Stretching First Order Equational Logic: Proofs with Partiality, Subtypes and Retracts

Joseph A. Goguen
Dept. Computer Science & Engineering
University of California at San Diego, La Jolla CA 92093-0114 USA

Abstract: It is widely recognized that equational logic is simple, (relatively) decidable, and (relatively) easily mechanized. But it is also widely thought that equational logic has limited applicability because it cannot handle subtypes or partial functions. We show that a modest stretch of equational logic effectively handles these features. Space limits preclude a full theoretical treatment, so we often sketch, motivate and exemplify.

1 Introduction

First order equational logic (**EL**) has significant conceptual, theoretical and computational advantages, to the extent that I suggest it *should* be used if it *can* be used for a given application. But there are many applications where ordinary first order EL does not seem sufficiently expressive. This paper describes an extension that greatly expands its expressiveness and applicability, at little cost to its advantages.

EL was untyped at birth [3], but later extended to many sorts in various ways, of which [1] was perhaps first and [5] notationally simplest; extensions to overloaded function symbols and conditional equations were also important; see [10] for technical and historical details. Section 2 quickly reviews many sorted EL, and Section 3 covers the next important extension, order sorted EL [11], including an inductive proof for a typical partial function. A final section discusses a further extension to hidden EL.

2 Many Subsorted Equational Logic

Overloaded many sorted EL (**MSEL**) permits overloaded operation symbols, which will be important for the work of this paper. This is based on sorted sets [5]: Given a set S, whose elements are called **sorts**, an S-**sorted set** A is a family of sets A_s, one for each $s \in S$. Then an S-sorted **signature** Σ is an $(S^* \times S)$-sorted set $\{\Sigma_{w,s} \mid \langle w, s\rangle \in S^* \times S\}$. The elements of $\Sigma_{w,s}$ are called **operation symbols** of **arity** w, **sort** s, and **rank** $\langle w, s\rangle$; in particular, $\sigma \in \Sigma_{[],s}$ is a **constant symbol** ($[]$ denotes the empty string). Σ is a **ground signature** iff $\Sigma_{[],s} \cap \Sigma_{[],s'} = \emptyset$ whenever $s \neq s'$ and $\Sigma_{w,s} = \emptyset$ unless $w = []$. By convention, $|\Sigma| = \bigcup_{w,s} \Sigma_{w,s}$ and $\Sigma' \subseteq \Sigma$ means $\Sigma'_{w,s} \subseteq \Sigma_{w,s}$ for each w, s. Similarly, **union** is defined by $(\Sigma \cup \Sigma')_{w,s} = \Sigma_{w,s} \cup \Sigma'_{w,s}$. A common special case is union with a ground signature X, for which we use the notation $\Sigma(X) = \Sigma \cup X$.

A Σ-**algebra** M consists of an S-sorted set also denoted M, plus an **interpretation** of Σ in M, which is a family of arrows $i_{s_1\ldots s_n,s}\colon \Sigma_{s_1\ldots s_n,s} \to [M^{s_1\ldots s_n} \to M_s]$ for each rank $\langle s_1\ldots s_n, s\rangle \in S^* \times S$, which interpret the operation symbols in Σ as actual operations on M. For constant symbols, the interpretation is given by $i_{[],s}\colon \Sigma_{[],s} \to M_s$. Usually we write just σ for $i_{w,s}(\sigma)$, but if we need to make the dependence on M explicit, we may write σ_M. M_s is called the **carrier** of M of sort s. Given Σ-algebras M, M', a Σ-**homomorphism** $h\colon M \to M'$ is an S-sorted arrow $h\colon M \to M'$ such that

$h_s(\sigma_M(m_1, ..., m_n)) = \sigma_{M'}(h_{s_1}(m_1), ..., h_{s_n}(m_n))$ for each $\sigma \in \Sigma_{s_1...s_n,s}$ and $m_i \in M_{s_i}$ for $i = 1, ..., n$, and such that $h_s(c_M) = c_{M'}$ for each constant symbol $c \in \Sigma_{[],s}$.

Given an S-sorted signature Σ, the S-sorted set T_Σ of (**ground**) **Σ-terms** is the smallest set of lists of symbols that contains the constants, $\Sigma_{[],s} \subseteq T_{\Sigma,s}$, and such that given $\sigma \in \Sigma_{s_1...s_n,s}$ and $t_i \in T_{\Sigma,s_i}$ then $\sigma(t_1 \dots t_n) \in T_{\Sigma,s}$. We view T_Σ as a Σ-algebra by interpreting $\sigma \in \Sigma_{[],s}$ as just σ, and $\sigma \in \Sigma_{s_1...s_n,s}$ as the operation sending $t_1, \dots, t_n$ to the list $\sigma(t_1 \dots t_n)$. Then T_Σ is called the **Σ-term algebra**. Note that because of overloading, terms do not always have a unique parse. The following is the key property of this algebra:

> **Theorem 1:** (Initiality) Given any signature Σ and any Σ-algebra M, there is a unique Σ-homomorphism $T_\Sigma \to M$.

Given Σ and a ground signature X disjoint from Σ, we can form the $\Sigma(X)$-algebra $T_{\Sigma(X)}$ and then view it as a Σ-algebra by forgetting the names of the new constants in X; let us denote this Σ-algebra by $T_\Sigma(X)$. It has the following universal **freeness** property: Given a Σ-algebra M and $a\colon X \to M$, there is a unique Σ-homomorphism $\overline{a}\colon T_\Sigma(X) \to M$ extending a, in the sense that $\overline{a}_s(x) = a_s(x)$ for each $x \in X_s$ and $s \in S$.

A **Σ-equation** consists of a ground signature X of **variable symbols** (disjoint from Σ) plus two $\Sigma(X)$-terms of the same sort $s \in S$; we may write such an equation abstractly in the form $(\forall X)\ t = t'$ and concretely in the form $(\forall x, y, z)\ t = t'$ when $|X| = \{x, y, z\}$ and the sorts of x, y, z can be inferred from their uses in t and in t'. A **specification** P is a pair (Σ, A), consisting of a signature Σ and a set A of Σ-equations. Conditional equations are similar, but we omit them here.

A Σ-algebra M **satisfies** a Σ-equation $(\forall X)\ t = t'$ iff for any $a\colon X \to M$ we have $\overline{a}(t) = \overline{a}(t')$ in M, written $M \models_\Sigma (\forall X)\ t = t'$. A Σ-algebra M satisfies a set A of Σ-equations iff it satisfies each one, written $M \models_\Sigma A$. The class of all algebras that satisfy A is called the **variety** defined by A. Given sets A and A' of Σ-equations, let $A \models A'$ mean $M \models A'$ for all A-models M. Then we have the following:

> **Theorem 2:** (Initiality) $T_{\Sigma,A} = T_\Sigma/{\equiv_A}$ is an initial (Σ, A)-algebra, where $\equiv_A$ is the Σ-congruence generated by ground instances of equations in A.

And again we get the free algebras $T_{\Sigma,E}(X)$.

The word "abstract" in "abstract algebra" means "uniquely defined up to isomorphism" and initial algebras are easily shown abstract in this sense. Moreover, the word "abstract" in "abstract data type" has *exactly* the same meaning, since an ADT is defined to be the isomorphism class of initial algebras of a specification [12]; this is no mere pun, but a significant fact about software engineering. Another sign we are on the right track is that any computable algebra has an equational specification, as first proved by Bergstra and Tucker [2]; moreover, this specification tends to be simple and intuitive in practice. (M is **reachable** iff the unique Σ-homomorphism $T_\Sigma \to M$ is surjective.)

> **Theorem 3:** (Computability) Given a reachable computable Σ-algebra M with Σ finite, there is a finite specification $P = (\Sigma', A')$ such that $\Sigma \subseteq \Sigma'$, such that Σ' has the same sorts as Σ, and such that M is Σ-isomorphic to T_P viewed as a Σ-algebra.

The following simple result is much used in equational theorem proving, but rarely stated:

> **Fact:** (Theorem of Constants) Given a signature Σ, a ground signature X disjoint from Σ, a set A of Σ-equations, and $t, t' \in T_{\Sigma(X)}$, then $A \models_\Sigma (\forall X)\ t = t'$ iff $A \models_{\Sigma \cup X} (\forall \emptyset)\ t = t'$.

It says we can regard the universally variables as new constants instead.

2.1 Basic OBJ Notation

OBJ gives a notation which has been implemented to permit proving things about such specifications [8, 13]; this is sufficiently powerful that it can be used as a programming language for small applications, and as a prototyping language for larger applications. OBJ modules to be interpreted **loosely**, i.e., that are to have the whole variety as their semantics, begin with the keyword `theory` (or `th`) and close with the keyword `endth`. Between these two keywords come declarations for sorts and operations, plus variables and equations. For example, the following OBJ code specifies the theory of automata:

```
th AUTOM is
  sorts Input State Output .
  op s0 : -> State .
  op f : Input State -> State .
  op g : State -> Output .
endth
```

Any number of sorts can be declared following `sorts` (or equivalently, `sort`), and operations are declared with their arity between the `:` and the `->`, and their sort following the `->`. The keyword pair `obj...endo` indicates that **initial** semantics is intended. For example, the Peano natural numbers with addition are specified by

```
obj NATP is
  sort Nat .
  op 0 : -> Nat .
  op s_ : Nat -> Nat [prec 2].
  op _+_ : Nat Nat -> Nat .
  vars M N : Nat .
  eq 0 + N = N .
  eq s M + N = s(M + N).
endo
```

In "mixfix" operator declarations, underbar characters the expression before the colon, the are place holders showing where the operation's arguments should go; hence successor is prefix and addition is infix above. The low number 2 in the annotation "`[prec 2]`" indicates that successor is strongly binding.

2.2 Term Rewriting

Term rewriting is the most important implementation for EL. The **rewrite relation** of a set of equations E is defined as follows: a term t rewrites to a term t' in one step, written $t \Rightarrow t'$, iff some subterm of t matches the left side of some equation in E and t' is the result of replacing that subterm in t by the corresponding substitution instance of the right side of the equation; this requires that no equation in E has variables in its left side that are not in its right side. Then t **rewrites** to t' iff there exist rewrites $t \Rightarrow t_1 \Rightarrow ...t_n = t'$, in which case we write $t \overset{*}{\Rightarrow} t'$. OBJ implements term rewriting with the command `red` t, the result of which is t' such that $r \overset{*}{\Rightarrow} t'$ with t' **reduced**, in the sense that no equation applies to it; in general, there may be no such t' or many; OBJ just produces one if it exists. A term rewriting system is **canonical** iff terminating and Church-Rosser; this allows deciding equality of terms. But since every rewrite sequence proves equality of its initial and final terms, canonicity need not shown before trying a rewrite, despite the fact that some EL systems require such a check. Many systems have been specified, implemented and prototyped in OBJ3, and many proofs have also been done [13].

3 Partiality, Subsorts and Retracts

It seems little known that overloaded order sorted algebra with retracts gives a rigorous theory for many kinds of partial function, and that reasoning about such functions can be mechanized in OBJ3 [13]. This section sketches overloaded order sorted EL (**OSEL**), including retracts [11, 6, 15] and their implementation and use in OBJ3, with examples. Things are much the same as for MSEL, modulo some technicalities; therefore we shall be rather informal. An OSEL signature adds an ordering relation to the sort set of a MSEL signature, and an OSEL algebra M respects that relation, in that if $s \leq s'$ then $M_s \subseteq M_{s'}$, and if σ is in both $\Sigma_{w,s}$ and $\Sigma_{w',s'}$ with $w \leq w'$ then the corresponding operations of M agree on M_w. OSEL homomorphisms must also respect the subsort relation in an appropriate sense. Then the term algebra construction carries over and gives an initial algebra. But we usually assume **regular** signatures, so that terms have a unique parse of least sort [11]; otherwise subterms must be annotated with sort information. Adding the mild technical condition of local filtration yields **coherent** signatures, for which equations and their satisfaction, as well as congruences and quotients, go much as before. We get the analog of Theorem 2, so that both loose and initial specifications carry over, as do free algebras and the Theorem of Constants; incoherent signatures do not seem to occur in natural examples. The OBJ3 notation and implementation handle OSEL much the same as MSEL, modulo some subtle points about order sorted rewriting that we omit here [7, 14]. OSEL can specify not just computable functions, but also semi-computable functions.

3.1 Retracts

Untyped logics are too permissive, allowing many expressions that make no sense. But strongly typed logics like OSEL can be too strict. For example, given the following specification for stacks, ("`pr Nat`" indicates that OBJ3's module for builtin natural numbers `NAT` is "protecting" imported, i.e., restricting the initial model of the whole spec to the signature of `NAT` gives the initial model of `NAT`.)

```
obj STACK is sorts Stack NeStack .
  pr NAT .
  subsort NeStack < Stack .
  op empty : -> Stack .
  op push : Nat Stack -> NeStack .
  op pop_ : NeStack -> Stack .
  op top_ : NeStack -> Nat .
  var N : Nat . var S : Stack .
  eq pop push(N,S) = S .
  eq top push(N,S) = N .
endo
```

then terms like

```
     top pop push(7,push(3,empty))
```

do not parse, because `pop` delivers the wrong sort of argument to `top`, even though after evaluation, `top` actually gets `push(3,empty)`, which has sort `NeStack`.

Retracts solve this problem by transforming ill-formed terms over a signature Σ into well-formed terms over a signature $\Sigma^{\otimes}$ which extends Σ with some new operation symbols called **retracts** and some new equations: whenever $s \leq s'$ in Σ, then $\Sigma^{\otimes}$ adds $r_{s',s} : s' \to s$ to Σ and $E^{\otimes}$ adds $r_{s',s}(x) = x$ to E, where x has sort s. Parsing must also be extended to give the "benefit of the doubt" to terms like that above which could potentially become well formed after reduction, by filling the gaps between actual sorts and required sorts with retracts. For example, the above term is parsed as

```
    top r:NeStack>Stack(pop push(7,push(3,empty)))
```

which OBJ3 indeed reduces to 3, using the retract equation as a rewrite rule. On the other hand, reductions of truly erroneous terms contain retracts and serve as informative error messages. OSEL with retracts combines the flexibility of untyped logic with the error checking of strong typing.

The following result, proved in [11], shows that adding retracts is "safe" in the sense of not interfering with free models of the original specification. Faithfulness is a very weak condition that holds, for example, if (Σ, E) has no models where some carriers are empty and others are not, or if E is Church-Rosser; all specs in this paper are faithful.

Theorem 4: (Conservative extension) If Σ is coherent and (Σ, E) is faithful, then for each X, the natural homomorphism $T_{\Sigma,E}(X) \to T_{\Sigma^{\otimes},E^{\otimes}}(X)$ is injective.

Given a specficiation $P = (\Sigma, E)$, we often add "error supersorts" s' for each old sort s; then a partial operation of sort s is defined to have target sort s'. Adding retracts then gives three layers, one for the original P, one for terms of old sorts with retracts, and the third for supersort terms. Equations with a retract on the left side can "trap" retracts to produce useful new behaviors; this is much used below.

3.2 Sort Constraints

Sort constraints are declarations that a certain term has (or should have) a certain sort, under certain conditions. The theory is developed in [11, 14]; although sort constraints would handle the problems of this paper, they have not yet been fully implemented.

3.3 A Partial Specification and a Partial Proof

Subtraction on the natural numbers is a typical partial function, with $x - y$ defined iff $y \leq x$. We specify this as an operation to the error supersort `ENat` of `Nat`, using retract equations to do the usual reductions on the domain $y \leq x$; note that each operation on `Nat` is overloaded with a counterpart on `ENat`.

```
obj PNAT is sorts Nat ENat EBool .
  subsort Nat < ENat .
  subsort Bool < EBool .
  op 0 : -> Nat .
  op s_  : Nat -> Nat [prec 2].
  op s_  : ENat -> ENat [prec 2].
  op _+_ : Nat Nat -> Nat .
  op _+_ : ENat ENat -> ENat .
  op _*_ : Nat Nat -> Nat .
  op _*_ : ENat ENat -> ENat .
  op _<=_ : Nat Nat -> Bool .
  op _<=_ : ENat ENat -> EBool .
  op _-_ : Nat Nat -> ENat .
  vars X Y : Nat .  var E : ENat .
  eq 0 + X = X .
  eq (s X) + Y = s(X + Y) .
  eq 0 * X = 0 .
  eq (s X) * Y = (X * Y) + Y .
```

```
    eq X <= 0 = X == 0 .
    eq 0 <= X = true .
    eq (s X) <= (s Y) = X <= Y .
    eq r:ENat>Nat(X - 0) = X .
    cq r:ENat>Nat(s X - s Y) = X - Y if Y <= X .
  endo
```
Now we do some calculations. Of course, `+`, `*` and `<=` behave just as expected. Since subtraction has value sort `ENAT`, retracts can never be added to an expression involving it unless it is parsed as sort `NAT`; this is the purpose of the operation `[_]` below. The notation `open` indicates that material up to `close` is added to the current module (in this case `PNAT`), and then forgotten.
```
  open .
  op [_] : Nat -> Nat .
  var X : Nat .  eq [X] = X .
  red s 0 - s s 0 .
  red 0 *(s 0 - s s 0) .
  red [s s 0 - s 0] .
  red [s 0 - s s 0] .
  close
```
The first term is irreducible because its sort is `ENAT` and there are no equations of that sort; the second is irreducible for the same reason. The third reduction gives `s 0` as expected, while the value of the fourth is `r:ENat>Nat(s 0 - s s 0)`, since the term is forced to the sort `Nat`. Because of the overloading and the retract equations, Theorem 4 does not guarantee conservation here. However, we can check directly that these additions cannot interfere with the free subalgebra $T_{\Sigma,E}$.

Now we use induction to prove the equation

$(\forall l, m, n)\ n - (l + m) = n - l - m$,

assuming its left side is defined, i.e., $l + m \leq n$, using an OBJ "proof score" in the style of [8]. Theorem proving for partial functions is harder than calculation. The key point, already seen in our calculations, is to represent terms of sort `ENat` that should have sort `Nat` by retracts, and add equations to manipulate them.

The new constants `l`, `m`, `n` come from using the Theorem of Constants to handle universal quantifiers. As usual for non-trivial proofs, some "lemmas" are needed; the first equation below is a simple result about the natural numbers, easily proved separately by induction. The second equation is more interesting because it involves subtraction, but it too is easy. The third equation is the most interesting, since it involves a retract. In fact, it involves a retract on each side, and therefore cannot violate conservation; its purpose is to get a useful normal form for retract expressions. The problem is that OBJ does not know, and cannot be told, that a symbolic expression like $n - l$ has sort `Nat`. Of course we can tell it that $l \leq n$, but this is not enough to lower the sort. Therefore we use expressions with retracts, like $r_{\texttt{ENat},\texttt{Nat}}(n - l)$. (`openr` below indicates that the following material will be added to the previous module and retained.)
```
  openr .  *** constants plus lemmas
  ops l m n : -> Nat .
  vars L M N : Nat .
    cq N <= s M = true if N <= M .
    cq s N - M = s(N - M) if M <= s N .
    eq r:ENat>Nat(s E) = s r:ENat>Nat(E) .
  close
```
Now we do the base case. The assertions $l = 0$ and $m = 0$ are again just facts about the naturals, proved from the assumption $l + m \leq n = 0$. The operator `==` checks whether or not the reduced forms of its two argument terms are syntactically identical; if they are, then it returns `true`, and we know it has proved the terms equal.

```
open .  *** base case
eq n = 0 .
*** now since l + m <= 0 = true
  eq l = 0 .
  eq m = 0 .
red n -(l + m) == (n - l) - m .
close
```

For the induction step, we first state the induction hypothesis, and then two consequences of it. The second, $m \leq n - l$, follows from the domain constraint in the hypothesis, that $l + m \leq n$, but must be expressed with a sort constraint, for reasons described above.

```
open .  *** induction step
*** induction hypothesis
  eq n -(l + m) = (n - l) - m .
  eq l + m <= n = true .
*** thus
  eq l <= n = true .
  eq m <= r:ENat>Nat(n - l) = true .
red s n - (l + m) == (s n - l) - m .
close
```

All this code has been executed, and the two reductions above indeed give `true`; therefore our result is proved (modulo the lemmata). We have also specified category theory, where the composition operation is partial, and done some proofs using retracts. It would not be hard to support this style of reasoning with a preprocesser for OBJ.

4 Further Hidden Research

The quest to stretch equational logic in other useful ways continues. The way with which I am most familiar is called "hidden equational logic" or "hidden algebra." It handles states along with associated features of current interest in computer science, including concurrency, nondeterminism, distribution and inheritance. This adventure is still at an early stage, but some promising results can be found in [9] with further references. Many related adventures are under way, including a dual adventure called coalgebra [16], and the new CafeOBJ industrial strength OBJ implementation, which directly implements hidden algebra and includes a powerful support environment [4].

References

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

[2] Jan Bergstra and John Tucker. Characterization of computable data types by means of a finite equational specification method. In J.W. de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming, Seventh Colloquium*, pages 76–90. Springer, 1980. Lecture Notes in Computer Science, Volume 81.

[3] Garrett Birkhoff. On the structure of abstract algebras. *Proceedings of the Cambridge Philosophical Society*, 31:433–454, 1935.

[4] Kokichi Futatsugi and Ataru Nakagawa. An overview of CAFE specification environment. In *Proceedings, ICFEM'97*. 1997. Hiroshima, November 1997.

[5] Joseph Goguen. Semantics of computation. In Ernest Manes, editor, *Proceedings, First International Symposium on Category Theory Applied to Computation and Control*, pages 151–163. Springer, 1975. San Fransisco, February 1974. Lecture Notes in Computer Science, Volume 25.

[6] Joseph Goguen and Răzvan Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4:363–392, 1994.

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

[8] Joseph Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs.* MIT, 1996.

[9] Joseph Goguen and Grant Malcolm. A hidden agenda. Technical Report CS97–538, UCSD, Dept. Computer Science & Eng., May 1997. Early version in *Proc., Conf. Intelligent Systems: A Semiotic Perspective, Vol. I*, ed. J. Albus, A. Meystel and R. Quintero, Nat. Inst. Science & Technology (Gaithersberg MD, 20–23 October 1996), pages 159–167.

[10] Joseph Goguen and José Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics*, 11(3):307–334, 1985.

[11] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992. Drafts exists from as early as 1985.

[12] Joseph Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Raymond Yeh, editor, *Current Trends in Programming Methodology, IV*, pages 80–149. Prentice Hall, 1978.

[13] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Algebraic Specification with OBJ: An Introduction with Case Studies.* Academic, to appear. Also Technical Report SRI-CSL-88-9, August 1988, SRI International.

[14] Claude Kirchner, Hélène Kirchner, and José Meseguer. Operational semantics of OBJ3. In T. Lepistö and Aarturo Salomaa, editors, *Proceedings, 15th International Colloquium on Automata, Languages and Programming, Tampere, Finland, July 11-15, 1988*, pages 287–301. Springer, 1988. Lecture Notes in Computer Science, Volume 317.

[15] José Meseguer and Joseph Goguen. Order-sorted algebra solves the constructor selector, multiple representation and coercion problems. *Information and Computation*, 103(1):114–158, March 1993. Revision of a paper presented at the second LICS, 1987.

[16] Horst Reichel. An approach to object semantics based on terminal co-algebras. *Mathematical Structures in Computer Science*, 5:129–152, 1995.