last	string	X

From these definitions, it is clear that $t\backslash\varepsilon$ is the same as t. In the example of Fig. 5, the subterm at address father.id is name(last$\Rightarrow X$: string).

Given a term $t = \langle \Delta, \psi, \tau \rangle$, a symbol f, (respectively, a tag X, a term t') is said to *occur* in t if there is an address w in Δ such that $\psi(w) = f$ (respectively, $\tau(w) = X$, $t\backslash w = t'$). The following proposition is immediate and follows by definition.[6]

4.5. Proposition. *Given a term $t = \langle \Delta, \psi, \tau \rangle$, the following statements are equivalent:*
(1) *t is a regular term;*
(2) *the number of subterms occurring in t is finite;*
(3) *the number of symbols occurring in t is finite;*
(4) *the number of tags occurring in t is finite.*

4.6. Definition. In a term, any two addresses bearing the same tag are said to *corefer*. Thus, the *coreference* relation κ of a term $t = \langle \Delta, \psi, \tau \rangle$ is a relation defined on Δ as the *kernel* of the tag function τ; i.e., $\kappa = \mathbf{Ker}(\tau) = \tau \circ \tau^{-1}$.

We immediately note that κ is an equivalence relation since it is the kernel of a function. A κ-class is called a *coreference class*. For example, in the term in Fig. 5, the addresses father.id.last and id.last corefer. It follows from Proposition 4.5 that a coreference relation on a regular term domain has finite index.

A term t is *referentially consistent* if the same subterm occurs at all addresses in a coreference class. That is, if $\mathscr{C}$ is a coreference class in Δ/κ, then $t\backslash w$ is *identical* at *all* addresses w in $\mathscr{C}$. Thus, if a term is referentially consistent, then by definition, for any $w_1, w_2 \in \Delta$, if $\tau(w_1) = \tau(w_2)$ then, for all w such that $w_1.w \in \Delta$, we must have necessarily $w_2.w \in \Delta$ also, and $\tau(w_1.w) = \tau(w_2.w)$. Therefore, if a term is referentially consistent, κ is in fact more than a simple equivalence relation: it is a *right-invariant* equivalence—a *right-congruence*—on Δ. That is, for any two addresses w_1 and w_2, if $w_1 \kappa w_2$, then $w_1.w \kappa w_2.w$ for any w such that $w_1.w \in \Delta$ and $w_2.w \in \Delta$.

4.7. Definition. A *well-formed term* (henceforth, wft) is a term which is referentially consistent.

We can use this property to justify another syntactic convention. Namely, whenever a tag occurs without a subterm, what is meant is that the subterm elsewhere referred to in the term by this tag is implicitly present. If there is no such subterm, the implicit subterm is understood to be $\top$. For example, in the term

$$\text{foo}(l_1 \Rightarrow X;\ l_2 \Rightarrow X:\text{bar};\ l_3 \Rightarrow Y;\ l_4 \Rightarrow Y)$$

the subterm at address l_1 is 'bar', and the subterm at address l_4 is $\top$. In fact, we shall never write explicitly the symbol $\top$ in a term.

This syntactic convention makes it also possible to consider infinite terms such as the one shown in Fig. 6, where a cyclic tagging occurs at addresses father and

[6] Also established in [14] for regular first-order terms.

```
person(id⇒name(last⇒X: string);
       born⇒date(day⇒integer;
                  month⇒monthname;
                  year⇒integer);
       father⇒Y: person(id⇒name(last⇒X);
                         son⇒person(father⇒Y)))
```

Fig. 6. An example of cyclic tagging in a term structure.

father.son.father. Syntactically, cycles may also be present in more pathological ways such as illustrated in Fig. 7, where one must follow a complex path of cross-references.

A term is *referentially acyclic* if there is no cyclic tagging occurring in it. A *cyclic term* is one which is not referentially acyclic. Thus, the terms in Figs. 6 and 7 are *not* referentially acyclic.

A wft is then best pictured as a *labelled directed graph* as illustrated in Fig. 8 which is the graph representation of the wft

$$\begin{aligned}X_0:f_1(&l_1\Rightarrow X_1:f_2(l_2\Rightarrow X_2;\\&\qquad\qquad l_3\Rightarrow f_3);\\&l_4\Rightarrow X_2;\\&l_5\Rightarrow f_4(l_6\Rightarrow X_1;\\&\qquad l_7\Rightarrow X_3:f_5;\\&\qquad l_8\Rightarrow X_3;\\&\qquad l_9\Rightarrow X_0)).\end{aligned}$$

As will be seen later, the similarity of the graph in Fig. 8 with a finite-state diagram is not coincidental.

The set of well-formed terms will be denoted by $\mathcal{WFT}$, and the subset of $\mathcal{WFT}$ of *acyclic* wft's by $\mathcal{WFAT}$.

We shall not give any semantic value to the tags aside from the coreference classes they define. The following relation α on $\mathcal{WFT}$ is to handle *tag renaming*. This means that α is relating wft's which are identical up to a renaming of the tags that preserves coreference classes.

4.8. Definition. Two terms $t_1=\langle\Delta_1,\psi_1,\tau_1\rangle$ and $t_2=\langle\Delta_2,\psi_2,\tau_2\rangle$ are *alphabetical variants* of one another (noted $t_1\,\alpha\,t_2$) if and only if (1) $\Delta_1=\Delta_2$; (2) $\mathbf{Ker}(\tau_1)=\mathbf{Ker}(\tau_2)$; and (3) $\psi_1=\psi_2$.

$$\begin{aligned}\text{foo}(&l_1\Rightarrow X_1:\text{foo}_1(k_1\Rightarrow X_2);\\&l_2\Rightarrow X_2:\text{foo}_2(k_2\Rightarrow X_3);\\&\qquad\vdots\\&l_i\Rightarrow X_i:\text{foo}_i(k_i\Rightarrow X_{i+1});\\&\qquad\vdots\\&l_n\Rightarrow X_n:\text{foo}_n(k_n\Rightarrow X_1))\end{aligned}$$

Fig. 7. An example of complex cyclic tagging in a term structure.

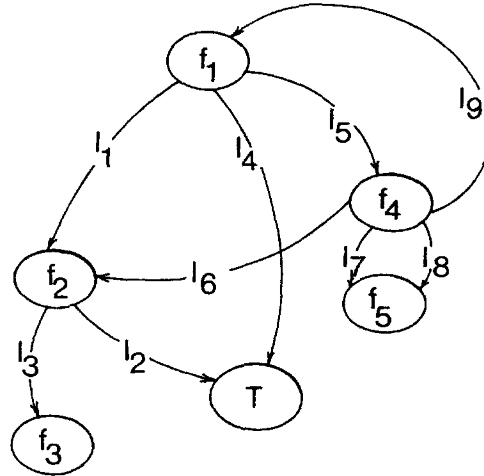

Fig. 8. Graph representation of a wft.

Interpreting these structures as commutative diagrams between sets, it follows that the symbols $\top$ and $\bot$ denote, respectively, the whole universe—*the least informative type*—and the empty set—*the overdefined, or inconsistent, type.*[7] Hence, a term in which the symbol $\bot$ occurs is to be interpreted as inconsistent. To this end, we can define a relation $\Downarrow$ on $\mathcal{WFT}$, called *bottom smashing*, where $t_1 \Downarrow t_2$ if and only if $\bot$ occurs in both t_1 and t_2, to be such that all equivalence classes except $[\bot]$ are *singletons.* Clearly, if $\bot$ occurs in a term, it also occurs in all terms in its α-class. Hence, by the way they have been defined, the relations α and $\Downarrow$ are such that their *union* $\simeq = \alpha \cup \Downarrow$ is an equivalence relation. Thus, we have the following definition.

4.9. Definition. A *ψ-type* is an element of the quotient set $\Psi = \mathcal{WFT}/\simeq$. An *acyclic* ψ-type is an element of the quotient set $\Psi_0 = \mathcal{WFAT}/\simeq$.

4.2. The subsumption ordering

The partial ordering on symbols can be extended to terms in a fashion which is reminiscent of the algebraic notion of *homomorphic extension.* We define the *subsumption* relation on the set Ψ as follows.

4.10. Definition. Let $t_1 = \langle \Delta_1, \psi_1, \tau_1 \rangle$ and $t_2 = \langle \Delta_2, \psi_2, \tau_2 \rangle$ be two wft's. We say that t_2 subsumes t_1, and we write $t_1 \leqslant t_2$, if and only if *either*, $t_1 \simeq \bot$ *or*

$$\Delta_2 \subseteq \Delta_1, \tag{1}$$

$$\mathbf{Ker}(\tau_2) \subseteq \mathbf{Ker}(\tau_1), \tag{2}$$

$$\forall w \in \mathcal{L}^*, \psi_1(w) \leqslant \psi_2(w). \tag{3}$$

It is easy to verify that a subsumption relation on Ψ is defined by $[t_1] \leqslant [t_2]$ if and only if $t_1 \leqslant t_2$ is well-defined (i.e., it does not depend on particular class

[7] See Appendix D.

representatives) and it is an *ordering* relation.[8] The reader is invited to verify the claim made in the example at the beginning of Section 4.

This notion of subsumption is related to the (in)famous IS-A ordering in semantic networks [8, 9]. It expresses the fact that, given a ψ-type t, any ψ-type t' defined on at least the same domain, with at least the same coreference classes, and with symbols at each address which are less than the symbols in t at the corresponding addresses, is a subtype of t. Indeed, such a t' is *more specified* than t.

The 'homomorphic' extension of the ordering on Σ to the subsumption ordering on Ψ can be exploited further. Indeed, if *least upper bounds* (lub) and *greatest lower bounds* (glb) are defined for finite nonempty subsets of Σ, then this property carries over to Ψ.

4.11. Theorem. *If the signature Σ is a lattice, then so is Ψ.*

Proof. We must define lubs and glbs of ψ-types. The easier of these is the join and is defined as $t_1 \sqcup t_2 = \langle \Delta, \psi, \tau \rangle$ such that

$$\Delta = \Delta_1 \cap \Delta_2, \tag{4}$$

$$\tau : \Delta \to \mathcal{T} \text{ is such that } \mathbf{Ker}(\tau) = \kappa_1 \cap \kappa_2, \tag{5}$$

$$\forall w \in \Delta,\ \psi(w) = \psi_1(w) \vee \psi_2(w). \tag{6}$$

It is clear that the intersection of Δ_1 and Δ_2 is itself a term domain, and the largest such that is contained in both. Now, recall that the intersection of the coreference relations κ_1 and κ_2 is also the greatest equivalence relation which is contained in both. That it is also right-invariant is obvious since, for all w_1 and w_2 in $\mathcal{L}^*$,

$$w_1(\kappa_1 \cap \kappa_2) w_2 \text{ iff } w_1 \kappa_1 w_2 \text{ and } w_1 \kappa_2 w_2$$

which implies, for all w in $\mathcal{L}^*$,

$$w_1.w \kappa_1 w_2.w \quad \text{and} \quad w_1.w \kappa_2 w_2.w,$$

which is equivalent to

$$w_1.w(\kappa_1 \cap \kappa_2) w_2.w.$$

Now, by (6), to all addresses in the symmetric difference of the two term domains is assigned the symbol $\top$; thus, the condition in Definition 4.3 requiring that $\psi(\mathcal{L}^* - \Delta) = \top$ is met.

As for ψ at an address w in Δ, (6) guarantees that the symbol $\psi(w)$ be the least in Σ which is greater than both symbols at this address in both terms. By Proposition 4.5, there are only finitely many such symbols, and since Σ is a lattice, they admit a lub.

Finally, (6) preserves referential consistency since each $(\kappa_1 \cap \kappa_2)$-class is assigned a consistent symbol, provided that is the case for each κ_1-class and κ_2-class. Therefore, conditions (4)–(6) do define a lub for two ψ-terms.

[8] This justifies that, in the sequel, we shall conventionally denote a ψ-type by one of its class representatives, understanding that what is meant is modulo *tag renaming* and *bottom smashing*.

Defining the meet operation needs a little more work. The union of two term domains being a term domain, it is safe to say that the term domain of the greatest lower bound of t_1 and t_2 must at least contain the union $\Delta = \Delta_1 \cup \Delta_2$. Also, by a similar argument, the coreference relation must contain at least the smallest equivalence relation on Δ containing both κ_1 and κ_2; namely, the relation

$$\kappa = \bigcup_{n \geq 0} (\kappa_1^{\Delta} \circ \kappa_2^{\Delta})^n,$$

where κ_i^{Δ} is the reflexive extension of κ_i from Δ_i to Δ, for $i = 1, 2$.

Recall again that taking the transitive closure of the composition of the extended relations is indeed the least equivalence on the union domain Δ whose restrictions to Δ_1 and Δ_2 contain κ_1 and κ_2. Yet, this is not quite sufficient since it is not guaranteed that this relation be right-invariant, as shown in the examples in Figs. 9 and 10.

Therefore, it is necessary to close κ so that it be a right-congruence. That is, Δ and κ must be completed by incrementally adding, to any κ-class C in the partition of Δ, any string $w_1.w$ such that there exists some w_2 in Δ such that $w_1 \kappa w_2$ and

$$\begin{aligned}
\Delta_1 &= \{\varepsilon, a, b, c, d\}\\
\kappa_1 &= \{\{\varepsilon\}, \{a, b\}, \{c, d\}\}\\
\Delta_2 &= \{\varepsilon, a, b, c, b.e, c.e\}\\
\kappa_2 &= \{\{\varepsilon\}, \{a\}, \{b, c\}, \{b.e, c.e\}\}\\
\Delta &= \Delta_1 \cup \Delta_2 = \{\varepsilon, a, b, c, d, b.e, c.e\}\\
\kappa_1^{\Delta} &= \{\{\varepsilon\}, \{a, b\}, \{c, d\}, \{b.e\}, \{c.e\}\}\\
\kappa_2^{\Delta} &= \{\{\varepsilon\}, \{a\}, \{b, c\}, \{b.e, c.e\}, \{d\}\}\\
\kappa &= (\kappa_1^{\Delta} \circ \kappa_2^{\Delta})^* = \{\{\varepsilon\}, \{a, b, c, d\}, \{b.e, c.e\}\}\\
\kappa^{[*]} &= \{\{\varepsilon\}, \{a, b, c, d\}, \{a.e, b.e, c.e, d.e\}\}\\
\Delta^{[*]} &= \{\varepsilon, a, b, c, d, a.e, b.e, c.e, d.e\}
\end{aligned}$$

Fig. 9. An example of acyclic right-invariant closure construction.

$$\begin{aligned}
\Delta_1 &= a(ba)^* + (ab)^*\\
\kappa_1 &= \{a(ba)^*, (ab)^*\}\\
\Delta_2 &= \{\varepsilon, a, c\}\\
\kappa_2 &= \{\{\varepsilon\}, \{a, c\}\}\\
\Delta &= \Delta_1 \cup \Delta_2 = a(ba)^* + (ab)^* + c\\
\kappa_1^{\Delta} &= \{a(ba)^*, (ab)^*, \{c\}\}\\
\kappa_2^{\Delta} &= \{\{\varepsilon\}, \{a, c\}\} \cup \bigcup_{u \in (ab)^+ + (ba)^+} \{u\}\\
\kappa &= (\kappa_1^{\Delta} \circ \kappa_2^{\Delta})^* = \{a(ba)^* + c, (ab)^*\}\\
\kappa^{[*]} &= \{(a+c)(ba)^*, ((a+c)b)^*\}\\
\Delta^{[*]} &= (a+c)(ba)^* + ((a+c)b)^*
\end{aligned}$$

Fig. 10. An example of cyclic right-invariant closure construction.

$w_2.w \in C$. Formally, this is achieved by constructing the following sequence of relations on $\mathscr{L}^*$: $\kappa^{[0]} = \kappa$, and for $n > 0$,

$$\kappa^{[n]} = \kappa^{[n-1]} \cup \{\langle u.w, v.w\rangle \in (\mathscr{L}^*)^2 \mid u\kappa^{[n-1]}v\}.$$

It is immediate to verify that by its very construction, the limit

$$\kappa^{[*]} = \bigcup_{n=0}^{\infty} \kappa^{[n]}$$

of this sequence is right-invariant, and the least such that contains κ. Now, taking

$$\Delta^{[*]} = \bigcup (\mathscr{L}^*/\kappa^{[*]})$$

we obtain the right term domain.

Finally, the symbols for each class of this closure are computed as the meets in Σ of the sets of symbols associated to all the addresses in the class by each term's ψ-functions. Again by Proposition 4.5, and because Σ is assumed to be a lattice, this is well-defined. This clearly preserves referential consistency, and guarantees the maximal consistent symbol for each class.

In summary, $t_1 \sqcap t_2 = \langle \Delta, \psi, \tau \rangle$ such that

$$\Delta = (\Delta_1 \cup \Delta_2)^{[*]}, \tag{7}$$

$$\tau : \Delta \to \mathscr{T} \text{ is such that } \mathbf{Ker}(\tau) = \kappa^{[*]}, \tag{8}$$

$$\forall [w] \in \Delta/\kappa^{[*]},\ \psi([w]) = \bigwedge \{\psi_i(u) \mid u \in [w],\ i = 1, 2\}. \tag{9}$$

It is thus established that conditions (7)–(9) define a glb, modulo smashing the term to $\perp$ anytime condition (9) would produce the symbol $\perp$. Therefore, $\sqcup$ is a *join* operation and $\sqcap$ is a *meet* operation with respect to the subsumption ordering defined in Definition 4.10. □

Let us illustrate these lattice operations $\sqcup$ and $\sqcap$ with an example. Figure 11 shows a signature which is a finite (non-modular) lattice. Given this signature, the two types in Fig. 12 admit as meet and join the types in Fig. 13, respectively.

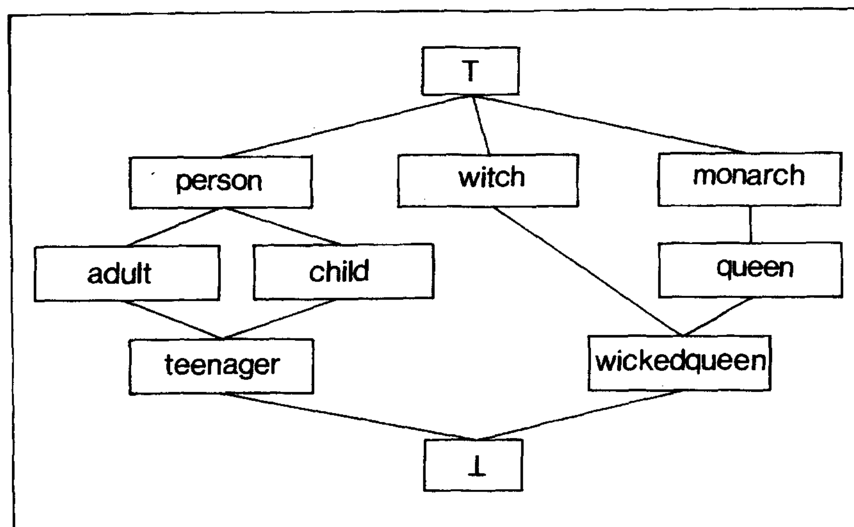

Fig. 11. A signature which is a lattice.

```
child(knows⇒X : person(knows⇒queen;
                              hates⇒Y : monarch);
      hates⇒child(knows⇒Y;
                  likes⇒wickedqueen);
      likes⇒X)

adult(knows⇒adult(knows⇒witch);
      hates⇒person(knows⇒X : monarch;
                   likes⇒X))
```

Fig. 12. Two wft's.

```
person(knows⇒person;
       hates⇒person(knows⇒monarch;
                    likes⇒monarch))

teenager(knows⇒X : adult(knows⇒wickedqueen;
                         hates⇒Y : wickedqueen);
         hates⇒child(knows⇒Y;
                     likes⇒Y);
         likes⇒X)
```

Fig. 13. Lub and glb of the two types in Fig. 12.

The meet and join operation on Ψ are essentially extensions of the *unification* [31, 57] and *generalization* [55] operations on regular first-order terms. Indeed, these operations are special cases of our definitions when (i) Σ is a *flat* lattice, (ii) a coreference class may contain more than one element iff all of its elements are leaves and (iii) the symbols occurring at these leaves are restricted to be $\top$. An efficient ψ-term unification algorithm is detailed in [2, 3].

An important remark is that the set Ψ_0 of acyclic ψ-types also has a lattice structure.

4.12. Theorem. *If Σ is a lattice, then so is Ψ_0. However, Ψ_0 is not a sublattice of Ψ.*

Proof. The proof is immediate and only sketched. The join operation Ψ_0 is the same as in Ψ, but the meet operation is modified so that if the glb in Ψ of two acyclic terms contains a cycle, then their glb in Ψ_0 is $\perp$.

It is thus clear that Ψ_0 is not a sublattice of Ψ since the meet in Ψ of two acyclic wft's is not necessarily acyclic. This is similar to the so-called '*occur-check*' in unification of first-order terms. Consider, for example,

$$t_1 = f(l_1 \Rightarrow X : f;\ l_2 \Rightarrow f(l_3 \Rightarrow X)), \qquad t_2 = f(l_1 \Rightarrow X : f;\ l_2 \Rightarrow X),$$

and so

$$t_1 \sqcap t_2 = f(l_1 \Rightarrow X : f(l_3 \Rightarrow X);\ l_2 \Rightarrow X). \qquad \square$$

4.3. *A distributive lattice of types*

Keeping in mind a 'type-as-set' interpretation of the calculus of ψ-types,[9] we must yet wonder whether lattice-theoretic properties of meet and join reflect those of intersection and union. Unfortunately, this is not the case with Ψ. Indeed, the lattice of ψ-types is not so convenient as to be *distributive*, even if the signature Σ is itself distributive.

For example, consider the flat (distributive) lattice $\Sigma = \{\top, a, f, \bot\}$. Clearly,

$$f \sqcap (f(l \Rightarrow a) \sqcup a) = f,$$

but

$$(f \sqcap f(l \Rightarrow a)) \sqcup (f \sqcap a) = f(l \Rightarrow a).$$

This shows that $\mathcal{WFT}$ is *not* distributive.[10]

This is not the only ailment of $\mathcal{WFT}$ as a type system. Recall that in order to obtain the benefit of a lattice structure as stated in Theorem 4.11, there is a rather strong demand that the type signature Σ be itself a lattice. For a signature that would be any poset, this nice result is unfortunately lost. Although in practice programs deal with finite sets of atomic types, it is quite unreasonable to require that all meets and joins of those atomic types be explicitly defined. What should be typically specified in a program is the minimal amount of type information which is to be relevant to the program. Clearly, such a signature of type symbols should be not necessarily more than a finite, incompletely specified poset of symbols.

It is hence necessary to go further than the construction of $\mathcal{WFT}$ in order to obtain a satisfactory type system which would not make unreasonable demand for atomic type information. Fortunately, it is possible not to impose so drastic demands on Σ, and yet construct a more powerful lattice than $\mathcal{WFT}$; i.e., a distributive lattice.

The idea is very simple, and is based on observing that the join operation in Ψ is too 'greedy'. Indeed, if one wants to specify that an object is of type 'foo' or 'bar' when no explicit type symbol in Σ is known as their lub, then this object is induced to be of type $\top$. Clearly, there is a loss of information in this process since it is not correct to infer that the given object is of the least informative type—'*anything*'—just because Σ does not happen to contain explicitly a symbol for the lub of 'foo' and 'bar'. All that can be correctly said is that the given object is of *disjunctive* type foo$\sqcup$bar.

We next define such a more adequate type lattice. It may be construed as a very simplistic powerdomain construction to handle indeterminacy [51]; in our case, *variant records*. All the formal details can be found in Appendix C.

Given a poset S, the set $\mathbf{2}^{(S)}$ of finite nonempty subsets of *maximal elements* of S is called the *restricted power of S*. If S is a Noetherian poset, the set $\mathbf{2}^{[S]}$ of *all* such subsets of maximal elements is called the *complete* restricted power of S. Given

[9] See Appendix D.

[10] A similar result pertaining to subsumption of first-order clauses was pointed out by Plotkin in [52].

a Noetherian poset S, and some subset S' of S, the set $[S']$ is the set of maximal elements of S'.

We shall call $\mathfrak{L}$ the set $\mathbf{2}^{[\Psi]}$, and $\mathfrak{L}_0$ the set $\mathbf{2}^{[\Psi_0]}$. Clearly, $\mathfrak{L}_0$ is a subset of $\mathfrak{L}$. We shall write a singleton $\{t\}$ in $\mathfrak{L}$ simply as t.

As explained in Appendix C, subsumption among elements of $\mathfrak{L}$ is defined as $T_1 \sqsubseteq T_2$ if and only if *every* ψ-type in T_1 is subsumed by *some* ψ-type in T_2.

Let us define a notational variant of elements of $\mathfrak{L}$ which has the convenience of being more *compact* syntactically. Consider the object shown in Fig. 14. The syntax used is similar to the one which has expressed ψ-types. However, *sets* of terms rather than terms may occur at some addresses.

```
person(sex⇒{male, female};
       father⇒Y:person(sex⇒male);
       mother⇒Z:person(sex⇒female);
       parent⇒{Y, Z})
```

Fig. 14. Example of an ϵ-term.

This notation may be viewed as a compact way of representing sets of ψ-types. For example, the object in Fig. 14 represents a set of *four* ψ-types which can be obtained by expansion, keeping *one* element at each address. Such terms are called ϵ-terms.

An ϵ-term can be transformed into a set of ψ-types—its *ψ-expansion*. The ψ-expansion of an ϵ-term is the set of all possible ψ-types which can be inductively obtained by keeping only one ψ-type at each address. The reader familiar with first-order logic could construe this process as being similar to transforming a logical formula into its disjunctive normal form. A detailed ψ-expansion algorithm is described in [4].

We are now ready to construct a distributive lattice of ϵ-types. First, we relax the demand that the signature Σ be a lattice. Assuming it is a Noetherian poset we can embed it into a lower semi-lattice $\mathbf{2}^{[\Sigma]}$ preserving existing glb's. Then, we can define the meet operation on Ψ so that whenever the meet of two symbols in not a singleton, the result is expanded using Ψ-expansion.

4.13. Theorem. *If the signature Σ is a Noetherian poset, then so is the lattice Ψ_0; but the lattice Ψ is not Noetherian.*

Proof. We must show that there is no infinitely ascending chain in Ψ_0. By definition of the subsumption ordering, this means that there must not be infinitely *descending* chains of ψ-term domains and coreference relations, as well as no infinitely *ascending* chains of symbols in Σ. The latter is assumed by hypothesis. As for the former two conditions, it is clear that they are true since term domains for wft's in Ψ_0 are finite sets of addresses.

However, this is no longer true if we consider the infinite regular domains of wft's in Ψ. The following counterexample exhibits an infinitely ascending chain of wft's in Ψ.[11]

For any a in $\mathscr{L}$ and any $f \neq \bot$ in Σ, define the sequence $t_n = \langle \Delta_n, \psi_n, \tau_n \rangle$ $(n > 0)$ as follows:

$$\Delta_n = a^*, \qquad \psi_n(\Delta_n) = f,$$

$$\Delta_n/\kappa_n = \Delta_n/\mathbf{Ker}(\tau_n) = \{\{\varepsilon\}, \{a\}, \ldots, \{a^{n-1}\}, a^n.a^*\}.$$

This clearly defines an infinite strictly ascending sequence of regular wft's since, for all $n > 0$,

$$\Delta_{n+1} \subseteq \Delta_n, \qquad \psi_n(\Delta_n) \leq \psi_{n+1}(\Delta_{n+1}), \qquad \kappa_{n+1} \subset \kappa_n.$$

In our syntax, this corresponds to the infinite sequence:

$$t_0 = X : f(a \Rightarrow X),$$

$$t_1 = f(a \Rightarrow X : f(a \Rightarrow X)),$$

$$t_2 = f(a \Rightarrow f(a \Rightarrow X : f(a \Rightarrow X))),$$

$$\vdots$$

$$t_n = f(a \Rightarrow f(a \Rightarrow \cdots f(a \Rightarrow X : f(a \Rightarrow X)) \cdots)) \quad (n+1\ a\text{'s}),$$

$$\vdots \quad \square$$

We define two binary operations $\sqcap$ and $\sqcup$ on the set $\mathfrak{L}_0$. For any two sets T_1 and T_2 elements of $\mathfrak{L}_0$:

$$T_1 \sqcap T_2 = \lceil \{t_1 \wedge t_2 \mid t_1 \in T_1, t_2 \in T_2\} \rceil, \qquad T_1 \sqcup T_2 = \lceil T_1 \cup T_2 \rceil, \tag{10}$$

where $\wedge$ is the meet operation defined on Ψ_0.

The following pair of theorems are corollaries to Theorem C.4 of Appendix C. Indeed, for *any* poset Σ containing $\top$ and $\bot$, we have the following theorem.

4.14. Theorem. *The poset $\mathfrak{L}_0$ of finitary ψ-types is a* distributive *lattice with meet $\sqcap$, with join $\sqcup$, and admits $\{\top\}$ as greatest element and $\{\bot\}$ as least element.*

It is not possible to define lattice operations (10) for $\mathfrak{L}$ because Ψ is not Noetherian. Indeed, the set of maximal elements of arbitrary poset elements in $\mathfrak{L}$ cannot be defined. However, if only *finite* sets of regular wft's are considered, then we have our next theorem.

[11] This counterexample is due to Dowling (private communication).

4.15. Theorem. *The poset* $\mathbf{2}^{(\Psi)}$ *of* finite *sets of incomparable regular wft's is a distributive lattice.*

However, $\mathbf{2}^{(\Psi)}$ is not complete. It is also true that the subset $\mathbf{2}^{(\Psi_0)}$ of $\mathbf{2}^{(\Psi)}$ is a distributive lattice, but it is not a sublattice of $\mathfrak{L}$. In general, the glb of elements of $\mathbf{2}^{(\Psi_0)}$ is a lower bound of the glb of these elements taken in $\mathbf{2}^{(\Psi)}$.

The last result of this section is a corollary to Theorem C.7, and is important for the next section.

4.16. Theorem. *If the signature* Σ *is a Noetherian poset, then the lattice* $\mathfrak{L}_0$ *of* all *sets of finitary wft's is a complete Brouwerian lattice.*

Let us answer a question that might be hovering in the reader's mind.[12] The fact that the lattice $\mathfrak{L}_0$ is a complete Brouwerian lattice will be needed for showing the existence of solutions of systems of simultaneous equations. Unfortunately, Theorem 4.16 does not hold for $\mathfrak{L}$, the lattice of all regular terms, since the lattice $\mathfrak{L}$ is not complete. Hence, the results described in the rest of this article are stated only for finitary wft's.

5. Solving equational type specifications

Consider the equations in Fig. 15. Each equation is a pair made of a symbol and an ϵ-term, and may for now intuitively be understood as a definition. We shall call a set of such definitions a *knowledge base.*[13]

5.1. Definition. A *knowledge base* is a function from Σ to $\mathfrak{L}_0$ which is the identity everywhere except for a finite subset of $\Sigma-\{\bot,\top\}$.

For reasons made practical later, we shall further define a particular form of knowledge base which we shall call *canonical.* Namely, we are interested in those knowledge bases which take symbols of Σ either into a non-atomic ψ-term (i.e., an element of $\Psi_0-\Sigma$), or into a non-singleton set of symbols of Σ. More formally, we have the following definition.

5.2. Definition. A knowledge base $\mathcal{KB}$ is *canonical* or in *standard form* if

$$\forall s_1\in\Sigma, \forall s_2\in\Sigma,\quad \mathcal{KB}(s_1)\subseteq\mathcal{KB}(s_2)\ \Rightarrow\ s_1=s_2$$

and, for all s in Σ, either $\mathcal{KB}(s)\in\Psi_0-\Sigma$, or $\mathcal{KB}(s)\in\mathbf{2}^{[\Sigma-\{s\}]}$.

[12] Namely, 'So what?...'.

[13] Or *type specification*, or *type schema*.... Nevertheless, 'knowledge base' is a deliberate choice since what is defined is in essence an abstract semantic network.

```
list   = {nil, cons};
append = {(front⇒nil;
           back⇒X : list;
           whole⇒X),
          (front⇒cons(head⇒X;
                      tail⇒Y);
           back⇒Z : list;
           whole⇒cons(head⇒X;
                      tail⇒U);
           patch⇒append(front⇒Y;
                        back⇒Z;
                        whole⇒U))}
```

Fig. 15. A specification for appending two lists.

In words, in a canonical knowledge base no two distinct right-hand sides may contain one another; and a right-hand side of an equation can either be a single ψ-term or a set of signature symbols.

The knowledge base in Fig. 15 is not canonical. Indeed, neither $\mathcal{KB}(\text{append}) \in \Psi_0$ nor $\mathcal{KB}(\text{append}) \in 2^{[\Sigma]}$.

However, the reader will verify that the knowledge base shown as in Fig. 16 is in standard form. Indeed, any knowledge base $\mathcal{KB}$ can be put in standard form as follows. For all $f \in \Sigma$,

(1) if $\mathcal{KB}(f)$ is not a singleton, then replace any ψ-type element t of $\mathcal{KB}(f)$ which is not a symbol in Σ by a new symbol s not already in Σ and augment the knowledge base with $\mathcal{KB}(s) = t$;

(2) if $\mathcal{KB}(f) \in \Sigma$, replace all occurrences of g in $\mathcal{KB}$ by f, and delete g from Σ;

(3) if $\mathcal{KB}(f) \subseteq \mathcal{KB}(g)$, for some $g \in \Sigma$, remove $\mathcal{KB}(f)$ from $\mathcal{KB}(g)$, replace it with a new symbol s not already in Σ and augment the knowledge base with $\mathcal{KB}(s) = \mathcal{KB}(f)$.

```
list    = {nil, cons};

append  = {append0, append1};

append0 = {front⇒nil;
           back⇒X : list;
           whole⇒X);

append1 = (front⇒cons(head⇒X;
                      tail⇒Y);
           back⇒Z : list;
           whole⇒cons(head⇒X;
                      tail⇒U);
           patch⇒append(front⇒Y;
                        back⇒Z;
                        whole⇒U))
```

Fig. 16. A canonical specification for appending two lists.

Hence, without loss of generality, we shall only consider knowledge bases in standard form. We shall now justify the need for restricting our attention only to canonical knowledge bases.

So far, the partial ordering on Σ has been assumed predefined. However, given a knowledge base such as the one in Fig. 16, it is quite easy to quickly infer a minimal consistent such ordering. For example, examining the knowledge base in Fig. 16, it is evident that this example's signature must be a superset of the set

$$\{\text{list}, \text{cons}, \text{nil}, \text{append}, \text{append0}, \text{append1}\}$$

and that the partial ordering on Σ must be such that

$$\text{nil} < \text{list}, \qquad \text{cons} < \text{list},$$

$$\text{append0} < \text{append}, \qquad \text{append1} < \text{append}.$$

We shall call such a minimal consistent strict ordering an *implicit symbol ordering*. Now, the reason for introducing a standard form is that this ordering can be automatically extracted from the specification of a *canonical* knowledge base as explained in the following definition.

5.3. Definition. Given a canonical knowledge base $\mathcal{KB}$ its *implicit symbol ordering* is the least strict ordering relation $<$ on Σ, if one exists, such that

(1) if $\mathcal{KB}(f) = g(l_1 \Rightarrow t_1; \ldots; l_n \Rightarrow t_n)$, then $f < g$;

(2) if $\mathcal{KB}(f) = \{f_1, \ldots, f_n\}$, then $f_i < f$ for all $i = 1, \ldots, n$.

Naturally, that such an ordering exists is subject to circularity checks. Thus, we use the following definition.

5.4. Definition. A (canonical) knowledge base is *well-defined* if and only if it admits an implicit symbol ordering.

The knowledge base in Fig. 16 is well-defined. It specifies four types. The next section describes how to use this type information computationally. Intuitively, an *interpretation* of an ϵ-type in the context of this knowledge base is obtained by 'rewriting' defined symbols in the given type according to the specifications.

5.1. A KBL interpreter

Given a well-defined knowledge base $\mathcal{KB}$, we define an interpreter for KBL by three rules of 'computation' of a functional $[\![\ ]\!]_{\mathcal{KB}}$ which maps $\mathfrak{L}_0$ into $\mathfrak{L}_0$. Interpreting ϵ-types is done by

$$[\![\{t_1, \ldots, t_n\}]\!]_{\mathcal{KB}} = \bigsqcup_{i=1}^{n} [\![t_i]\!]_{\mathcal{KB}} \tag{11}$$

and for ψ-types by

$$[\![f(l_1 \Rightarrow t_1; \ldots; l_n \Rightarrow t_n)]\!]_{\mathcal{KB}} = [\![\mathcal{KB}(f) \sqcap (l_1 \Rightarrow [\![t_1]\!]_{\mathcal{KB}}; \ldots; l_n \Rightarrow [\![t_n]\!]_{\mathcal{KB}})]\!]_{\mathcal{KB}}. \tag{12}$$

In other words, Rules (11) and (12) 'evaluate' ϵ-types as follows:

(1) a set of ψ-types is evaluated by evaluating all its elements and keeping only maximal elements;

(2) a ψ-type is evaluated by first evaluating its subterms, then by expanding its root symbol; i.e., substituting the root symbol by its knowledge base value by taking the meet of this value and the ψ-type whose root symbol has been erased (replaced by $\top$).

They define an operational semantics which reflects the 'type-as-set' semantics of ϵ-types and ψ-types described in Appendix D, in the sense that they compute unions and intersections of sets. The symbol substitution process is to be informally interpreted as importing the information encapsulated in the symbol into the context of another type.

Let us trace what this interpreter does, one step at a time, on an example. Consider the knowledge base in Fig. 16, and the following 'input':

```
append(front⇒cons(head⇒1;
                  tail⇒cons(head⇒2;
                            tail⇒nil));
       back⇒cons(head⇒3;
                 tail⇒nil)).
```

This is a ψ-term with a root symbol defined by $\mathcal{KB}$. Hence, applying Rule (12), the interpreter expands 'append' into the set {append0, append1}, to yield

```
{append0(front⇒cons(head⇒1;
                    tail⇒cons(head⇒2;
                              tail⇒nil));
         back⇒cons(head⇒3;
                   tail⇒nil)),
 append1(front⇒cons(head⇒1;
                    tail⇒cons(head⇒2;
                              tail⇒nil));
         back⇒cons(head⇒3;
                   tail⇒nil))}.
```

Using Rule (11), each of these two ϵ-terms is further expanded according to the definitions of their root symbols. Thus, the first one (append0) yields $\perp$ since the glb of the subterms at address 'front' is $\perp$. Hence, by the maximal restriction operation, we are left with only

```
(front⇒cons(head⇒1;
            tail⇒cons(head⇒2;
                      tail⇒nil));
 back⇒cons(head⇒3;
           tail⇒nil);
```

```
whole⇒cons(head⇒1;
           tail⇒U);
patch⇒append(front⇒cons(head⇒2;
                        tail⇒nil);
             back⇒cons(head⇒3;
                       tail⇒nil);
             whole⇒U)).
```

This expansion process continues, again using Rule (12) to expand the subterm at address 'patch'[14]

```
(front⇒cons(head⇒1;
            tail⇒cons(head⇒2;
                      tail⇒nil));
 back⇒cons(head⇒3;
           tail⇒nil);
 whole⇒cons(head⇒1;
            tail⇒cons(head⇒2;
                      tail⇒U));
 patch⇒(front⇒cons(head⇒2;
                   tail⇒nil);
        back⇒cons(head⇒3;
                  tail⇒nil);
        patch⇒append(front⇒nil;
                     back⇒cons(head⇒3;
                               tail⇒nil);
                     whole⇒U);
        whole⇒cons(head⇒2;
                   tail⇒U))).
```

Finally, the following term is obtained which cannot be further expanded. As one could intuitively expect, the interpretation of 'append' for the given input has thus produced a type whose 'whole' attribute is the result of a list concatenation of its 'front' to its 'back' list attributes. The attribute 'patch' represents the history of the computation.

```
(front⇒cons(head⇒1;
            tail⇒cons(head⇒2;
                      tail⇒nil));
 back⇒cons(head⇒3;
           tail⇒nil);
 whole⇒cons(head⇒1;
            tail⇒cons(head⇒2;
```

[14] We shall omit mentioning the details of cleaning-up ⊥ by maximal restriction.

```
                                    tail⇒cons(head⇒3;
                                              tail⇒nil)));
          patch⇒(front⇒cons(head⇒2;
                                tail⇒nil);
                 back⇒cons(head⇒3;
                                tail⇒nil);
                 patch⇒(front⇒nil;
                        back⇒cons(head⇒3;
                                       tail⇒nil);
                        whole⇒cons(head⇒3;
                                        tail⇒nil));
                 whole⇒cons(head⇒2;
                                 tail⇒cons(head⇒3;
                                                tail⇒nil)))).
```

At this stage, it is beneficial to put things into perspective. Following is a discussion on the nature of the model of computation of KBL and how it relates to term rewriting and nondeterministic recursive program schemes.

5.2. Graph rewriting

Computation in KBL amounts essentially to some sort of directed acyclic graph (DAG) rewriting. In fact, it bears much resemblance with computation with nondeterministic program schemes [15, 49], and macro-languages and tree grammars [30]. This section presents a formal characterization of computation in KBL along the lines of the algebraic semantics of tree grammars [6, 30]. Symbol rewriting presented in this section is very close to the notion of second-order substitution defined in [14] and macro-expansion defined in [30].

We show that a KBL program can be seen as a system of equations. Thanks to the lattice properties of finite wft's, we establish that such systems of equations admit a least fixed-point solution. The particular order of computation of KBL informally presented in the previous section is formally defined. We call it *fan-out computation order*, which rewrites symbols closer to the root first. This order of computation is also shown to be maximal, in the sense that it yields 'greater' ϵ-types than any other order of computation. The complete correctness of fan-out rewriting of KBL with respect to its least fixed-point semantics is also established. That is, we show that the fan-out normal form of a term is equal to the least fixed-point solution.

5.2.1. Wft substitution

We introduce and give some properties of the concept of wft substitution. Roughly, given a wft t such that a symbol sf occurs at address u in t, one can substitute some other wft t' for f at address u in t by 'pasting-in' t' in t at that address.

Given a wft $t = \langle \Delta, \psi, \tau \rangle$ and some string u in $\mathscr{L}^*$, we define the wft $u.t$ to be the least wft which contains t at address u. This can be better visualized as the wft

obtained by attaching the wft t at the end of the string u. That is, $u.t = \langle u.\Delta, u.\psi, u.\tau\rangle$, where

- $u.\Delta = \{w \in \mathcal{L}^* \mid w = u.v, v \in \Delta\}$;
- $u.\psi(w) = \begin{cases} \psi(v) & \text{if } w = u.v, \\ \top & \text{otherwise;} \end{cases}$
- $u.\tau : u.\Delta \to \mathcal{T}$ is such that $u.\tau(v) = u.\tau(w)$ iff $v = u.v'$, $w = u.w'$ and $\tau(v') = \tau(w')$.

Let u_i $(i = 1, \ldots, n)$ be mutually non-coreferring addresses in Δ and let f_i $(i = 1, \ldots, n)$ be symbols in Σ. Then, the wft $t[u_1 : f_1, \ldots, u_n : f_n]$ is the wft $\langle \Delta, \phi, \tau\rangle$, where ϕ coincides with ψ everywhere except for the coreference classes of the u_i's, where $\phi([u_i]) = f_i$ for $i = 1, \ldots, n$. It is clear that the term thus obtained is still well-formed.

5.5. Definition. Let $t = \langle \Delta, \psi, \tau\rangle$ be a wft and u some address in Δ, and let t' be some other wft. The term $t[t'/u]$ is defined as

$$t[t'/u] = t[u : \top] \sqcap u.t'.$$

This operation must not be confused with the classical tree *grafting* operation which *replaces* a subtree with another tree. The operation defined above *super-imposes* a term on a subterm with the exception of the root symbol of that subtree which becomes equal to the root of the replacing tree. In particular, note that $\perp$ may result out of such a substitution.

As an example of wft substitution, if t is the wft

```
(front⇒cons(head⇒X1:1;
            tail⇒X2:cons(head⇒2;
                         tail⇒nil));
 back⇒X3:cons(head⇒3;
              tail⇒nil);
 whole⇒cons(head⇒X1;
            tail⇒X4);
 patch⇒append(front⇒X2;
              back⇒X3;
              whole⇒X4))
```

and t' is the wft

```
(front⇒cons(head⇒X;
            tail⇒Y);
 back⇒Z;
 whole⇒cons(head⇒X;
            tail⇒U);
 patch⇒append(front⇒Y;
              back⇒Z;
              whole⇒U)),
```

then $t[t'/\text{patch}]$ is the wft

$$
\begin{array}{l}
(\text{front}\Rightarrow\text{cons}(\text{head}\Rightarrow X1:1;\\
\qquad\qquad\text{tail}\Rightarrow X2:\text{cons}(\text{head}\Rightarrow X5:2;\\
\qquad\qquad\qquad\qquad\text{tail}\Rightarrow X6:\text{nil}));\\
\ \text{back}\Rightarrow X3:\text{cons}(\text{head}\Rightarrow 3;\\
\qquad\qquad\text{tail}\Rightarrow\text{nil});\\
\ \text{whole}\Rightarrow\text{cons}(\text{head}\Rightarrow X1;\\
\qquad\qquad\text{tail}\Rightarrow X7:\text{cons}(\text{head}\Rightarrow X5;\\
\qquad\qquad\qquad\qquad\text{tail}\Rightarrow X4));\\
\ \text{patch}\Rightarrow(\text{front}\Rightarrow X2;\\
\qquad\text{back}\Rightarrow X3;\\
\qquad\text{patch}\Rightarrow\text{append}(\text{front}\Rightarrow X6;\\
\qquad\qquad\qquad\text{back}\Rightarrow X3;\\
\qquad\qquad\qquad\text{whole}\Rightarrow X4);\\
\qquad\text{whole}\Rightarrow X7)).
\end{array}
$$

Next, we give a series of 'surgical' lemmas about this substitution operation which will be needed in proving key properties of KBL's computation rule. In all the proofs of these lemmas, we shall omit considering the trivial cases where $\perp$ may result from substitutions since none of the stated lemmas will be affected by these situations.

The first lemma states the intuitively clear fact that which address is picked out of a coreference class in a substitution does not affect the result. This situation is made clearer when depicted as in Fig. 17.

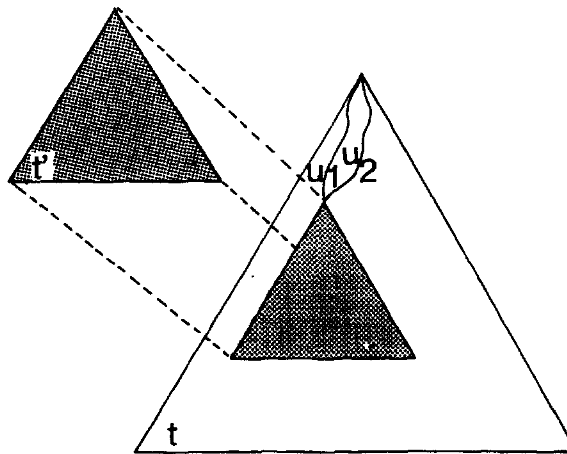

Fig. 17. Substitution at coreferring addresses.

5.6. Lemma. *Let $t=\langle\Delta, \psi, \tau\rangle$ and t' be wft's, and let u_1 and u_2 be two coreferring addresses in Δ. Then,*

$$(t[t'/u_1])[t'/u_2]=t[t'/u_1]=t[t'/u_2]=(t[t'/u_2])[t'/u_1].$$

Proof. Since u_1 and u_2 corefer in t, we can write

$$t[u_1:\top]=t[u_2:\top] \tag{13}$$

and thus, clearly,

$$t[u_1:\top]\sqcap u_1.t'=t[u_2:\top]\sqcap u_2.t'.$$

That is, $t[t'/u_1]=t[t'/u_2]$. Since u_1 and u_2 still corefer in this wft, the same steps as above yield

$$(t[t'/u_1])[t'/u_2]=(t[t'/u_2])[t'/u_1].$$

Now, by (13),

$$(t[t'/u_1])[u_2:\top]=(t[t'/u_1])[u_1:\top]$$

and therefore, by coreference of u_1 and u_2,

$$(t[t'/u_1])[u_2:\top]\sqcap u_2.t'=(t[t'/u_1])[u_1:\top]\sqcap u_1.t';$$

that is,

$$(t[t'/u_1])[t'/u_2]=(t[t'/u_1])[t'/u_1]=t[t'/u_1]. \quad \square$$

5.7. Definition. An address u *covers* an address v in a wft if there exists an address u' in $[u]$ such that $v=u'.w$ for some $w\neq\varepsilon$ in $\mathscr{L}^*$. That is, in other words, u covers v in t if $u\notin[v]$ and v occurs in $t\backslash u$.

Next, it is important to analyse the extent to which a sequence of substitution operations is affected by the particular order in which they are performed. Specifically, order of two substitutions will not matter if the addresses do not cover each other; however, order of substitutions will matter if one of the two addresses covers the other. We first need a small technical lemma.

5.8. Lemma. *If u and v are addresses in a wft t which do not cover each other, then, for any wft t',*

$$(t[u:\top]\sqcap u.t')[v:\top]=t[u:\top,v:\top]\sqcap u.t'.$$

Proof. Since the term t is acyclic and addresses u and v do not cover each other, they do not corefer in t. Moreover, it is clear that $v\notin u.\Delta'_{t'}$. Hence, in the least coreference merging, κ_{t_1} and κ_{t_2} addresses u and v still do not corefer (or a cycle would occur and the meet would be $\bot$). Therefore, changing the symbol at address v in t to $\top$ before or after taking the meet of $t[u:\top]$ and $u.t'$ yields the same result. $\square$

The next lemma gives a sufficient condition for commutativity of wft substitution.

5.9. Lemma. *Let $t=\langle\Delta,\psi,\tau\rangle$, t_1, and t_2 be wft's, and let u_1, u_2 be two addresses in Δ which do not cover each other. Then,*

$$(t[t_1/u_1])[t_2/u_2]=(t[t_2/u_2])[t_1/u_1].$$

Proof. By definition,

$$(t[t_1/y_1])[t_2/u_2] = (t[u_1:\top]\sqcap u_1.t_1)[u_2:\top]\sqcap u_2.t_2$$

which, by Lemma 5.8, is equal to

$$(t[u_1:\top,\ u_2:\top]\sqcap u_1.t_1)\sqcap u_2.t_2.$$

But clearly,

$$t[u_1:\top,\ u_2:\top] = t[u_2:\top,\ u_1:\top].$$

Hence, by associativity and commutativity of $\sqcap$, and using Lemma 5.8 again in the reverse direction, this must be equal to $(t[t_2/u_2])[t_1/u_1]$. □

The second lemma complements the previous one and shows that the order of substitution matters for covering addresses. However, the wft resulting from performing first the 'outermost' substitution subsumes the wft resulting from performing the 'innermost' substitution first.

5.10. Lemma. *If two addresses u_1 and u_2 in a wft t are such that u_1 covers u_2, then*

$$(t[t_2/u_2])[t_1/u_1] \leqslant (t[t_1/u_1])[t_2/u_2]$$

for any wft t_1 and t_2.

Proof. Because of associativity and commutativity of term domain union and coreference closure, it is clear that the order of performing the substitutions will not affect the resulting domains and coreferences. Therefore, the only things that may differ in the results are the symbols at addresses u_1 and u_2. The picture in Fig. 18 may help illustrate the argument.

For address u_1, performing the substitution at u_1 first leaves there the symbol $\psi_{t_1}(\varepsilon)$. Therefore, performing next the substitution at u_2 will not affect this symbol since u_1 covers u_2 and the acyclic condition prevents the address u_1 in t from being affected by some lower coreference. Following the same argument, the reverse order of substitutions yields first $\psi_t(u_1)$ at address u_1, then $\psi_{t_1}(\varepsilon)$. Hence, the eventual symbol at u_1 is unaffected by the order of performing the substitutions.

However, this is not the case for u_2. Indeed, substituting at u_1 first, then at u_2 eventually yields the symbol $\psi_{t_1}(\varepsilon)$ at u_2. On the other hand, if substituting at u_2 first yields a symbol f, this symbol may be further coerced down when substituting at u_1. □

The objective of these lemmas is to help show that the particular order of performing substitution performed by the KBL interpreter yields an ϵ-type that subsumes all ϵ-types obtained by any other order of computation. Next, the *fan-out computation* Theorem 5.13 will be proved to that effect, using the above technical lemmas.

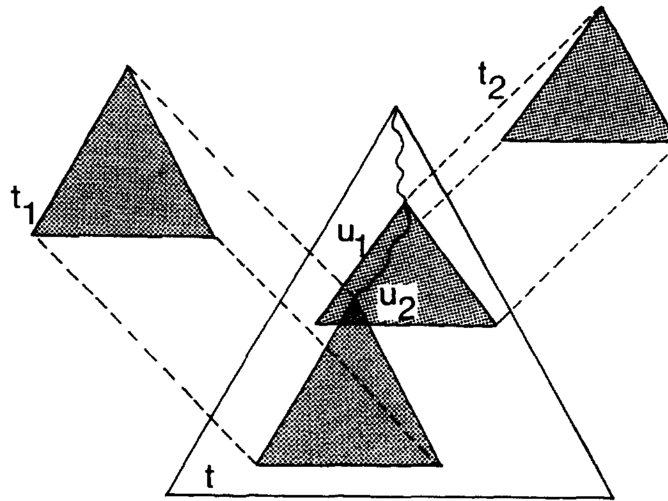

Fig. 18. Substitutions at covering addresses.

The following notion will be useful in expressing an ordering on the addresses of a wft. The *radius* of an address is a measure of how 'close to the root' an address is; that is, the shortest (in length) in its coreference class. Given a string u in $\mathcal{L}^*$, $|u|$ denotes its length; i.e., the number of labels which constitute u.

5.11. Definition. Let $t = \langle \Delta, \psi, \tau \rangle$ be a wft; then, the *radius* of an address u in Δ is defined as $\rho(u) = \min_{v \in [u]}(|v|)$. That such a minimum number exists for all classes is obvious.

Recall that Lemma 5.6 states that a substitution can be performed at any address in a coreference class with the same result. For this reason, it will be implicit in all substitutions henceforth considered that the address at which the substitution is performed is of minimal length in its class.

5.12. Definition. A sequence of addresses u_i, $i = 1, \ldots, n$ of a wft t is in *fan-out* order if and only if $i < j$ implies $\rho(u_i) \leqslant \rho(u_j)$.

For example, in the wft

$$
\begin{aligned}
t = f_1(l_1 &\Rightarrow X_1 : f_2(l_2 \Rightarrow X_2; \\
&\qquad\qquad l_3 \Rightarrow f_3); \\
l_4 &\Rightarrow X_2; \\
l_5 &\Rightarrow f_4(l_6 \Rightarrow X_1; \\
&\qquad\quad l_7 \Rightarrow X_3 : f_5; \\
&\qquad\quad l_8 \Rightarrow X_3))
\end{aligned}
\tag{14}
$$

the sequence ε, $l_5.l_6$, l_4, $l_5.l_7$, $l_1.l_3$ is in fan-out order. However, the sequence ε, $l_5.l_6$, l_4, $l_1.l_3$, l_5 is not.

We shall lighten our notation by writing $t[t_1/u_1][t_2/u_2]$ rather than $(t[t_1/u_1])$ $[t_2/u_2]$.

5.13. Theorem. *Let t be a wft, and $U=\{u_1,\ldots,u_n\}$ a set of mutually non-coreferring addresses of t such that the sequence $\{u_i\}_{i=1}^n$ is in fan-out order. Let π be a permutation of the set $\{1,\ldots,n\}$ such that $\{u_{\pi(i)}\}_{i=1}^n$ is also in fan-out order. Then, for any set of wft's $\{t_1,\ldots,t_n\}$,*

$$t[t_1/u_1]\ldots[t_n/u_n]=t[t_{\pi(1)}/u_{\pi(1)}]\ldots[t_{\pi(n)}/u_{\pi(n)}]. \tag{15}$$

Moreover, if the permutation π destroys fan-out order, then

$$t[t_1/u_1]\ldots[t_n/u_n]\geqslant t[t_{\pi(1)}/u_{\pi(1)}]\ldots[t_{\pi(n)}/u_{\pi(n)}]. \tag{16}$$

Proof. To prove (15), we observe that since the sequences $\{u_i\}$ and $\{u_{\pi(i)}\}$ are both fan-out permutations of U, each must be partitioned into a sequence of m subsequences U_j and $U_{\xi(j)}$, respectively, where $1\leqslant m\leqslant n$ and $j=1,\ldots,m$, where ξ is a permutation of the set $\{1,\ldots,m\}$, such that

- all addresses in each subsequence have equal radius; and,
- for every $j=1,\ldots,m$, subsequences U_j and $U_{\xi(j)}$ have same size.

Now, by the acyclicity condition and the fact that the u_i's are mutually non-coreferring, all addresses in a subsequence must be mutually non-covering. By Lemma 5.9, this means that the difference of order of substitutions between a pair of subsequences U_j and $U_{\xi(j)}$, for any fixed j, does not affect the result. Whence (15) follows.

Now, if π perturbs the fan-out order, pairs of addresses $u_{\pi(i)}$ and $u_{\pi(j)}$ $(1\leqslant i<j\leqslant n)$ in the sequence may be such that $u_{\pi(i)}$ covers $u_{\pi(j)}$. By Lemma 5.10, each of these pairs will contribute to 'decrease' the ultimate wft. Hence, fan-out order of substitution is one which yields the maximal wft. And this entails (16). □

Symbol substitution is extended to ϵ-types as follows: for any t in Ψ_0, any T in $\mathfrak{L}_0$, and any u in Δ_t,

$$t[T/u]=\bigsqcup_{t'\in T} t[t'/u].$$

5.2.2. Symbol-rewriting systems

5.14. Definition. A *symbol rewriting system* (SRS) on Σ is a system S of n equations $\mathrm{S}:\{s_i=E_i\}_{i=1}^n$, where $s_i\in\Sigma$ and $E_i\in\mathfrak{L}_0$, for $i=1,\ldots,n$.

Given such a system S, the subset $E=\{s_1,\ldots,s_n\}$ of Σ is called the set of S-*expandable* symbols. Its complement $N=\Sigma-E$ is called the set of *non*-S-*expandable* symbols. The same notion of canonical SRS is obviously definable as that of a knowledge base.

An example of an SRS is given in Fig. 16. In there, we have

$$E=\{\text{list, append, append0, append1}\},\qquad N=\{\text{nil, cons}\}.$$

Let $\mathbf{S}:\{s_i = E_i\}_{i=1}^{n}$ be an SRS. It defines a *one-step rewriting* relation $\rightarrow$ on $\mathfrak{L}_0$ as follows.

5.15. Definition. $T_1 \rightarrow T_2$ if and only if there exist a wft $t \in T_1$, some address u in Δ_t, and some index $i \in \{1, \ldots, n\}$ for which $\psi_t(u) = s_i$, such that

$$T_2 = (T_1 - \{t\}) \sqcup t[E_i/u].$$

In words, this expresses the fact that the ϵ-type T_2 is obtained from the ϵ-type T_1 by picking out some element of T_1, substituting for *one* of its occurrences of some expandable symbol the right-hand side of this symbol in S, and adjoin the result to the set, keeping only maximal elements. This process is illustrated by the first step of the trace of KBL shown in Section 5.1.

We shall denote by $\rightarrow^k$ $(k \geqslant 0)$ the relation $\rightarrow$ composed with itself k times, and by $\rightarrow^*$ the reflexive and transitive closure of $\rightarrow$; that is, the relation $\bigcup_{k=0}^{\infty} \rightarrow^k$.

The notation for the sets Ψ of ψ-types and $\mathfrak{L}$ of ϵ-types has hitherto been implicitly understood to depend on the signature of symbols Σ. Since it will now become necessary to make this dependency more explicit, we shall use the notation $\Psi[\Sigma]$ and $\mathfrak{L}[\Sigma]$.

5.16. Definition. Let S be an SRS, and T be an ϵ-type. The S-*normal form* of T is defined as

$$\mathcal{N}(T) = \bigsqcup \{T' \in \mathfrak{L}_0[N] \mid T \overset{*}{\rightarrow} T'\}.$$

That is, the lub of all terms containing no more expandable symbols which can be rewritten from T. Since $\mathfrak{L}_0$ is a complete lattice, this is well-defined.

Notice that a normal form is defined as a join of all possible rewritings of an ϵ-type. Thus, by Theorem 5.13, we can and will, without loss, restrict this definition to sequences of rewritings in fan-out order only.

To lighten notation, we shall use vector notation to denote elements of $\mathfrak{L}_0^n$ the set of n-tuples of ϵ-types; e.g., $\boldsymbol{T} = \langle T_1, \ldots, T_n \rangle$, where $T_i \in \mathfrak{L}_0$ $(i = 1, \ldots, n)$. Hence, a symbol rewriting system S of n equations is denoted by a single vector equation $\boldsymbol{s} = \boldsymbol{E}$. Given such an SRS, we shall use either indices in $\{1, \ldots, n\}$ or the symbols s_i to index the components of a vector $\boldsymbol{T}$ in $\mathfrak{L}_0^n$; i.e., $T_{s_i} = T_i$. There should be no confusion since the s_i's will be assumed distinct. Vector rewriting is the appropriate obvious extension to vectors of ϵ-types of the $\rightarrow$ relation, and so is the definition of vector normal form $\mathcal{N}(\boldsymbol{T})$.

Given an SRS $\mathbf{S}: \boldsymbol{s} = \boldsymbol{E}$ and a wft t, $\mathbf{X}(t, \boldsymbol{s})$ denotes the set of (minimum radius) addresses in t whose symbols are S-expandable. That is,

$$\mathbf{X}(t, \boldsymbol{s}) = \{u \in \Delta_t \mid \psi_t(u) = s_i, \text{ for some } i = 1, \ldots, n\}. \tag{17}$$

All indexing of $\mathbf{X}(t, \boldsymbol{s}) = \{u_1, \ldots, u_m\}$ will henceforth be implicitly assumed to be

fan-out indexings; that is, such that the sequence $\{u_1, \ldots, u_m\}$ is in fan-out order. For example, taking the wft t of (14) and $s = \langle f_2, f_4, f_5 \rangle$ we have $\mathbf{X}(t, s) = \{l_1, l_5, l_5.l_7\}$.

Our objective here is to define the operation of applying a fan-out sequence of substitutions of ϵ-types to a wft t at all expandable addresses of t. This operation is denoted $t[\boldsymbol{T}/\boldsymbol{s}]$ and defined as:

$$t[\boldsymbol{T}/\boldsymbol{s}] = t[T_{\psi_t(u_1)}/u_1] \ldots [T_{\psi_t(u_m)}/u_m], \tag{18}$$

where $\{u_1, \ldots, u_m\} = \mathbf{X}(t, s)$. By Theorem 5.13, it is evident that this is a well-defined operation. It will be important to keep in mind that (18) is essentially a finite meet of wft's. We shall condense notation of (18) to

$$t[\boldsymbol{T}/\boldsymbol{s}] = t[T_{\psi_t(u)}/u]_{u \in \mathbf{X}(t,s)}.$$

Let us illustrate this operation on a small example. We are interested in the expression of $t[\boldsymbol{T}/\boldsymbol{s}]$, taking $\boldsymbol{s} = \langle s_1, s_2 \rangle$ and $\boldsymbol{T} = \langle T_1, T_2 \rangle$ with

$$t = s_1(l_1 \Rightarrow s_2;\ l_3 \Rightarrow s_1),$$

$$T_1 = \{f(l_1 \Rightarrow X;\ l_2 \Rightarrow X), g\}, \qquad T_2 = s_1(l_2 \Rightarrow X;\ l_3 \Rightarrow X).$$

The set of expandable addresses for s in t thus is $\mathbf{X}(t, s) = \{\varepsilon, l_1, l_3\}$ corresponding to the sequence of symbols (in fan-out order) s_1, s_2, s_1. Hence, the sequence of substitutions starts with s_1 at ε:

$$\begin{aligned}&\{f(l_1 \Rightarrow X : s_2;\ l_2 \Rightarrow X;\ l_3 \Rightarrow s_1),\\ &\ g(l_1 \Rightarrow s_2;\ l_3 \Rightarrow s_1)\},\end{aligned}$$

then continues with s_2 at l_1:

$$\begin{aligned}&\{f(l_1 \Rightarrow X : s_1(l_2 \Rightarrow Y;\ l_3 \Rightarrow Y);\ l_2 \Rightarrow X;\ l_3 \Rightarrow s_1),\\ &\ g(l_1 \Rightarrow s_1(l_2 \Rightarrow X;\ l_3 \Rightarrow X);\ l_3 \Rightarrow s_1)\},\end{aligned}$$

and finally ends with s_1 at l_3:

$$\begin{aligned}&\{f(l_1 \Rightarrow X : s_1(l_2 \Rightarrow Y;\ l_3 \Rightarrow Y);\\ &\quad l_2 \Rightarrow X;\\ &\quad l_3 \Rightarrow f(l_1 \Rightarrow Y : l_1;\ l_2 \Rightarrow Y)),\\ &\ g(l_1 \Rightarrow s_1(l_2 \Rightarrow X;\ l_3 \Rightarrow X);\\ &\quad l_3 \Rightarrow s_1(l_2 \Rightarrow Y;\ l_3 \Rightarrow Y)),\\ &\ f(l_1 \Rightarrow X : s_1(l_2 \Rightarrow Y;\ l_3 \Rightarrow Y);\\ &\quad l_2 \Rightarrow X;\\ &\quad l_3 \Rightarrow g),\end{aligned}$$

$$g(l_1 \Rightarrow s_1(l_2 \Rightarrow X; l_3 \Rightarrow X);$$
$$l_3 \Rightarrow g))\}$$

which is the value of $t[T/s]$.

This operation is extended to $\mathfrak{L}_0^n$, i.e., to vectors of ϵ-types, as follows: $T[T'/s]$ is the vector of $\mathfrak{L}_0^n$ whose ith component is defined as

$$(T[T'/s])_i = \bigsqcup_{t \in T_i} t[T'/s]. \tag{19}$$

5.17. Definition. An element T of $\mathfrak{L}_0^n$ is a *solution* of the equation $s = E$ if and only if $E[T/s] = T$.

We now proceed to show that an SRS viewed as a system of equations in $\mathfrak{L}_0^n$ always has a solution which corresponds to the least fixed-point of a vector function from $\mathfrak{L}_0^n$ to itself. Such a function $\mathcal{F}$ is defined for an SRS $s = E$ as

$$\mathcal{F}(T) = E[T/s]. \tag{20}$$

5.18. Proposition. *The function $\mathcal{F}$ from $\mathfrak{L}_0^n$ to itself defined by* (20) *is continuous.*

Proof. From (19) and the definition of wft substitution, it is evident that any component of $\mathcal{F}(T)$ is a join of finite meets. By isotonicity of lattice operations, it is thus clear that $\mathcal{F}$ is monotone.

To prove that it is also continuous, we must show that $\mathcal{F}$ preserves lub's of infinite chains. Although notation makes it cumbersome to express, the argument is straightforward and follows by recalling the property of complete Brouwerian lattices stated as Theorem B.2 in Appendix B. Indeed, by definition,

$$\mathcal{F}_i\left(\bigsqcup_{k=0}^{\infty} T_k\right) = \bigsqcup_{e \in E_i} e\left[\left(\bigsqcup_{k=0}^{\infty} T_k\right)_{\psi_e(u)} \Big/ u\right]_{u \in \mathbf{X}(e,s)},$$

where $\mathcal{F}_i$ is the ith component function of $\mathcal{F}$. Thus, by definition of the product lattice operations in terms of the operations on component lattices,

$$\mathcal{F}_i\left(\bigsqcup_{k=0}^{\infty} T_k\right) = \bigsqcup_{e \in E_i} e\left[\bigsqcup_{k=0}^{\infty} ((T_k)_{\psi_e(u)})/u\right]_{u \in \mathbf{X}(e,s)}.$$

Now, by Theorem B.2 we know that in $\mathfrak{L}_0$ joins are completely distributive on meets. Hence,

$$\mathcal{F}_i\left(\bigsqcup_{k=0}^{\infty} T_k\right) = \bigsqcup_{k=0}^{\infty}\left(\bigsqcup_{e \in E_i} e[(T_k)_{\psi_e(u)}/u]_{u \in \mathbf{X}(e,s)}\right).$$

Therefore,

$$\mathcal{F}\left(\bigsqcup_{k=0}^{\infty} T_k\right) = \bigsqcup_{k=0}^{\infty} \mathcal{F}(T_k). \qquad \square$$

As a result, Tarski's Least Fixed-Point Theorem [63] guarantees that $\mathcal{F}$ has a least fixed point given by

$$\mathbf{Y}\mathcal{F} = \mathcal{F}^*(\bot) = \bigsqcup_{k=0}^{\infty} \mathcal{F}^k(\bot);$$

that is, since $\boldsymbol{E}[\mathbf{Y}\mathcal{F}/\boldsymbol{s}] = \mathbf{Y}\mathcal{F}$, we showed that $\mathbf{Y}\mathcal{F}$ is the least solution of the equation $\boldsymbol{s} = \boldsymbol{E}$.

Let us take again a small example to illustrate. Consider the single (non-canonical) equation:

tree = {leaf, node(left⇒tree; right⇒tree)},

where leaf < tree and node < tree. We thus have, $\mathcal{F}_{\text{tree}}(\bot) = \{\text{leaf}\}$, then $\mathcal{F}^2_{\text{tree}}(\bot)$ is given by

{leaf, node(left⇒leaf; right⇒leaf)}

and so $\mathcal{F}^3_{\text{tree}}(\bot)$ is

```
{leaf,
 node(left⇒leaf; right⇒leaf),
 node(left⇒leaf;
      right⇒node(left⇒leaf;
                 right⇒leaf)),
 node(left⇒node(left⇒leaf;
                right⇒leaf);
      right⇒leaf),
 node(left⇒node(left⇒leaf;
                right⇒leaf);
      right⇒node(left⇒leaf;
                 right⇒leaf))}
```

and so on.

The reader can now see that the successive powers of the 'tree' component function $\mathcal{F}$ generate all possible binary trees. Indeed, the *meaning* of the type 'tree' is precisely $\mathcal{F}^*_{\text{tree}}(\bot)$, the infinite set ($\epsilon$-type) of all such terms. Hence, effectively solving type equations gives a constructive meaning to recursively defined types.

The reader may wonder at this point how the example given in Section 5.1 (appending two lists) is related to computing a *vector* fixed point. Naturally, the explanation is that, given a knowledge base $\mathcal{KB}$ and an ϵ-type input E, we can add a new equation of the form $? = E$, where ? is a special *query* symbol not already in Σ. Then, the *answer* to the query is the component $(\mathbf{Y}\mathcal{F})_?$ of the solution of the augmented system.

5.2.3. Correctness

In order to establish that the fixed-point solution of an SRS does correspond to the value computed by KBL, it is necessary to establish the *correctness* of the KBL

interpreter. Namely, one must show that the normal form obtained by infinite rewritings is equal to the least solution of the system of equations.

We first need two technical lemmas; namely, Lemmas 5.19 and 5.20. These lemmas make intuitive sense and are extrapolations of similar facts for tree-grammars. Although we have developed proof sketches which reduce these lemmas further to be consequences of more elementary ones, we have not yet satisfied ourselves with the correctness of these proofs—which are rather complicated, and ought to be simpler.[15] Before stating these two lemmas, let us define some useful functions.

For any T in $\mathfrak{L}_0^n$ we define

$$\mathcal{G}(T) = T \sqcup \mathcal{F}(T) \quad \text{and} \quad \mathcal{G}^*(T) = \bigsqcup_{k=0}^{\infty} \mathcal{G}^k(T).$$

Then, for any T_1, T_2, and T_3 in $\mathfrak{L}_0^n$, we have the following two lemmas.

5.19. Lemma. *If* $T_1 \to^* T_2$, *then* $T_2[T_3/s] \sqsubseteq T_1[\mathcal{G}^*(T_3)/s]$.

5.20. Lemma. *If* $T_2 \sqsubseteq T_1[\mathcal{N}(s)/s]$, *then* $T_1 \to^* T_2$.

With these two lemmas we then necessarily have the following theorem.

5.21. Theorem. *The least fixed-point solution to an SRS* S *is identical to the* S*-normal form of the signature symbols; that is,*

$$\mathbf{Y}\mathcal{F} = \mathcal{N}(s).$$

Proof. We first establish $\mathcal{N}(s) \sqsubseteq \mathbf{Y}\mathcal{F}$. Let T in $\mathfrak{L}_0^n[N]$ such that $s \to^* T$. Using Lemma 5.19 with $T_1 = s$, $T_2 = T$, and $T_3 = \bot$ we have

$$T[\bot/s] \sqsubseteq s[\mathcal{G}^*(\bot)/s]. \tag{21}$$

But, since T is in $\mathfrak{L}_0^n[N]$, it has no expandable symbols. Hence, $T[\bot/s] = T$. Now, by monotonicity of $\mathcal{F}$ we know that $\mathcal{F}^k(\bot) \sqsubseteq \mathcal{F}^{k+1}(\bot)$. This readily entails $\mathcal{G}^*(\bot) = \mathcal{F}^*(\bot)$. With these remarks, together with the fact that $s[V/s] = V$ for any V, (21) becomes

$$T \sqsubseteq \mathcal{F}^*(\bot) = \mathbf{Y}\mathcal{F}.$$

Therefore, $\mathcal{N}(s) \sqsubseteq \mathbf{Y}\mathcal{F}$.

To prove the inverse inequality, it suffices to show that $\mathcal{N}(s)$ is a fixed point of $\mathcal{F}$. To that end, we first show that $\mathcal{F}(\mathcal{N}(s)) \sqsubseteq \mathcal{N}(s)$. Taking $T_2 = \mathcal{F}(\mathcal{N}(s))$ in Lemma 5.20, it comes that $E \to^* \mathcal{F}(\mathcal{N}(s))$ and hence, that $s \to^* \mathcal{F}(\mathcal{N}(s))$. Hence, $\mathcal{F}(\mathcal{N}(s)) \sqsubseteq \mathcal{N}(s)$.

[15] See [30, pages 28-29, Lemmas 2.38 and 2.39]. At the time of this writing, Irène Guessarian has communicated us outlines of proofs for them that we cannot fully appreciate at this point for lack of details—and time [29].

For the other direction, we use Lemma 5.19 with $T_1 = E$, and $T_2 = T_3 = \mathcal{N}(s)$. It thus comes

$$\mathcal{N}(s)[\mathcal{N}(s)/s] \sqsubseteq E[\mathcal{G}^*(\mathcal{N}(s))/s];$$

that is, since by definition a normal form does not have expandable symbols,

$$\mathcal{N}(s) \sqsubseteq E[\mathcal{G}^*(\mathcal{N}(s))/s]. \tag{22}$$

Now, since we first established that $\mathcal{F}(\mathcal{N}(s)) \sqsubseteq \mathcal{N}(s)$, it is clear that $\mathcal{G}(\mathcal{N}(s)) \sqsubseteq \mathcal{N}(s)$. And therefore, that $\mathcal{G}^*(\mathcal{N}(s)) \sqsubseteq \mathcal{N}(s)$. Combined with (22), this implies that $\mathcal{N}(s) \sqsubseteq \mathcal{F}(\mathcal{N}(s))$. □

6. Extension of research

6.1. Negative information

There are two possible ways of integrating negative information in wft's. The first one captures exclusion rather than equality among subterms of a wft. The second considers relative complementation among wft's.

6.1.1. Capturing inequalities

The tags in a wft essentially define a set of equalities among subterms of the wft. Thus, computing the meet of two wft's computes a right-congruence closure to propagate these equalities by merging coreference classes. Instead of tags, we could (less concisely) express coreferences in a tag-less wft by a finite set (conjunction) of equations of the form $u_i = v_i$, $i = 1, \ldots, n$, where the u_i's and the v_i's are addresses in the wft. One could then ask whether it is possible to add inequations in such a conjunction of address equations, in order, for example, to capture mutual exclusion constraints among subterms of a wft.

Clearly, the answer is 'yes'. Two very interesting articles by Oppen [50], and Nelson and Oppen [48] discuss fast decision procedures based on congruence closure on graphs to prove the unsatisfiability of a finite quantifier-free conjunction of equations and inequations among first-order terms. The method first computes the congruence classes corresponding to the equations only. Then, it simply checks whether any pair of terms supposed not to be equal as specified by the inequations are not put in the same congruence class. If at least one such pair is found, then the given conjunction is unsatisfiable. The method is inspired by another paper by Downey, Sethi and Tarjan [19] which applies the UNION/FIND technique to problems involving congruence closure.

The same procedure can be adapted in the case of wft's. Hence, mutual exclusion among subterms of a wft can be accommodated. In fact, the only tricky part would be to think up a syntax to capture mutual exclusion among addresses in a nice and compact way as do tags for equality among addresses.

6.1.2. Complemented types

The second possibility that we outline deals with capturing some negative information. It turns out that (relative) type complementation is feasible. More precisely, we can allow both positive and negative types to appear under labels of ϵ-terms in the form of pairs $t_1 \backslash t_2$. Intuitively, this specifies anything which is subsumed by t_1 but not by t_2. For example, a relation specifying that John owns a pet which is not a cat might be

$$\text{owns} = (\text{owner} \Rightarrow \text{john}; \text{pet} \Rightarrow \text{animal} \backslash \text{cat}).$$

A type interpretation ι is readily extended as follows:

$$\iota[\![t_1 \backslash t_2]\!] = \iota[\![t_1]\!] - \iota[\![t_2]\!].$$

Considering the set $\mathfrak{L} \times \mathfrak{L}$, what we need to express is that $t_1 \backslash t_2$ has no meaning if $t_1 \sqsubseteq t_2$. Thus, we may define a *smash* function $\downarrow$ from $\mathfrak{L} \times \mathfrak{L}$ to itself:

$$\downarrow(t_1 \backslash t_2) = \begin{cases} \bot & \text{if} \quad t_1 \sqsubseteq t_2, \\ t_1 \backslash (t_1 \sqcap t_2) & \text{otherwise.} \end{cases}$$

That is $\downarrow$ is a retract of $\mathfrak{L} \times \mathfrak{L}$.

Next, we can define the following binary operation $\wedge$ on $\downarrow(\mathfrak{L} \times \mathfrak{L})$:

$$t_1 \backslash t_2 \wedge t_3 \backslash t_4 = \downarrow((t_1 \sqcap t_2) \backslash (t_2 \sqcup t_4)). \tag{23}$$

It is straightforward to check that the following proposition holds.

6.1. Proposition. *The operation defined in* (23) *is associative, commutative, and idempotent.*

Hence, we find the following proposition.

6.2. Proposition. *The operation* $\wedge$ *defined in* (23) *defines a lower semi-lattice structure on* $\mathfrak{L} \times \mathfrak{L}$ *for the ordering defined by*

$$t_1 \backslash t_2 \leqslant t_3 \backslash t_4 \ \textit{iff} \ (t_1 \backslash t_2) \wedge (t_3 \backslash t_4) = t_1 \backslash t_2.$$

We can then proceed to the construction of Theorem C.4, and then justify the definition of the $\sqcap$ and $\sqcup$ operations on the set of sets of pairs of ϵ-terms in exactly the same fashion. The lattice thus constructed is then distributive, and complete and Brouwerian if limited to finite wft's on a Noetherian signature.

The point made here is that one can use the exact semantic scheme detailed in Section 5.1 to interpret the language of complemented types outlined above.

6.2. Polymorphic types

We can show that KBL could easily be extended to provide for the possibility of defining *parameterized* or *polymorphic types* [45].

Let us give an example. If we want to specify a type representing a list of integers, the following will work:

$$\text{integer-list} = \{\text{nil}, \text{cons}(\text{head} \Rightarrow \text{integer}; \text{tail} \Rightarrow \text{integer-list})\}.$$

However, if the intent is to define, in a generic way, a *homogeneous* list type for any base type and not only for *integer*, KBL—as defined—cannot express this.

Again, looking at what is needed provides a straightforward solution. Indeed, the original terms in $\mathscr{WFT}$ are built on a signature which is a poset of symbols. This is all which has been needed to construct the lattice of ϵ-types. Therefore any poset shall do as well. A poset which does better is the set of first-order algebraic terms on a given partially ordered signature of symbols Σ and a set of type variables $\mathscr{V}$ (pre-)ordered by first-order instantiation as usual. With this poset rather than Σ, KBL gains universal type polymorphism.

If type variables are denoted by small Greek letters, the definition of a (universal) polymorphic homogeneous list type may thus be:

$$\text{list}[\alpha] = \{\text{nil}, \text{cons}(\text{head} \Rightarrow \alpha, \text{tail} \Rightarrow \text{list}[\alpha])\}.$$

Other potential such extensions may be to second-order (respectively ω-order) terms [14] (respectively [31]) ordered by second-order (respectively ω-order) term instantiation.

It is not difficult to extend the syntax to accommodate these features.

6.3. *Further research*

In addition to presenting some definite results, the work described in this article opens a fecund vein for further research. One could think of many ideas to explore along the lines of the approach started here, ranging from the very theoretical to the very practical. We propose some specific questions that could lead to potential improvements on what has been discussed here.

6.3.1. *Psi-expansion considered unnecessary*

Although the lattice construction of ϵ-types from ψ-types leads to a semantically satisfactory type system, it makes an implementation of a type system rather inefficient because of the cost of ψ-expansion.

While playing with a KBL prototype implementation, we have found that many programs are unnecessarily plagued by combinatorial explosion. Indeed, systematically expanding all ϵ-types leads obviously to exponential growth of computation. We think that, most often, not expanding subterms which are sets would give the interpreter much efficiency. The idea is simply that since the interpreter works by pruning out non-maximal elements of a set of types, most types could be eliminated in block in one step of computation before being expanded, rather than individually after being expanded.

Of course, the catch is that one must then deal with computing meets on graph structures which look more like nondeterministic automata than deterministic ones; that is, on hypergraphs rather than graphs. The idea is, if nontrivial, not impossible.

However, the use of the UNION/FIND method must be altered since in a hypergraph, a node has a set of nodes as successor. Hence, since a coreference class becomes a set of sets, the FIND of a class representative can no longer be as efficient, because equality of sets is up to a permutation. Nevertheless, the problem is a challenging one, and must be thoroughly explored.

6.3.2. Epsilon-types as a type system for a programming language

The example of KBL as a programming language where types are 'first-class citizens' is perhaps an interesting theoretical exercise, but an extreme one! Indeed, the calculus of types presented in Section 4 is independent of the particular use made of it for solving type equations. In fact, it could complement advantageously any programming language as a typing facility. However, one must be warned that this may not be done carelessly. One must make sure that, in providing a programming language like PROLOG (say) with a type system like the one described here, the distribution of information between types and procedures (predicates, in the case of PROLOG), does not lead to redundancy, or worse, inconsistency. That is to say, one must be thoroughly aware of legal ways of combining the two systems of information. In fact, Scott's semantic theory of information systems [60] provides excellent tools to construct new information systems from old ones. A formalization of the types presented here as an information system will give the key to investigate their potential as a practical type system. Such works as [11], in the context of relational database theory, and [2, 3], in logic programming, are interesting upshots of this approach. In the latter, a small restriction is imposed on the nature of the type calculus which makes it an intermediary calculus between Ψ and $\mathfrak{L}$. Recursive type equations are partially solved at compile-time by fan-out rewriting limited to the maximal length of recursive cycles, and completely for finite branches.

6.3.3. Higher-order types

The types considered here have been strictly of first-order kind. That is, function types have not been considered. It is however interesting to wonder what could be obtained from lifting this restriction. There are really two ways of introducing higher-order beings in the ϵ-type calculus.

Functional types—The first way is the more natural. It consists of allowing functional types—i.e., denoting sets of functions, since we are following a type-as-set semantics. That is, a function type $t_1 \rightarrow t_2$ denotes the set of all (computable) functions from type t_1 to type t_2.

However, a strangely non-intuitive quirk in the partial ordering of types happens for these types which was discovered by MacQueen, Sethi, and Plotkin [39]. Namely, if $\mathbf{T}$ is the type lattice, the ordering on $[\mathbf{T} \rightarrow \mathbf{T}]$ is not the product ordering, but the ordering of the product of the dual lattice of $\mathbf{T}$ with $\mathbf{T}$. The ordering is so-called *anti-monotonic*—or *contravariant*—in the first component. Hence, if higher-order functional types are considered, an odd 'flip-flop' of orderings must be taken into account. MacQueen, Sethi, and Plotkin have shown that recursive type equations

have solutions for these types. But the technique they propose to prove this uses an involved argument about a contracting mapping in an appropriately defined metric space. The reason is that one may no longer count on a least fixed-point argument since the solution functional for the type equation is not monotone! Again, this is a challenge to take in the context of our type system, perhaps at the expense of forbidding such problematic recursion in type equations as $s = s \to t$; that is, in effect, stratifying type orders.

Attribute variables—The other way to introduce higher-order information is a more operational one. Namely, since attributes are interpreted as functions, one may wonder what happens if ***variable attributes*** are allowed in the syntax of ψ-types. That is to say that a variable label would match any address in a meet computation. The idea is reminiscent of higher-order unification in the lambda-calculus [31]. We suspect that this may lead to undecidable problems; but again it is worth exploring its possibilities.

6.3.4. Integrating logic

As stated earlier, the kind of information to which this work has been restricted, is essentially a logic of equality. It is interesting to ask whether other relations than equality could be dealt with in a similar way. Indeed, it seems possible to augment the expressive power of ψ-types by allowing logical formulae to be attached to a ψ-type specifying constraints in a richer logic than one limited to (in)equality. It would be a propositional logic whose sentential variables would be term addresses. By the principle of coreference, all predicates would then have the peculiar property to be 'right-invariant' in all their arguments. The kind of efficient decision procedure devised by Dowling and Gallier [18] would then be a great candidate to extend the lattice operations.

7. Conclusion

In this article, we have proposed a novel approach to an old problem. Namely, we have developed a formal calculus of record-like type structures, analysed the order- and lattice-theoretic consequences of interpreting these type structures as sets of objects, and offered a mathematical construction which allows this interpretation to be operationally meaningful. A particular language of types has been described which interprets programs in the form of a system of simultaneous equations. A fixed-point semantics has been studied and used to justify the existence of least solutions to these equations. Correctness of the operational semantics of the interpreter with respect to the fixed-point semantics has been discussed. Finally, extensions of the type calculus have been proposed.

Our work is by no means complete. It is intended as an illustration of a certain basis on which to expect variations. It is hoped that this research be just a beginning . . .

Appendix A. Lattice ideals

First, we recall the concept of lattice *ideal*.

A.1. Definition. An *ideal* of an upper semi-lattice L, $\leqslant$, $\vee$ is a nonempty subset $\mathscr{I}$ of L such that

(1) $\forall a \in \mathscr{I}$, $\forall x \in L$, if $x \leqslant a$, then $x \in \mathscr{I}$;

(2) $\forall a \in \mathscr{I}$, $\forall b \in \mathscr{I}$, $a \vee b \in \mathscr{I}$.

Informally, one may say that a nonempty subset of an upper semi-lattice is an ideal if it is '*downward-closed*' and '*join-closed*'.

Given an element a of an upper semi-lattice L the set $\underline{a} = \{x \in L \mid x \leqslant a\}$ is clearly an ideal of L. It is called the *principal ideal* generated by a. The next theorem states an important standard result.

A.2. Theorem. *The set $\mathfrak{I}(L)$ of all ideals of any lattice L, ordered by set inclusion, is a complete lattice. Furthermore, the set of all principal ideals of L is a sublattice of $\mathfrak{I}(L)$ which is isomorphic with L.*

Theorem A.2 justifies the embedding of any lattice into the complete lattice of all its ideals. This construction is commonly known as *completion by ideals*. Thus, we have the following corollary.

A.3. Corollary. *Any lattice can be embedded as a sublattice into a complete lattice.*

This construction preserves glb's but it does not preserve lub's (it is not *self-dual*). However, it preserves modularity and distributivity.[16]

The next easy result will be useful in later proofs.

A.4. Proposition. *Every ideal of a Noetherian lattice is principal.*

Proof. The proof is inductive in nature. Given an ideal $\mathscr{I}$ of a Noetherian lattice L, we construct a sequence of elements of $\mathscr{I}$ as follows. We start with some arbitrary element a_0 in $\mathscr{I}$. If a_0 is the greatest element of $\mathscr{I}$, then $\mathscr{I} = \underline{a_0}$ is principal. Otherwise, there must be a b_0 in $\mathscr{I}$ such that a_0 and b_0 are not comparable. We define $a_1 = a_0 \vee b_0$. Necessarily, $a_0 < a_1$. We repeat the same construction that we did for a_0 on a_1, and so on. Clearly, this sequence cannot continue for ever since there are no infinitely ascending chains in L. Therefore, there must be an element a_n ($n \geqslant 0$) in the sequence such that $\mathscr{I} = \underline{a_n}$. $\square$

[16] The completion by ideals is not the only possible completion. Another well-known construction, *the completion by cuts*, is self-dual: any poset can be plunged into a complete lattice such that both existing glb's and existing lub's are preserved. However, modularity (and hence distributivity) is not kept. We shall not detail this latter construction, and we refer the reader to [7].

Therefore, we have the following corollary.

A.5. Corollary. *Any Noetherian lattice is complete.*

Next, we recall a not so well-known kind of lattice.

Appendix B. Brouwerian lattices

B.1. Definition. A *Brouwerian lattice* L is a lattice such that, for any given elements a and b, the set $\{x \in L \mid a \wedge x \leq b\}$ contains a greatest element.

An interesting property of Brouwerian lattices is that (i) any Brouwerian lattice is distributive but, not conversely; and (ii) any Boolean lattice is Brouwerian, but not conversely. Thus, the class of Brouwerian lattices lies strictly between the class of distributive lattices and the class of Boolean lattices. Apart from its lattice-theoretic properties, a Brouwerian lattice is interesting as it forms the basis of an intuitionistic propositional logic, due to Brouwer [10, 20].[17]

We finally state a theorem which is a key property of Brouwerian lattices.

B.2. Theorem. *A complete lattice is Brouwerian if and only if the join operation is completely distributive on the meets*; *that is, for all $x \in L$ and all $Y \subseteq L$,*

$$x \wedge \bigvee Y = \bigvee_{y \in Y} (x \wedge y).$$

The following theorem is due to Stone.

B.3. Theorem. *The ideals of a distributive lattice form a complete Brouwerian lattice.*

Appendix C. Powerlattice constructions

In this section, we describe two constructions which are extensions of posets preserving the ordering and existing glb's. In fact, they are 'hidden' constructions by ideals for Noetherian posets. These constructions are universal; i.e., they are independent of the particular instances of sets on which one may eventually use

[17] The connection between intuitionistic logic and lattice theory is due to Birkhoff [7].

them. Although quite simple, these constructions are not so common, and this is why we chose to give them in detail.

C.1. A semi-lattice construction

A poset is *Noetherian* if it does not contain infinitely ascending chains.

The *restricted power* of a poset P, noted $2^{(P)}$, is the set of nonempty finite subsets of pairwise *incomparable* elements of P.

The *complete restricted power* $2^{[P]}$ of P is the set of *all* nonempty subsets of incomparable elements of P. It is clear that $2^{(P)} \subseteq 2^{[P]}$.

Such subsets are also called *cochains*, or *crowns*, and are partially ordered by the relation $\sqsubseteq$ defined as follows:

$$X \sqsubseteq Y \text{ iff } \forall x \in X, \exists y \in Y, x \leqslant y.$$

We must verify that the following proposition holds.

C.1. Proposition. *The relation $\sqsubseteq$ defines a partial ordering on $2^{[P]}$.*

Proof. Reflexivity and transitivity are straightforward.

As for antisymmetry, let X and Y in $2^{[P]}$ be such that $X \sqsubseteq Y$, and $Y \sqsubseteq X$. By definition of $X \sqsubseteq Y$, for any x in X, there is an element y of Y such that $x \leqslant y$. But since $Y \sqsubseteq X$, for that element y, there is a z in X such that $y \leqslant z$. Hence, by transitivity of $\leqslant$, $x \leqslant z$ in X. However, X is a cochain. Thus, all elements of X are incomparable. Therefore, we must have $x = z$, and so the intermediate element y is such that $x = y$. As a result, x is an element of Y.

We have proved that $X \subseteq Y$; a symmetric argument entails $Y \subseteq X$. Therefore, $X = Y$. □

Let P, $\leqslant$ be a poset. The *canonical injection* of P into $2^{(P)}$ is the function which takes any element x of P into the singleton $\{x\}$. It is clear that this is an injection. Moreover, it is an order homomorphism since

$$\forall x \in P, \forall y \in P, \quad \{x\} \sqsubseteq \{y\} \text{ iff } x \leqslant y. \tag{24}$$

Let P be a Noetherian poset. Given any subset X of P, we define its *maximal restriction* $\lceil X \rceil$ as the set of maximal elements of X. Since P has no infinitely ascending chains, this is well defined, even for infinite subsets.

The following construction shows how one can embed any Noetherian poset P into its complete restricted power $2^{[P]}$ and obtain a lower semi-lattice. The construction is quite simple and is based on defining the right meet operation on $2^{[P]}$ which will preserve existing glb's in P.

Given some element x of P, we note $\underset{\sim}{x}$ the subset of P of all lower bounds of x. That is, $\underset{\sim}{x} = \{y \in S \mid y \leqslant x\}$. Then, for any two elements a, b in P, $\underset{\sim}{a} \cap \underset{\sim}{b}$ is the set of common lower bounds of a and b in P.

The following $\sqcap$ operation can be defined for any pair of subsets X, Y in $2^{[P]}$:

$$X \sqcap Y = \left\lceil \bigcup_{a \in X}^{b \in Y} \underline{a} \cap \underline{b} \right\rceil. \tag{25}$$

Informally, we may describe what is performed by the operation of (25) as 'skimming the cream off the crown' of the set of all common lower bounds of all pairs of elements.

C.2. Theorem. *Let P, $\leqslant$ be a poset. Then, $2^{[P]}$, $\sqsubseteq$, $\sqcap$, as defined above, is a lower semi-lattice.*

Proof. Let z be in $X \sqcap Y$ as defined by (25), for some X and Y in $2^{[P]}$. By definition, z is a common lower bound of a pair of elements x_0 in X, and y_0 in Y. Therefore, $X \sqcap Y \sqsubseteq X$ and $X \sqcap Y \sqsubseteq Y$. Now, consider some Z in $2^{[P]}$ such that $Z \sqsubseteq X$ and $Z \sqsubseteq Y$. Then, for any z in Z, there must be some $x_0 \in X$ and some $y_0 \in Y$ such that $z \leqslant x_0$ and $z \leqslant y_0$. Hence, z is a common lower bound for this pair $\langle x_0, y_0 \rangle$, and must be either in $X \sqcap Y$, or less than or equal to some element of $X \sqcap Y$ which is maximal in $\bigcup_{a \in X}^{b \in Y} \underline{a} \cap \underline{b}$. In any case, $Z \sqsubseteq X \sqcap Y$. $\square$

Note that if two elements x and y in P already have a unique glb z in P, it follows that $\{x\} \sqcap \{y\} = \{z\}$. Hence, this construction is a structure embedding, since it preserves the ordering and the glbs when they exist in P.

Now, we are justified to take the freedom of writing simply x rather than $\{x\}$ for any single element of a poset P, and extend P to $2^{[P]}$, the glb-preserving lower semi-lattice extension of P. And this is the 'least' such possible structure, because if P is already a lower semi-lattice, then it is isomorphic to its canonical injection into $2^{[P]}$. Therefore, we have the following corollary.

C.3. Corollary. *Any Noetherian poset can be embedded into a lower semi-lattice such that existing glb's are preserved.*

C.2. A distributive lattice construction

The second construction that we describe shows how to take a Noetherian lower semi-lattice into a distributive lattice.

Let M, $\leqslant$, $\wedge$ be a lower semi-lattice. We define two binary operations $\sqcap$ and $\sqcup$ on $2^{[M]}$ as follows:

$$X \sqcap Y = \lceil \{x \wedge y \mid x \in X, y \in Y\} \rceil, \qquad X \sqcup Y = \lceil X \cup Y \rceil. \tag{26}$$

Let M, $\leqslant$, $\wedge$ be an arbitrary Noetherian lower semi-lattice. Then, we have the following theorem.

C.4. Theorem. *The structure $2^{[M]}$, $\sqsubseteq$, $\sqcap$, $\sqcup$ as defined above forms a distributive lattice.*

Proof. We first verify that $X \sqcap Y$ is indeed a lower bound of X and Y. This is clear by definition since any element of $X \sqcap Y$ is a lower bound of some element of X

and of some element of Y. It is also clear by definition that $X \sqcup Y$ is an upper bound of X and Y since any element of X (respectively Y) is either in $X \sqcup Y$, or less than or equal to some element of $X \sqcup Y$.

To show that $X \sqcap Y$ is the *greatest* lower bound of X and Y, let us assume that there exists a $Z \in \mathbf{2}^{[M]}$ such that $Z \sqsubseteq X$ and $Z \sqsubseteq Y$. Then, for any $z \in Z$, there exist some $x \in X$ and some $y \in Y$ such that $z \leqslant x$ and $z \leqslant y$. Thus, $z \leqslant x \wedge y$. But, $x \wedge y$ is either in $X \sqcap Y$, or less than or equal to some element of $X \sqcap Y$. Therefore, $Z \sqsubseteq X \sqcap Y$.

To show that $X \sqcup Y$ is the *least* upper bound of X and Y, let us assume that there is a $Z \in \mathbf{2}^{[M]}$ such that $X \sqsubseteq Z$ and $Y \sqsubseteq Z$. But, any element of $X \sqcup Y$ is an element of either X or Y. Consequently, any such element is less than or equal to some element of Z. Therefore, $X \sqcup Y \sqsubseteq Z$.

It is thus established that $\mathbf{2}^{[M]}$ is a lattice. The last thing to prove is that this lattice is distributive. To this end, it suffices to show that, for any X, Y, Z in $\mathbf{2}^{[M]}$,

$$(X \sqcup Y) \sqcap Z \sqsubseteq (X \sqcap Z) \sqcup Y. \tag{27}$$

Let t be an arbitrary element of $(X \sqcup Y) \sqcap Z$. By definition of $\sqcap$, there must exist some u in $X \sqcup Y$ and some v in Z such that $t = u \wedge v$. Now, by definition of join (26), u must be in X or in Y. If $u \in X$, then $u \wedge v$ must be less than or equal to some element u_1 of $X \sqcap Z$. If $u_1 \in (X \sqcap Z) \sqcup Y$, our point is made. So, let us assume that u_1 is not an element of $(X \sqcap Z) \sqcup Y$. By definition of $\sqcup$, and the fact that $u_1 \in (X \sqcap Z) \cup Y$, the only consistent way that this may be possible is if there is some u_2 in $(X \sqcap Z) \sqcup Y$ which is maximal in $(X \sqcap Z) \cup Y$ and such that $u_1 \leqslant u_2$. Therefore, for the case where $u \in X$, we have shown that

$$(X \sqcup Y) \sqcap Z \sqsubseteq (X \sqcap Z) \sqcup Y.$$

Now, for the case where $u \in Y$, since $t = u \wedge v$ we have $t \leqslant u$. Again, let us assume that $u \notin (X \sqcap Z) \sqcup Y$. As before, the only way this could be possible is if there exists some u_1 in $(X \sqcap Z) \sqcup Y$ which is maximal in $(X \sqcap Z) \cup Y$ and $u \leqslant u_1$. Hence, for the case where $u \in Y$ also,

$$(X \sqcup Y) \sqcap Z \sqsubseteq (X \sqcap Z) \sqcup Y. \quad \square$$

Again, we can readily observe that the construction of Theorem C.4 preserves the ordering relation and meet operation of M in the sense that, $\forall x \in M$, $\forall y \in M$,

$$\{x\} \sqsubseteq \{y\} \text{ iff } x \leqslant y; \qquad \{x\} \sqcap \{y\} = \{x \wedge y\} \tag{28, 29}$$

and hence, we have the following corollary.

C.5. Corollary. *Any Noetherian lower semi-lattice can be embedded into a distributive lattice such that glb's are preserved.*

It is important to remark that the construction taking a Noetherian lower semi-lattice into a distributive lattice is an extension of the one taking a Noetherian poset

into a lower semi-lattice in the sense that when the poset happens to be a lower semi-lattice, the two constructions are identical. Indeed, in that case, (26) and (25) are equivalent.

There is a close connection between the ordering on the complete restricted power of a poset and inclusion of ideals, as the following proposition indicates.

C.6. Proposition. *Let L be a lattice, and $\mathcal{I}$ and $\mathcal{J}$ two ideals of L. Then,*

$$\mathcal{I} \subseteq \mathcal{J} \text{ iff } \forall x \in \mathcal{I}, \exists y \in \mathcal{J} \text{ such that } x \leq y.$$

Proof. Trivially, if $\mathcal{I} \subseteq \mathcal{J}$, then any x in $\mathcal{I}$ is also in $\mathcal{J}$, and hence x itself is such a y. Conversely, if $x \in \mathcal{I}$ and $y \in \mathcal{J}$ are such that $x \leq y$, then since $\mathcal{J}$ is an ideal, we must have $x \in \mathcal{J}$. □

This suggests that this last construction is actually much stronger than it looks, as expressed in the next theorem.

C.7. Theorem. *For any Noetherian lower semi-lattice M, the lattice $2^{[M]}$ is a complete Brouwerian lattice.*

Proof. If we show that the lattice $2^{[M]}$ is isomorphic with the complete distributive lattice of all its ideals, it will follow from Theorem B.3 that it is a complete Brouwerian lattice. A way to do this is to establish that every ideal of $2^{[M]}$ is principal, and by Theorem A.2 the theorem will follow.

Let $\mathcal{I}$ be some arbitrary ideal of $2^{[M]}$. Since there are no infinitely ascending chains in M, the subset of M defined as

$$\mathcal{I}^* = \bigsqcup_{X \in \mathcal{I}} X = \left\lceil \bigcup_{X \in \mathcal{I}} X \right\rceil$$

is well-defined. This set is the set of maximal elements of the union of all subsets of incomparable elements of M which are in $\mathcal{I}$. Since $\mathcal{I}$ is an ideal, it is join-closed. And hence, $\mathcal{I}$ is the principal ideal generated by $\mathcal{I}^*$. □

C.8. Corollary. *Any Noetherian lower semi-lattice can be embedded into a complete Brouwerian lattice so that existing glb's are preserved.*

Appendix D. A semantics of type inheritance

We give here a 'type-as-set' denotational semantics of the ψ-term calculus of partially-ordered type structures.

We assume the existence of an abstract interpretation universe $\mathcal{U}$ of objects, where our types take meaning. A type is an intensional denotation of a set of elements in this universe. For example, the type 'person' denotes the class of all

objects in $\mathcal{U}$ which are categorized as persons. Some consensual agent is postulated—e.g., a programmer or an interpreter—for which such a categorization is meaningful. For example, it is reasonable to suppose that the reader's understanding of the English word "person" concurs with ours as far as our common sense interpretation; namely, a particular subclass of the class $\mathcal{U}$ of all objects. Hence, in particular, the least informative type ($\top$) denotes the whole universe $\mathcal{U}$; and the overdefined type ($\bot$) is the inconsistent type, and denotes the empty set—the type of no object.

The subtype relation is interpreted as set inclusion in the semantic universe $\mathcal{U}$. For example, if the set of students is contained in the set of persons, then the type 'student' is a subtype of the type 'person'.

Let T be such a set of types, endowed with a subtype ordering relation $\leqslant$. A type semantics is an order homomorphism:

$$\iota : \langle T, \leqslant \rangle \to \langle \mathbf{2}^{\mathcal{U}}, \subseteq \rangle,$$

where $\mathbf{2}^{\mathcal{U}}$ is the set of all subsets of $\mathcal{U}$. Namely,

$$\iota[\![\top]\!] = \mathcal{U}, \qquad \iota[\![\bot]\!] = \emptyset \tag{30}$$

and, for all s, t in T,

$$s \leqslant t \;\Rightarrow\; \iota[\![s]\!] \subseteq \iota[\![t]\!]. \tag{31}$$

Furthermore, if glb's exist, it is desired that

$$\iota[\![s \wedge t]\!] = \iota[\![s]\!] \cap \iota[\![t]\!]. \tag{32}$$

In addition to signature type denotation, the information content of attributes and inheritance of attributes must be given a denotation which is congruent with the constructor types. For example, given that the type 'person' is interpreted as a set of objects of $\mathcal{U}$, specifying the types of certain attributes of 'person' is a means to denote a further restriction of the type 'person', e.g., talking about the class of persons whose last name is a character string, rather than anything ($\top$). Thus, an attribute denotes the *intension* of a function between subsets of the universe $\mathcal{U}$. Attribute concatenation denotes function composition, and attribute coreference denotes the fact that certain functional diagrams commute.

More precisely, let Σ, $\leqslant$ be a partially-ordered type signature, and let ι be a type semantics for it. Now, we need to indicate how to extend consistently this type interpretation to one for ψ-terms. Let us define a monoid homomorphism η from $\mathcal{L}^*$ with string concatenation, to the set $\mathcal{U}^{\mathcal{U}}$ of functions from $\mathcal{U}$ to $\mathcal{U}$, with function composition. That is,

- for each label l in $\mathcal{L}$, $\eta[\![l]\!]$ is a function in $\mathcal{U}^{\mathcal{U}}$;
- $\eta[\![\varepsilon]\!]$ is the identity on $\mathcal{U}$;
- $\forall u, v \in \mathcal{L}^*$, $\eta[\![u.v]\!] = \eta[\![v]\!] \circ \eta[\![u]\!]$.

The type semantics ι is extended to ψ-terms by the denotational semantic equations (33)-(36). These equations can be construed as 'evaluation' rules for all possible syntactic cases. That is, the set which is the meaning of a given ψ-term is obtained

by repeatedly applying equations (33)-(36). These rules are clearly well-founded (i.e., there cannot be an infinite interpretation sequence using them) because of the finiteness of a ψ-term's domain and coreference relation index. Also, the order in which these equations are applied does not matter because of commutativity of set intersection.

Equation (33) treats the simple attribute case:

$$\iota[\![f(l \Rightarrow t)]\!] = \{x \in \iota[\![f]\!] \mid \exists y \in \iota[\![t]\!],\ \eta[\![l]\!](x) = y\}. \tag{33}$$

It is now clear that the identification with $\perp$ of all ψ-terms where $\perp$ occurs is justified by this semantics. Indeed, by (33), it comes immediately that

$$\iota[\![f(l \Rightarrow \perp)]\!] = \iota[\![\perp]\!] = \emptyset.$$

Equation (33) is generalized to many attributes as follows:

$$\iota[\![f(l_1 \Rightarrow t_1; \ldots; l_n \Rightarrow t_n)]\!] = \bigcap_{i=1}^{n} \iota[\![f(l_i \Rightarrow t_i)]\!]. \tag{34}$$

Attribute coreference means that compositions of attribute functions commute, as expressed by (35):

$$\iota[\![\psi_1]\!] = \{x \in \iota[\![\psi_2]\!] \mid \eta[\![l_0. \ldots . l_n]\!](x) = \eta[\![k_0. \ldots . k_m]\!](x)\}, \tag{35}$$

where

$$\psi_1 = f(l_0 \Rightarrow g_1(l_1 \Rightarrow \cdots g_n(l_n \Rightarrow X : t) \cdots); \quad k_0 \Rightarrow h_1(k_1 \Rightarrow \cdots h_m(k_m \Rightarrow X) \cdots))$$

and

$$\psi_2 = f(l_0 \Rightarrow g_1(l_1 \Rightarrow \cdots g_n(l_n \Rightarrow t) \cdots); \quad k_0 \Rightarrow h_1(k_1 \Rightarrow \cdots h_m(k_m \Rightarrow t) \cdots)).$$

Finally, cyclic coreference corresponds to fixed-points of attribute functions, as expressed in (36):

$$\iota[\![\psi_1]\!] = \{x \in \iota[\![\psi_2]\!] \mid \eta[\![l_0. \ldots . l_n]\!](x) = x\}, \tag{36}$$

where

$$\psi_1 = X : f_1(l_1 \Rightarrow \cdots f_n(l_n \Rightarrow X))$$

and

$$\psi_2 = f_1(l_1 \Rightarrow \cdots f_n(l_n \Rightarrow f_1)).$$

As for ϵ-types, they are constructed from ψ-types to be interpreted as disjunctive types through a powerlattice embedding. That such an embedding is semantically sound becomes clear when one understands the semantics of a type such as $\{t_1, \ldots, t_n\}$. In a 'type-as-set' semantics, the lub of the types t_i $(i = 1, \ldots, n)$ should

also be the lub of everything that is subsumed by every t_i. Hence, it comes naturally that

$$\iota[\![\{t_1, \ldots, t_n\}]\!] = \bigcup_{i=1}^{n} \iota[\![t_i]\!].$$

It is not difficult to verify that axioms (30)-(32) hold for this type semantics.

Acknowledgment

The author is indebted to Bob Boyer and Roger Nasr for their careful proof-reading and friendly support. Many thanks to Irène Guessarian for teaching me about algebraic semantics, and for her very keen insight for proofs. The contents and form of this article are strongly influenced by her work. The author is also grateful to Maurice Nivat for encouraging him to write it. Finally, much gratitude is expressed to the MCC-AI program for tolerating a benevolent use of their facilities for the redaction of this manuscript.

References

[1] H. Aït-Kaci, Solving type equations by graph rewriting, *Proc. 1st Internat. Conf. on Rewriting Techniques and Applications*, Dijon, France, 1985, Lecture Notes in Computer Science **202** (Springer, Berlin, 1986) 158–179.

[2] H. Aït-Kaci and R. Nasr, LOGIN: A logic programming language with built-in inheritance, *J. Logic Programming* 3 (3) (1986) 185–215.

[3] H. Aït-Kaci and R. Nasr, Logic and inheritance, *Proc. 13th ACM Symp. on Principles of Programming Languages*, St-Petersburg, FL (1986) 219–228.

[4] H. Aït-Kaci, A lattice-theoretic approach to computation based on a calculus of partially-ordered type structures, Ph.D. Thesis, Dept. Computer and Information Science, University of Pennsylvania, Philadelphia, PA, 1984.

[5] J.F. Allen and A.M. Frisch, What's in a semantic network, *Proc. 20th Ann. Meeting of the Association for Computational Linguistics*, 1982.

[6] G. Berry and J.J. Levy, Minimal and optimal computations of recursive programs, *J. ACM* **26** (1979) 148–175.

[7] G. Birkhoff, *Lattice Theory* (American Mathematical Society, Providence, RI, 1940; third revised edition 1979).

[8] R.J. Brachman, A new paradigm for representing knowledge, BBN Rept. 3605, Bolt, Beranek, and Newman, Inc., Cambridge, MA, 1978.

[9] R.J. Brachman, What IS-A is and isn't: An analysis of taxonomic links in semantic networks, *Computer* **16** (10) (1983) 30–35.

[10] L.E.J. Brouwer, On order in the continuum, and the relation of truth to non-contradictority, *Proc. Section of Sciences* **54** (Koninklijke Nederlandse Akademie van Wetenschappen, Amsterdam, 1951) 357–358.

[11] O.P. Buneman, Datatypes for database programming, in: M. Atkinson, O.P. Buneman and R. Morrison, eds., *Proc. Internat. Workshop on Persistence and Data Types in Programming Languages and Databases* (Universities of Glasgow and Saint-Andrews, 1985) 295–308.

[12] L. Cardelli, A semantics of multiple inheritance, in: G. Kahn, D. MacQueen and G. Plotkin, eds., *Proc. Internat. Symp. on the Semantics of Data Types*, Sophia-Antipolis, France, 1984, Lecture Notes in Computer Science **173** (Springer, Berlin, 1984) 51-68.
[13] C.F. Clocksin and W.M. Mellish, *Programming in PROLOG* (Springer, Berlin, 1980).
[14] B. Courcelle, Fundamental properties of infinite trees, *Theoret. Comput. Sci.* **25** (1983) 95-169.
[15] B. Courcelle and M. Nivat, The algebraic semantics of recursive program schemes, in: J. Winkowski, ed., *Proc. Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science **64** (Springer, Berlin, 1978) 16-30.
[16] A. Deliyanni and R.A. Kowalski, Logic and semantic networks, *Comm. ACM* **22** (3) (1979) 184-192.
[17] J. Donahue, On the semantics of data types, *SIAM J. Comput.* **8** (4) (1979) 546-560.
[18] W.F. Dowling and J.H. Gallier, A fast algorithm for testing the satisfiability of propositional Horn formulae, Tech. Rept. MS-CIS-83-26, Dept. Computer and Information Science, University of Pennsylvania, Philadelphia, PA, 1983.
[19] P.J. Downey, R. Sethi and R.E. Tarjan, Variations on the common subexpression problem, *J. ACM* **27** (4) (1980) 758-771.
[20] M. Dummett, *Elements of Intuitionism* (Oxford University Press, Oxford, U.K., 1977).
[21] N.V. Findler, ed., *Associative Networks: Representation and Use of Knowledge by Computers* (Academic Press, New York, 1979).
[22] J.A. Goguen, Order sorted algebras: Exceptions and error sorts coercions and overloaded operators, Semantics and Theory of Computation Report **14**, Computer Science Dept., UCLA, 1978.
[23] J.A. Goguen, J.W. Thatcher, E.G. Wagner and J.B. Wright, Initial algebra semantics and continuous algebras, *J. ACM* **24** (1) (1977) 68-95.
[24] J.A. Goguen, J.W. Thatcher and E.G. Wagner, An initial algebra approach to the specification correctness and implementation of abstract data types, in: R.T. Yeh, ed., *Current Trends in Programming Methodology IV* (Prentice-Hall, Englewood Cliffs, NJ, 1978) 80-149.
[25] J.A. Goguen and J. Meseguer, An initiality primer, SRI International Computer Science Laboratory, Draft, 1983.
[26] J.A. Goguen and J.J. Tardo, An introduction to OBJ: A language for writing and testing formal algebraic program specifications, *Proc. IEEE Conf. on Specifications of Reliable Software*, Cambridge, MA (1979) 170-189.
[27] S. Gorn, Data representation and lexical calculi, *Inform. Process. & Manag.* **20** (1, 2) (1984) 151-174.
[28] S. Gorn, Explicit definitions and linguistic dominoes, in: J.F. Hart and S. Takasu, eds., *Systems and Computer Science* (University of Toronto Press, Toronto, Ontario, 1965) 77-105.
[29] I. Guessarian, Personal communication, February, 1986.
[30] I. Guessarian, *Algebraic Semantics*, Lecture Notes in Computer Science **99** (Springer, Berlin, 1981).
[31] G. Huet, Résolution d'équations dans des langages d'ordre $1, 2, \ldots, \omega$, Thèse de Doctorat d'Etat, Université de Paris VII, France, 1976.
[32] D.J. Israel, On interpreting semantic network formalisms, BBN Rept. 5117, Bolt, Beranek and Newman, Inc., Cambridge, MA, 1982.
[33] K. Jensen and N. Wirth, *Pascal User Manual and Report* (Springer, Berlin, 1974).
[34] G. Kahn, D.B. MacQueen and G.D. Plotkin, eds., *Proc. Internat. Symp. on the Semantics of Data Types*, Sophia-Antipolis, France, Lecture Notes in Computer Science **173** (Springer, Berlin, 1984).
[35] B.W. Kernighan and D.M. Ritchie, *The C Programming Language* (Prentice-Hall, Englewood Cliffs, NJ, 1978).
[36] R.A. Kowalski, Logic for data description, in: H. Gallaire and J. Minker, eds., *Logic and Data Bases* (Plenum Press, New York, 1978) 77-103.
[37] H. Ledgard, *ADA: An Introduction; ADA Reference Manual* (Springer, Berlin, 1980).
[38] B.H. Liskov, E. Moss, C. Schaffert, B. Scheifler and A. Snyder, *CLU Reference Manual* (MIT Laboratory for Computer Science, Cambridge, MA, 1979).
[39] D.B. MacQueen, G.D. Plotkin and R. Sethi, An ideal model for recursive polymorphic types, *Proc. 11th ACM Symp. on Principles of Programming Languages*, Salt-Lake City, UT (1984) 165-175.
[40] D.B. MacQueen and R. Sethi, A higher order polymorphic type system for applicative languages, *Proc. Symp. on Lisp and Functional Programming*, Pittsburgh, PA (1982) 243-252.
[41] N. McCracken, An investigation of a programming language with a polymorphic type structure, Ph.D. Thesis, Dept. Computer Science, Syracuse University, New York, 1979.
[42] A.D. McGettrick, *ALGOL 68: A First and Second Course* (Cambridge University Press, Cambridge, U.K., 1978).

[43] J.R. McSkimin and J. Minker, A predicate calculus based semantic network for question-answering systems, in: N. Findler, ed., *Associative Networks—The Representation and Use of Knowledge by Computers* (Academic Press, New York, 1979).
[44] J. Meseguer and J.A. Goguen, Initiality induction and computability, in: M. Nivat and J. Reynolds, eds., *Application of Algebra to Language Definition and Compilation* (Cambridge University Press, Cambridge, U.K., 1984).
[45] R. Milner, A theory of type polymorphism in programming, *J. Comput. System Sci.* **17** (3) (1978) 348-375.
[46] M. Minsky, A framework for representing knowledge, in: P.H. Winston, ed., *The Psychology of Computer Vision* (McGraw-Hill, New York, 1975) 211-277.
[47] D.R. Musser, Abstract data type specification in the AFFIRM system, *IEEE Proc. Conf. on Specifications of Reliable Software*, Cambridge, MA (1979) 45-57.
[48] G. Nelson and D.C. Oppen, Fast decision procedures based on congruence closure, *J. ACM* **27** (2) (1980) 356-364.
[49] M. Nivat, On the interpretation of recursive polyadic program schemes, Symposia Mathematica **XV** (Istituto Nazionale di Alta Mathematica, Rome, 1975) 225-281.
[50] D.C. Oppen, Reasoning about recursively defined data structures, *J. ACM* **27** (3) (1980) 403-411.
[51] G.D. Plotkin, A powerdomain construction, *SIAM J. Comput.* **5** (1976) 452-487.
[52] G.D. Plotkin, Lattice theoretic properties of subsumption, Memorandum MIP-R-77, Dept. Machine Intelligence and Perception, University of Edinburgh, U.K., 1977.
[53] G.D. Plotkin, *Lecture Notes on Domain Theory*, Draft, 1983.
[54] M.R. Quillian, Semantic memory, in: M. Minsky, ed., *Semantic Information Processing* (MIT Press, Cambridge, MA, 1968) 216-270.
[55] J.C. Reynolds, Transformational systems and the algebraic structure of atomic formulas, in: D. Michie, ed., Machine Intelligence **5** (Edinburgh University Press, 1970) 135-151.
[56] R.B. Roberts and I.P. Goldstein, The FRL primer, Memorandum 408, AI Laboratory, MIT, Cambridge, MA, 1977.
[57] J.A. Robinson, A machine-oriented logic based on the resolution principle, *J. ACM* **12** (1) (1965) 23-41.
[58] W.C. Rounds and R. Kasper, A complete logical calculus for record structures representing linguistic information, *Proc. 1st Ann. Symp. on Logic in Computer Science*, IEEE Computer Society, Cambridge, MA (1986) 38-43.
[59] D. Scott, Data types as lattices, *SIAM J. Comput.* **5** (3) (1976) 522-587.
[60] D. Scott, Domains for denotational semantics, *9th Internat. Conf. on Automata Languages and Programming*, Lecture Notes in Computer Science **140**, Berlin (1982) 577-613.
[61] R.L. Sites, Algol W reference manual, Tech. Rept. CS-230, Computer Science Department, Stanford University, Stanford, CA, 1972.
[62] J.E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, Series in Computer Science **1** (MIT Press, Cambridge, MA, 1977).
[63] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, *Pacific J. Mathematics* **5** (1955) 285-309.
[64] W. Teitelman, *INTERLISP Reference Manual* (Xerox PARC, Palo Alto, CA, 1978).
[65] D. Weinreb and D. Moon, *Lisp Machine Manual* (MIT, Cambridge, MA, 1981).
[66] C. Wadsworth, *Lecture Notes on Denotational Semantics*, Hand-written draft, 1979.