A Polymorphic Type Model for SETL

Fritz Henglein
Courant Institute of Mathematical Sciences
New York University
251 Mercer St.
New York, N.Y. 10012, USA
Internet: henglein@nyu.nyu.edu

July 17th, 1987

Abstract

We present a polymorphic type model for a very high level algorithmic language with set theoretic dictions largely similar to SETL [SDDS86]. The model is based on ML-style polymorphism [Mil78], yet extends it in two fundamental ways. It is possible to infer undeclared recursive types, and it adds a restricted form of polymorphic types, termed finitary polymorphic types. This, in contrast to ML, permits generic operators such as '+' to be handled in the type inference system. We briefly address the modifications to the unification based type inference algorithm for ML to deal with these extensions.

1 Introduction

SETL [SDDS86] is a very high level programming language with set theoretic dictions. It is a largely declaration-free algorithmic language with a weak typing discipline that is geared towards prototyping applications. It is safe to say that the prominent high level features of SETL were instrumental in the successful validation of the first Ada compiler [DaC84]. Practical experience, however, also indicates that the weak typing discipline — checks for type consistency are only done at run time, not at compile time — results in enormous execution time and memory overhead, as well as a large amount of type errors that go undetected until run time.

To improve the execution characteristics, Tenenbaum [Ten74] and Weiss [Wei86] devised type finding algorithms that collect type information from variable usage and and data flow information. Tenenbaum's type finder was in fact implemented as part of the SETL optimizer [FSS75]. Its payoff has been very limited, though, mostly because it has to be able to analyze "anything goes" programs.

For this reason the design of the successor of SETL, tentatively termed SETL-2, incorporates amongst its design goals the adoption of a strong typing discipline without, however, compromising the "spirit" of SETL. More specifically, the new typing discipline should permit variable declarations, yet mandate as few as possible; it should provide a syntactic criterion for representation independence [Rey74, Don79, Fok81] so that there is no need to maintain type information at run time; it should be sufficiently robust in the context of an evolving overall language design; and, of course, it should admit a theoretically and practically efficient type checking/inference algorithm in order not to create a bottleneck in compiler performance.

In section 2 we introduce the type model for our language. We give an informal exposition of syntax and semantics in section 3. This is followed by a logical specification of the type inference problem for our language in section 4. In section 5 we touch upon the implementation of the novel features in our type model within a unification based type inference algorithm. Section 6 contains some examples of type correct program fragments, and section 7 finishes the paper with some concluding remarks.

2 The Basic Type Model

The type model specifies the collection of types we are working with. We will not present a formal semantics for types here, but appeal to their intuitive character as "sets of objects". We feel this is sufficient at this point to justify the typing rules we will describe in section 4.

2.1 Monomorphic Types

2.1.1 Simple Types

The types are presented in the form of *type expressions*, which, for convenience' sake, we simply call types, too. First, we introduce the *simple types* in grammatical form. The simple types comprise all the data types for which there are operators and constants in the programming language that construct values of those types.

```
TYPE     ::=   BASETYPE |
               set(TYPE) |
               sequence(TYPE) |
               tuple(TYPE ...) |
               smap(TYPE ...)TYPE |
               procedure(TYPE ...)TYPE
               union(a1 <- TYPE, a2 <- TYPE,..., an <- TYPE)
               record(a1 <- TYPE, a2 <- TYPE,..., an <- TYPE)
```

```
BASETYPE ::=    integer | real | bool | atom | char
```

The base types are the well-known primitive (computational) data types `integer`, `real`, `boolean`, and `character`. `atom` is an abstract primitive type for which equality is the only defined operation. There are no constant denotations of type `atom`; however, atoms can be generated with the newat primitive, which is similar to gensym in LISP systems. For any type τ, `set(`τ`)` is the type of all finite sets whose elements are of type τ, `sequence(`τ`)` is the type of finite sequences with elements of type τ, and `smap(`$\tau_1, \tau_2, \ldots, \tau_n$`)`$\sigma$ is the type of finite single-valued maps whose arguments have types $\tau_1, \tau_2, \ldots, \tau_n$, respectively, and whose range elements have type σ; `tuple(`$\tau_1, \tau_2, \ldots, \tau_n$`)` is the type of n-tuples whose elements have types $\tau_1, \tau_2, \ldots, \tau_n$ in the appropriate positions[1]; `procedure(`$\tau_1\tau_2, \ldots, \tau_n$`)`$\sigma$ is the type of all procedures with n arguments of types $\tau_1, \tau_2, \ldots, \tau_n$, respectively, and whose result is of type σ; `union(a1<-`τ_1`, a2<-`τ_2`, ..., an<-`τ_n`)` stands for the type of tagged unions, the tags being the tokens a1, a2,..., an with associated values of types $\tau_1, \tau_2, \ldots, \tau_n$; finally, `record(a1<-`$\tau_1$`, a2<-`$\tau_2$`, ..., an<-`$\tau_n$`)` is the type of labelled records with labels (fieldnames) a1, a2,..., an, and corresponding values of types $\tau_1, \tau_2, \ldots, \tau_n$.

This type model departs in several respects from the SETL type model formulated in [Ten74] or [Wei86]. For one, no free disjoint union type is available such as `integer`|`real`. Instead tagged unions are provided. With tagged unions we not only avoid problems having to do with implicit coercions from a type τ to a free union type containing τ as an alternand, they are also more general in some sense since we can draw distinctions between values of the same type having different tags associated with them. Furthermore, we include procedures as first class objects, which we think necessary for a language of very high level. We also provide a labelled record type, the absence of which has been criticized repeatedly and has led to the extensive use of SETL macros to emulate a record facility.

2.1.2 Recursive Types

The usefulness of recursive types was recognized early on and found its expression in many strongly typed programming languages beginning with ALGOL68 [vWMPK69]. In all of these languages recursive types have to be declared explicitly, and variables, likewise, have to be declared explicitly to be of a recursive type. Even ML, which is largely declaration free, mandates declarations for recursive types in such a way as to make the detection of expressions of such a type trivial.

Weiss in his thesis [Wei86] extended the type model of [Ten74] with recursive types and observed that expressions with recursive types can be found with data flow analysis. It has been known that recursive types can also be inferred in the syntax-directed polymorphic type discipline of ML by removing the "occurs check" from the unification algorithm employed in it, although references seem to be hard to come by [Rey85, page 113]. Since our discipline is basically syntax-oriented, our type inference algorithm (see section 5) illustrates the latter technique.

To accommodate recursive types we add the following productions to the grammar for simple types.

```
TYPE      ::=    fix VAR . TYPE
BASETYPE  ::=    VAR
VAR       ::=    <type variables>
```

The presence of type variables in type expressions is semantically quite puzzling at first. What is the type denoted by an expression that contains a type variable? It is best to think of

[1]Note that our usage of tuple and sequence is congruent with mathematical usage, but exactly *opposite* from the terminology of [Wei86], where our sequences are tuples and our tuples are sequences.

such type expressions as *type functions*, which map argument types to a result type[2]. With this understanding, a recursive type `fix` α`.f(`α`)` is the smallest type τ that satisfies the constraint τ `= f(`τ`)`. Of course, if `f(`α`)` contains another type variable, `fix` α`.f(`α`)` more properly defines a type *function*, but we will avoid such complications in our informality in order to keep things simple.

Recursive types let us introduce ML-style lists, which are not primitive (i.e. simple) data types in our model. The type of lists of integers is denoted by `fix` α`.tuple(integer,`α`)`. More generally, the list type over elements of any type τ is `rec` α`.tuple(`τ`,`α`)`. For reasons of simplicity, we have only included directly recursive types. Mutually recursive types are notationally more cumbersome, but constitute no additional technical problems.

The simple types together with the recursive types constitute the collection of *monomorphic types.*

2.2 Polymorphic Types

If we want to write a function that returns the length of an input list in a strongly typed languages with only monomorphic types, such as Pascal, we have to provide a separate function definition for every element type of the input lists even though that type is inessential for the function. While this is a nuisance in a language with mandatory variable declarations, it is unacceptable in a largely declaration-free language since, although all the copies would be identical, as many of them are needed as there are list element types in calls. To associate a type with the length function we need to introduce some form of *type polymorphism.* Before we present our form of type polymorphism, though, we will go over some of the common models of polymorphism to provide some reasoning for our approach to polymorphism from the point of view of SETL.

A very powerful form of polymorphism in a programming language can be found in the Second Order Lambda Calculus (SOL) [Gir71, Rey74]. SOL is, however, a fully declared language: every variable is explicitly declared with its type. At present these declarations cannot be omitted and replaced by automatic type inference since the decidability of the type inference problem for SOL is still an open problem. Similarly, the languages Russell [BDD80] and Poly [Mat83, Mat] are based on the notion of types as first class objects and full parameter declaration. Since this is in conflict with our design goals we opted not to adopt these powerful polymorphic type disciplines. The situation is not much different in Cardelli's *inclusion* or *subtype* polymorphism [Car84, Rey85, CW85]. It is based on a partial ordering between types and the axiom that an object of type τ has also type τ' for any type τ' greater than τ. This corresponds to implicit coercions from the smaller type τ to the bigger τ. An example would be the coercion of an integer typed value into the (same) value of the free union type `integer`|`real`. Under appropriate conditions, every typable expression has a unique *minimal* type [Rey81, Rey85]. This property hinges critically on the requirement that the types of function parameters be declared. Since this, also, is in conflict with our design goal of assuring a maximum of declaration-freeness, we have excluded inclusion polymorphism at this point. As a consequence, free union types are not part of our type model currently.

Our form of polymorphism builds on and extends the socalled *parametric polymorphism* as it can be found in ML [GMM+78], Hope [BMS80], Miranda [Tur85], or B [Mee83]. We first describe the kind of infinitary polymorphism in our language that is common to all of these languages. Then we proceed to describe our novel extension, terming it finitary in contrast to infinitary polymorphism.

[2]Type expressions with no type variables can thus be identified with constant functions.

2.2.1 Infinitary Parametric Polymorphic Types

Infinitary parametric polymorphic types, or simply *polytypes* [Mil78], are expressed by unbounded quantification over type variables. We amend our type grammar with the following rules.

```
POLYTYPE  ::=     forall VAR . POLYTYPE |
                  TYPE
```

A polymorphic type `forall` α`.f(`α`)` is the biggest type in an appropriate domain of types contained in the type `f(`τ`)` for every *monomorphic* type τ. In a set-theoretic domain this is simply the intersection of the `f(`α`)`'s for all monomorphic τ's. A procedure of type `forall` α`. procedure (sequence(`α`)) integer` can be applied to sequences of *any* type.

2.2.2 Finitary Parametric Polymorphic Types

Finitary parametric polymorphic types, or simply *oligotypes*, capture operators sometimes confusingly called generic [Rey85], which is why we prefer to refer to them as *dynamically overloaded* [Hen87]. We express an oligotype by bounded quantification over type variables, as indicated in the following additional grammar rule.

```
POLYTYPE   ::=     forall VAR in {TYPE ...}. POLYTYPE
```

The meaning of the type expression `forall` γ `in` $\{\tau_1, \tau_2, \ldots, \tau_n\}$`.f(`$\gamma$`)` is the biggest type contained in `f(`γ`)` for every type γ in $\{\tau_1, \tau_2, \ldots, \tau_n\}$. `forall` γ `in` $\{$`set(`α`), sequence(`α`)`$\}$`. procedure(`α, γ`)bool` is the oligotype of the SETL operator `in`.

Some examples in section 6 will demonstrate the significance of oligotypes.

3 Syntax and Semantics

In this section we will present the basic syntax and an informal semantics of an abstract language that is meant to capture the salient features of SETL-2 *as they relate to typing issues.* We are able to greatly reduce the conceptual complexity of the basic type inference problem and make it largely independent from the idiosyncracies of SETL in this way. Relating the syntax to concrete SETL provides the informal semantics of our language.

3.1 Syntax

The syntax is given below in an essentially abstract form. It is spiced up with familiar concrete elements to facilitate understanding.

```
program          ::=     'program' exp

exprs            ::=     exp ...
exp              ::=     set |
                         sequence |
                         procedure |
                         tuple |
                         smap |
                         union |
                         record |
                         quantexpr |
                         unionappl |
                         tagcheck |
                        recordappl |
                        application |
                        identifier |
                        constant |
                        '(' vardecls block ')'
set             ::=     '{' exp : vardecls '|' exp '}'
sequence        ::=     '[' exp : vardecls '|' exp ']'
procedure       ::=     'procedure' '(' vardecls ')' exp
tuple           ::=     '[' exprs ']'
smap            ::=     '{' '[' vardecl exp ']' '|' exp
union           ::=     '[' field ']'
record          ::=     '{' fields '}'
fields          ::=     field ...
field           ::=     label '<-' exp
label           ::=     identifier
quantexpr       ::=     quantifier vardecl '|' exp
quantifier      ::=     'forall' | 'exists'
unionappl       ::=     exp 'as' label
tagcheck        ::=     exp 'is' label
recordappl      ::=     exp '.' label
tupleappl       ::=     exp '[' constant ']'
application     ::=     operator exprs
operator        ::=     '()' | 'with' | 'less' | ':=' |
                        '=' | '<' | 'read' | 'print' |
                        'om' | 'newat'

block           ::=     constdecl block |
                        statements
statements      ::=     statement ...
statement       ::=     'yield' exp |
                        'return' |
                        'if' exp statements statements |
                        'while' exp statements |
                        'for' vardecl ':' exp statements |
                        exp

vardecls        ::=     vardecl ...
vardecl         ::=     identifier
constdecl       ::=     identifier '=' exp
```

identifier and constant are lexical categories.

3.2 Semantics

3.2.1 Expressions

We give a list of expressions denoting a set, a sequence, a procedure (as part of a constant declaration), a tuple, and a single-valued map, respectively. The notation for our language is on the left with the corresponding formulations in SETL on the right.

```
{x: x | x in S and x > 0}           {x : x in S | x > 0}

[y+1: y | y in S and y > 0]         [y+1: y in S | y > 0]
```

```
lte = procedure (x,y)               procedure lte(x,y);
        ( yield x < y                            return x < y;
          return )                  end procedure;

[5, 5.0, 'hello']                   [5, 5.0, 'hello']

{[x,x+1]: x | x in S and x > 0}        {[x,x+1] : x in S | x > 0}
```

We make no difference between an iterator and a boolean valued expression in set and tuple formers — in contrast to SETL — since we don't have to worry as to whether the expression is executable or not.[3] Our notation mirrors to some degree the approach taken in Miranda [Tur85], yet provides more flexibility. The first expression denotes the set of all elements in `S` that are greater than `0`. The second expression describes the sequence of successors of positive elements in `S`. If `S` is a set, the order of the elements in the sequence is indeterminate; if `S` is a sequence the order is determined by the order in `S`. The fourth expression is simply a 3-tuple (triple) with the integer element `5`, the real value `5.0`, and the character sequence `hello`. The fifth expression, finally, represents a single-valued map mapping all positive elements in `S` to their successors.

The correspondence between quantified expressions in our language and SETL is given below.

```
forall x | x in S ==> x in T        forall x in S | x in T

exists y | y in S and y in T        exists y in S | y in T
```

The first expression returns true if and only if all elements in `S` are also in `T`, and the second expression evaluates to true if and only if there is an element in `S` that is also in `T`.

As mentioned before, our language provides data types not directly available in SETL. The additional operations on these are exercised in the following code fragment.

```
u := [ complex <- [5.0, 5.7] ]
if (u is complex) then
        print ``Real part is'' (u as complex)[1]

r := { birthdate <- [3, 6, 1961], name <- ``Fritz'' }
print ``Age is'' r.birthdate
```

In this example we first assign a tagged value to `u`. Then we check if `u` currently has the tag `complex`, and if so print the constant string `Real part is` followed by the first component of `u`, which is assumed to have the tag `complex`. Note that `u as complex` returns the tuple associated with the tag `complex` *if* `u` currently has that tag, otherwise it generates a dynamic exception that aborts program execution[4]. Of course we could add an exception handling facility to be able to resume execution. Since for typing purposes this is inessential, we have chosen not to do so. Coming back to the example code above, in the fourth line we assign a record with fieldnames `birthdate` and `name` to `r`. In the last line, the 3-tuple associated with `birthdate` in `r` is printed.

[3]It seems possible to develop flexible syntactic conditions (a la "safety" in relational calculus [Ull82, pp. 159ff]) to insure that set (tuple) expressions in our language evaluate to finite sets (tuples) only.

[4]Alternatively we could also make it return `om`, but then we could not distinguish `[real <- 5.0] as complex` from `[complex <- om] as complex`.

3.2.2 Statements and Blocks

Statements are just like expressions; the only difference is that they have no associated values. Their principal purpose is to modify the the store and the environment. The statement forms that have a direct equivalent in SETL are shown below.

```
for x : x in S                  (for x in S)
   if x < 0 then                   if x < 0 then
      x := x+1                        x := x+1;
   else                            end;
      om

while y > 0                     (while y > 0)
   y := y-1                        y := y-1;
                                end;
```

In this code `om` is a statement, a "no-op", similar to skip in ALGOL68, only present to make explicit that nothing is to be executed in the else clause. `om` has type `forall` $\alpha.\alpha$, which is synonymous to `void` in our type system (compare section 4). We interpret the `for` statement to be the repeated execution of the statements in the `for` loop body for every value of `x` that satisfies the boolean expression in the loop header. The `while` loop is standard in both form and semantics.

Our language has, in contrast to SETL, a block structure. Every block can have a value associated with it determined by a `yield` statement. The `return` statement quits and thus finishes execution of a block. The value of the block is the one produced by the last `yield` statement executed. Note that the `return` statement in SETL combines the functionalities of `yield` and `return` in our language. In fact `return e` in SETL is equivalent to `yield e; return` in our language. The block in the example below returns an integer value.

```
digit := ( if exists x | x in S
              print ``nonempty''
              yield 1
           else
              yield 0
         )
```

If there is no `yield` statement in a block, or no `yield` statement is executed, `om` is returned.

4 Specification of the Type Inference Problem

Strong typing is a part of the static semantics of a language. It prunes the set of grammatically correct programs such that any program left over is *type correct*, that is, it cannot incur a type error during program execution. Since we have not formulated a formal semantics for our language, we have to refer to [Mil78] again for a precise definition of type correctness. Intuitively, a type error is the application of an operation to incompatible operands, such as adding an integer to a string, applying the integer 3 as a function, or assigning an integer valued expression to a variable in one statement and a real value in another.

We characterize the set of programs that we consider type correct with an inference system. Since our reasoning proceeds along the programming language syntax we formulate logical statements, called *typings*, for programs, expressions, statements and blocks. The following two subsections contain the language of typings and the inference system that specifies the set of valid typings, respectively.

4.1 Typings

Typings in our type inference system are the equivalent of propositions in propositional calculus or well-formed formulas in first-order logic. Just like those, a typing can either be true or false. Before we can describe the composition of typings, we have to define *type expressions* and *type assignments.*

We already introduced type expressions in section 2, but to have all grammar rules in one place they are given again just below. The type expressions are all the strings derivable from the nonterminal POLYTYPE.

```
POLYTYPE  ::=    forall VAR . POLYTYPE |
                 forall VAR in {TYPE ...}. POLYTYPE |
                 TYPE
TYPE      ::=    BASETYPE |
                 set(TYPE) |
                 sequence(TYPE) |
                 tuple(TYPE ...) |
                 smap(TYPE ...)TYPE |
                 procedure(TYPE ...)TYPE
                 union(a1<-TYPE, a2<-TYPE,..., an<-TYPE)
                 record(a1<-TYPE, a2<-TYPE,..., an<-TYPE)
                 fix VAR . TYPE
BASETYPE   ::=   integer | real | bool | atom | char |
                 VAR
VAR        ::=   <identifiers>
```

A type assignment is a finite mapping from program identifiers to type expressions. It specifies a set of assumptions about the types of the identifiers in its domain.

There are typings for programs, expressions, statements, and blocks, respectively. `A` is a type assigment, `e` a program expression, and σ a type expression, then `A |- e:`σ is a typing. It should be read "Under the type assumptions `A` the expression `e` is type correct and has type σ." The typings for statements and blocks have the same parts, but we will write them down and read them differently. If `A` is a type assignment, `s` a statement, a sequence of statements or a block, and τ a type expression, then `A,`τ `|- s` is a typing. It should be read something like "Given the type of the current block is τ, then, under the type assumptions `A`, the statement (sequence of statements, block) `s` is type correct." The current block refers to the innermost parenthesized block containing s.[5] Finally, if `p` is a program, `|- p` reads "Program `p` is type correct". Of course, not all such typings are acceptable. The acceptable typings are exactly the typings derivable in the type inference system of the next subsection. We consider all the programs `p`, and only those, type correct for which we can derive `|- p` (as the reading of `|- p` suggests).

4.2 The Type Inference System

In the presentation of the inference system we use several notational abbreviations. First, `A[x:`τ`]` denotes the type assigment identical to `A` in all arguments but, possibly, `x`; the type associated with `x` is τ. Second, for `A[x1:`τ_1`][x2:`τ_2`]...[xn:`τ_n`]` we simply write `A[x:`τ`]` whenever no ambiguity can arise. Third, we allow ourselves to write `A |- e1:` τ_1`, e2:` τ_2`, ..., en:` τ_n for `A |- e1:` τ_1`, A |- e2:` τ_2`, ..., A |- en:` τ_n; similarly, `A,`τ `|- s1, s2, ..., sn` for `A,`τ `|- s1, A,`τ `|- s2, ..., A,`τ `|- sn`. Fourth, we write `void` for `forall` $\alpha.\alpha$. Fifth and last, the type $\sigma[\alpha{:=}\tau]$ is the same as σ except that all free occurrences of α in σ are replaced by the monotype τ.

[5]Remember that the execution of a `yield` statement assigns a value to the current block.

Since we have quite a numerous collection of axioms and inference rules for our programming language we break it down into strictly syntax-oriented rules and the other rules, which capture recursive types, oligotypes, and polytpes.

4.2.1 Syntax-Oriented Axioms and Rules

The syntax-oriented rules are partitioned into rules that derive program typings, rules that derive expressions, and rules that derive statements and blocks.

Rule for Programs

```
|- (expression) e:void
-----------------------
|- program e
```

This rule expresses the fact that a program is type correct if the expression it is made up from is type correct under *no* assumptions and has result type void.

Axioms and Rules for Expressions

```
A |- (constant) c:$\tau_c$ (where $\tau_c$ is the type of c)

A[x:$\sigma$] |- (identifier) x:$\sigma$


A[x:$\sigma$] |- e:bool, e':$\tau$
-------------------------------
A |- { e':  x | e }:set($\tau$)
A |- [ e':  x | e ]:sequence($\tau$)

A[x:$\sigma$] |- e:$\tau$
-----------------------------------
A |- procedure (x) e:procedure($\sigma$)$\tau$

A |- e1:$\sigma_1$, e2:$\sigma_2$,..., en:$\sigma_n$
---------------------------------------
A |- [e1, e2,..., en]:tuple($\sigma_1$,$\sigma_2$,...,$\sigma_n$)

A[x:$\sigma$] |- e:$\tau$, e':bool
----------------------------
A |- { [x, e] | e' }:smap($\sigma$)$\tau$

A |- e:$\tau$
---------------------------------------------------
A |- [a<-e]:union(a<-$\tau$,b1<-$\sigma_1$,b2<-$\sigma_2$,/ldots,bn<-$\sigma_n$)

A |- e1:$\sigma_1$, e2:$\sigma_2$,..., en:$\sigma_n$
-------------------------------------------------------------
A |- {a1<-e1,a2<-e2,...,an<-en}:record(a1<-$\sigma_1$,a2<-$\sigma_2$,...,an<-$\sigma_n$)

A[x:$\sigma$] |- e:bool
----------------------
```

```
A |- forall x | e:bool
A |- exists x | e:bool

A |- e:union(a<-$\tau$,b1<-$\sigma_1$,b2<-$\sigma_2$,...,bn<-$\sigma_n$)
------------------------------------------
A |- e as a:$\tau$
A |- e is a:bool

A |- e:record(a<-$\tau$,b1<-$\sigma_1$,b2<-$\sigma_2$,...,bn<-$\sigma_n$)
-------------------------------------------
A |- e.a:$\tau$

A |- e:tuple($\sigma_1$,$\sigma_2$,...,$\sigma_i$,...,$\sigma_n$)
------------------------------
A |- e[i]:$\sigma_i$

A |- (operator) op:procedure($\sigma_1, \sigma_2, \ldots, \sigma_n$)$\tau$
A |- e1:$\sigma_1$, e2:$\sigma_2$,..., en:$\sigma_n$
------------------------------------------
A |- op e1, e2,..., en:$\tau$
```

The rules can quite easily be phrased in natural language with the readings we have provided before. For example,

```
A[x:$\sigma$] |- e:bool
----------------------
A |- forall x | e:bool
A |- exists x | e:bool
```

says that, if `e` has type **bool** under the assumption that `x` has some type σ, then both the quantified expressions `forall x | e` and `exists x | e` are type correct and have type `bool` with no assumption made about `x`. The absence of a type assumption about `x` reflects the fact that `x` is locally scoped inside both quantified expressions. Other assumptions about free variables in `e` — expressed in `A` — are not affected by this rule.

Axioms and Rules for Statements and Blocks

```
A,$\tau$ |- return

A |- (expression) e:$\tau$
---------------------
A,$\tau$ |- yield e

A |- e:bool A,$\tau$ |- s1, s2
----------------------------
A,$\tau$ |- if e then s1 else s2

A |- e:bool A,$\tau$ |- s
---------------------
A,$\tau$ |- while e s
```

```
A[x:σ] |- e:bool A[x:σ],τ |- s
-----------------------------
A,τ |- for x:  e s

A |- (expression) e:void
-------------------------
A,τ |- (statement) e

A,τ |- (statement) s1, (statement) s2,..., (statement) sn
---------------------------------------------------------
A,τ |- s1 s2 ...sn

A[c:σ] |- e:σ A[c:σ],τ |- s
----------------------------
A,τ |- c = e; s
```

While most of the rules are quite easily understood, the last rule deserves some elaboration. It makes polymorphic objects (mostly procedures) declared in a declaration of the form `c = e` accessible by the following parts of a block. This is comparable to the `let` construct in ML. The main difference, though, from a typing point of view is that a polymorphic `c` is also considered polymorphic in its definition `e` here, whereas in ML it would be treated monomorphically in `e`. This is one of the main reasons why ML's type inference algorithm is complete for its inference system [Mil78], while it isn't for our inference system even without recursive types and oligotypes. How hard complete type inference is in our model is an open problem (compare also the concluding remarks in [Mee83]).

4.2.2 Rules for Recursion and Polymorphism

The rules for recursive types, polytypes, and oligotypes are syntactic manifestations of (some of) the semantic properties we ascribed to them in section 2.

Rules for Recursion

```
A |- fix α.τ
-------------------
A |- τ[α:=fix α.τ]

A |- τ[α:=fix α.τ]
-------------------
A |- fix α.τ
```

These two rules simply express that recursive type expressions can be arbitrarily folded and unfolded along occurrences of recursive type variables.

Rules for Polytypes

```
A |- e:σ (α not free in A, but free in σ)
------------------------------------------
A |- e:forall α.σ

A |- e:forall α.σ (τ any monotype)
-----------------------------------
A |- e:σ[alpha:=τ]
```

Rules for Oligotypes

$$\frac{\texttt{A |- e:}\sigma[\alpha\texttt{:=}\tau_1]\texttt{, e:}\sigma[\alpha\texttt{:=}\tau_2]\texttt{,...,}\sigma[\alpha\texttt{:=}\tau_n] \quad (\alpha\ \texttt{not free in A, but free in}\ \sigma)}{\texttt{A |- e:forall}\ \alpha\ \texttt{in}\ \{\tau_1\texttt{,}\tau_2\texttt{,...,}\tau_n\}\texttt{.}\sigma}$$

$$\frac{\texttt{A |- e:forall}\ \alpha\ \texttt{in}\ \{\tau_1\texttt{,}\tau_2\texttt{,...,}\tau_i\texttt{,...,}\tau_n\}\texttt{.}\sigma}{\texttt{A |- e:}\sigma[\alpha\texttt{:=}\tau_i]}$$

5 Type Inference Algorithm

We are currently developing a type inference algorithm for our type inference system. The algorithm is based on the ML type checker (algorithm J of [Mil78]). It has two main extensions, however, corresponding to the addition of recursive types and oligotypes in our type model. The algorithm will be described in a paper under preparation that examines theoretical properties of our type system and some of its derivatives. Here we only touch upon the salient points of the extensions over the ML-style algorithm, of which the reader is hoped to have a basic understanding.

A method for integrating inference of recursive types is described in [ASU86, chapter 6]. It corresponds to eliminating the socalled "occurs" check in unification. In a conventional unification step we have to check if a nontrivial substitution term `t` for a variable `v` contains `v`. This is the occurs check. If so, unification is aborted and a failure indicated. In our case we substitute `fix v.t` for `v` and continue.

To handle oligotypes in unification, we maintain a set of *possible type expression* with bounded (type) variables. Our algorithm assumes that none of these type expressions are variables and that their *outermost constructors* are pairwise distinct. The outermost constructors of `fix` α`.`τ, `forall` α `in` $\{\tau_1, \tau_2, \ldots, \tau_n\}$`.`$\sigma$ and `forall` α`.`ρ are the the outermost constructors of τ, σ, and ρ, respectively. We now describe how to unify a bounded variable β with a type expression τ. Since β is bounded, it has an associated set `S` of possible types. If τ is an unbounded variable α, the unified term is β and the most general unifying substitution is $\{\alpha{:=}\beta\}$. If τ is a bounded variable with a set `T` of possible type expressions, then pair up type expressions from `S` and `T` with equal outermost constructors, which can be done uniquely because of our conditions above. If there is no such pair, unification aborts with failure. Otherwise unify these pairs in some order (saving up the unifying substitutions) and create a new bounded variable γ that has the unified terms as possible type expressions. The most general unifying substitution is the sequential composition of the substitutions saved up and $\{\beta{:=}\gamma, \tau{:=}\gamma\}$. The unified term is γ. If τ is a type expression with outermost constructor `c`, check if there is an expression σ in S with the same outermost constructor. If so, substitute σ for β and unify σ with τ.

6 Dynamic Overloading and Oligotypes

The main reason for introducing oligotypes into our type model is the abundance of dynamically overloaded operators and constants in SETL. Due to the weak typing of current SETL, the notion of overloading in it is completely dynamic: Conflicts are only resolved at run time. In all the strongly typed languages we know of overloading is purely syntactic and is resolved at compile time. That dynamic overloading is more general than syntactic overloading can be seen in the following ML and SETL function declarations.

```
fun add (x,y) =
        x + y;
```

The ML operator `+` is overloaded with integer addition and real addition. Since this conflict cannot be resolved within the context of the ML function definition of `add`, the ML system signals a type error. Notice that the corresponding SETL procedure declaration

```
procedure add(x,y);
        return x + y;
end procedure;
```

can of course not generate a compile type type error since SETL is weakly typed, but, even if it were strongly typed, it should not result in a type error since the deliberate use of such overloading is an instrumental part of the flexibility of SETL. The `add` procedure exhibits the sort of polymorphism captured by our oligotypes, since it can be called with two integer values, two real values or even two sets or two sequences with the same element types.

Dynamically overloaded operators are quite common in SETL. We present a list of them with their oligotypes in a strongly typed system. Some obvious abbreviations in type expressions have been made; in particular we write ∀ for `forall`.

```
in:  ∀ α.∀ β in {set(α),sequ(α)}.proc(α,β)bool
+:  ∀ α.∀ β in {set(α),sequ(α),integer,real}.proc(β,β)β
−:  ∀ α.∀ β in {set(α),integer,real}.proc(β,β)β
<:  ∀ β in {integer,real,sequ(character)}.proc(β,β)bool
with:  ∀ α.∀ β in {set(α),sequ(α)}.proc(β,α)β
{}:  ∀ α.set(α)
arb:  ∀ α.proc(set(α)).α
```

Another more paradigmatic example of the usage of dynamic overloading is the following quicksort based sorting routine.

```
qsort = procedure (S)
        ( var x, L, R
          if S = {} then
             yield []
          else
             x := arb S
             L := { y: y | y in S and y < x }
             R := { y: y | y in S and x < y }
             yield qsort(L) + ([] with x) + qsort(R)
        )
```

This sorting routine has type `forall y in {integer, real, sequence (character)}. procedure (set(y)) sequence(y)`. Note that a type inference algorithm without oligotypes would treat the `<` as syntactically overloaded. Since it cannot resolve the overload conflict within the context given, it would fail and signal an error.

7 Concluding Remarks

We have presented a quite extensive type model for SETL-2, the strongly typed successor of SETL. We have argued that parametric polymorphism is best suited for the style of declaration-free programming usually found in SETL. Due to the abundance of dynamically overloaded operators in SETL we have extended the classical type model of parametric polymorphism to finitary polymorphic types, which we call oligotypes. These oligotypes capture in a strongly typed framework the difference between dynamic overloading found in SETL and syntactic overloading in a language such as ML. We believe that the standard type inference algorithm for parametric polymorphism can be extended to handle both recursive types and oligotypes. Unless the type system is restricted, we believe, though, that complete type inference is infeasible. Although oligotypes model dynamic overloading as it is found in SETL quite naturally, a comprehensive investigation into the theory of oligotypes is necessary to clarify the issues.

References

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* 1986.

[BDD80] H. Boehm, A. Demers, and J. Donahue. An informal description of russell. Technical Report TR-80-430, Dept. of Computer Science, Cornell U., 1980.

[BMS80] R. Burstall, D. MacQueen, and D. Sannella. Hope: An experimental applicative language. In *Stanford LISP Conference 1980*, pages 136–143, 1980.

[Car84] L. Cardelli. A semantics of multiple inheritance. In *Int. Symp. on Semantics of Data Types*, volume 173 of *LNCS*, pages 51–68. Springer-Verlag, 1984.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. Technical Report CS-85-14, Dept. of Computer Science, Brown University, August 1985.

[DaC84] R. DaCosta. The history of ada. *Defense Science and Electronics*, 1984.

[Don79] J. Donahue. On the semantics of 'data type'. *SIAM J. Computing*, 8(4):546–560, 1979.

[Fok81] M. Fokkinga. *On the Notion of Strong Typing*, pages 305–320. North-Holland, 1981.

[FSS75] S. Freudenberger, J. Schwartz, and M. Sharir. Experience with the setl optimizer. *ACM TOPLAS*, 5(1):26–45, Jan. 1975.

[Gir71] J. Girard. Une extension de l'interpretation de Godel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symp.*, pages 63–92, 1971.

[GMM$^+$78] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in LCF. In *Proc. 5th ACM POPL*, pages 119–130, 1978.

[Hen87] F. Henglein. Overloading in setl is not (syntactic) overloading after all. Technical Report (SETL Newsletter) 220, New York University, April 1987.

[Mat] D. Matthews. Introduction to poly. [see POLYMORPHISM].

[Mat83] D. Matthews. Poly report. *Polymorphism*, I(2), April 1983.

[Mee83] L. Meertens. Incremental polymorphic type checking in B. In *Proc. 10th ACM Symp. on Principles of Programming Languages (POPL)*, pages 265–275, 1983.

[Mil78] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[Rey74] J. Reynolds. Towards a theory of type structure. In *Proc. Programming Symposium*, volume 19 of *LNCS*, pages 408–425. Springer-Verlag, 1974.

[Rey81] J. Reynolds. The essence of algol. In *Algorithmic Languages*, pages 345–372. North-Holland, 1981.

[Rey85] J. Reynolds. Three approaches to type structure. In *Proc. TAPSOFT*, LNCS, pages 97–138. Springer-Verlag, 1985.

[SDDS86] J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, 1986.

[Ten74] A. Tenenbaum. Type determination for very high level languages. Technical Report NSO-3, Courant Institute of Mathematical Sciences, New York University, 1974.

[Tur85] D. Turner. Functional programs as executable specifications. In C. Hoare and J. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 29–50. Prentice/Hall, 1985.

[Ull82] J. Ullman. *Database Systems*. Computer Science Press, 1982.

[vWMPK69] A. van Wijngaarden, B. Mailloux, J. Peck, and C. Koster. Report on the algorithmic language algol 68. Technical report, Mathematisch Centrum, Amsterdam, 1969.

[Wei86] G. Weiss. Recursive data types in setl: Automatic determination, data language description, and efficient implementation. Technical Report 201, Courant Institute of Mathematical Sciences, New York University, March 1986.