Solving Type Equations by Graph Rewriting

Hassan Ait-Kaci

Microelectronics and Computer Technology Corporation[1]
9430 Research Boulevard
Austin, Texas 78759-6509
(512) 834-3354

1. Introduction

The first part of this paper focuses on syntactic properties of record-like 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 [8, 14, 15]. 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 paper.

2. 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 [5] or OBJ [11] 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 determines 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 attributes of attributes, *etc.*, 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,*

[1]Research described in this paper was done while the author was at the University of Pennsylvania, Philadelphia.

- *all* the coreference constraints binding in t_2 are also binding in t_1.

For example, understanding the symbols `student`, `person`, `philadelphia`, `cityname` to denote sets of objects, and if `student` $<$ `person` and `philadelphia` $<$ `cityname` denote set inclusion, then the type:

```
student(id => name(last => X:string);
        lives_at => Y:address(city => philadelphia);
        father => person(id => name(last => X);
                         lives_at => Y));
```

should be a subtype of:

```
person(id => name;
       lives_at => address(city => cityname);
       father => person);
```

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

2.1. 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 $\mathcal{L}$ be a set of *label symbols*, and let $\mathcal{T}$ be a set of *tag symbols*, both non-empty and countably infinite. I 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 [1]. It will suffice to mention that type symbols in Σ denote sets of objects, and label symbols in $\mathcal{L}$ denote the *intension* of functions. This semantics takes the partial ordering on type symbols into set inclusion, and label concatenation as 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 [8, 12, 13], I define a *term domain on* $\mathcal{L}$ to be the *skeleton* built from label symbols of a such a commutative diagram. This is nothing but the graph of arrows that one draws to picture functional maps. Formally,

Definition 1: A *term (or tree) domain* Δ on $\mathcal{L}$ is a set of finite strings of labels of $\mathcal{L}$ such that:

- Δ is *prefix-closed; i.e.,* if $\mathtt{u.v} \in \mathcal{L}^*$ and $\mathtt{u.v} \in \Delta$ then $\mathtt{u} \in \Delta$;
- Δ is *finitely branching; i.e.,* if $\mathtt{u} \in \Delta$, then the set $\{\mathtt{u.a} \in \Delta \mid \mathtt{a} \in \mathcal{L}\}$ is finite.

It follows from this definition that the empty string e 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 $\mathcal{L}$ = {`id`, `born`, `day`, `month`, `year`, `first`, `last`, `father`}, a term-domain on $\mathcal{L}$ may be Δ_1 = {e, `born`, `born.day`, `born.month`, `born.year`, `id`, `id.last`, `father`, `father.id`, `father.id.first`}. A term domain need not be finite; for instance, the regular expression Δ_2 = `a(ba)*+(ab)*`, where `a`, `b` $\in \mathcal{L}$, denotes a regular set (on {`a`,`b`}, say) which is closed under prefixes, and finitely branching; thus, it is a term domain and it is infinite.

Given a term domain Δ, an address `w` in Δ, we define the *sub-domain of* Δ *at address* `w` to be the term domain Δ\`w` = {`w'` | `w.w'` $\in \Delta$}. In the last example, the sub-domain at address `born` of Δ_1 is the set {e, `day`, `month`, `year`}, and the sub-domain of Δ_2 at address `a.b` is Δ_2 itself.

Definition 2: A term domain Δ is a *regular* term domain if the set of all sub-domains of Δ defined as **Subdom**(Δ) = {Δ\`w` | `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 **Subdom**(Δ_2) = {Δ_2, `b`.Δ_2}. In what follows, I will 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, I 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 -- an *association list*. An example of such an object is shown in figure 2-1.

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

Figure 2-1: 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 above term is the set of addresses {`e, id, born, born.day, born.month, born.year, father`}.

The example in figure 2-1 shows a possible description of what one may intend to use as a 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 sub-structures appearing 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 figure 2-2.

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

Figure 2-2: An example of tagging in a term structure

> **Definition 3:** A *term* is a triple (Δ, ψ, τ) where Δ is a term domain on $\mathcal{L}$, ψ is a *symbol* function from $\mathcal{L}^*$ to Σ such that $\psi(\mathcal{L}^* - \Delta) = \{\top\}$, and τ is a *tag* function from Δ to $\mathcal{T}$. A term is finite (*resp.* regular) if its domain is finite (*resp.* regular).

Such a definition illustrated for the term in figure 2-2 is captured in the table in figure 2-3. Note the *"syntactic sugar"* implicitly used in figure 2-2. Namely, I shall omit writing explicitly tags for addresses which are not sharing theirs. In the sequel, by "term" it will be meant "regular term".

Given a term $t = (\Delta, \psi, \tau)$, an address w in Δ, the *subterm of t at address w* is the term $t\backslash w = (\Delta\backslash w, \psi\backslash w, \tau\backslash w)$ where $\psi\backslash w$: $\mathcal{L}^* \rightarrow \Sigma$ and $\tau\backslash w$: $\Delta\backslash w \rightarrow \mathcal{T}$ are defined by:

- $\psi\backslash w(w') = \psi(w.w')\ \forall w' \in \mathcal{L}^*$;
- $\tau\backslash w(w') = \tau(w.w')\ \forall w' \in \Delta\backslash w$.

From these definitions, it is clear that $t\backslash e$ is the same as t. In example of figure 2-2, the subterm at address `father.id` is `name(last => X:string)`.

Given a term $t = (\Delta, \psi, \tau)$, a symbol `f`, (*resp.*, a tag `X`, a term `t'`) is said to *occur* in t if there is an address w

Addresses (Δ)	*Symbols (ψ)*	*Tags (τ)*
e	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

Figure 2-3: (Δ, ψ, τ)-definition of the term in figure 2-2

in Δ such that $\psi(w) = f$ (*resp.*, $\tau(w) = X$, $t\backslash w = t'$). The following proposition is immediate and follows by definition.[2] .

Proposition 4: Given a term $t = (\Delta, \psi, \tau)$, the following statements are equivalent:

- t is a regular term;
- The number of subterms occurring in t is finite;
- The number of symbols occurring in t is finite;
- The number of tags occurring in t is finite.

It follows that a coreference relation on a regular term domain has finite index.

Definition 5: In a term, any two addresses bearing the same tag are said to corefer. Thus, the *coreference* relation κ of a term $t = (\Delta, \psi, \tau)$ is a relation defined on Δ as the *kernel* of the tag function τ; *i.e.*, $\kappa = \mathrm{Ker}(\tau) = \tau \circ \tau^{-1}$.

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 figure 2-2, the addresses `father.id.last` and `id.last` corefer.

A term t is *referentially consistent* if the same subterm occurs at all addresses in a coreference class. That is, if C is a coreference class in Δ/κ then $t\backslash w$ is *identical* for *all* addresses w in C. Thus, if a term is referentially consistent, then by definition for any w_1, w_2 in Δ, if $\tau(w_1) = \tau(w_2)$ then for all w such that $w_1.w \in \Delta$, 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, or a *right-congruence*, on Δ. That is, for any two addresses w_1, 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 $w_2.w \in \Delta$.

Definition 6: A *well-formed term* (wft) is a term which is referentially consistent.

I shall use this property to justify another syntactic "sweetness": whenever a tag occurs in a term without a subterm, what is meant is that the subterm elsewhere referred to in the term by an address bearing this tag is implicitly present. If there is no such subterm, the implicit subterm is $\top$. For example, in the term `foo(`l_1 `=> X; `l_2 `=> X:bar; `l_3 `=> Y; `l_4 `=> Y)`, the subterm at address l_1 is `bar`, and the subterm at address l_4 is $\top$. In what follows, $\top$ will never be written explicitly in a term.

Note that it is quite possible to consider *infinite* terms such as shown in figure 2-4. For example, at the addresses `father` and `father.son.father`, is a phenomenon which I call *cyclic tagging*.

Syntactically, cycles may also be present in more pathological ways such as pictured in figure 2-5, where one must follow a path of cross-references.

[2] Also established in [8]

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

Figure 2-4: An example of simple cyclic tagging in a term structure

$$
\begin{aligned}
&foo(l_1 \Rightarrow X_1{:}foo_1(k_1 \Rightarrow X_2);\\
&\quad l_2 \Rightarrow X_2{:}foo_2(k_2 \Rightarrow X_3);\\
&\quad \ldots\\
&\quad l_i \Rightarrow X_i{:}foo_i(k_i \Rightarrow X_{i+1});\\
&\quad \ldots\\
&\quad l_n \Rightarrow X_n{:}foo_n(k_n \Rightarrow X_1));
\end{aligned}
$$

Figure 2-5: An example of complex cyclic tagging in a term structure

A term is *referentially acyclic* if there is no cyclic tagging occurring in the term. A *cyclic term* is one which is not referentially acyclic. Thus, the terms in figures 2-4 and 2-5 are *not* referentially acyclic. A wft is then best pictured as a *labelled directed graph* as illustrated in figure 2-6 which is the *graph representation* of the wft below. Thus, labels act as arcs between nodes bearing type symbols. Tags are *physical pointers* to nodes, indicating which nodes are shared.

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

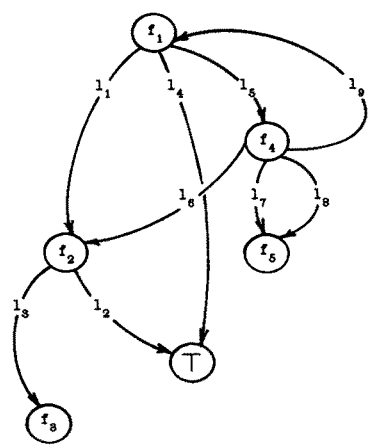

Figure 2-6: Graph representation of a wft

In figure 2-6, the similarity with finite states diagrams is not coincidental. And thus, it follows that a term is referentially acyclic if and only if its term domain is finite. Also, any term (cyclic or not) expressed in the above syntax is a regular term.

The set of well-formed terms is denoted $\mathcal{WFT}$. The set of well-formed acyclic terms is denoted $\mathcal{WFAT}$ and is a subset of $\mathcal{WFT}$.

I 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 which preserves the coreference classes.

Definition 7: Two terms $t_1 = (\Delta_1, \psi_1, \tau_1)$ and $t_2 = (\Delta_2, \psi_2, \tau_2)$ are *alphabetical variants* of one another (noted $t_1 \ \alpha \ t_2$) if and only if:

1. $\Delta_1 = \Delta_2$;
2. $\mathtt{Ker}(\tau_1) = \mathtt{Ker}(\tau_2)$;
3. $\psi_1 = \psi_2$.

Interpreting these structures as commutative diagrams betweens sets, it comes that the symbols $\top$ and $\bot$ denote, respectively, the whole universe -- *"anything"* -- and the empty set -- *"inconsistent"*. Hence, a term in which the symbol $\bot$ occurs is to be interpreted as being inconsistent. To this end, we can define a relation $\Downarrow$ on $\mathcal{WFT}$ -- *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. In the way they have been defined, the relations α and $\Downarrow$ are such that their *union* $\approx \ = \alpha \cup \Downarrow$ is an equivalence relation. Thus,

Definition 8: A *ψ-type* is an element of the quotient set $\Psi = \mathcal{WFT}/\approx$. An *acyclic* ψ-type is an element of the quotient set $\Psi_0 = \mathcal{WFAT}/\approx$.

2.2. 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*. I define the *subsumption* relation on the set Ψ as follows.

Definition 9: A term $t_1 = (\Delta_1, \psi_1, \tau_1)$ *is subsumed* by a term $t_2 = (\Delta_2, \psi_2, \tau_2)$ (noted $t_1 \preceq t_2$), if and only if *either*, $t_1 \approx \bot$; *or*,

1. $\Delta_2 \subseteq \Delta_1$;
2. $\mathtt{Ker}(\tau_2) \subseteq \mathtt{Ker}(\tau_1)$;
3. $\psi_1(w) \leq \psi_2(w), \ \forall w \in L^*$.

It is easy to verify that a subsumption relation on Ψ defined by $[t_1] \preceq [t_2]$ if and only if $t_1 \preceq t_2$ is well-defined (*i.e.*, it does not depend on particular class representatives) and it is an *ordering* relation.[3]

This notion of subsumption is related to the (in)famous *IS-A* ordering in semantic networks [5, 6], and the *tuple ordering* in the so-called semantic relation data model [4]. 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 any subsets of Σ, then this property carries over to Ψ.

Theorem 10: If the signature Σ is a lattice, then so is Ψ.

Rather than giving formal definitions for the meet and join operations on Ψ, let us illustrate the extended lattice

[3] In the sequel, I shall use the (*abusive*) convention of denoting a ψ-type by one of its class representatives, understanding that what is meant is modulo *tag renaming* and *smashing*.

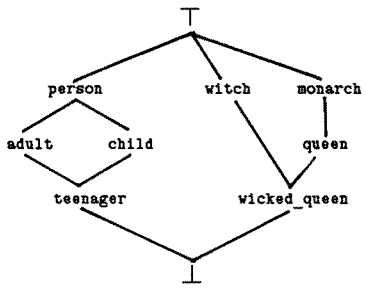

Figure 2-7: A signature which is a lattice

operations with an example. Figure 2-7 shows a signature which is a finite (non-modular) lattice. Given this signature, the two types in figure 2-8 admit as meet and join the types in figure 2-9.

```
child(knows => X:person(knows => queen;
                        hates => Y:monarch);
      hates => child(knows => Y;
                     likes => wicked_queen);
      likes => X);

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

Figure 2-8: Two wft's

```
person(knows => person;
       hates => person(knows => monarch;
                       likes => monarch));

teenager(knows => X:adult(knows => wicked_queen;
                          hates => Y:wicked_queen);
         hates => child(knows => Y;
                        likes => Y);
         likes => X);
```

Figure 2-9: LUB and GLB of the two types in figure 2-8

The reader is referred to [1] for the detailed definitions of the meet and join operation on Ψ. It suffices here to say that they are essentially extensions of the *unification* [15, 20] and *generalization* [19] operations on regular first-order terms. Indeed, these operations are special cases of my 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 the symbols occurring at these leaves are restricted to be $\top$.

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

Theorem 11: If Σ is a lattice, then so is Ψ_0. However Ψ_0 is *not* a sublattice of Ψ.

The join operation is the same, 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 $\bot$. However, Ψ_0 is not a sublattice of Ψ, since the meet in Ψ of two

acyclic wft's is not necessarily acyclic. Consider, for example[4]

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

$$t_2 = f(l_1 \Rightarrow X : f;\ l_2 \Rightarrow X)$$

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

2.3. A Distributive Lattice of Types

Accepting the "type-as-set" interpretation of the calculus of ψ-types, it is yet necessary to wonder whether lattice-theoretic properties of meet and join reflect those of intersection and union. Unfortunately, this is not the case with Ψ. The lattice of ψ-types is not so convenient as to be *distributive*, even if the signature Σ is itself distributive. As a counter-example, consider the flat (distributive) lattice $\Sigma = \{\top, a, f, \bot\}$. Indeed,

$$f \wedge (f(1 \Rightarrow a) \vee a) = f$$

$$(f \wedge f(1 \Rightarrow a)) \vee (f \wedge a) = f(1 \Rightarrow a)$$

and this proves that $\mathcal{WFT}$ is not distributive.[5]

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 10, 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. In practice, programs deal with finite sets of primitive types. Even then, it would be quite unreasonable to require that all meets and joins of those primitive 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 primitive 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 GLB, then $\top$ is returned. Clearly, it is not correct to infer that the given object is of type *"anything"* just because Σ does not happen to contain explicitly a symbol for the GLB `foo` and `bar`. All that can be correctly said is that the given object is of *disjunctive* type `foo`$\vee$`bar`.

I next give a brief summary of a construction of such a more adequate type lattice. It may be construed as a powerdomain construction to handle indeterminacy [17]; in our case, *variant records*. It is not possible to detail this construction here. The interested reader is referred to [1].

A poset is *Noetherian* if it does not contain infinitely ascending chains. Given a set S, the set $\mathcal{P}^{(S)}$ of finite non-empty subsets of *maximal elements* of S is called the *restricted power of* S. If S is a Noetherian poset, the set $\mathcal{P}^{[S]}$ of *all* such subsets of maximal elements is called the *complete* restricted power of S. Given a Noetherian poset S, and $S' \subseteq S$, $\Re(S')$ is the set of maximal elements of S'.

I shall call $\mathcal{L}$ the set $\mathcal{P}^{[\Psi]}$, and $\mathcal{L}_0$ the set $\mathcal{P}^{[\Psi_0]}$. Clearly, $\mathcal{L}_0$ is a subset of $\mathcal{L}$. I shall denote a singleton $\{t\}$ in $\mathcal{L}$ simply by t.

---

[4] A similar phenomenon happens in unification of first-order terms where it is reason for the so-called *"occur-check"* testing whether a variable occurs in a term when trying to unify that variable with the term.

[5] A similar result was pointed out by G.Plotkin in [18].

Definition 12: Subsumption in £ is defined by, $T_1 \sqsubseteq T_2$ if and only if *every* ψ-type in T_1 is subsumed by *some* ψ-type in T_2.

Let's define a notational variant of elements of £ which will have the advantage of being more *compact* syntactically. Consider the object shown in figure 2-10. The syntax used is similar to the one which has expressed ψ-types up to now. However, *sets* of terms rather that terms may occur at some addresses.

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

Figure 2-10: Example of a ϵ-term

This notation may be viewed as a compact way of representing a sets of ψ-types. For example, the object in figure 2-10 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*; *i.e.*, 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.

We are now ready to construct a distributive lattice of ϵ-types. First, we relax the demand that the signature Σ be lattice. Assuming it is a Noetherian poset we can embed it in a meet-semilattice $\mathcal{Z}^{[\Sigma]}$ 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.

Theorem 13: If the signature Σ is a Noetherian poset then so is the lattice Ψ_0; but the lattice Ψ is *not* Noetherian.

The following counter-example exhibits an infinitely ascending chain of wfts in Ψ. For any $\mathbf{a}$ in $\mathcal{L}$ and any $\mathbf{f}$ in Σ, define the sequence $\mathbf{t}_n = (\Delta_n, \psi_n, \tau_n)$, $n \geq 1$ as follows:

$$\Delta_n = \mathbf{a}^*;$$

$$\psi_n(\Delta_n) = \mathbf{f};$$

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

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

$$\Delta_{n+1} \subseteq \Delta_n;$$

$$\psi_n(\Delta_n) \leq \psi_{n+1}(\Delta_{n+1});$$

$$\kappa_{n+1} \subset \kappa_n.$$

In our syntax, this corresponds to the sequence:

$\mathbf{t}_0$ = X : f(a => X),

$\mathbf{t}_1$ = f(a => X : f(a => X)),

$\mathbf{t}_2$ = f(a => f(a => X : f(a => X))), ... ,

$\mathbf{t}_n$ = f(a => f(a => ... f(a => X : f(a => X))...)), ...
<---- n+1 a's ---->

We define two binary operations $\sqcap$ and $\sqcup$ on the set $£_0$. For any two sets T_1 and T_2 in $£_0$:

$T_1 \sqcap T_2 = \Re(\{t \mid t = t_1 \wedge t_2,\ t_1 \in T_1,\ t_2 \in T_2\})$;

$T_1 \sqcup T_2 = \Re(T_1 \cup T_2)$.

where $\wedge$ is the meet operation defined on Ψ_0. Then, for *any* poset Σ containing $\top$ and $\bot$,

Theorem 14: The poset $\mathcal{L}_0$ is a *distributive* lattice whose meet is $\sqcap$, whose join is $\sqcup$, and whose top and bottom are $\{\top\}$ and $\{\bot\}$.

It is not possible to define lattice operations for $\mathcal{L}$ because Ψ is not Noetherian. Hence, the set of maximal elements of a set cannot be defined for *all* sets. However, if only finite sets of regular wft's are considered, then:

Theorem 15: The poset $\mathcal{Z}^{(\Psi)}$ of *finite* sets of incomparable regular wft's is a distributive lattice.

However, it is not complete. It is also true that $\mathcal{Z}^{(\Psi_0)} \subseteq \mathcal{Z}^{(\Psi)}$ and $\mathcal{Z}^{(\Psi_0)}$ is a distributive lattice, but it is not a sublattice of $\mathcal{L}$. In general, the GLB of elements of $\mathcal{Z}^{(\Psi_0)}$ is a lower bound of the GLB of these elements taken in $\mathcal{Z}^{(\Psi)}$.

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, written as $a \rightarrow b$. An interesting point is that *(i)* any Brouwerian lattice is distributive but, *not conversely*; and *(ii)* any Boolean lattice is Brouwerian, but *not conversely* [3]. Thus, the class of Brouwerian lattices lies strictly *between* the class of distributive lattices and the class of Boolean lattices.

Theorem 16: If the signature Σ is a Noetherian poset then the lattice $\mathcal{L}_0$ of *all* sets of finite wfts is a complete Brouwerian lattice.

To answer the question that might be hovering in the reader's mind,[6] the fact that the lattice $\mathcal{L}_0$ is a complete Brouwerian lattice reveals itself invaluable for showing the existence of solutions to systems of equations. Apart from its lattice theoretic properties, a Brouwerian lattice is interesting as it forms the basis of an *intuitionistic* propositional logic, due to L.E.J.Brouwer [7, 10].

Unfortunately, theorem 16 does not hold for $\mathcal{L}$ the lattice of all regular terms since the lattice $\mathcal{L}$ is not complete. On the other hand, I do not know whether $\mathcal{L}$ is Brouwerian.

3. Programs as Recursive Type Equations

Consider the equations in figure 3-1. Each equation is a pair made of a symbol and an ϵ-term, and may intuitively be understood as a *definition*. I shall call a set of such definitions a *knowledge base*.[7]

Definition 17: A *knowledge base* is a function from $\Sigma - \{\bot\}$ to $\mathcal{L}_0$ which is the identity almost everywhere except for a finite number of symbols.

So far, the partial order on Σ has been assumed predefined. However, given a knowledge base, it is quite easy to quickly infer what I shall call its *implicit symbol ordering*. For example, examining the knowledge base in figure 3-1, it is evident that the signature Σ must contain the set of symbols {`list`, `cons`, `nil`, `append`, `append_0`, `append_1`}, and that the partial ordering on Σ is such that `nil` $<$ `list`, `cons` $<$ `list`, `append_0` $<$ `append`, `append_1` $<$ `append`. In general, this ordering can always be extracted from the specification of a knowledge base.

Definition 18: A knowledge base is *well-defined* if and only if it admits an implicit symbol ordering.

---

[6] Namely, *"So what?..."*

[7] Or *program*, or *type environment*... Nevertheless, *knowledge base* is a deliberate choice since what is defined is in essence an *abstract semantic network*.

```
list = {nil, cons};

append = {append_0, append_1};

append_0 =
      (front => nil;
       back => X:list;
       whole => X);

append_1 =
      (front => cons(head => X; tail => Y);
       back => Z:list;
       whole => cons(head => X; tail => U);
       patch => append(front => Y; back => Z; whole => U));
```

Figure 3-1: A specification for appending two lists

I want to describe an *interpretation* of any given type in the context of this knowledge base so that *expanding* the input according to the specifications will produce a consistently typed object. A ψ-type is evaluated by *"expanding"* its root symbol if its knowledge base value is not itself; *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 $\top$). If the root symbol is mapped to itself by the knowledge base, the process is applied recursively to the subterms. Recalling the "type-as-set" semantics of ϵ-types and ψ-types, this process essentially computes unions and intersections of sets. The symbol substitution process is to be interpreted as *importing* the information encapsulated in the symbol into the context of another type.

Let's *trace* what the interpreter does, one step at a time, on an example. Let's suppose that the knowledge base in figure 3-1 is defined. Consider the following input:

```
append(front => cons(head => 1;
                     tail => cons(head => 2;
                                  tail => nil));
       back => cons(head => 3;
                    tail => nil));
```

Next, the interpreter expands `append` into `{append_0, append_1}`:

```
{append_0(front => cons(head => 1;
                        tail => cons(head => 2;
                                     tail => nil));
          back => cons(head => 3;
                       tail => nil)),

 append_1(front => cons(head => 1;
                        tail => cons(head => 2;
                                     tail => nil));
          back => cons(head => 3;
                       tail => nil))};
```

Each of these two basic ϵ-terms is further expanded according to the definitions of their heads. However, the first one (`append_0`) yields $\perp$ since the meet of the subterms at `front` is $\perp$. Hence, by $\Re$-reduction, 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));
```

The process continues, expanding the subterms:[8]

```
(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. The interpretation of **append** has thus correctly produced a type whose **whole** is the concatenation of its **front** to its **end**. The result could be isolated by projection on the field **whole** if desired. The attribute **patch** is 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;
                            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))));
```

Computation in KBL amounts essentially to term rewriting. In fact, it bears much resemblance with computation with non-deterministic program schemes [9, 16], and macro-languages and tree grammars [14]. This section attempts a formal characterization of computation in KBL along the lines of the algebraic semantics of

[8] For what remains, I shall leave out the details of cleaning-up $\perp$ by $\mathfrak{R}$-reduction.

tree grammars [2, 14]. Symbol rewriting presented in this section is very close to the notion of second-order substitution defined in [8] and macro-expansion defined in [14].

It is next shown that a KBL program can be seen as a system of equations. Thanks to the lattice properties of finite wft's, such a system of equations admits a least fixed-point solution. The particular order of computation of KBL, the *"fan-out computation order"*, which rewrites symbols closer to the root first is formally defined and shown to be maximal; *i.e.*, it yields "greater" ϵ-types than any other order of computation. Unfortunately, the complete "correctness" of KBL is not established. That is, it is not known (yet) whether the normal form of a term is equal to the fixed-point solution. However, as steps in this direction, two technical lemmas are conjectured to which a proof of the correctness is corollary.

All wft's considered hereafter are *finite*. Hence, I shall not bother mentioning the adjective "finite" when dealing with wft's for the rest of this paper.

3.1. Wft substitution

I next introduce and give some properties of the concept of wft substitution. Roughly, given a wft t such that a symbol f 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 $t = (\Delta, \psi, \tau)$ and some string u in $\mathcal{L}^*$, I define the wft $u.t$ to be the smallest wft containing t at address u; that is, $u.t = (u.\Delta, u.\psi, u.\tau)$ where

- $u.\Delta = \{w \in \mathcal{L}^* \mid w = u.v,\ v \in \Delta\}$;
- $u.\psi(w) = $ *if* $w = u.v$ *then* $\psi(v)$ *else* $\top$;
- $u.\tau : u.\Delta \rightarrow T$ *such that* $u.\tau(v) = u.\tau(w)$ *iff* $v = u.v'$, $w = u.w'$ *and* $\tau(v') = \tau(w')$.

This can be better visualized as the wft obtained by attaching the wft t at the end of the string u.

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 (Δ, ϕ, τ), 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 obtained is still well-formed.

> **Definition 19:** Let $t = (\Delta, \psi, \tau)$ be a wft and u some address in Δ, and let t' be a wft. The term $t[t'/u]$ is defined as $t[t'/u] = t[u:\top] \wedge u.t'$.

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. Note that $\perp$ may result out of such a substitution. To illustrate this operation, 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 $t[t'/patch]$ is

```
(front => cons(head => X1 : 1;
               tail => X2 : cons(head => X5 : 2;
                                 tail => X6 : nil));
 back => X3 : cons(head => 3;
                   tail => nil);
 whole => cons(head => X1;
               tail => X7 : cons(head => X5;
                                 tail => X4));
 patch => (front => X2;
           back => X3;
           patch => append(front => X6;
                           back => X3;
                           whole => X4);
           whole => X7)).
```

Next, I give a series of "surgical" lemmas about this substitution operation which will be needed in proving key properties of KBL's computation rule. 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 is made apparent as depicted in figure 3-2.

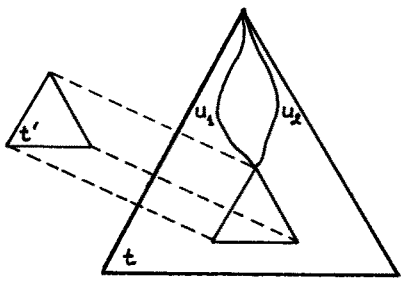

Figure 3-2: Substitution at coreferring addresses

Lemma 20: Let $t = (\Delta, \psi, r)$ and t' be wft's, and let u_1, u_2 be two coreferring addresses in Δ. Then, $(t[t'/u_1])[t'/u_2] = t[t'/u_1] = t[t'/u_2] = (t[t'/u_2])[t'/u_1]$

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 in $\mathcal{L}^*$. That is, u covers v in t if v occurs in t\u.

Next, it is important to analyze the extent to which a sequence of substitutions 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.

Lemma 21: If u and v are addresses in a wft t which do not cover each other, then for any wft t',

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

The next lemma gives a sufficient condition for commutativity.

Lemma 22: Let $t = (\Delta, \psi, r)$, t_1, t_2 be wft's, and let u_1, u_2 be two addresses in Δ which do not cover each other. Then,

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

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

resulting from performing the "innermost" substitution first. The picture in figure 3-3 may help illustrate the argument.

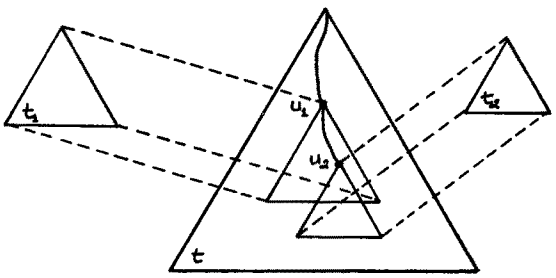

Figure 3-3: Substitutions at covering addresses

Lemma 23: If two addresses u_1 and u_2 in a wft t are such that u_1 covers u_2, then

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

for any wft's t_1 and t_2.

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 26 is proposed to that effect, using the above technical lemmas.

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

Definition 24: Let $t = (\Delta, \psi, \tau)$ be a wft; then, the *radius* of an address u in Δ is defined as $\rho(u) = \mathtt{Min}(\{|v| \mid v \in [u]\})$.

That such a minimum number exists for all classes is clear. Recall that lemma 20 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 considered hereafter that the address at which the substitution is performed is a minimal length in its class.

Definition 25: 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 $\rho(u_i) \leq \rho(u_j)$.

For example, in the wft:

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

the sequence e, $l_5 \cdot l_6$, l_4, $l_5 \cdot l_7$, $l_1 \cdot l_3$ is in fan-out order. However, the sequence e, $l_5 \cdot l_6$, l_4, $l_1 \cdot l_3$, l_5 is not. In the sequel, I shall lighten the notation $(t[t_1/u_1])[t_2/u_2]$ to $t[t_1/u_1][t_2/u_2]$.

The following theorem is a consequence of the lemmas just presented.

Theorem 26: 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, $i=1,\ldots,n$ is in fan-out order. Let π be a permutation of the set

$\{1,\ldots,n\}$ such that $\pi(u_1)$, $i=1,\ldots,n$ is also in fan-out order. Then, for any set of wft's $\{t_1,\ldots,t_n\}$,

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

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

$$t[t_1/u_1]\ldots[t_n/u_n] \preceq t[t_{\pi(1)}/u_{\pi(1)}]\ldots[t_{\pi(n)}/u_{\pi(n)}]. \quad (2)$$

Substitution is extended to ϵ-types as follows: for any t in Ψ_0, T in $\pounds_0$, and any u in Δ_t,

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

3.2. Symbol Rewriting Systems

Definition 27: A *Symbol Rewriting System* (SRS) on Σ is a system S of n equations $S: s_i = E_i$, where $s_i \in \Sigma$ and $E_i \in \pounds_0$, for $i = 1,\ldots,n$.

Given such a system S, I shall denote by E the subset $\{s_1,\ldots,s_n\}$ of Σ of S-*expandable* symbols, and N the set $\Sigma-E$ of *non*-S-*expandable* symbols of Σ. An example of a SRS is given by figure 3-1. There, we have E = {list, append, append_0, append_1} and N = {nil, cons}.

Definition 28: Let S: $s_i = T_i$ be a SRS. It defines a *one-step rewriting* relation $\xrightarrow[S]{}$ on $\pounds_0$ as follows: $T_1 \xrightarrow[S]{} T_2$ if and only if there is 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 on page 11.

I shall denote by $\xrightarrow[S]{k}$ for $k\geq 0$ the relation $\xrightarrow[S]{}$ composed with itself k times, and by $\xrightarrow[S]{*}$ the reflexive and transitive closure of $\xrightarrow[S]{}$; that is, the relation $\cup_{k=0}^{\infty} \xrightarrow[S]{k}$.

In the foregoing, the notation for the sets Ψ of ψ-types and $\pounds$ of ϵ-types was implicitly understood to depend on the signature of symbols Σ. Whenever it will be necessary to make this more explicit I shall use the notation $\Psi[\Sigma]$ and $\pounds[\Sigma]$.

Definition 29: Let S be a SRS, and t be a wft. The S-*normal form* of an ϵ-type T is defined as

$$\mathcal{N}(T) = \sqcup\{T' \in \pounds_0[N] \mid T \xrightarrow[S]{*} T'\}$$

That is the LUB of all terms containing no more expandable symbols that can be rewritten from T. Since $\pounds_0$ is a complete lattice, this is well-defined. Notice that a normal form is defined as a join of *all* possible rewriting of an ϵ-type. Thus, by theorem 26, we can restrict this definition only to sequences of rewritings in fan-out order without losing anything in the definition of a normal form.

To lighten notation, I shall make use of vector notation to denote elements of $\pounds_0^n$ the set of n-tuples of ϵ-types ; *e.g.*, $\vec{T} = \langle T_1,\ldots,T_n\rangle$, $T_i \in \pounds_0$, $i=1,\ldots,n$. Hence, a symbol rewriting system S of n equations, is denoted by a single vector equation $\vec{s} = \vec{E}$. Given such a SRS, I shall use either indices in $\{1,\ldots,n\}$ or the symbols s_i to index the components of a vector $\vec{T}$ in $\pounds_0^n$; *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 $\xrightarrow[S]{}$ relation, and so is the definition of vector normal form $\vec{\mathcal{N}}(\vec{T})$.

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

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

Any indexing of $\mathbf{X}(t,\vec{s}) = \{u_1,\ldots,u_m\}$ will be assumed to be a *fan-out indexing*. That is, one such that the sequence $(u_1,\ldots,u_m)$ is in fan-out order. For example, taking the wft t on page 15 and $\vec{s} = \langle f_2, f_4, f_5\rangle$ we have $\mathbf{X}(t,\vec{s}) = \{1_1, 1_5, 1_5.1_7\}$.

The 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 $t[\vec{T}/\vec{s}]$ and defined as:

$$t[\vec{T}/\vec{s}] = t[T_{\psi_t(u_1)}/u_1]\ldots[T_{\psi_t(u_m)}/u_m] \tag{3}$$

where $\{u_1,\ldots,u_m\} = \mathbf{X}(t,\vec{s})$. By theorem 26, it is evident that this is a well-defined operation. I shall condense notation in 3 to:

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

Let's illustrate this operation on a small example. Let's take $\vec{s} = \langle s_1, s_2\rangle$ and $\vec{T} = \langle T_1, T_2\rangle$ with

$$T_1 = \{f(1_1 \Rightarrow X;\ 1_2 \Rightarrow X),\ g\}$$

$$T_2 = h(1_2 \Rightarrow X;\ 1_3 \Rightarrow X)$$

and the term

$$t = s_1(1_1 \Rightarrow s_2;\ 1_3 \Rightarrow s_1).$$

The set of expandable addresses for $\vec{s}$ in t thus is:

$$\mathbf{X}(t,\vec{s}) = \{e,\ 1_1,\ 1_2\}$$

corresponding to the symbols (in fan-out order) s_1, s_2, s_1. Hence, the sequence of substitutions starts with s_1 at e:

$$\begin{array}{l}\{\ f(1_1 \Rightarrow X : s_2;\\ \quad 1_2 \Rightarrow X;\\ \quad 1_3 \Rightarrow s_1),\\ \ g(1_1 \Rightarrow s_2;\\ \quad 1_3 \Rightarrow s_1)\ \}\end{array}$$

then continues with s_2 at 1_1:

$$\begin{array}{l}\{\ f(1_1 \Rightarrow X : h(1_2 \Rightarrow Y;\ 1_3 \Rightarrow Y);\\ \quad 1_2 \Rightarrow X;\\ \quad 1_3 \Rightarrow s_1),\\ \ g(1_1 \Rightarrow h(1_2 \Rightarrow X;\ 1_3 \Rightarrow X);\\ \quad 1_3 \Rightarrow s_1)\ \}\end{array}$$

and finally ends with s_1 at 1_2:

```
{ f(l1 => X : h(l2 => Y; l3 => Y);
    l2 => X;
    l3 => f(l1 => Y : l1; l2 => Y),
  g(l1 => h(l2 => X; l3 => X);
    l3 => h(l2 => Y; l3 => Y)),
  f(l1 => X : h(l2 => Y; l3 => Y);
    l2 => X;
    l3 => g),
  g(l1 => h(l2 => X; l3 => X);
    l3 => g) }
```

which is the value of $t[\vec{T}/\vec{s}]$

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

$$(\vec{T}[\vec{T}'/\vec{s}])_i = \sqcup_{t \in T_i} t[\vec{T}'/\vec{s}]. \tag{4}$$

Definition 30: An element $\vec{T}$ of $\mathcal{L}_0^n$ is a *solution* of the equation $\vec{s} = \vec{E}$ if and only if

$$\vec{E}[\vec{T}/\vec{s}] = \vec{T}.$$

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

$$\mathcal{F}(\vec{T}) = \vec{E}[\vec{T}/\vec{s}]. \tag{5}$$

Proposition 31: The function $\mathcal{F}$ from $\mathcal{L}_0^n$ to itself defined by 5 is continuous.[9]

As a result, $\mathcal{F}$ has a least fixed-point given by

$$\mathbf{Y}\mathcal{F} = \mathcal{F}^*(\vec{\perp}) = \sqcup_{k=0}^{\infty} \mathcal{F}^k(\vec{\perp}).$$

Now, since

$$\vec{E}[\mathbf{Y}\mathcal{F}/\vec{s}] = \mathbf{Y}\mathcal{F}$$

$\mathbf{Y}\mathcal{F}$ is the solution of the equation $\vec{s} = \vec{E}$.

Let's take again a small example to illustrate. Consider the single equation:

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

with `leaf` $<$ `tree`, `node` $<$ `tree`. Hence, $\mathcal{F}_{\texttt{tree}}(\vec{\perp})$ = `{leaf}`; then, $\mathcal{F}^2_{\texttt{tree}}(\vec{\perp})$ is given by:

```
{leaf, node(left => leaf; right => leaf}
```

and so $\mathcal{F}^3_{\texttt{tree}}(\vec{\perp})$ is:

---

[9] This is where the fact that $\mathcal{L}_0$ is a complete Brouwerian lattice is important. Indeed, the proof of this proposition uses a characteristic property of these structures.

```
{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 should see now 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 $\mathcal{F}^*_{\texttt{tree}}(\vec{\perp})$ the infinite set (ϵ-type) of all such terms. Hence, solving type equation does give the meaning of recursively defined types.

The reader may wonder at this point how the example given in the beginning of this section on appending two lists is related to computing a *vector* fixed-point. To see this, given a knowledge base KB, we can add a new equation called *query* of the form ? = E, where ? is a special symbol not already in Σ. Then, the *answer* to the query is the component $(\mathbf{Yf})_?$ of the solution of the augmented system.

3.3. Correctness

In order to establish that the fixed-point solution of a 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.

Unfortunately, I have not *(yet)* worked out a complete proof for the correctness theorem. A *"conditional"* proof is obtained if two technical lemmas can be proved. These lemmas make intuitive sense and are extrapolations of similar facts for tree-grammars.[10] I conjecture them for now.

For any $\vec{T}$ in $\pounds_0^n$ define

$$\mathcal{G}(\vec{T}) = \vec{T} \sqcup \mathcal{F}(\vec{T})$$

and

$$\mathcal{G}^*(\vec{T}) = \sqcup_{k=0}^{\infty} \mathcal{G}^k(\vec{T})$$

Then, provided that, for any $\vec{T}_1$, $\vec{T}_2$, $\vec{T}_3$, in $\pounds_0^n$,

Lemma 32: $\vec{T}_1 \xrightarrow[\vec{S}]{*} \vec{T}_2$ *implies* $\vec{T}_2[\vec{T}_3/\vec{S}] \sqsubseteq \vec{T}_1[\mathcal{G}^*(\vec{T}_3)/\vec{S}]$;

and,

Lemma 33: $\vec{T}_2 \sqsubseteq \vec{T}_1[\vec{N}(\vec{S})/\vec{S}]$ *implies* $\vec{T}_1 \xrightarrow[\vec{S}]{*} \vec{T}_2$;

then,

Theorem 34: $\mathbf{Y}\mathcal{F} = \vec{N}(\vec{S})$.

---

[10] See [14], pages 28-29, lemmas 2.38 and 2.39.

4. Conclusion

I have described a syntactic calculus of partially ordered structures and its application to computation. A syntax of record-like terms and a "type subsumption" ordering were 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 was proposed which defines an operational semantics for KBL -- a Knowledge Base Language. It was shown that a KBL program can be seen as a system of equations. Thanks to the lattice properties of finite structures, a system of equations admits a least fixed-point solution. The particular order of computation of KBL, the *"fan-out computation order"*, which rewrites symbols closer to the root first was formally defined and shown to be maximal. Unfortunately, the complete "correctness" of KBL is not yet established. That is, it is not known at this point whether the normal form of a term is equal to the fixed-point solution. However, as steps in this direction, two technical lemmas were conjectured to which a proof of the correctness is corollary.

References

[1] Ait-Kaci, H.
A Lattice Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures.
PhD thesis, Computer and Information Science, University of Pennsylvania, 1984.

[2] Berry, G., and Levy, J.J.
Minimal and Optimal Computations of Recursive Programs.
Journal of the ACM 26:148-75, 1979.

[3] Birkhoff, G.
Colloquium Publications. Volume 25: *Lattice Theory.*
American Mathematical Society, Providence, RI, 1940.
Third (revised) edition, 1979.

[4] Borkin, S.A.
Series in Computer Science. Volume 4: *Data Models: A Semantic Approach for Database Systems.*
The M.I.T. Press, Cambridge, MA, 1980.

[5] Brachman, R.J.
A New Paradigm for Representing Knowledge.
BBN Report 3605, Bolt Beranek and Newman, Cambridge, MA, 1978.

[6] Brachman, R.J.
What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks.
Computer 16(10):30-35, October 1983.

[7] Brouwer, L.E.J.
On Order in the Continuum, and the Relation of Truth to Non-Contradictority.
In *Proceedings of the Section of Sciences 54*, pages 357-358. Koninklijke Nederlandse Akademie Van Wetenschappen, 1951.
Series A, Mathematical Sciences.

[8] Courcelle, B.
Fundamental Properties of Infinite Trees.
Theoretical Computer Science 25:95-169, 1983.

[9] Courcelle, B., and Nivat, M.
The Algebraic Semantics of Recursive Program Schemes.
In J.Winkowski (editor), *Mathematical Foundations of Computer Science Proceedings*, pages 16-30. Springer-Verlag, Berlin, W.Germany, 1978.
Lecture Notes in Computer Science *64*.

[10] Dummett, M.
Elements of Intuitionism.
Oxford University Press, Oxford, UK, 1977.

[11] Goguen, J.A., and Tardo, J.J.
An Introduction to OBJ: a Language for Writing and Testing Formal Algebraic Program Specifications.
In *Proceedings of the IEEE Conference on Specifications of Reliable Software*, pages 170-189. Cambridge, MA, 1979.

[12] Gorn, S.
Explicit Definitions and Linguistic Dominoes.
In J.F. Hart and S. Takasu (editors), *Systems and Computer Science*, pages 77-105. University of Toronto Press, Toronto, Ontario, 1965.

[13] Gorn, S.
Data Representation and Lexical Calculi.
Information Processing & Management 20(1-2):151-174, 1984.
Also available as technical report MS-CIS-82-39, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA.

[14] Guessarian, I.
Lecture Notes in Computer Science. Volume 99: *Algebraic Semantics.*
Springer-Verlag, Berlin, W.Germany, 1981.

[15] Huet, G.
Resolution d'Equations dans des Langages d'Ordre 1, 2, ..., ω.
PhD thesis, Universite de Paris VII, France, September, 1976.

[16] Nivat, M.
On the Interpretation of Recursive Polyadic Program Schemes.
In *Symposia Mathematica*, pages 225-81. Istituto Nazionale di Alta Mathematica, Rome, Italy, 1975.

[17] Plotkin, G.D.
A Powerdomain Construction.
SIAM Journal on Computing 5, 1976.

[18] Plotkin, G.D.
Lattice Theoretic Properties of Subsumption.
Memorandum MIP-R-77, Department of Machine Intelligence and Perception, University of Edinburgh, June, 1977.

[19] Reynolds, J.C.
Transformational Systems and the Algebraic Structure of Atomic Formulas.
In D. Michie (editor), *Machine Intelligence 5*, chapter 7. Edinburgh University Press, 1970.

[20] Robinson, J.A.
A Machine-Oriented Logic Based on the Resolution Principle.
Journal of the ACM 12(1):23-41, 1965.