What is Unification?

A Categorical View of Substitution, Equation and Solution*

Joseph A. Goguen
Programming Research Group, University of Oxford
SRI International, Menlo Park CA 94025
Center for the Study of Language and Information,
Stanford University 94305

Abstract: From a general perspective, a *substitution* is a transformation from one space to another, an *equation* is a pair of such substitutions, and a *solution* to an equation is a substitution that yields the same value when composed with (i.e., when substituted into) the substitutions that constitute the given equation. In some special cases, solutions are called *unifiers*. Other examples include Scott domain equations, unification grammars, type inference, and differential equations. The intuition that the composition of substitutions should be associative when defined, and should have identities, motivates a general concept of *substitution system* based on category theory. Notions of morphism, congruence, and quotient are given for substitution systems, each with the expected properties, and some general cardinality bounds are proved for most general solution sets (which are minimal sets of solutions with the property that any other solution is a substitution instance of one in the set). The notions of equation and solution are also generalized to systems of equations, i.e., to constraint solving, and applied to clarify the notions of "compositionality" and "unification" in linguistic unification grammar. This paper is self-contained as regards category theory, and indeed, could be used as an introductory tutorial on that subject.

1 Introduction

Although certain simple ideas about substitutions, equations and solutions have been implicit in the categorical literature at least since Lawvere's 1963 thesis [31], it has been difficult for computer scientists, linguists, and even mathematicians, to relate this work to their own concerns, and appreciate how simple it really is. This paper provides an introduction to these ideas, particularly developing the applications to unification. Unification has become an important topic in computer science because of its connections to mechanical theorem proving and so-called "logic" programming, and in linguistics because of its connections to so-called "unification grammar." However, there are many other interesting examples.

This paper develops a very general notion of "substitution system" that not only includes algebraic theories in the sense of Lawvere, but also seems able to encompass virtually every kind of equation found in mathematics, computer science, and linguistics. We develop some

*This research was performed at SRI International with the support of Office of Naval Research Contracts N00014-85-C-0417 and N00014-86-C-0450, NSF Grant CCR-8707155, and a gift from the System Development Foundation.

of the general theory of substitution systems, including congruences, quotients, and some sufficient conditions for the existence of most general unifiers. This material can perhaps can be considered the beginnings of a General Unification Theory, extending the classical framework of term unification, as developed for example by Siekmann [45].

Some readers may wish to see some motivation for pursuing a subject that is so very abstract and general. Here are four reasons:

1. Relationships among different applications may be revealed; sometimes these can be surprising and/or suggestive (for example, solving polynomials, inferring types, and understanding natural language can all be seen as solving equations in substitution systems).

2. The need to prove essentially the same results in many different contexts may be avoided (for example, our very general results in Section 6.2 about the uniqueness of most general solution sets apply to all the examples mentioned above).

3. New algorithms and applications may be suggested as extensions of existing special cases (for example, Section 7.5 suggests using order sorted unification to extend Milner's polymorphic type inference to handle subtypes).

4. Old concepts may be simplified, clarified, and/or generalized. For example, Section 5 shows that many inadequacies of traditional syntactic approaches to term substitution are overcome by systematically exploiting the freeness of term algebras, and by making the source and target variable sets explicit. The precise sense in which unification grammars involve unification is clarified in Section 7.8. Other examples include the very general results on floorings in Section 6.2, and on quotients and colimits of arbitrary substitution systems in Section 6.5.

Elegant category theoretic expositions often require sophisticated technical apparatus; since this paper assumes no previous familiarity with category theory, it can reach only a modest level of sophistication. Our discussions often start with informal motivation, so if something seems too vague, please read on until you see it in **boldface**, which marks a formal definition; *italics* are used for informal emphasis. Also, there are many unproved assertions; unless explicitly stated otherwise, they have short proofs, and the reader is urged to use them as exercises to test and improve his/her understanding. Some may find this a relatively painless way to learn some basic category theory. The standard advanced introduction to category theory is Mac Lane [29], while Goldblatt [26] provides a gentler approach that includes topoi; neither book discusses Lawvere theories, but (for example) Schubert [42] does. The only prerequisites for this paper are some elementary set theory and an interest in its subject matter.

I thank Timothy Fernando and Peter Rathmann for their comments on this paper, and Dr. José Meseguer both for his valuable comments and for his collaboration on many of its ideas.

1.1 An Example

Let us consider the simple case of polynomials with integer coefficients, such as $p = x^2 + 2y + 3$. We will use the notation $\{p_1 \Rightarrow x_1, p_2 \Rightarrow x_2, ..., p_n \Rightarrow x_n\}$ for the *substitution* that

substitutes the polynomial p_i for the variable x_i for $i = 1, ..., n$; for example, $f = \{2z \Rightarrow x, 0 \Rightarrow y\}$ substitutes $2z$ for x and 0 for y. Now let[1] "$f;p$" denote the result of applying the substitution f to p, and then simplifying according to the usual laws of polynomial algebra; for example, with f and p as above, we have $f;p = 4z^2 + 3$. We may consider the application $f;p$ to be well defined provided that only the variables of f occur in p (this intuition is further refined in Section 2 below).

Next, given also $g = \{2x \Rightarrow u, y + 1 \Rightarrow z\}$, we can form $g;(f;p) = 4y^2 + 8y + 7$ as well as $g;f = \{2y + 2 \Rightarrow x, 0 \Rightarrow y\}$ and, of course, $(g;f);p = 4y^2 + 8y + 7$. It is no coincidence that $g;(f;p) = (g;f);p$. For uniformity of notation, we can also regard polynomials as substitutions, e.g., $\{x^2 + 2y + 3 \Rightarrow p\}$ for the above p.

If we are also given $\{x^2 + y + z \Rightarrow q\}$, what does it mean to solve the equation $p = q$? It means to find a substitution h such that $h;p = h;q$. There are many, including $h_1 = \{3 \Rightarrow x, -1 \Rightarrow y, 2 \Rightarrow z\}$, $h_2 = \{-5 \Rightarrow x, u - 1 \Rightarrow y, u + 2 \Rightarrow z\}$, $h_3 = \{u \Rightarrow x, v \Rightarrow y, v + 3 \Rightarrow z\}$, and $h_4 = \{1 \Rightarrow x, y \Rightarrow y, y+3 \Rightarrow z, x+y \Rightarrow w\}$. Notice that h_1 and h_2 are both "substitution instances" of h_3, and that h_4 is invalid, because it substitutes for a variable w that doesn't occur in either p or q. Also notice what it means that h_2 is a substitution instance of h_3: there is a substitution j such that $h_2 = j;h_3$, namely $j = \{-5 \Rightarrow u, u - 1 \Rightarrow v\}$. In fact, h_3 is a "most general solution," in the sense that *any* other solution h (that does not substitute for extraneous variables) is a substitution instance of h_3.

This polynomial example is used informally as motivation throughout the paper, and finally developed more formally in Section 6.3.3. But first, we greatly generalize this example, and also clarify our conventions regarding variables in the composition of substitutions.

2 Substitutions and Categories

The most basic intuitions about substitutions concern their *composition*. In particular, composition should satisfy associative and identity laws. Since we wish to capture the idea that there may be *constraints* on what can be substituted where, the composition operation should be *partial* rather than total, reflecting the idea that $f;g$ is defined only when the target type of f, denoted $\partial_1 f$, equals the source type of g, denoted $\partial_0 g$.

Thus, a substitution system should have a set $\mathcal{S}$ of **substitutions**, a set $|\mathcal{S}|$ of **types**, a partial **composition** operation on $\mathcal{S}$ denoted ";" plus **source** and **target** operations denoted $\partial_0, \partial_1 : \mathcal{S} \to |\mathcal{S}|$ such that:

(1) $f;g$ is defined iff $\partial_1 f = \partial_0 g$.

(2) $\partial_0(f;g) = \partial_0 f$ and $\partial_1(f;g) = \partial_1 g$.

(3) $(f;g);h = f;(g;h)$ whenever all compositions are defined.

(4) For each $T \in |\mathcal{S}|$, there is a substitution id_T such that $\partial_0 id_T = \partial_1 id_T = T$ and such that $f;id_T = f$ and $id_T;g = g$ whenever these compositions are defined.

[1] Mathematics traditionally writes composition in the opposite order from that used in this paper, often using the notation "$\circ$". Here, we follow the computer science notation ";" for sequential composition of operations. While it would be an exaggeration to say that mathematics "got it wrong," it does seem fair to say that many things go smoother in the "backwards" notation, and indeed, many category theorists ordinarily use it.

Axiom (4) implies that there is exactly one identity on each object S: any two are equal since $id_S; id'_S = id_S$ and $id_S; id'_S = id'_S$.

Many readers will not be surprised to learn that these axioms define neither more nor less than **categories**, and from here on we will introduce and then use standard bits of terminology and notation from category theory. In particular, we will say "morphism," "map" or "arrow" instead of "substitution," and to say "object" instead of "type." Also, we will say "category" instead of "substitution system," particularly because these axioms are only a first step toward a more complete definition to be given later. Also, we write $f : R \to S$ when $\partial_0 f = R$ and $\partial_1 f = S$, and we let $\mathcal{S}[R, S]$ denote the collection of all maps from R to S in $\mathcal{S}$.

The following discussion will show that even this very simple machinery is sufficient to define some interesting concepts, including isomorphism, substitution instance, and ground substitution.

A morphism $f : S \to T$ in a category $\mathcal{C}$ is an **isomorphism** iff there is another morphism $g : T \to S$ such that $f; g = id_S$ and $g; f = id_T$. We may write f^{-1} for g and g^{-1} for f, and we may also write $S \cong T$, and call f and g **invertible morphisms**. We will see some substitution systems where the isomorphisms are just one-to-one renamings of variables.

Isomorphisms have a number of simple, useful and easily verified properties, including the following, which assume that $f : R \to S$ and $g : S \to T$ are isomorphisms in a category $\mathcal{C}$:

(a) $(f^{-1})^{-1} = f$.

(b) $(f; g)^{-1} = g^{-1}; f^{-1}$.

(c) $id_S^{-1} = id_S$.

(d) $\cong$ is an equivalence relation on $|\mathcal{C}|$.

Let us again consider our polynomial substitution system, which from here on will be denoted $\mathcal{P}$. Its types are finite sets of variables, and if $f : R \to S$, then S contains the variables that f substitutes *into*, while R contains the variables that may be used in the polynomials that are substituted for the variables in S. The constraint on when a composition $f; g$ is defined guarantees that f must tell what happens to all and only the variables that can occur in polynomials in g. The identity on T is the substitution $\{x \Rightarrow x \mid x \in T\}$. It is an interesting exercise to determine the isomorphisms of $\mathcal{P}$, but perhaps it is better to delay this until the details have been developed formally in Section 6.3.3.

Returning now to the general case (but remembering the polynomials for motivation), we say that a substitution $h' : R' \to S$ is an **instance** of a substitution $h : R \to S$ iff there is a substitution $j : R' \to R$ such that $h' = j; h$. This defines a relation on substitutions called **subsumption**, and we write $h' \preceq h$ to indicate that h **subsumes** h'. It is easy to check that subsumption is transitive and reflexive, i.e., a pre-ordering.

Figure 1 shows a so-called *commutative diagram* that renders the definition of substitution instance graphically. In such a diagram, the edges are morphisms in some category $\mathcal{C}$ and the nodes are objects in $\mathcal{C}$. The diagram is said to *commute* iff, given any two nodes

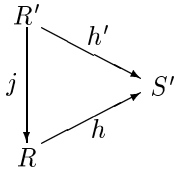


Figure 1: Substitution Instance

R and S, the composition of the morphisms along any two paths from R to S in the diagram are equal as morphisms in $\mathcal{C}$. Such diagrams can often help with visualizing a set of equations among morphisms, or as in this case, substitutions.

Returning again to the polynomial substitution system $\mathcal{P}$, some substitutions are *constants*, in the sense that what is substituted for variables does not involve any variables, i.e., is an integer. We can capture this concept categorically by postulating a "constant type" 1 that is the source for all constants; for $\mathcal{P}$, it is the *empty* set of variables. Then a constant of sort S is just a substitution $c\colon 1 \to S$, e.g., for $\mathcal{P}$, a substitution that involves no variables. But how can we characterize 1? In fact, 1 has a simple categorical property that (we will soon see) characterizes it uniquely up to isomorphism: there is exactly one substitution with target 1 for each possible source object, i.e.,

(5) There is a type $1 \in |\mathcal{S}|$ such that for each $S \in |\mathcal{S}|$, there is one and only one substitution $S \to 1$ in $\mathcal{S}$, which we may denote $!_S$ or even just $!$.

In category theory, an object 1 satisfying axiom (5) is called a **final** or **terminal** object. In $\mathcal{P}$, the arrows $!_S$ are empty functions. The basic fact about final objects is the following:

Proposition 1 Let F and F' be final objects in a category $\mathcal{C}$. Then the unique morphisms $i\colon F \to F'$ and $i'\colon F' \to F$ are inverse isomorphisms.

Proof: First notice that we have $i;i'\colon F \to F$ and $i';i\colon F' \to F'$ in $\mathcal{C}$. Since F and F' are both final, and since there can be only one morphism each $F \to F$ and $F' \to F'$, and since id_F and id'_F are such morphisms, the given compositions must equal these identities. □

Let us call a substitution $f : S \to T$ a **constant** if $S = 1$, and let us call it a **ground substitution** if it factors through 1, i.e., if $f = !;c$, i.e., if $f \preceq c$, for some constant $c\colon 1 \to S$. Then

Fact 2 $f;g;h$ is ground whenever g is ground. The only invertible ground morphisms are the boring ones $!\colon 1 \to F$ where F is another final object. If $h \preceq g$ and g is ground, then h is also ground. □

To summarize, this section develops categories with final objects as a model for substitution systems. Although this model has very few axioms and concepts, it already allows us to define renaming and ground substitutions, as well as subsumption, and to develop some of their basic properties. However, it is only a first step toward the definition of substitution system finally given in Section 4.

3 Equations and Solutions

What is an equation? Our view is that it is just a pair of substitutions with the same source and the same target, i.e., an **equation** is a pair $\langle f, g\rangle$ with $\partial_0 f = \partial_0 g$ and $\partial_1 f = \partial_1 g$; we will often use the notation $f, g : S \to T$. Sections 5 and 6 will show that this definition includes the expected examples, and in particular, that terms can be seen as substitutions.

Given this view of equations, a **solution** to an equation $f, g\colon S \to T$ is just a substitution $h\colon R \to S$, for some type R, such that $h; f = h; g$.

Usually, we prefer a **most general** solution if there is one, i.e., a solution such that any other solution is a substitution instance of it, i.e., a solution $h\colon R \to S$ such that for any solution $h'\colon R' \to S$ there is a "factoring" substitution $j\colon R' \to R$, such that $j; h = h'$.

In many examples, including our polynomial example, a most general solution h can be chosen so that whenever $h' \preceq h$ there is a *unique* j such that $h' = j; h$. Let us say that a substitution h such that $f; h = g; h$ implies $f = g$ is **monic** (another term is **monomorphism**). If h is monic and most general, then the factoring substitution j is necessarily unique. We will soon see that this uniqueness of the factoring substitution guarantees the uniqueness of h (up to isomorphism), which is a deepening of our intuition about most general substitutions. This discussion motivates the following

Definition 3 A **final solution**[2] to an equation $f, g\colon S \to T$ is a substitution $h\colon R \to S$ such that any other solution h' to $\langle f, g\rangle$ is of the form $j; h$ for a *unique* substitution j. □

Recall that a **maximum** element m for a partial order $\preceq$ has the property that $a \preceq m$ for all a.

Fact 4 A solution is most general iff it is maximum with respect to $\preceq$. Moreover, any monic most general solution is final. □

We now prove the uniqueness of final solutions by using abstract nonsense[3]. Although the construction is certainly abstract, it is really rather simple, and is also useful in many other situations. Given $f, g : S \to T$, we construct a category $\mathcal{SOL}(f, g)$ of solutions, in which final solutions are final objects:

- Let $\mathcal{SOL}(f, g)$ have the solutions of $\langle f, g\rangle$ as its objects, i.e., the substitutions $h\colon R \to S$ (for some R) such that $h; f = h; g$.
- Given solutions $h : R \to S$ and $h' : R' \to S$ to $\langle f, g\rangle$, we define a **morphism of solutions** $h' \to h$ to be a substitution $j\colon R' \to R$ such that $j; h = h'$. See Figure 2.

Notice that $h' \preceq h$ iff $h' \to h$. We immediately have

Proposition 5 $\mathcal{SOL}(f, g)$ as defined above is a category, and h is a final solution for $\langle f, g\rangle$ iff it is final in $\mathcal{SOL}(f, g)$. □

[2] I regret the connotations of this phrase, but "terminal solution" seemed almost equally unfortunate, yet less suggestive. I also considered reversing all arrows, so that the phrase "initial solution" could be used, and I might well have done so, had it not required rewriting virtually the entire paper.

[3] This is a sort of technical term in category theory, referring to cases where something is proved without actually looking at how it has been constructed.

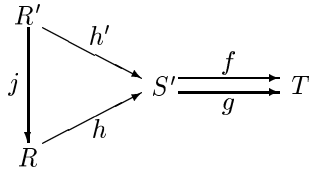


Figure 2: A Morphism of Solutions

Now Proposition 1 implies that final solutions are unique up to isomorphism in $\mathcal{SOL}$, which means in particular that their source types are isomorphic. We cannot, of course, expect that every equation has a solution, let alone a final solution, but it is comforting to know that final solutions are unique when they exist. Section 6.1 will show that the final solution concept includes "most general unifier." In the standard language of category theory, a final solution for $\langle f, g \rangle$ is an **equalizer** of f and g.

A property of the form "there exists a unique morphism such that ..." is called a **universal property**, and we have now seen two of these: the characterizing properties for equalizers and for final objects. We will see other examples soon.

We close this section with an easy generalization of the notion of *equation*: instead of a pair $\langle f, g \rangle$, we may consider any set ? of morphisms all having the same source and the same target. A solution is then a substitution that makes them all equal, and we get a category $\mathcal{SOL}(?)$ of solutions of ?, in which final solutions are again final objects, and therefore unique (up to isomorphism). All our results generalize to this situation, but this will not be explicitly mentioned.

4 Types, Variables and Products

Let us return to the example of Section 1, polynomials with integer coefficients, and examine more closely how it fits into the framework of Sections 2 and 3. Our goal is to define a substitution system $\mathcal{P}$ whose morphisms are the polynomial substitutions, and our first step is to describe its types, the elements of $|\mathcal{P}|$. To this end, let $X_\omega = \{x_0, x_1, x_2, \ldots\}$ and let $|\mathcal{P}|$ be the set of all *finite* subsets of X_ω including the empty subset $\emptyset$. Also, we let u, v, w, x, y, z denote elements of X_ω and call them **variables**.

We (informally) let the morphisms $f : S \to T$ in $\mathcal{P}$ be the substitutions of the form $\{f_x \Rightarrow x \mid x \in T\}$ where f_x is a polynomial with integer coefficients involving only variables from S. Composition $f; g$ is of course substitution of the polynomials in f for the variables in the polynomials of g, and the identity $T \to T$ is $\{x \Rightarrow x \mid x \in T\}$. The final object is the empty type, $\emptyset$, because the only substitution $T \to \emptyset$ is the empty one. The morphisms $\emptyset \to T$ are (tuples of) constants, $\{i_x \Rightarrow x \mid x \in T\}$ with each i_x an integer. (All this will be developed more formally in Section 6.3.3.)

Among the morphisms $S \to T$ in $\mathcal{P}$ are some very simple ones, called **type morphisms**, that are of the form $\{f_x \Rightarrow x \mid x \in T\}$ where $f_x \in S$, i.e., where each f_x is just a variable in

S. It is easy to be convinced that the composition of two type morphisms is again a type morphism, and also that identity morphisms are again type morphisms. Then types with type morphisms form a category, which we will denote by $\mathcal{X}$ from here on.

Now $\mathcal{X}$ is a subcategory of $\mathcal{P}$, in the following sense: A category $\mathcal{C}'$ is a **subcategory** of a category $\mathcal{C}$ if the objects of $\mathcal{C}'$ are also objects of $\mathcal{C}$, and if the morphisms of $\mathcal{C}'$ from R to S are also morphisms of $\mathcal{C}$ from R to S, such that compositions and identities in $\mathcal{C}'$ agree with those in $\mathcal{C}$; let us write $\mathcal{C}' \subseteq \mathcal{C}$. A subcategory $\mathcal{C}'$ of $\mathcal{C}$ is **broad** iff $|\mathcal{C}'| = |\mathcal{C}|$, and is **full** iff $\mathcal{C}'[R,S] = \mathcal{C}[R,S]$ for all $R, S \in |\mathcal{C}'|$.

Fact 6 If $\mathcal{C}' \subseteq \mathcal{C}$ is broad and full, then $\mathcal{C}' = \mathcal{C}$. □

Notice that a type morphism $f : R \to S$ in $\mathcal{P}$ is just an assignment of an element of R to each element of S, i.e., a *function* $S \to R$ that we will denote f^o. Moreover, given type morphisms $f : R \to S$ and $g : S \to T$, their composition $f;g : R \to T$ corresponds to the functional composition $g^o; f^o$, i.e.,

$$(f;g)^o = g^o; f^o.$$

Now let us construct $\mathcal{X}$ quite precisely: its objects (called "types") are the finite subsets of X_ω; its morphisms $f : R \to S$ are functions $f^o : S \to R$; its composition is defined by the formula above; and $id_S = 1_S^o$ where $1_S : S \to S$ is the identity function on the set S.

We could entirely dispense with variables if we wanted, by making them implicit in types. This illustrates an important theme in category theory, that isomorphic objects are "essentially the same." Under this view, we could use *any* finite set as a type, i.e., we could use anything at all as a variable symbol. In particular, instead of the finite subsets of $X_\omega = \{x_0, x_1, x_2, \ldots\}$ we could use the finite sets of integers[4], i.e., the finite subsets of $\omega = \{0, 1, 2, \ldots\}$. In fact, *variables play only an auxiliary role*, as a notational convenience. The real issue, as we will see below, is the product structure of types.

Sometimes it is convenient to use the following "canonical" subsets of X_ω

$$X_n = \{x_0, x_1, x_2, \ldots, x_{n-1}\},$$

with $X_0 = \emptyset$ by convention. Let $\mathcal{N}$ denote the category with objects X_n and with the functions $f^o : X_n \to X_m$ as the arrows $X_m \to X_n$. $\mathcal{N}$ is a full subcategory of $\mathcal{X}$.

The type morphisms in $\mathcal{P}$ can be used to "untuple" substitutions. Given a type T and $x \in T$, let $p_x : T \to \{x\}$ denote the type morphism corresponding to the inclusion function $\{x\} \to T$. Now given $f : S \to T$, the composition $f; p_x$ "pulls out" the x component of f. For example, if $f = \{2x \Rightarrow w, y+1 \Rightarrow x, x+2y \Rightarrow y\}$, then $f; p_x = f_x = \{y+1 \Rightarrow x\}$. It is also very useful to "tuple up" substitutions, e.g., to form $\{2x \Rightarrow w, y+1 \Rightarrow x, x+2y \Rightarrow y\}$ from its three components. Suppose we are given a substitution $f_x : S \to \{x\}$ for each $x \in T$; then what we want is $f : S \to T$ such that $f; p_x = f_x$ for each $x \in T$. As it happens, there is always a *unique* such f for polynomial substitutions, and our intuition suggests that satisfying the equations $f; p_x = f_x$ should completely characterize f in any substitution system.

It is convenient to use language and notation for tupling that is inspired by Cartesian products of sets, which will turn out to be a special case of the general construction given below. Thus, we will call $p_x : T \to \{x\}$ for $x \in T$ the x-**projection** of T; also, we will write

[4]However, to avoid ambiguous notation, it would be necessary to somehow distinguish integers used as variable symbols from integers used as coefficients, for example by enclosing them in brackets.

$\langle f_x \mid x \in T \rangle$ for the construction of f from its components f_x and call it the **tupling** of the f_x (all this is formally defined below).

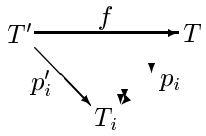


Figure 3: A Morphism of Cones

These considerations motivate a general construction, in the setting of an arbitrary category $\mathcal{C}$, for tupling an arbitrary family of $f_i : S \to T_i$ of morphisms, for $i \in I$. To characterize the tupled morphism $f = \langle f_i \mid i \in I \rangle$, we must also characterize its target T, which is the **product object** of the T_i. We do this through its family of **projections**, which are the basic feature of products:

- Given a family of objects T_i for $i \in I$, let us call a family $p_i : T \to T_i$ of morphisms a **cone** over the T_i.
- Given also another cone over the T_i, say $p'_i : T' \to T_i$, a **cone morphism** $f : T' \to T$ over the T_i is a morphism $f : T' \to T$ such that $f ; p_i = p'_i$ for each $i \in I$. See Figure 3.

We thus obtain a category $\mathcal{C}(T_i)$ of cones over the family T_i, and we define a **product** of the T_i to be a *final object* in this category. The "apex" of this cone is the product object, and its morphisms are the projections. Once again, Proposition 1 gives uniqueness up to isomorphism. Moreover, given a cone $f_i : S \to T_i$ over the T_i, the unique morphism $S \to T$, where T is the apex of the product cone, is the tupled substitution $\langle f_i \mid i \in I \rangle$ of the f_i.

Of course, a given category may fail to have certain products. When $I = \emptyset$, any final object is a product for the empty family. When $I = \{1, 2\}$, we get a **binary** product, and write $T_1 \times T_2$ for its object. For the general case, we use the notation $\prod_{i \in I} T_i$ for the product object.

We will see in the subsection below that, for our polynomial example, the product of types is given by their disjoint union.

Proposition 7 A category has all finite products iff it has all binary products and a final object. In such a category,

- $1 \times T \cong T \times 1 \cong T$ when 1 is final.
- $T \times T' \cong T' \times T$.
- $(T \times T') \times T'' \cong T \times (T' \times T'')$.
- Given a finite set I and T_i for $i \in I$, let $T = \prod_i T_i$. Then each $f : S \to T$ is of the form $\langle f_i \mid i \in I \rangle$ for uniquely determined $f_i : S \to T_i$. In particular, if $p_i : T \to T_i$ are the projections, then

- $f_i = f; p_i$
- $id_T = \langle p_i \mid i \in I \rangle$.

(Only the first two results need a final object.) □

We can summarize our discussion so far with the following: A **type system** $\mathcal{T}$ is a category with finite products, and a **substitution system** is a category $\mathcal{S}$ with finite products having a type system $\mathcal{T}$ as a broad subcategory such that products in $\mathcal{S}$ are the same[5] as products in $\mathcal{T}$. By abuse of notation, we will denote the pair $\langle \mathcal{S}, \mathcal{T} \rangle$ by just $\mathcal{S}$; also, we may call $\mathcal{T}$ the **base** of $\mathcal{S}$. (These definitions are still a bit vague, and will be refined further in Section 6.5.)

The following sections will show that many interesting notions of type and substitution fall within this general framework. (However, the reader is warned that some models of these axioms have morphisms that look nothing like substitutions.)

4.1 Finite Sets and Duality

Let us look more closely at types and their products in our polynomial example. Recall that this type system $\mathcal{X}$ has as its objects the finite subsets of X_ω and that its morphisms $f : S \to T$ are given by functions $f^o : T \to S$, with composition $f; g$ defined by $(f; g)^o = g^o; f^o$.

We can understand $\mathcal{X}$ better by studying the closely related category $\mathcal{FSET}$ of finite sets; this will also shed light on the strange reversal of composition order in the above equation. $\mathcal{FSET}$ has finite sets as its objects and functions as their morphisms. This category (or more precisely, a subcategory of it) is *dual* to $\mathcal{X}$, in the following sense:

Given a category $\mathcal{C}$, its **dual category** $\mathcal{C}^o$ has the same objects as $\mathcal{C}$ (i.e., $|\mathcal{C}| = |\mathcal{C}^o|$), with morphisms $f^o : S \to T$ in $\mathcal{C}^o$ being morphisms $f : T \to S$ in $\mathcal{C}$, with $id^o_T = id_T$ and with $f^o; g^o = (g; f)^o$. In our example, $\mathcal{X}^o$ is the full subcategory of $\mathcal{FSET}$ having as its objects the finite subsets of X_ω. In general, we have

Fact 8 If $\mathcal{C}$ is a category, so is its opposite $\mathcal{C}^o$. Moreover, $\mathcal{C}^{oo} = \mathcal{C}$. □

A very nice feature of category theory is that every categorical concept has a *dual* concept. For example, the dual of final object is **initial object**, an object such that there is a unique morphism *from* it to any other. Similarly, the dual of product is **coproduct**: Given two objects T_1 and T_2, their coproduct in $\mathcal{C}$ consists of an object C and two **injections** $j_i : T_i \to C$ with a co-universal property. We can make this more precise and general by using the category $\mathcal{C}(T_i)$ of cones in $\mathcal{C}$ over a family T_i of objects in $\mathcal{C}$ for $i \in I$: A coproduct of the T_i is a final object in $\mathcal{C}^o(T_i)$, i.e., a product of the T_i in $\mathcal{C}^o$. It now follows that coproducts are determined uniquely up to isomorphism, as usual by Proposition 1.

Epics are dual to monics; specifically, $h : X \to Y$ is an **epimorphism** iff whenever $f, g : Y \to Z$ satisfy $h; f = h; g$ then $f = g$.

Returning to $\mathcal{FSET}$, if we are given disjoint sets T_1 and T_2, then their union $T_1 \cup T_2$ is a coproduct, with the inclusions as injections. More generally, given any finite sets T_1 and

[5] This means that their final objects are the same and their product cones are the same; in particular, their projections are identical.

T_2, their coproduct object in $\mathcal{FSET}$ is their **disjoint union**, which could be any finite set T whose cardinality is the sum of those of T_1 and T_2. One such set is $T = T_1 \times \{1\} \cup T_2 \times \{2\}$, with injections $j_i : T_i \to T$ sending t in T_i to the pair $\langle t, i \rangle$ in T.

An advantage of not being committed to any particular construction for disjoint union is that, since any isomorphic set will do, we are not tied down to any particular choice of variables. In particular, the awkward phenomenon of "renaming away," which appears for example in the usual approaches to narrowing, is entirely avoided.

Returning now to the polynomial substitution system $\mathcal{P}$ and its type system $\mathcal{X}$, we see for example that $\{x_0, x_1, x_2\} = \{x_0\} \times \{x_1\} \times \{x_2\}$, so that we can write

$$\{2x_1 + x_2^2 + x_0 x_1 \Rightarrow x_0\} : \{x_0\} \times \{x_1\} \times \{x_2\} \to \{x_0\},$$

which seems rather neat.

$\mathcal{FSET}$ also has finite products, given by the usual Cartesian product construction, i.e., $T_1 \times T_2 = \{\langle t_1, t_2 \rangle \mid t_1 \in T_1, t_2 \in T_2\}$ with $p_i : T \to T_i$ sending $\langle t_1, t_2 \rangle$ to t_i for $i = 1, 2$. Any one point set is a final object.

5 Classical Term Substitution

Both mathematics and computer science provide many interesting substitution systems besides the polynomial example that we have been using for motivation. This section is devoted to the case of classical terms.

5.1 Terms

We first define terms over a given fixed set of function symbols. Terms appear as a basic data structure in many functional and logic programming languages, for example, OBJ [23, 9, 10, 19], Hope [4], and Prolog [6]. An unsorted (or one sorted) **signature** consists of a set Σ whose elements are called **function symbols** and a function $\alpha : \Sigma \to \omega$ assigning an **arity** to each symbol; $\sigma \in \Sigma$ is a **constant** symbol when $\alpha(\sigma) = 0$. Let Σ_n be $\{\sigma \in \Sigma \mid \alpha(\sigma) = n\}$. Now define the set T_Σ of all Σ-**terms** to be the smallest set of *strings* over the alphabet $\Sigma \cup \{\underline{(}, \underline{)}\}$ (where $\underline{(}$ and $\underline{)}$ are special symbols disjoint from Σ) such that

- $\Sigma_0 \subseteq T_\Sigma$ and
- given $t_1, \ldots, t_n \in T_\Sigma$ and $\sigma \in \Sigma$, then $\sigma\underline{(}t_1 \ldots t_n\underline{)} \in T_\Sigma$.

The discussion below will show that the Σ-terms form a Σ-algebra, and that a simple universal property characterizes this algebra.

A Σ-**algebra** is a set A with a function $A_\sigma : A^n \to A$ for each $\sigma \in \Sigma_n$; note that if $\sigma \in \Sigma_0$, then A_σ is essentially an element of A, since A^0 is a one point set. Given Σ-algebras A and B, a Σ-**homomorphism** $h : A \to B$ is a function h such that

$$h(A_\sigma(a_1, \ldots, a_n)) = B_\sigma(h(a_1), \ldots, h(a_n)).$$

Then Σ-algebras and Σ-homomorphisms form a category, denoted $\mathcal{ALG}_\Sigma$. We now view T_Σ as a Σ-algebra as follows:

- For $\sigma \in \Sigma_0$, let $(T_\Sigma)_\sigma$ be the string σ.
- For $\sigma \in \Sigma_n$, let $(T_\Sigma)_\sigma$ be the function sending $t_1, \ldots, t_n$ to the string $\sigma\underline{(}t_1 \ldots t_n\underline{)}$.

Thus, $\sigma(t_1, \ldots, t_n) = \sigma(t_1 \ldots t_n)$, and from here on we prefer to use the first notation. The key property of T_Σ is its *initiality* in $\mathcal{ALG}_\Sigma$:

Theorem 9 For any Σ-algebra A, there is a unique Σ-homomorphism $T_\Sigma \to A$. $\Box$

Initiality is dual to finality, and the proof of Proposition 1 applies to show that any two initial objects in a category are not only isomorphic, but are isomorphic by a unique homomorphism. In particular, this shows that our use of strings was not essential in T_Σ: any construction giving an isomorphic algebra would be just as good, for example, lists, trees, and many other representations could have been used. The proof of Theorem 9 is by induction over the construction of T_Σ and is not so simple as previous proofs that have been omitted.

5.2 Freebies

We can extend Theorem 9 to a concept called "free algebras" using some tricks on signatures. Given a signature Σ and a set X of elements called "variable symbols," let $\Sigma(X)$ denote the new signature with constants $\Sigma(X)_0 = \Sigma_0 \cup X$ and with $\Sigma(X)_n = \Sigma_n$ for $n > 0$. We can now form the initial $\Sigma(X)$-algebra $T_{\Sigma(X)}$ and then view $T_{\Sigma(X)}$ as a Σ-algebra just by forgetting about the additional constants in X; let us denote this Σ-algebra by $T_\Sigma(X)$, and let us also denote the set inclusion function of X into $T_\Sigma(X)$ by i_X.

Corollary 10 $T_\Sigma(X)$ is the **free Σ-algebra on** X, in the sense that any function $f : X \to A$ from X to a Σ-algebra A extends uniquely to a Σ-homomorphism, denoted $\overline{f} : T_\Sigma(X) \to A$. (See Figure 4.)

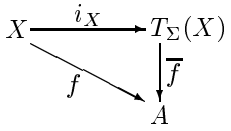


Figure 4: The Free Σ-Algebra on X

Proof: We construct a category $\mathcal{F}_X$ whose objects are functions $f : X \to A$ from X to some Σ-algebra A, and whose morphisms from $f : X \to A$ to $g : X \to B$ are functions $j : A \to B$ such that $f; j = g$. The composition in $\mathcal{F}_X$ of $j : A \to B$ from $f : X \to A$ to $g : X \to B$ with $j' : B \to C$ from $g : X \to B$ to $h : X \to C$ is then just $j; j' : A \to C$. (See Figure 5.)

It suffices to show that $i_X : X \to T_\Sigma(X)$ is initial in $\mathcal{F}_X$, which follows from these observations: (1) objects in $\mathcal{F}_X$ are in one-to-one correspondence with $\Sigma(X)$-algebras, such that (2) $\mathcal{F}_X$-homomorphisms correspond to $\Sigma(X)$-homomorphisms and (3) $i_X : X \to T_\Sigma(X)$ corresponds to $T_{\Sigma(X)}$, and finally, (4) $T_{\Sigma(X)}$ is initial in $\mathcal{ALG}_{\Sigma(X)}$. $\Box$

Given $t \in T_{\Sigma(X)}$, let $var(t)$, the set of variables that occur in t, be the least $Y \subseteq X$ such that $t \in T_{\Sigma(Y)}$.

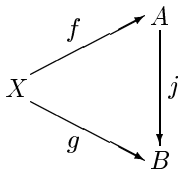


Figure 5: Composition in $\mathcal{F}_X$

5.3 Term Substitution Systems

We are now ready to define a substitution system $\mathcal{S}_\Sigma$ whose morphisms are the substitutions on the terms over a given signature Σ. Using the same base type system $\mathcal{X}$ as in the polynomial example (it has the finite subsets of $X_\omega = \{x_0, x_1, x_2, \ldots\}$ as its objects), we define a **substitution** $X \to Y$ in $\mathcal{S}_\Sigma$ to be a function $Y \to T_\Sigma(X)$. Given substitutions $f\colon Y \to T_\Sigma(X)$ and $g\colon Z \to T_\Sigma(Y)$, we define their **composition** as substitutions, denoted $f;g : X \to Z$, to be the composition $g;\overline{f} : Z \to T_\Sigma(X)$ of functions where $\overline{f} : T_\Sigma(Y) \to T_\Sigma(X)$ is the free extension given by Corollary 10. Finally, we define a base type morphism $f\colon X \to Y$ to be a function $Y \to T_\Sigma(X)$ that factors through the inclusion $i_X\colon X \to T_\Sigma(X)$; thus a base type morphism $X \to Y$ corresponds to a function $Y \to X$, as expected.

It is now fun to prove the associative law for the composition of term substitutions, $f;(g;h) = (f;g);h$, i.e.,

$$h;\overline{g;\overline{f}} = (h;\overline{g});\overline{f},$$

for which it suffices to show that

$$\overline{g;\overline{f}} = \overline{g};\overline{f}$$

because the composition of functions is associative. But this last equation follows directly from the universal property of $T_\Sigma(X)$, as can be seen from contemplating the diagram in Figure 6. Moreover, i_X is clearly the identity on X, and thus $\mathcal{S}_\Sigma$ is indeed a category.

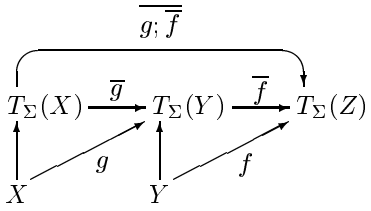


Figure 6: The Associative Law for Substitution

Notice that a term $t \in T_\Sigma(X)$ can appear as a morphism $t\colon Y \to \{x\}$ for any Y containing X and any $x \in X_\omega$. However, if we use the canonical variable sets $X_n = \{x_0, x_1 \ldots, x_{n-1}\}$ then $\sigma \in \Sigma_n$ appears only as $\sigma\colon X_n \to X_1$.

Let us check that $\mathcal{S}_\Sigma$ has products, and that these products agree with those in its base

type system $\mathcal{X}$. First notice that $\emptyset$ is a final object (actually the only final object), since there is a unique function $\emptyset \to T_\Sigma(X)$ for any X. By Proposition 7, it now suffices to construct binary products. In fact, X_{m+n} is a product of X_m and X_n, i.e.,

$$X_m \times X_n \cong X_{m+n}$$

with projection $p\colon X_{m+n} \to X_m$ given by the inclusion function $X_m \to X_{m+n}$ and with projection $p'\colon X_{m+n} \to X_n$ given by the function $X_n \to X_{m+n}$ sending i to $i+m$. We leave this as an exercise.

Now let's look at a simple example.

Example 11 Let $\Sigma_0 = \{0\}, \Sigma_1 = \{s\}, \Sigma_2 = \{+\}$, and $\Sigma_n = \emptyset$ for $n > 2$. Taking the liberty of writing $+$ with infix syntax, two typical Σ-terms are $x_0 + s(x_1 + 0)$ and $s(x_1 + s(0)) + x_2$, and a typical substitution is

$$\{x_0 + s(x_1 + 0) \Rightarrow x_1,\ s(x_1 + s(0)) + x_2 \Rightarrow x_3\}\colon \{x_0, x_1, x_2\} \to \{x_1, x_3\}.$$

□

Traditional syntactic approaches define a substitution to be something like a finite partial function from an infinite set of variables to the set of all terms on those variables; [30] gives a nice technical and historical survey of such approaches, also pointing out some of their difficulties. Our discussion above shows that algebra can greatly ease proofs about substitution; for example, it is simple to prove associativity by exploiting the freeness of term algebras (see Figure 6). Moreover, many technical annoyances about "renaming away" variables, etc., disappear by making source and target variable sets explicit. Finally, many issues are clarified, such as most general unifiers and the sense in which they are unique (again, see [30] for some history and discussion of the problem).

6 What is Unification?

This section contains the technical core of the paper. We first consider classical term unification, which is the inspiration for the whole subject. This is followed by a very general discussion of solution sets, giving useful sufficient conditions for several pleasant properties, including some cardinality bounds. Next, congruences and quotients are developed for substitution systems, motivated by substitution and solution modulo equations. Then we consider families of substitution systems, and give our "official" definition of substitution system. This material can be considered the beginnings of a General Unification Theory. Many sorted algebra and order sorted algebra are also considered.

6.1 Classical Term Unification

Although term unification goes back at least[6] to Herbrand's 1930 thesis [27], it rose to its present importance in computer science because of the close connections with mechanical theorem proving developed by Alan Robinson [39] and systematically exploited in so-called "logic" programming (for example, see [32]).

Using the framework of Section 5, let us define a **most general unifier** of two given terms f, g that only involve variables in $X \subseteq X_\omega$ to be a most general solution of the

[6] According to [45], suggestions can also be found in the notebooks of Emil Post.

equation $f, g : X \to \{x_0\}$. Although most general solutions are not always final, there is always a closely related final solution in $\mathcal{S}_\Sigma$. Following [34], call a substitution $f : X \to Y$ in $\mathcal{S}_\Sigma$ **sober** iff $X = \bigcup_{y \in Y} vars(f_y)$ where $f = \{f_y \mid y \in Y\}$. Then,

Proposition 12 A substitution $h : X \to Y$ in $\mathcal{S}_\Sigma$ is monic iff it is sober. Moreover, if $f, g : Y \to Z$ in $\mathcal{S}_\Sigma$ has a most general solution, then it has a final solution.

Proof: The first assertion is clear from knowing that substitution works by "sticking in" terms, but it seems awkward to prove from the formal definition based on free algebras, and I would be grateful to see a simple proof along those lines.

For the second assertion, given a most general solution $h : X \to Y$ of f, g, let $h' : X' \to Y$ be the substitution with X' containing exactly the variables that actually occur in the components h_y of h, and with $h'_y = h_y$ for $y \in Y$. Then h' is most general because h was, and it is monic because it is sober. Fact 4 now implies that h' is final. □

The most important result about classical term unification, which may be called the **Herbrand-Robinson Theorem**, can be stated as follows:

Theorem 13 If a pair $\langle f, g \rangle$ has a solution, then it has a final solution. □

A **unification algorithm** finds a most general unifier when there is one, and the hard part of the traditional proofs of the Herbrand-Robinson theorem is to show that some unification algorithm necessarily terminates. A very nice categorical formulation of a unification algorithm is given by Rydeheard and Burstall [40], who also sketch a largely categorical proof of its correctness, and explain the obstacles that they found to a purely categorical proof. Meseguer, Goguen and Smolka [34] develop general categorical results about unification for application to the order sorted case. The results in this and the next subsection somewhat improve those, but unfortunately are still not quite sufficient to prove the classical Herbrand-Robinson Theorem.

I realized that most general unifiers can be seen as equalizers in 1972, but have only recently appreciated what could be done with the viewpoint implicit in this observation; this mostly arose from working on [34], and from reading [40] and [45]. The usual definitions of unifier are (in my view) much too syntactic, since they depend on a notion of substitution that is too syntactic, e.g., an endo-function on the set of all terms in all variables, as in Schmidt-Schauss [41]. Taking too syntactic a view of substitution can also give misleading results. For example, Walther [51] showed that most general unifiers only exist in order sorted algebra when the subsort graph is a forest; however, this result only holds for his notion of unifier, and is perhaps best seen as demonstrating the inadequacy of that notion; see [34] for further discussion of this point. Another advantage of the categorical view is that it generalizes very naturally, as demonstrated below.

6.2 Floorings and Solution Sets

We can summarize and generalize the above discussion by saying that a unifier is a solution in some substitution system, and a most general unifier is a most general solution. However, there are many examples beyond classical term unification that do *not* have most general

unifiers in this simple sense, including our ongoing integer polynomial example[7]. Such examples require a more general notion of most general solution. In keeping with our whole viewpoint and development, it is convenient to do this by generalizing the notion of final object. This material, which extends and clarifies aspects of [34], is somewhat more difficult than what we have seen so far, and some readers may prefer to skip it the first time through.

Definition 14 A **flooring**[8] in a category $\mathcal{C}$ is a set $\mathcal{F}$ of objects of $\mathcal{C}$ such that for each $C \in |\mathcal{C}|$ there is some $F \in \mathcal{F}$ with a morphism $C \to F$ in $\mathcal{C}$. A **minimal flooring** is a flooring such that no proper subset is a flooring. A minimal flooring $\mathcal{F}$ is **uniquely minimal** if for each $C \in |\mathcal{C}|$ and each $F \in \mathcal{F}$, there is *at most one* morphism $C \to F$ in $\mathcal{C}$. Then a **solution set** for an equation $\langle f, g \rangle$ is a flooring in $\mathcal{SOL}(f, g)$, a **most general solution set** is a minimal flooring in $\mathcal{SOL}(f, g)$, and a **final solution set** is a uniquely minimal flooring in $\mathcal{SOL}(f, g)$. □

The terminology of Mac Lane [29] would say *weak final set* instead of flooring.

Lemma 15 A flooring $\mathcal{F}$ is minimal iff whenever there is a morphism $F \to F'$ in $\mathcal{C}$ for $F, F' \in \mathcal{F}$, then $F = F'$.

Proof: We first show by contradiction that the first condition implies the second. If $F \to F'$ for $F, F' \in \mathcal{F}$ with $F \neq F'$, then $\mathcal{F} - \{F'\}$ is also a flooring, contradicting the minimality of $\mathcal{F}$. Conversely, let us assume of $\mathcal{F}$ that $F \to F'$ in $\mathcal{C}$ implies that $F = F'$, and also assume that $\mathcal{F} - \{G\}$ is a flooring for some $G \in \mathcal{F}$. Then there is some $H \in \mathcal{F} - \{G\}$ such that $G \to H$. Therefore $G = H$, another contradiction. □

Not every flooring contains a minimal subflooring; however, any finite flooring certainly has a minimal subflooring. Moreover,

Proposition 16 If a category $\mathcal{C}$ has a minimal flooring $\mathcal{F}$, then any flooring $\mathcal{G}$ contains a subset $\mathcal{G}'$ that is a minimal flooring, and moreover, there is an isomorphism $m : \mathcal{F} \to \mathcal{G}'$ such that there are morphisms $F \to m(F)$ and $m(F) \to F$ in $\mathcal{C}$.

Proof: Since $\mathcal{G}$ is a flooring, for each $F \in \mathcal{F}$ there is some $G \in \mathcal{G}$ such that $F \to G$ in $\mathcal{C}$; let $G = m(F)$ and let $\mathcal{G}' = \{m(F) \mid F \in \mathcal{F}\}$. Then $m : \mathcal{F} \to \mathcal{G}'$ is surjective and $\mathcal{G}'$ is also a flooring. Since $\mathcal{F}$ is a flooring, for each $G \in \mathcal{G}'$, there is some $F \in \mathcal{F}$ such that $G \to F$; let $F = m'(G)$. Then we have $F \to m(F) \to m'(m(F))$ and so minimality of $\mathcal{F}$ implies that $m'(m(F)) = F$. Therefore m is also injective, and hence an isomorphism.

To show the minimality of $\mathcal{G}'$, let us assume some morphism $m(F) \to m(F')$. Then injectivity of m implies that $F = F'$, and so we get minimality by Lemma 15. □

The following says that any two minimal floorings are essentially the same; in particular, they have the same cardinality.

Corollary 17 If $\mathcal{F}$ and $\mathcal{G}$ are both minimal floorings of $\mathcal{C}$, then there is an isomorphism $m : \mathcal{F} \to \mathcal{G}$ such that $F \to m(F)$ and $m(F) \to F$ for each $F \in \mathcal{F}$. □

[7] For example, consider the equation $x^2 = 4$.

[8] This term is intended to suggest the dual of "covering," since "cocovering" and "overing" seem unsuitable.

We can use this result to classify equations $\langle f, g\rangle$ by the cardinality of a most general solution set $\mathcal{F}$ as follows: the equation is **unitary** if $\mathcal{F}$ has exactly one element (i.e., it has a most general solution); it is **finitary** if $\mathcal{F}$ is finite[9]; and it is **infinitary** if $\mathcal{F}$ is infinite. Then, a substitution system $\mathcal{S}$ is: **unitary** if every solvable equation is unitary; is **finitary** if every solvable equation is finitary; is **infinitary** if every solvable equation is either finitary or infinitary, and some equation is infinitary; and otherwise, is **nullary** (i.e., some equation does not have a most general solution set). Siekmann [45] gives this classification for classical term unification modulo equations, and the present discussion lifts it to General Unification Theory; Adachi [1] applies similar results in a similar way.

A nullary substitution system is given in Section 6.3.4 below. However, most substitution systems of practical interest are not nullary, by the following proposition. Given a category $\mathcal{C}$, let $C < C'$ mean that $C \to C'$ but not $C' \to C$. Then call $\mathcal{C}$ **Noetherian** if there is no infinite ascending chain $C_1 < C_2 < \ldots$ in $\mathcal{C}$. An object M is **maximal** with respect to $<$ iff there is no object M' with $M < M'$.

Proposition 18 Any Noetherian category has a minimal flooring.

Proof: Let $\mathcal{M}$ be the set of all maximal objects in $\mathcal{C}$, and let $C \in |\mathcal{C}|$. If C is not maximal, then there is some C_1 with $C < C_1$; if C_1 is not maximal, there is some C_2 with $C_1 < C_2$; etc. In this way, either $C < M$ for some $M \in \mathcal{M}$, or else we get an infinite ascending chain, contradicting the Noetherian assumption. Thus, every $C \in |\mathcal{C}|$ is $<$ some $M \in \mathcal{M}$. Therefore, $\mathcal{M}$ is a flooring of $\mathcal{C}$, and it is minimal by construction. □

Recall that $f \preceq g$ means that $f = j; g$ for some $j \in \mathcal{C}$, and let $f \prec g$ mean that $f \preceq g$ but not $g \preceq f$. Now define $\mathcal{C}$ to be **finite factoring** if there is no infinite ascending chain $f_1 \prec f_2 \prec \ldots$, and notice that a substitution system $\mathcal{S}$ is finite factoring iff each $\mathcal{SOL}(f, g)$ is Noetherian.

Corollary 19 No finite factoring substitution system $\mathcal{S}$ is nullary. □

Uniquely minimal flooring might seem a very specialized concept, but actually it is not. We already know that final objects and most general term unifiers are special cases. Moreover, in the category with all fields as objects and field homomorphisms as morphisms, the fields $\mathcal{Z}_p$ for $p \geq 0$ (with $\mathcal{Z}_0$ the rational numbers) are an initial covering (the dual concept to uniquely minimal flooring). Let us call a substitution system $\mathcal{S}$ **mono solution factoring** iff any solution h of any pair f, g has a factorization $e; m$ with m monic in $\mathcal{S}$ and a solution of f, g. Then

Proposition 20 In a mono solution factoring substitution system, if $\langle f, g\rangle$ has a most general solution set, then it has a final solution set.

Proof: First, notice that if $\mathcal{F}$ is a most general solution set for f, g, i.e., a minimal flooring of $\mathcal{SOL}(f, g)$, and if each $F \in \mathcal{F}$ is monic, then $\mathcal{F}$ is uniquely minimal. For, if there were some solution C with $a, b\colon C \to F$ (for $F \in \mathcal{F}$) with $a; F = b; F = c$ then $a = b$ since F is monic.

[9] Notice that unitary equations are also finitary.

Now let $\{F_i \mid i \in I\}$ be a most general solution set, and let $F_i = E_i; M_i$ be a factorization for each F_i. Then $\{M_i \mid i \in I\}$ is a solution set because $\{F_i \mid i \in I\}$ is. Since there is a most general solution set, Proposition 16 gives us a subflooring of $\{M_i \mid i \in I\}$ that is monic, minimal, and therefore uniquely minimal. □

For example, since $\mathcal{S}_\Sigma$ is mono solution factoring by the proof of Proposition 12, and is easily seen to be finite factoring, it follows from Propositions 20 and 18 that every solvable equation in $\mathcal{S}_\Sigma$ has a final solution set. In order to prove the Herbrand-Robinson Theorem, it would now suffice to show that each $\mathcal{SOL}(f, g)$ has upper bounds with respect to $\preceq$. Unfortunately, I do not know any simple abstract algebraic proof of this fact.

6.3 Substitution Modulo Equations

Many interesting examples arise by imposing a fixed set $\mathcal{E}$ of equations on the function symbols in Σ, and then considering other equations *modulo* $\mathcal{E}$. In particular, our ongoing polynomial example, and so-called AC (for associative-commutative) unification, both arise this way. We will let the given set $\mathcal{E}$ of Σ-equations define a congruence $\equiv_\mathcal{E}$ on $\mathcal{S}_\Sigma$, and then the substitution system that we want will appear as the quotient $\mathcal{S}_{\Sigma,\mathcal{E}}$ of $\mathcal{S}_\Sigma$ by $\equiv_\mathcal{E}$.

The definitions are really no more complex for the general case. Given a substitution system $\mathcal{S}$, a **substitution congruence** on $\mathcal{S}$ is a family $\{\equiv_{R,S} \mid R, S \in |\mathcal{S}|\}$ of equivalence relations on the morphism sets $\mathcal{S}[R, S]$ of $\mathcal{S}$, such that

(1) $f \equiv_{R,S} f'$ and $g \equiv_{S,T} g'$ imply $f; g \equiv_{R,T} f'; g'$ (if the compositions are defined), and

(2) $f; p_i \equiv_{R,S_i} g; p_i$ for $f, g\colon R \to S$ with $S = \prod_i S_i$ where $p_i\colon S \to S_i$ are its projections, implies $f \equiv_{R,S} g$ (if the compositions are defined).

Then the **quotient substitution system** $\mathcal{S}/\equiv$ has the same objects $|\mathcal{S}|$ as $\mathcal{S}$ (and as its base type system $\mathcal{T}$), and has for its morphisms in $\mathcal{S}/\equiv [R, S]$ the equivalence classes $[f]$ of the morphisms f in $\mathcal{S}[R, S]$, with their composition defined by

(3) $[f]; [g] = [f; g]$.

We must now show that this gives a substitution system, and in particular, that

Proposition 21 $\mathcal{S}/\equiv$ is a category with finite products.

Proof: Let $\mathcal{Q}$ denote $\mathcal{S}/\equiv$ in this proof. That the composition defined by (3) is really well-defined follows from (1), and is left to the reader, who should also check that $\mathcal{Q}$ has identities. Next, we check that $\mathcal{Q}$ has a final object. Since each $\mathcal{S}[S, 1]$ has just one element, namely !, each $\mathcal{S}/\equiv [S, 1]$ also has just one element, namely $[!] = \{!\}$. Using Proposition 7, it suffices to show that $\mathcal{Q}$ has binary products. Let T_1, T_2 be objects, and let T be their product in $\mathcal{S}$, with projections p_1, p_2; then T is also their product in $\mathcal{S}/\equiv$, with projections $[p_1], [p_2]$. For, if $[q_i]\colon S \to T_i$ (for $i = 1, 2$) is another cone in $\mathcal{Q}$, then $q_i\colon S \to T_i$ is a cone in $\mathcal{S}$, and so we get $q = \langle q_i \mid i \in \{1, 2\}\rangle\colon S \to T$ such that $q; p_i = q_i$. Therefore $[q]$ satisfies $[q]; [p_i] = [p_i]$ by (3), and the uniqueness of $[q]$ satisfying these two equations follows directly from (2). □

Now let $\mathcal{T}'$ be the subcategory of $\mathcal{S}/\equiv$ with its objects the same as those of $\mathcal{S}$, and with its morphisms the $\equiv$-classes in $\mathcal{S}$ of morphisms from $\mathcal{T}$, i.e., with $\mathcal{T}'[R,S] = \{[f] \mid f \in \mathcal{T}[R,S]\}$. Then $\mathcal{T}'$ is a broad subcategory of $\mathcal{S}/\equiv$, and $\mathcal{S}/\equiv$ and $\mathcal{T}'$ together form a substitution system that we will denote $\mathcal{S}/\equiv$. Let us call $\equiv$ **non-degenerate** iff no two distinct morphisms of $\mathcal{T}$ are identified by $\equiv$. If $\equiv$ is non-degenerate, then $\mathcal{S}/\equiv$ can be seen as still having the base type system $\mathcal{T}$. All the standard examples are non-degenerate.

The lemma below shows that any family of relations generates a congruence. We will use this result to define the congruence $\equiv_{\mathcal{E}}$ generated by a set $\mathcal{E}$ of Σ-equations. Then we let $\mathcal{S}_{\Sigma,\mathcal{E}}$ be $\mathcal{S}_\Sigma/\equiv_{\mathcal{E}}$.

Lemma 22 Given a family $\{R_{S,T} \mid S,T \in |\mathcal{S}|\}$ of relations on the morphism sets $\mathcal{S}[S,T]$ of a substitution system $\mathcal{S}$, there is a *least* substitution congruence $\equiv$ on $\mathcal{S}$ such that $R_{S,T} \subseteq \equiv_{S,T}$ called the congruence **generated** by $R_{S,T}$.

Proof: It is well-known that the intersection of equivalence relations is an equivalence relation, and it is easy to check that the intersection of any family of equivalence relations satisfying (1) also satisfies (1), and of equivalence relations satisfying (2) also satisfies (2). Moreover, since the family of relations each of which identifies everything, is certainly a substitution congruence, there is at least one substitution congruence containing the given relation $R_{S,T}$. Therefore, it makes sense to take the intersection of *all* substitution congruences containing the given relation family $R_{S,T}$ and the result is guaranteed to be the least substitution congruence containing $R_{S,T}$. □

So now, we only need to define the relation family $R_{S,T}$ generated by $\mathcal{E}$ to get the substitution congruence $\equiv_{\mathcal{E}}$ generated by $\mathcal{E}$. In fact, we let each $R_{S,T}$ be the set of pairs $\langle f,g\rangle$ such that $f,g : S \to T$ is an equation in $\mathcal{E}$. Then $\mathcal{S}_{\Sigma,\mathcal{E}} = \mathcal{S}_\Sigma/\equiv_{\mathcal{E}}$. The next few subsections give examples of this construction.

6.3.1 Associative Unification

This example assumes a signature Σ containing a binary function symbol, say $*$, satisfying the associative equation,

$$x * (y * z) = (x * y) * z \ ,$$

where we have used infix notation for clarity. Plotkin [38] shows that this substitution system theory is infinitary, i.e., all solvable equations have a most general solution set, but some solvable equation has an infinite most general solution set. For example, if Σ also has a binary function symbol f, then the two terms

$$x * f(a,b)$$
$$f(y,b) * x$$

have the following infinite most general solution set,

$$\{f(a,b) \Rightarrow x, a \Rightarrow y\}$$
$$\{f(a,b) * f(a,b) \Rightarrow x, a \Rightarrow y\}$$
$$\{f(a,b) * (f(a,b) * f(a,b)) \Rightarrow x, a \Rightarrow y\}$$
$$\ldots\ldots$$

[38] also gives a unification algorithm for this case.

6.3.2 AC Unification

This case assumes that Σ contains a binary function symbol $*$ satisfying not only the associative equation, but also the commutative equation,

$$x * y = y * x \ .$$

This system is finitary, and Stickel [49] and others have given unification algorithms for it.

6.3.3 Polynomials

Our ongoing polynomial example can now be developed rigorously, by giving a signature and some equations that define polynomials. Here both the signature and the equation set are infinite. Let $\Sigma_0 = \mathcal{Z}$ (the integers), $\Sigma_1 = \emptyset$, $\Sigma_2 = \{+, *\}$, and $\Sigma_n = \emptyset$ for $n > 2$, where $+$ and $*$ are both AC, with the following additional equations

$$x + 0 = x$$
$$x * 0 = 0$$
$$x * (y + z) = (x * y) + (x * z)$$

plus an infinite number of equations that are the complete addition and multiplication tables for $\mathcal{Z}$, including for example, $2 + 2 = 4$ and $329 * -13 = -4277$. (There are various ways to do the same job with a finite signature and equation set, but the above approach seems more amusing.)

6.3.4 A Nullary Example

The following example is due to Fages: let $\Sigma_0 = \{1, a\}, \Sigma_1 = \{g\}, \Sigma_2 = \{*\}$, $\Sigma_n = \emptyset$ for $n > 2$, and let $\mathcal{E}$ contain the equations

$$1 * x = x$$
$$g(y * x) = g(x) \ .$$

Then each of the following is a solution to the equation $g(x) = g(a)$,

$$\{a \Rightarrow x\}$$
$$\{y_1 * a \Rightarrow x\}$$
$$\{(y_2 * y_1) * a \Rightarrow x\}$$
$$\ldots\ldots$$

and since each is a substitution instance of the one below, there is no most general solution, i.e., no maximum element, and $\mathcal{SOL}(g(x), g(a))$ is not Noetherian. Moreover, in considering Corollary 19, notice that $\mathcal{S}_{\Sigma,\mathcal{E}}$ is not finite factoring, since we have

$$\{a \Rightarrow x\} \prec \{y_1 * a \Rightarrow x\} \prec \{(y_2 * y_1) * a \Rightarrow x\} \prec \ \ldots .$$

6.4 Lawvere Theories

All the above examples are *Lawvere theories*. Lawvere's original formulation is perhaps less intuitive but more elegant: substitution systems with base type system $\mathcal{N}$. But the choice of base system is inessential, since any substitution system with base $\mathcal{X}$ has an equivalent substitution system with base $\mathcal{N}$ (Mac Lane [29], page 91 gives the precise definition of equivalence for categories); moreover, this system is unique up to isomorphism.

Although it took us a rather long journey to construct the systems $\mathcal{S}_{\Sigma,\mathcal{E}}$, what is wonderful is that, in a certain sense, this journey was unnecessary, by Theorem 23 below. But first, we must define functor and isomorphism of categories.

A functor is essentially a morphism of categories. Given categories $\mathcal{C}$ and $\mathcal{C}'$, a **functor** $F : \mathcal{C} \to \mathcal{C}'$ consists of a function $|F| : |\mathcal{C}| \to |\mathcal{C}'|$ plus for each $A, B \in |\mathcal{C}|$, a function $F_{A,B} : \mathcal{C}[A, B] \to \mathcal{C}'[|F|(A), |F|(B)]$ such that

1. $F(id_A) = id_{|F|(A)}$ and
2. $F(f;g) = F(f); F(g)$ whenever $f;g$ is defined.

The composition of $F : \mathcal{C} \to \mathcal{C}'$ with $G : \mathcal{C}' \to \mathcal{C}''$ is defined the obvious way, and turns out to be associative and to have identities $id_{\mathcal{C}}$ given by $|id_{\mathcal{C}}|(A) = A$ and $(id_{\mathcal{C}})_{A,B}(f) = f$, for any category $\mathcal{C}$. Thus (ignoring any possible difficulties with set theoretic foundations, as is now quite usual), we get a category $\mathcal{CAT}$ of all categories and functors. An **isomorphism of categories** is an isomorphism in $\mathcal{CAT}$, which turns out to be just a functor $F : \mathcal{C} \to \mathcal{C}'$ with $|F|$ and each $F_{A,B}$ an isomorphism.

Theorem 23 Any substitution system with base $\mathcal{X}$ is isomorphic to some $\mathcal{S}_{\Sigma,\mathcal{E}}$. □

This is one of the main results about Lawvere theories; it is not trivial [31, 42].

6.5 Categories and Algebras of Substitution Systems

Actually, our definition of substitution system is not quite right, since the category of substitution systems with a given base type system does not have all coequalizers; the correct definition generalizes the inclusion of the base type system to a finite product preserving functor, and we will see that everything goes at least as smoothly as before.

Let us be more precise. If $\mathcal{C}$ and $\mathcal{C}'$ are categories with finite products (i.e., type systems), then a functor $F : \mathcal{C} \to \mathcal{C}'$ is **finite product preserving** if 1 is final in $\mathcal{C}'$ implies that $|F|(1)$ is final in $\mathcal{C}$, and if $\{f_i \mid i \in I\}$ is a product cone in $\mathcal{C}$ implies that $\{F(f_i) \mid i \in I\}$ is a product cone in $\mathcal{C}'$. Moreover, $F : \mathcal{C} \to \mathcal{C}'$ is **broad** if $|F|$ is surjective.

Definition 24 Let $\mathcal{B}$ be a category with finite products. Then a **substitution system** with **base** $\mathcal{B}$ is a broad finite product preserving functor $S : \mathcal{B} \to \mathcal{S}$, and a **morphism** of substitution systems with base $\mathcal{B}$, from $S : \mathcal{B} \to \mathcal{S}$ to $S' : \mathcal{B} \to \mathcal{S}'$, is a product preserving functor $F : \mathcal{S} \to \mathcal{S}'$ such that $F; S' = S$. □

This definition gives a category $\mathcal{SUBS}_{\mathcal{B}}$ of substitution systems with base $\mathcal{B}$ that contains the degenerate substitution systems, and from here on, Definition 24 is the "official" notion of substitution system. In particular, $\mathcal{SUBS}_{\mathcal{N}}$ is the category $\mathcal{LAW}$ of Lawvere theories. Moreover, these categories are *cocomplete*, i.e.,

Proposition 25 The categories $\mathcal{SUBS}_{\mathcal{B}}$ of substitution systems have all coproducts and coequalizers. □

This follows by general nonsense, using the fact that the categories in question are comma categories (see [29] or [17] for this material; [17] also gives some computer science examples). The identity functor $\mathcal{B} \to \mathcal{B}$ is of course initial in $\mathcal{SUBS}_{\mathcal{B}}$.

It is worth noticing that substitution systems could equally well have been developed in a dual manner, as categories with finite coproducts and an initial object.

Given a substitution system $S: \mathcal{B} \to \mathcal{S}$, an S-**algebra** is a finite product preserving functor $A: \mathcal{S} \to \mathcal{SET}$, and an S-**homomorphism** is a natural transformation between two such functors. This gives rise to a category $\mathcal{ALG}_S$ of S-algebras and S-homomorphisms, which is a subcategory of a functor category.

6.6 Many Sorted Algebra

Many sorted algebra assumes a set S of **sorts**, which are used to restrict the arguments and values of function symbols. This subsection will use notation introduced in lectures that I gave at the University of Chicago in 1969. It was once thought that many sorted algebra could be developed in the framework of classical Lawvere theories, but this is not so. Some generalization is required, and the way of viewing many sorted substitution systems as functors was worked out at IBM Research in 1972. Essentially, we will follow the same path as for $\mathcal{S}_{\Sigma,\mathcal{E}}$ in Section 6.3, but using S-indexed families in the base category instead of just sets. (Similar work was done even earlier by Benabou [3].)

Let $\mathcal{X}^s_\omega = \{x^s_0, x^s_1, x^s_2, \ldots\}$ be an infinite set of "variable symbols" for each $s \in S$, and let $\mathcal{X}_S$ be the type system whose objects are S-indexed families $\{X_s \mid s \in S\}$ of subsets X_s of X^s_ω and whose morphisms $\{X_s \mid s \in S\} \to \{X'_s \mid s \in S\}$ are S-indexed families $\{f_s: X_s \to X'_s \mid s \in S\}$ of functions. (It is an exercise to check that $\mathcal{X}_S$ is a category with finite products.)

An S-**sorted signature** is a set Σ with an **arity** function $\alpha: \Sigma \to S^* \times S$; now let $\Sigma_{w,s} = \{\sigma \in \Sigma \mid \alpha(\sigma) = \langle w, s\rangle\}$. The intuition is that if $w = s_1 \ldots s_n$ then $f \in \Sigma_{w,s}$ takes n arguments of sorts $s_1, \ldots, s_n$ and yields a value of sort s. We now extend the initial algebra construction of Section 5.1 to the many sorted context, defining the sets $T_{\Sigma,s}$ of all Σ-**terms** of sort s to be the smallest S-sorted family of sets of strings over the alphabet $\Sigma \cup \{\underline{(}, \underline{)}\}$ such that

- $\Sigma_{\lambda,s} \subseteq T_{\Sigma,s}$ and
- given $w = s_1 ... s_n$, $t_i \in T_{\Sigma,s_i}$ and $\sigma \in \Sigma_{w,s}$ then $\sigma\underline{(}t_1 ... t_n\underline{)} \in T_{\Sigma,s}$.

As in Section 5.1, these sets of terms form an algebra, and it is again characterized by initiality. To make this more precise, we need the following: Given a many sorted signature Σ, a Σ-**algebra** A consists of an S-indexed family $\{A_s \mid s \in S\}$ of sets and a function $A_\sigma: A_w \to A_s$ for each $\sigma \in \Sigma_{w,s}$ where $A_{s_1 \ldots s_n} = A_{s_1} \times \ldots \times A_{s_n}$ and where in particular, A_λ is some one point set $\{*\}$, so that $A_\sigma: \{*\} \to A_s$ for $\sigma \in \Sigma_{\lambda,s}$ gives a constant of sort s. Also, a Σ-**homomorphism** $h: A \to B$ is an S-indexed family $\{h_s: A_s \to B_s \mid s \in S\}$ such that

$$h_s(A_\sigma(a_1, ..., a_n)) = B_\sigma(h_{s_1}(a_1), ..., h_{s_n}(a_n))$$

for each $s \in S$, where $\sigma \in \Sigma_{w,s}$ with $w = s_1 ... s_n$ and $a_i \in A_{s_i}$ for $i = 1, ..., n$. Then we get, as before, a category $\mathcal{ALG}_\Sigma$ of Σ-algebras and Σ-homomorphisms. Also just as in Section 5.1, we can make T_Σ into a Σ-algebra, and it is initial in $\mathcal{ALG}_\Sigma$.

The next step is to define the **free** Σ-algebra $T_\Sigma(X)$, for X an S-indexed set, to be $T_{\Sigma(X)}$ viewed as a Σ-algebra, where $\Sigma(X)_{\lambda,s} = \Sigma_{\lambda,s} \cup X_s$ for $s \in S$, and $\Sigma(X)_{w,s} = \Sigma_{w,s}$

for $w \neq \lambda$. The proof of Corollary 10 works to show the same unique extension universal property.

Analogous to Section 5.3, given X and Y in $\mathcal{X}_S$ let us define a substitution $X \to Y$ to be an S-indexed function $f : T \to T_\Sigma(X)$, and we define composition and identity just as before, getting a substitution system that we will again denote $\mathcal{S}_\Sigma$. Using the results of Section 6.3, we can generate a congruence from a given set $\mathcal{E}$ of equations, and then form the quotient substitution system.

We can also argue that since $\mathcal{S}_\Sigma$ with Σ many sorted, is mono solution factoring and finite factoring, solvable equations have final solution sets. (In fact, this substitution system, like the unsorted case, is unitary, but our very general tools are not quite strong enough to prove it.)

6.7 Order Sorted Algebra

Order sorted algebra and order sorted unification can also be developed using substitution systems. Order sorted algebra was created in 1978 [15] to solve an outstanding problem in algebraic specification: it is difficult to deal with exceptions and partially defined functions using many sorted algebra; the difficulties are spelled out in some detail in [24]. The 1978 approach to order sorted algebra accomplished this goal, but was more complex than necessary. [22] gives a thorough development of basic theory for our current approach, while [34] discuss unification; [47] gives a slightly different approach. [18] gives details about the operational semantics that is the basis for the OBJ2 programming language [9, 10], and [28] does the same for OBJ3. Order sorted unification has interesting applications to knowledge representation with *is-a* hierarchies, and to polymorphic type inference where there are subtypes.

Three essential ideas of order sorted algebra are:

1. Impose a partial ordering $\leq$ on the set S of sorts.

2. Define an S-indexed family $\{F_s \mid s \in S\}$ where S is a poset, to be a family of sets such that $s \leq s'$ implies $F_s \subseteq F_{s'}$.

3. Define an S-sorted signature[10] to be a family $\{\Sigma_{w,s} \mid w \in S^*, s \in S\}$ satisfying the condition that $s \leq s'$ implies $\Sigma_{w,s} \subseteq \Sigma_{w,s'}$.

Believe it or not, the exposition given in Section 6.6 goes through for this generalization without any change at all, except that signatures must satisfy a technical condition called **regularity** [22] to insure that terms always have a well-defined *least parse*:

> given $w_0 \leq w$ and $\sigma \in \Sigma_{w,s}$ there must be some $\overline{w} \in S^*$ and $\overline{s} \in S$ such that $\sigma \in \Sigma_{\overline{w},\overline{s}}$ and $w_0 \leq \overline{w}$ such that $\sigma \in \Sigma_{w',s'}$ and $w_0 \leq w'$ implies $\overline{w} \leq w'$ and $s \leq s'$.

Additional technical conditions are needed for some results, for example, that every connected component of S has a maximum element [22, 34]. In particular, we can have unification for order sorted algebra modulo equations. [34] shows that under some simple syntactic conditions, $\mathcal{S}_\Sigma$ is unitary and has a linear unification algorithm.

[10] Viewing a signature as a family rather than as an arity function $\alpha : \Sigma \to S^* \times S$ has the advantage of allowing *overloading* of function symbols, since the *same* symbol can occur in several different sets $\Sigma_{w,s}$.

7 Further Explorations

An introductory paper is not a good place to discuss difficult examples in detail, but it may be nice to notice that they are examples. Consequently, what this section provides is little more than some sketches and relevant citations; there are probably also bugs. Readers are urged to work out further details.

7.1 Infinite Terms

Infinite terms are important in many areas, including logic programming [5], concurrency, and natural language processing [44]; Section 7.5 gives an example involving type inference. Our approach is to generalize the construction of $\mathcal{S}_\Sigma$ given in Section 5.3 above in such a way that both finite and infinitary unsorted terms will be special cases.

We begin by formalizing the notion of a free construction, since free algebras are the main device used in building $\mathcal{S}_\Sigma$. The crucial relationship is that between the free algebras and their generating sets. We can give a more precise description of this relationship using the "forgetful" functor $U : \mathcal{ALG}_\Sigma \to \mathcal{SET}$ which takes each Σ-algebra A to its "underlying" set, also denoted A, and each Σ-homomorphism $h : A \to B$ to its underlying function $h : A \to B$. Then $T_\Sigma(X)$ is characterized (up to isomorphism) by the following universal property:

> given any Σ-algebra A and any (set) function $f : X \to U(A)$, there is a unique Σ-homomorphism $\overline{f} : T_\Sigma(X) \to A$ such that $i_X ; U(\overline{f}) = f$, where $i_X : X \to U(T_\Sigma(X))$ is the inclusion function.

Now let us generalize: given a functor $U : \mathcal{A} \to \mathcal{SET}$ and a set X, an object $T(X) \in |\mathcal{A}|$ is **freely generated** by X in $\mathcal{A}$ (with respect to U) iff

> there is a function $i_X : X \to U(A)$ such that given any object A of $\mathcal{A}$ and any function $f : X \to U(A)$, there is a unique $\mathcal{A}$-morphism $\overline{f} : T(X) \to A$ such that $i_X ; U(\overline{f}) = f$.

Given $U : \mathcal{A} \to \mathcal{SET}$ such that each set X generates a free object $T(X)$ in $\mathcal{A}$, the next step is to generalize the construction of $\mathcal{S}_\Sigma$ to that of a category $\mathcal{S}_U$ whose objects are sets, and whose morphisms from X to Y are functions $Y \to U(T(X))$. Everything proceeds just as in Section 5.3: The composition $f ; g$ in $\mathcal{S}_\Sigma$ is defined to be $g ; \overline{f}$; the type morphisms from X to Y are functions $Y \to X$, and they induce morphisms $X \to Y$ that factor through $i_X : X \to T(X)$, thus giving rise to a functor $\mathcal{SET}^o \to \mathcal{S}_U$ which can be shown finite product preserving.

In fact, $\mathcal{S}_\mathcal{A}$ is $\mathcal{S}_\Sigma$ if we take $\mathcal{A}$ to be the category of Σ-algebras. But if we take $\mathcal{A}$ to be the category of continuous algebras [25] (or rational algebras, or some other variant), we will get various kinds of infinite terms as morphisms.

If we want to avoid uncountable sets of variables, we can restrict to the full subcategory whose objects are just the subsets of X_ω.

Also, notice that we can generalize the target category of the forgetful functor, from $\mathcal{SET}$ to the category $\mathcal{SET}_S$ of all S-indexed sets to get substitution systems with many sorted, or order sorted, infinite terms as morphisms.

Constructions of this kind have been rather thoroughly explored in category theory; for example, see [33].

7.2 Fixpoint Equations

Fixpoint equations are used in computer science to define many different structures, and least fixpoints are particularly used, because their existence can often be guaranteed by the well-known Tarski fixpoint theorem. A general context for these considerations is a function $f: X \to X$ on a poset X. Then a *fixpoint* of f is a solution $a: \{*\} \to X$ (in our sense) with one-point source, to the equation $f = id_X$.

When f is monotone, a most general solution for this equation is the inclusion function for the subset of all fixpoints of f. So the least fixpoint, if there is one, is the least element of the most general solution. It seems to me that the set of all fixpoints is a better starting place for discussion than the least fixpoint, because least fixpoints do not always exist, and because other fixpoints can sometimes be of equal or greater interest, e.g., the greatest fixpoint may be the one we want. Also, in this setting, there is no reason to restrict to equations of the form $f = id_X$ because an equation of the form $f = g$ also has an equalizer when f, g are monotone, and this again turns out to be the poset of all one-point solutions, including the least one-point solution if there is one.

Ait-Kaci [2] gives a fixpoint-based approach to solving what he calls "type equations" for the semantics of a language called KBL. However, this approach seems a bit awkward since infinitary structures are not really needed, and it has been subsumed by some elegant later work of Smolka and Ait-Kaci [46], reducing what they call "feature unification" to a kind of order sorted unification.

7.3 Scott Domain Equations

The so-called "domain equations" introduced by Scott for the Scott-Strachey "denotational" model-oriented semantics are an important technique in the semantics of programming languages [43]. Smyth and Plotkin [48] have developed an attractive categorical generalization of the fixpoint approach and shown that it encompasses Scott's approach, perhaps even with some advantages.

As a first step, let us generalize the setup of Section 7.2 by noticing that a partially ordered set is essentially the same thing as a category $\mathcal{C}$ with the following property:

(PO) Given $A, B \in |\mathcal{C}|$, there is at most one morphism between A and B in $\mathcal{C}$.

This follows by defining $A \to B$ iff $A \leq B$. Moreover, a monotone map between partially ordered sets corresponds to a functor between the corresponding categories. (More precisely, there is an equivalence between the category of posets with monotone maps and the category of small categories satisfying condition (PO).) Therefore, we may as well use categories instead of just posets. Generalizing Section 7.2, least fixpoints correspond to initial objects in most general solution categories; this works nicely, because $\mathcal{CAT}$ has equalizers.

The Smyth-Plotkin approach [48] uses an attractive generalization of the Tarski theorem, following Wand [52]. Let us say a category $\mathcal{C}$ is **ω-continuous** iff $\mathcal{C}$ has an initial object and every diagram of the form

$$C_0 \to C_1 \to ... \; C_n \to ...$$

has an **ω-colimit** in $\mathcal{C}$ (that is, an initial cone). Also, let us call an endofunctor $F: \mathcal{C} \to \mathcal{C}$ on $\mathcal{C}$ **ω-continuous** iff it preserves whatever ω-colimits exist in $\mathcal{C}$. Now, given an endofunctor $F: \mathcal{C} \to \mathcal{C}$, we define an **$F$-algebra** to be an object $C \in |\mathcal{C}|$ and a morphism $A: F(C) \to C$

in $\mathcal{C}$, and we define a **homomorphism** of F-algebras, from A to A' to be a morphism $h\colon C \to C'$ in $\mathcal{C}$ such that $A;h = F(h);A'$. Then F-algebras form a category, and

Theorem 26 Any ω-continuous endofunctor $F : \mathcal{C} \to \mathcal{C}$ on an ω-continuous category $\mathcal{C}$ has an initial F-algebra. Moreover, if $A\colon F(C) \to C$ is an initial F-algebra, then A is an isomorophism in $\mathcal{C}$. □

It is interesting to examine what happens here when $\mathcal{C}$ is a poset. ω-continuity of $\mathcal{C}$ means that countable chains have least upper bounds, and ω-continuity of F means that F preserves these least upper bounds. Then an F-algebra is some $C \in \mathcal{C}$ such that $F(C) \leq C$, and an initial F-algebra is a fixpoint of F, since isomorphisms in a poset are necessarily equalities. Of course, there may also be other F-algebras that are isomorphisms, i.e., that are fixpoints, but the initial one is the least fixpoint.

There are many different kinds of "domain," but all of them are at least partially ordered sets. Readers who have gotten this far will realize that various "bootstrapping" tricks are commonplace in applying category theory, and in particular, that categorical ideas are often applied to themselves. In particular, by taking $\mathcal{C}$ to be various suitable categories of domains, Smyth and Plotkin show that everything works, including the famous "arrow" domain constructors. However, the details are more complex than you might like, and I personally feel that an approach using a substitution system whose types are Cartesian closed categories would be more satisfactory; one would then take interpretations (algebras) into categories of domains.

7.4 Algebraic Domain Equations

Ehrich and Lipeck [8] have developed an algebraic analog of Scott's domain equations. We can give a rather elegant treatment of this example by using the substitution system whose types are the S-sorted substitution systems, for all S, with morphisms that allow changing and identifying sorts (actually, this example works even better using order sorted substitution systems). Another curious reversal of arrows occurs, and most general solutions appear as coequalizers rather than as equalizers. This approach allows treating any number of simultaneous equations in any number of variables. Of course, we do not get all the expressive power of Scott domain equations, because there are no infinitary structures. However, it is possible to carry out the same constructions with S-sorted infinitary substitution systems.

7.5 Polymorphic Type Inference

Milner's lovely paper on type polymorphism [35] shows how to infer the types of expressions in certain higher order programming languages by using classical term unification to solve systems of type equations; the key insight is to identify polymorphic types with ordinary first order terms that contain type variables. The exclusion of subtypes from this approach can be overcome by using type expressions in *order sorted algebra*. A first analysis leads to systems of *inequations* among type expressions; but any such system is equivalent to a system of equations among more general type expressions that allow type disjunction, because of the fact that

$$t \leq t' \text{ iff } t \vee t' = t'.$$

The resulting equations can then be solved using order sorted unification modulo the semilattice equations for $\vee$, which exactly capture the fact that $\leq$ is a partial ordering. This very general technique for viewing systems of inequations as systems of equations means that the techniques developed earlier in this paper can be applied to solving rather general systems of constraints. [35] also excludes types like $Stream(\alpha)$, as defined by

$$Stream(\alpha) = \mathrm{cons}(\alpha, Stream(\alpha)),$$

that (in some sense) are infinite. However, the methods of Section 7.1 seem adequate for this purpose, and I would venture to suggest that substitution systems might provide an appropriate framework for exploring even more general notions of type inference.

7.6 Database Query and Logical Programming Languages

There is a rather large body of work showing how unification, as embodied in Prolog and related languages, can be made the basic inference mechanism for sophisticated database query systems; for example, the papers in [11] represent an early stage in this evolution. I believe that it is very important to have a precise semantic foundation for any such endeavor. A more recent approach combines logic and functional programming to extend the Prolog framework with features such as abstract data types, multiple inheritance, generic modules, and both forward and backward chaining; some recent work in this area is collected in [7]. For example, [20] describes the Eqlog language, which has a rigorous semantics based upon (order sorted) Horn clause logic with equality. Some still more recent work extends this approach with object-oriented features; for example, see the FOOPS and FOOPlog languages in [21]. FOOPlog is a very powerful integrated database/programming system with a rigorous logical semantics, in which the solving of equations plays a basic role in answering database queries.

7.7 Generalizing the Notion of Equation

This subsection generalizes the notions of equation and solution in a way that can be very convenient in practice, even though in theory, it does not add any expressive power. The end of Section 3 suggested generalizing the notion of equation to a set ? of morphisms having the same source and the same target. Adachi [1] suggests a different generalization: an equation is a pair of morphisms with the same target. Actually, we may as well go the whole way and consider an equation to be an arbitrary **diagram** D in a category $\mathcal{C}$, i.e., a graph D with each node $n \in |D|$ labelled by an object $D_n \in |\mathcal{C}|$, and each edge $e : n \to n'$ of D labelled by a morphism $D_e : D_n \to D_{n'}$ in $\mathcal{C}$. Now we can generalize the definitions and results of Section 6.2. A **solution** F of D is a family $\{f_n \mid n \in |D|\}$ of morphisms $f_n : C \to D_n$ in $\mathcal{C}$ such that $f_n ; D_e = f_{n'}$ for each $e : n \to n'$ in D; we call C the **apex** of F. Similarly, a **morphism of solutions** $m : \{f_n \mid n \in |D|\} \to \{f'_n \mid n \in |D|\}$ is a morphism $m : C \to C'$ of the apices such that $m ; f'_n = f_n$ for each $n \in |D|$. This gives a category $\mathcal{SOL}(D)$ of solutions of D, and a **most general solution** is a final object in it. These notions of solution and most general solution are actually *identical* with the usual notions of **cone** over a diagram D and **limit** cone of D; see [29].

Continuing along this line, a **solution set** for a diagram D is a flooring in $\mathcal{SOL}(D)$, i.e., a set $\mathcal{F}$ of solutions of D such that for any solution G of D, there is some $F \in \mathcal{F}$ with a morphism $G \to F$. Then of course, a **minimal solution set** is a solution set such that no

proper subset is a solution set, and any two of these are isomorphic by Lemma 15, so that cardinality is a well defined invariant for minimal solution sets.

Notice that this notion of equation as diagram is general enough to include an arbitrary conjunction of (ordinary) equations, with variables shared among equations in any desired way; i.e., we can think of it as a system of equations, or a system of constraints. This framework also seems general enough to consider the Herbrand style of unification algorithm, which involves manipulating just such sets of equations. Adachi [1] also defines a notion of "kite" that generalizes Horn clauses, and gives a resolution algorithm that specializes to a Prolog-like interpreter. The above could be used to simplify this a bit.

It is well known in category theory that an arbitrary limit problem can be reduced to an equalizer problem, by first taking suitable products; for example, see [29], page 109. This implies that the generalization of equation and solution developed above adds no expressive power, so long as the appropriate products can be formed in $\mathcal{C}$.

7.8 Unification Grammars

So-called "unification grammar" formalisms have recently become important in linguistics. In these formalisms, a "meaning" is some kind of parameterized record structure, probably with sharing (i.e., a graph rather than a tree), and possibly with cycles; see [44] for a general overview. Of course, these structures are nothing like "meanings" in the sense that human beings experience meanings, or even in the sense that sentences in formal logic have meanings in models; they are purely syntactic, without any blood or any given denotation. In linguistics, such structures are called *feature structures*, or also "functional structures" or "dags," while in Artificial Intelligence they have been called "semantic networks" and "frame systems;" there are also many other names. Such structures have nodes (or slots) that may contain "logical variables" (in the sense of Prolog, or more exactly, of Eqlog [20]) that can unify with fragments of other structures to represent complex relationships; moreover, cyclic graphs represent what amount to infinite structures. Order sorted algebra is useful in this context, providing so-called *partiality*, the possibility that further record fields can be added later. These topics will be further explored in forthcoming work with Dr. José Meseguer; see also [46].

The ideas developed in this paper suffice to explain the meaning of "unification" in unification grammmar formalisms. To avoid commitment to any particular formalism, let us just assume that we are given a free **meaning algebra** $M(X)$ for each set X of parameters; more general types than sets can also be used. Plausible choices for M are given in [46] and [37], and others have been hinted at in this paper: We can now construct a substitution system whose objects are the parameter types, and whose morphisms are the (parameterized) meanings, by using the recipe of Section 7.1; let us denote this category $\mathcal{M}$. Then given a sentence (or more generally, a discourse), a unification grammar provides a **diagram** in $\mathcal{M}$, and the **limit** (possibly weak) of this diagram is the meaning of the sentence (or discourse) as a whole; this process is called **unification** in linguistics, and we can now see that it *really is unification* in the precise sense of this paper. We are solving the equation (generalized, in the sense of Section 7.7) given by the diagram, i.e., we are solving a system of local "constraints" to determine the meaning of the whole.

This process is *compositional*, since the meaning of the whole is composed from the

meanings of the parts, but the exact sense of compositionality involved is more general sense than that involved (for example) in Montague grammar [36], where meaning is compositional in the sense that it is given by the unique homomorphism from an initial algebra, i.e., it is given by "initial algebra semantics" in the sense of [14, 25] (see [50] for a general discussion of the relation between initial algebra semantics and compositionality in linguistics). It is also interesting to note that this approach is consistent with the "General Systems Theory" doctrine that the limit of a diagram solves an arbitrary system of constraints [12, 13, 16].

Also note that ambiguity of meaning can arise naturally in this setting, exactly because unification is not necessarily unitary.

7.9 Differential Equations

Standard techniques of functional analysis, such as differential operators on function spaces (e.g., see [53]), seem ideally suited for developing an exposition of differential equations within the framework of this paper. Standard cases may require restricting the morphisms allowed in the equations and/or in the solutions.

7.10 Cartesian Closed Categories and Topoi

Cartesian closed categories and topoi represent substantial extensions of the point of view developed here in somewhat different directions. Each has an underlying category of base types with finite products, as well as much more. Cartesian closed categories capture typed lambda calculi, and solving equations in a Cartesian closed category is higher order unification. Topoi, which were introduced by Lawvere and Tierney, are Cartesian closed categories with additional structure that captures set-like structures that arise in geometry, algebra and logic, thus providing a surprising unification of many areas of mathematics; [26] provides an introduction to these topics, and of course, we now know what it would mean to solve equations in topoi.

8 Summary

This paper develops a very general approach to the solution of equations, starting from the simple example of polynomials with integer coefficients, and gradually working toward a General Unification Theory that encompasses much more sophisticated examples such as order sorted unification, Scott domain equations, type inference, natural language semantics, and differential equations. A fair amount of basic category theory is introduced along the way, including initial and final objects, products and coproducts, equalizers and coequalizers, duality, limits, free constructions, and Lawvere theories. Substitution systems are a unifying theme, and some of their basic theory is developed, including morphisms, quotients, algebras, and cardinality bounds for most general solution sets.

References

[1] Takanori Adachi. Unification in categories. In Toshiaki Kurokawa, editor, *Several Aspects of Unification*, pages 35–43. ICOT, Technical Report TM-0029, 1984.

[2] Hassan Aït-Kaci. An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, 45:293–351, 1986.

[3] Jean Benabou. Structures algébriques dans les catégories. *Cahiers de Topologie et Géometrie Différentiel*, 10:1–126, 1968.

[4] Rod Burstall, David MacQueen, and Donald Sannella. Hope: an experimental applicative language. In *Proceedings, First LISP Conference*, volume 1, pages 136–143. Stanford University, 1980.

[5] Alain Colmerauer. Prolog and infinite trees. In Keith Clark and Sten-Åke Tärnlund, editors, *Logic Programming*, pages 231–251. Academic, 1982.

[6] Alan Colmerauer, H. Kanoui, and M. van Caneghem. Etude et réalisation d'un système Prolog. Technical report, Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Université d'Aix-Marseille II, 1979.

[7] Douglas DeGroot and Gary Lindstrom. *Logic Programming: Functions, Relations and Equations*. Prentice-Hall, 1986.

[8] Hans-Dieter Ehrich and Udo Lipeck. Algebraic domain equations. *Theoretical Computer Science*, 27:167–196, 1983.

[9] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian Reid, editor, *Proceedings, Twelfth ACM Symposium on Principles of Programming Languages*, pages 52–66. Association for Computing Machinery, 1985.

[10] Kokichi Futatsugi, Joseph Goguen, José Meseguer, and Koji Okada. Parameterized programming in OBJ2. In Robert Balzer, editor, *Proceedings, Ninth International Conference on Software Engineering*, pages 51–60. IEEE Computer Society, March 1987.

[11] Herve Gallaire and Jack Minker. *Logic and Data Bases*. Plenum, 1978.

[12] Joseph Goguen. Mathematical representation of hierarchically organized systems. In E. Attinger, editor, *Global Systems Dynamics*, pages 112–128. S. Karger, 1971.

[13] Joseph Goguen. Categorical foundations for general systems theory. In F. Pichler and R. Trappl, editors, *Advances in Cybernetics and Systems Research*, pages 121–130. Transcripta Books, 1973.

[14] Joseph Goguen. Semantics of computation. In Ernest G. Manes, editor, *Proceedings, First International Symposium on Category Theory Applied to Computation and Control*, pages 234–249. University of Massachusetts at Amherst, 1974. Also in Lecture Notes in Computer Science, Volume 25, Springer, 1975, pages 151-163.

[15] Joseph Goguen. Order sorted algebra. Technical Report 14, UCLA Computer Science Department, 1978. Semantics and Theory of Computation Series.

[16] Joseph Goguen. Sheaf semantics for concurrent interacting objects. *Mathematical Structures in Computer Science*, to appear 1991. Given as lecture at U.K.-Japan Symposium on Concurrency, Oxford, September 1989; draft in Report CSLI-91-155, Center for the Study of Language and Information, Stanford University, June 1991.

[17] Joseph Goguen and Rod Burstall. Some fundamental algebraic tools for the semantics of computation, part 1: Comma categories, colimits, signatures and theories. *Theoretical Computer Science*, 31(2):175–209, 1984.

[18] Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Operational semantics of order-sorted algebra. In W. Brauer, editor, *Proceedings, 1985 International Conference on Automata, Languages and Programming*. Springer, 1985. Lecture Notes in Computer Science, Volume 194.

[19] Joseph Goguen, Claude Kirchner, HélèneKirchner, Aristide Mégrelis, and José Meseguer. An introduction to OBJ3. In Jean-Pierre Jouannaud and Stephane Kaplan, editors, *Proceedings, Conference on Conditional Term Rewriting*, pages 258–263. Springer, 1988. Lecture Notes in Computer Science, Volume 308.

[20] Joseph Goguen and José Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In Douglas DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 295–363. Prentice-Hall, 1986. An earlier version appears in *Journal of Logic Programming*, Volume 1, Number 2, pages 179-210, September 1984.

[21] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming, with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT, 1987. Preliminary version in *SIGPLAN Notices*, Volume 21, Number 10, pages 153-162, October 1986.

[22] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Technical Report SRI-CSL-89-10, SRI International, Computer Science Lab, July 1989. Given as lecture at Seminar on Types, Carnegie-Mellon University, June 1983; many draft versions exist.

[23] Joseph Goguen and Joseph Tardo. An introduction to OBJ: A language for writing and testing software specifications. In Marvin Zelkowitz, editor, *Specification of Reliable Software*, pages 170–189. IEEE, 1979. Reprinted in *Software Specification Techniques*, Nehan Gehani and Andrew McGettrick, editors, Addison-Wesley, 1985, pages 391–420.

[24] Joseph Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. Technical Report RC 6487, IBM T.J. Watson Research Center, October 1976. In *Current Trends in Programming Methodology, IV*, Raymond Yeh, editor, Prentice-Hall, 1978, pages 80-149.

[25] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*,

24(1):68–95, January 1977. An early version is "Initial Algebra Semantics", with James Thatcher, IBM T.J. Watson Research Center Report RC 4865, May 1974.

[26] Robert Goldblatt. *Topoi, the Categorial Analysis of Logic*. North-Holland, 1979.

[27] Jacques Herbrand. Recherches sur la théorie de la démonstration. *Travaux de la Société des Sciences et des Lettres de Varsovie, Classe III*, 33(128), 1930.

[28] Claude Kirchner, HélèneKirchner, and José Meseguer. Operational semantics of OBJ3. In *Proceedings, 9th International Conference on Automata, Languages and Programming*. Springer, 1988. Lecture Notes in Computer Science, Volume 241.

[29] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, 1971.

[30] Jean-Louis Lassez, Michael Maher, and Kimbal Marriott. Unification revisited. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, 1988.

[31] F. William Lawvere. Functorial semantics of algebraic theories. *Proceedings, National Academy of Sciences, U.S.A.*, 50:869–872, 1963. Summary of Ph.D. Thesis, Columbia University.

[32] John Lloyd. *Foundations of Logic Programming*. Springer, 1984.

[33] Ernest Manes. *Algebraic Theories*. Springer, 1976. Graduate Texts in Mathematics, Volume 26.

[34] José Meseguer, Joseph Goguen, and Gert Smolka. Order-sorted unification. *Journal of Symbolic Computation*, 8:383–413, 1989. Preliminary version appeared as Report CSLI-87-86, Center for the Study of Language and Information, Stanford University, March 1987.

[35] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[36] Richard Montague. *Formal Philosophy: Selected Papers of Richard Montague*. Yale, 1974. Edited and with an introduction by Richard Thomason.

[37] Kuniaki Mukai. Unification over complex indeterminates in Prolog. Technical Report TR-113, ICOT, 1985.

[38] Gordon Plotkin. Building-in equational theories. *Machine Intelligence*, 7:73–90, November 1972.

[39] J. Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12:23–41, 1965.

[40] David Rydeheard and Rod Burstall. *Computational Category Theory*. Prentice-Hall, 1988.

[41] Manfred Schmidt-Schauss. Unification in many-sorted equational theories. In *Proceedings, 8th International Conference on Automated Deduction*, pages 538–552. Springer, 1986. Lecture Notes in Computer Science, Volume 230.

[42] Horst Schubert. *Categories*. Springer, 1972.

[43] Dana Scott. Lattice theory, data types and semantics. In Randall Rustin, editor, *Formal Semantics of Algorithmic Languages*, pages 65–106. Prentice Hall, 1972.

[44] Stuart Shieber. *An Introduction to Unification-Based Approaches to Grammar*. Center for the Study of Language and Information, 1986.

[45] Jörg Siekmann. Unification theory. In *Journal of Symbolic Computation*, 1988. Preliminary Version in *Proceedings*, European Conference on Artificial Intelligence, Brighton, 1986.

[46] Gert Smolka and Hassan Aït-Kaci. Inheritance hierarchies: Semantics and unification. Technical Report Report AI-057-87, MCC, 1987. In *Journal of Symbolic Computation*, 1988.

[47] Gert Smolka, Werner Nutt, Joseph Goguen, and José Meseguer. Order-sorted equational computation. In Maurice Nivat and Hassan Aït-Kaci, editors, *Resolution of Equations in Algebraic Structures, Volume 2: Rewriting Techniques*, page 299ff. Academic, 1989. Preliminary version in *Proceedings*, Colloquium on the Resolution of Equations in Algebraic Structures, held in Lakeway, Texas, May 1987; also appears as SEKI Report SR-87-14, Universität Kaiserslautern, December 1987.

[48] Michael Smyth and Gordon Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computation*, 11:761–783, 1982. Also Technical Report D.A.I. 60, University of Edinburgh, Department of Artificial Intelligence, December 1978.

[49] Mark Stickel. A unification algorithm for associative-commutative functions. *Journal of the Association for Computing Machinery*, 28:423–434, 1981.

[50] Peter van Emde Boas and Theo Janssen. The impact of frege's principle of compositionality for the semantics of programming and natural languages. Technical Report 79-07, University of Amsterdam, Department of Mathematics, 1979.

[51] Christoph Walther. A classification of many-sorted unification theories. In *Proceedings, 8th International Conference on Automated Deduction*, pages 525–537. Springer, 1986. Lecture Notes in Computer Science, Volume 230.

[52] Mitchell Wand. On the recursive specification of data types. In Ernest Manes, editor, *Proceedings, Symposium on Category Theory Applied to Computation and Control*, pages 214–217. Springer, 1975. Lecture Notes in Computer Science, Volume 25.

[53] Kosaku Yosida. *Functional Analysis*. Springer, 1968. Second Edition.

Contents

1 Introduction **1**
1.1 An Example . . . 2

2 Substitutions and Categories **3**

3 Equations and Solutions **6**

4 Types, Variables and Products **7**
4.1 Finite Sets and Duality . . . 10

5 Classical Term Substitution **11**
5.1 Terms . . . 11
5.2 Freebies . . . 12
5.3 Term Substitution Systems . . . 13

6 What is Unification? **14**
6.1 Classical Term Unification . . . 14
6.2 Floorings and Solution Sets . . . 15
6.3 Substitution Modulo Equations . . . 18
6.3.1 Associative Unification . . . 19
6.3.2 AC Unification . . . 20
6.3.3 Polynomials . . . 20
6.3.4 A Nullary Example . . . 20
6.4 Lawvere Theories . . . 20
6.5 Categories and Algebras of Substitution Systems . . . 21
6.6 Many Sorted Algebra . . . 22
6.7 Order Sorted Algebra . . . 23

7 Further Explorations **24**
7.1 Infinite Terms . . . 24
7.2 Fixpoint Equations . . . 25
7.3 Scott Domain Equations . . . 25
7.4 Algebraic Domain Equations . . . 26
7.5 Polymorphic Type Inference . . . 26
7.6 Database Query and Logical Programming Languages . . . 27
7.7 Generalizing the Notion of Equation . . . 27
7.8 Unification Grammars . . . 28
7.9 Differential Equations . . . 29
7.10 Cartesian Closed Categories and Topoi . . . 29

8 Summary **29**