A Type Algebra for SETL — Preliminary Proposal

Fritz Henglein

March 13, 1987

1 A little bit of background

SETL has a notion of several distinct (data) types, such as sets, maps, tuples, and simple types like integer, real, string, atom. The fact that objects of these types may only operate in a controlled fashion with each other makes SETL a *typed* language; since this is done at runtime, it is *weakly typed.* So, for example, applying addition to an integer value and a procedure doesn't make all that much sense to SETL and its runtime system will take a short cut to the end of the program execution at this point; this is in contrast to a machine language program which will happily do anything it can without giving terribly much attention to the type of the objects involved.

Runtime type checking, however, is costly in terms of memory management (space and organizational complexity) and execution speed. Moreover, since type checking is only performed at program execution time instead of program composition or compile time, type errors are detected very late.

To remedy — at least partially — the first deficit, namely the time and space cost of execution time type checking, there have been various attempts at finding the types of program objects (mostly variable occurrences) at compile time [Tenenbaum 74, Jones/Muchnick 76, Kaplan/Ullman 78, Suzuki 81, Weiss 86]; such information can consequently be used for circumventing some of the dynamic type checks. This work can be said to have arisen within a framework of "permissive" languages: It is assumed the programmer knows what s/he is doing, and a type finding algorithm tries to collect as much information as is possible in reasonable time, but the language itself imposes not many constraints on the programmer at composition time.

Another approach to type checking comes from *strongly typed* languages, in which (almost) all named program objects have to be declared to be of a certain type, and the compiler (or structure editor?!) will check the usage of objects against their type declarations. The produced code contains no dynamic type checks any more. This makes usually for very fast code at the expense of cluttering up the program with type declarations and other redundant information and restricting the flexibility of the programmer. In particular, code that only requires some structural properties of a type to work properly (e.g. sorting, which works on any data type that has a comparison relation) has to be written for every single specific type. Recently developed languages have tried to keep the type checking in the compilation phase while extending the flexibility of the language towards their weakly typed counterparts; foremost amongst these are ML [Milner 78], Poly [Matthews 83], Hope [Burstall, MacQueen, Sanella 80], Miranda [Turner 85], B [Meertens 83, 85].

SETL currently falls into the category of weakly typed languages, but imposing a strong typing discipline along the lines of ML on it promises to increase program efficiency substantially while not compromising its convenience and flexibility too much.

2 Types

While most people would agree with the view that types are semantically some sets of values, this "naive" view cannot be mathematically formalized because we would run into basic set theoretic contradictions. For this reason several quite involved mathematical models of types have been devised, usually based on domain theory and/or category theory. For our purposes the view that types are just sets of values is perfectly adequate, though (as of yet).

The "strong typing problem", as we will call it, can usually be decomposed into three subproblems:

1. provision of a type algebra describing

   - the types available (i.e. what the sets of values are that we consider types) in the form of a type language and
   - the algebraic properties of the latter

2. compilation of type requirements imposed by programming language constructs (e.g., the left-hand side and the right-hand side of an assignment statement have to have the same type)

3. specification of a valid typing, i.e. what constitutes an acceptable association of named program entities (usually identifiers in the program) with types in the type algebra, and ways of constructing such a typing

Naturally, these three subproblems interact with each other, but we believe that the type algebra is the first of them that has to be provided since there the most fundamental decisions are made and the other two subproblems can only be expressed formally after it is dealt with.

For this reason this note contains a preliminary proposal for a type algebra for SETL. We hope that the decisions incorporated herein are well-motivated in relevant past research and in the spirit of SETL itself.

3 Special Type Requirements for SETL

Since SETL, just like ML, is a language that mandates hardly any identifier declarations we have been guided by the approach to type inference taken in ML. SETL, however, has some features that necessitate a treatment more general than in ML.

First of all, SETL heavily uses overloaded operators (not functions, though). Since most of these overloaded uses cannot be resolved — and indeed should not be resolved — locally but only in a wider program context, there is a need for union types to capture this behavior without having to resort to restrictive type declarations or even type errors. Furthermore union types are essential in a language with overloading as the NP-completeness result of [Burstall, MacQueen, Sanella 80] shows.

Second, SETL supports the construction and use of recursive (data) types *without* unique constructors since no type declarations are necessary. In fact, tuple and set constructors can be viewed as totally overloaded constructors for all recursive types. This necessitates a careful treatment of (implicit) recursive types.

Third, SETL provides type testing predicates; their interaction with type inference has to be examined carefully.

And fourth, the polymorphism in SETL is more general for two reasons: As we expect to treat functions as first class types it should be possible to return polymorphic functions, and type instantiation of the arguments at the call site of a function with union typed arguments has to be treated separately.

4 The Type Language

We describe the set of types in our language in three stages:

1. constant types
2. type constructors
3. type combinators

Constant types are the smallest possible types of constant denotations occurring in a program, essentially just the singleton set consisting of the denoted value itself. These types allow a refined treatment of record types and, in general, a more precise type analysis.

Type constructors allow us to build the usual monomorphic world of types: integers, reals, sets of integers, tuples of reals, and so forth. Characteristic of them is that they constitute the “free” part of the type algebra, that is, no two different type expressions built solely from type constructors are considered equal (see minor exception later, though).

Type combinators contain the heart of the complex properties of our type algebra. We will introduce union types, recursive types, and polymorphic types. These types address the aforementioned special needs of SETL, but they also contain the hard core of the mathematical and algorithmic problems of type inference.

4.1 Constant Types

There is a constant type for every possible constant denotation in SETL. Since there are infinitely many of these, they can’t be listed here. Instead they are grouped into categories according to the smallest nonconstant types they can be coerced into (implicitly).

- integer constant types (e.g. 5, -8, 0)
- real constant types (e.g. 5.0, -.33, 1.9E87)
- string constant types (e.g. ’hello’, ’can”t do without it’)
- boolean constant types (namely false and true)
- omega constant type (namely om)

> Parenthetical remark: We hope that the use of 5 for both the value and the constant type consisting only of value 5 is not confusing, but rather emphasizes the close relationship of the value with its constant type.

We will use the notation [value¿: [type]for the statement “[value]is of type [type]” in the following. Legal type statements would be

```
5: 5
'hello': 'hello'
3.0: 3.0
5: integer
3.0: real
'hello': string
```

but not

```
5: 5.0
v: 5 (where v is a variable initialized to 1)
```

4.2 Type constructors

As we mentioned before, the type constructors provide the monomorphic type universe as it is present in Pascal or Algol68 or C for that matter, but without variant (union) types and recursive types (SETL also doesn't have any pointers).

The type constructors are most conveniently separated into nullary and nonnullary constructors.

4.2.1 Nullary Type Constructors

The following are types:

- integer (the integer values)
- real (the floating point values)
- boolean (the boolean values false and true)
- string (the character strings)
- OM (the undefined value om)

- atom (the blank atoms)
- unit (the "no-value" type)

4.2.2 Nonnullary Type Constructors

The nonnullary type constructors roughly fall into two classes: static and dynamic type constructors.

Static Nonnullary Type Constructors Let a, b, c, ... be types; then the following are also types:

- { a, b, c, ...}
- [a, b, c, ...]

Any finite number of types can appear inside the {} and []. Static types of this sort are mostly used to define named and indexed record types. [integer, real, string], for example, stands for the type of triples, i.e. an indexed record, consisting of an integer in the first position, a real in the second position, and a string in the third position. In contrast, {integer, real, string} stands for the type of three-element sets with exactly one integer, one real, and one string. Note that types inside {} can be repeated, but their position is irrelevant. The {} type seems quite useless in the previous example, but we can model named records with it: { ['year of birth', integer], ['weight', real], ['name', string] } can be viewed as the named record version of the indexed record above. Note that the availability of constant types facilitates this integration of record types into the language without a distinct record facility in the language. Furthermore, since not only string constants can function as keyword types, a more general named record type facility is provided.

Dynamic Nonnullary Type Constructors Let a, b be types; then the following are also types:

- set(a) (the sets of elements of type a)
- tuple(a) (the tuples of elements of type a)
- function(a,b) (the functions with domain type a and codomain type b)

- smap(a,b) (the finite single-valued maps with domain type a and range type b)
- mmap(a,b) (the finite multi-valued maps with domain type a and range type b)

Although mmap(a,b) could be viewed simply as a notational abbreviation for set([a,b]), one could argue that, instead, they should be different to distinguish between the different usages of a value as a map, as in f{x} := S, or simply as a set (no map operations). Of course there would be implicit coercions from mmaps to the appropriate sets, but not the other way around.

> Parenthetical remark: The distinction between smaps and mmaps would have an analog in functions and generators, if SETL ever were to incorporate generators.

Static types can always be coerced into the appropriate dynamic types should the need arise. {integer, integer, integer} can be coerced into set(integer), and [real, real, real] can be coerced into tuple(real). Once we will have introduced union types in the next subsection, even heterogeneous static types such as {integer, real} or [string, real] can be coerced into dynamic types.

4.3 Type combinators

The type combinators give us unioned/intersected, recursive, and polymorphic types.

(4.3.1) Finite Union Types and Finite Intersection Types

We can create new types by taking the finite union of types or the finite intersection of types. Union types contain what is sometimes called variant records or simply unions, while intersection types contain overloaded objects, mostly overloaded operators.

Let a, b be types; then

- $a \mid b$

is also a type, namely the union of the types a and b. E.g., integer — real is the type of numbers; integer — real — string — boolean — OM is the type of all basic values with constant denotations; set(integer) — set(real) is the type of sets containing only integers or only reals, but set(integer — real) is the type of sets containing integers and reals, possibly mixed.

Let a, b be types; then

- a & b

is also a type, namely the intersection of the types a and b. E.g., integer & real is the type of values that are both integers and reals. Depending on the interpretation of integer and real this is either the empty type (if integer values are considered a separate breed of objects from reals) or the integer type (if integer values are considered a subset of the reals). Most often, however, we will see intersection types in the context of overloading. The "+" operator in SETL, for example, has type

"+": (real x real → real) & (integer x integer → integer) & (string x string → string) & ... (sets and tuples)

It may come as a surprise at first that overloading does not correspond to union types but instead to intersection types, since intuitively "+" seems to "add" its basic functionalities together just like a union type. That this is not so, can be seen by considering the following line of reasoning.

If a variable x has type integer we can safely conclude that x has also type integer — real, but given y of type integer — real we don't know whether or not y has type integer. Now, if "+" had the type (integer x integer → integer) — (real x real → real) — ... we couldn't conclude that "+" has also type integer x integer → integer, but this has to be the case because of the very essence of overloading.

> Parenthetical remark: if you know a better way of exemplifying this, please let me know as soon as possible. It took me quite some time to realize that overloading is not modelled by union types, and now I am looking for a convincing way of making sure other people don't fall into the same mental trap.

4.3.1 Recursive types

Let x be a type variable, and let s[x] be a type expression in our type language with the additional type variable x, then

- μ x. s[x]

is a (directly recursive) type.

E.g., μ intlist = [integer, intlist] — [] is the type of integer lists; μ bintree = [bintree, integer, bintree] — [] is the type of binary trees with integer values. Some recursive types don't make much sense, though. μ x. x could be viewed as the (dynamic) "error" type; it contains no "real" element. Recursive types constructed from the function type constructor are quite peculiar. It is not clear if a type like μ x. function(x, integer) is very useful or even desireable, and only recently have [MacQueen, Plotkin, Sethi 84] come up with a model for such types.

For mutually recursive types we also introduce the following types. Let x, y, ... be type variables, and let s[x,y,...], t[x,y,...], ... be type expressions containing type variables x, y, ..., then

- μ [x, y, ...]. [s[x,y,...], t[x,y,...], ...]

is a (collection of mutually recursive) type(s) as long as the number of type expressions on the right is the same as the number of type variables on the left.

4.3.2 Polymorphic types

Let x be a type variable, and let s[x] be a type expression containing x, then

- V x. s[x]
- s[x]

are types. In the first case we have a fully independent polymorphic type, while in the second case we have a possibly dependent polymorphic type. A type statement

v: V x. s[x]

should be understood as "v has type s[x] simultaneously for every type x there is", but

v: s[x]

should stand for "v has type s[x] for any one type x stands for". To illustrate the difference consider the following program fragment.

```
procedure copy(n);
k := n;
return n;
end procedure;
```

We get the following type statements (x and y are type variables):

```
k: x
n: x
copy: V y. y -> y
```

k and n have the same type x, but *not* V x. x, since the type of n can be *any* type depending on the actual argument at a call site of copy, but it is not *every* type simultaneously, which would be expressed as V x. x. In any reasonable interpretation of types there is only one "element" that has every type: the error value, sometimes called "bottom" because of its domain theoretic relevance. The procedure copy, on the other hand, has all types y → y "simultaneously"; e. g., it is of type integer → integer as well as real → real as well as (μ x. [integer, x] — []) → (μ x. [integer, x] — []).

Universal polymorphism is related to intersection types. Whereas intersection types are the finite intersection of some given types, universal polymorphism describes the infinite, universal (over *all* types) intersection of a set of types given by a type pattern.

4.4 Grammar for type language

We will summarize the context-free language aspects of our type algebra in a (context free) grammar; the algebraic aspects are tackled in the next section.

```
TYPE ::= CONSTANT_TYPE |
MONO_TYPE_CONS |
POLY_TYPE_CONS |
TYPEVAR

CONSTANT_TYPE ::= INTEGER_CONSTANT_TYPE |
REAL_CONSTANT_TYPE |
STRING_CONSTANT_TYPE |
BOOLEAN_CONSTANT_TYPE |
OMEGA_CONSTANT_TYPE

INTEGER_CONSTANT_TYPE ::= <integer constants>
REAL_CONSTANT_TYPE ::= <real constants>
STRING_CONSTANT_TYPE ::= <string constants>
BOOLEAN_CONSTANT_TYPE ::= false | true
OMEGA_CONSTANT_TYPE ::= om

MONO_TYPE_CONS ::= NULLARY_TYPE |
NONNULLARY_TYPE_CONS
```

```
NULLARY_TYPE ::= integer |
real |
boolean |
string |
OM |
atom |
unit

NONNULLARY_TYPE_CONS ::= STAT_NN_TYPE_CONS |
DYN_NN_TYPE_CONS

STAT_NN_TYPE_CONS ::= { TYPE_LIST } |
[ TYPE_LIST ]

TYPE_LIST ::= TYPE  TYPE_LIST |
<empty>


DYN_NN_TYPE_CONS ::= set ( TYPE ) |
tuple ( TYPE ) |
function ( TYPE , TYPE ) |
smap ( TYPE , TYPE ) |
mmap ( TYPE , TYPE )

POLY_TYPE_CONS ::= UNION_TYPE_CONS |
INTERSECT_TYPE_CONS |
REC_TYPE_CONS |
POLY_TYPE_CONS

UNION_TYPE_CONS ::= TYPE ‘‘|’’ TYPE

INTERSECT_TYPE_CONS ::= TYPE & TYPE

REC_TYPE_CONS ::= DIR_REC_TYPE_CONS |
MUT_REC_TYPE_CONS

DIR_REC_TYPE_CONS ::= m TYPEVAR . TYPE_WITH_VAR

TYPEVAR ::= <type variable>
```

```
TYPE_WITH_VAR ::= TYPE

MUT_REC_TYPE_CONS ::= m [ TYPEVARLIST ] . [ T_W_V_LIST ]

TYPEVARLIST ::= TYPEVAR TYPEVARLIST |
<empty>

T_W_V_LIST ::= TYPE_WITH_VAR T_W_V_LIST |
<empty>

POLY_TYPE_CONS ::= IND_POLY_TYPE_CONS |
DEP_POLY_TYPE_CONS

IND_POLY_TYPE_CONS ::= V TYPEVAR . TYPE_WITH_VAR

DEP_POLY_TYPE_CONS ::= TYPE_WITH_VAR
```

5 Algebraic properties of types

As we shall see in the type requirements part of this trilogy, it is necessary to reason about, amongst other things, equality of types. We have noted several times that the type constructors have hardly any algebraic laws they satisfy, and so determining equality or inequality of two type expressions (and even their unification) involving only type constructors and constant types is pretty easy.

It is a different matter to decide equality of types (and in particular to do unification) in the presence of algebraic laws. In a companion note we have already pointed out that the unrestrained use of intersection types (overloading) makes type inference infeasible (see [Aho, Sethi, Ullman 86, ex. 6.25] and previous note on Overloading and NP-completeness). In this section we will present only the basic properties of our type algebra and leave a more involved investigation for later study when we will introduce the context in which they will be needed, namely S-unification (where S is an equational theory).

5.1 Union and intersection types

Union and intersection types have the obvious properties; they are associative, commutative, and idempotent. The latter sets them apart from disjoint unions, which lack idempotency. In fact they satisfy all the axioms of monotone set algebra (since they're essentially nothing else but set union and set intersection).

Symbolically, for types a, b we have

```
a | b  =  b | a a & b  =  b & a
(a | b) | c  = a | (b | c) (a & b) & c  = a & (b & c)
a | a  =  a a & a  =  a
```

Since overloading (intersection types) is permitted in a very restricted sense only in SETL, we won't have to examine the interaction of — and & in general.

5.2 Recursive Types

Given a recursive type μ x. s[x], we can "unfold" the definition of x by substituting s[x] for x in s[x].

Symbolically,

μ x. s[x] = μ x. s[x] { s[x]/x }

Of course, given an unfolded recursive type we can "fold" it back into the original type definition.

> Parenthetical remark: One has to be careful with the folding/unfolding rules since oftentimes they don't preserve equality. I'll look at this later.

5.3 Polymorphic types

Type variables bound by V can be renamed without changing the meaning of the type expression.

V y. s[y] = V z. s[z]

Other properties of polymorphic types will be introduced along with an ordering relation on types later on.

6 The question of equality and unification of types

Unification attempts to find a solution for the equation s[x,y,...] = t[x,y,...]. If unification is to be an algorithmically feasible task, then necessarily it should be possible to decide if equality holds between two type expressions.

[Solomon 78] has pointed out that equality of socalled context-free recursive types is computationally equivalent to the equivalence problem for deterministic push-down automata. Since it is not even known if the latter problem is decidable, the presence of these recursive types would make unification and thus type inference or type checking hopeless. Luckily our recursive types are more restrictive; in Solomon's terminology they are called regular recursive types. It is instructive to analyse a program that constructs values of a proper context-free recursive type and to see with what kind of type a hypothetical SETL type inference machine would come up. We will present such a case study in our next note.