Overloading in SETL is not (Syntactic) Overloading after all

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

April 8th, 1987

Abstract

Operator overloading is one of the characteristic features of SETL that contributes to SETL's intuitive character, ease of use, ease of modification, and its overall high level character.

However, as we point out, it is distinctly different from the usual, *static* kind of overloading. We present examples that elucidate these differences. The implications, mostly complications, for type analysis are introduced, and we point out where a type finding algorithm proposed by [DGDFJ87] is bound to fail since it derives its notion of overloading from classical syntactic overloading instead of the polymorphic overloading of SETL.

1 Introduction

Overloading in programming languages is an old feature. It was probably first incorporated in FORTRAN, where + denotes both integer addition and real addition, and it is found at least in some rudimentary form in all high-level programming languages. When properly used it emphasizes the similarity in which an overloaded operator works on different datatypes.

Parenthetical remark: This notion of similarity has a formal counterpart in abstract algebra, where a mapping $f : S \to T$ is said to be structure preserving (homomorphic) if it satisfies $f(x + y) = f(x) + f(y)$. The overloading

of + here is justified, since f correlates the structures of S and T and thus makes + in S and + in T very "similar". We think that this notion of similarity in algebra could also serve as the basis of a category theoretical or algebraic theory of overloading in programming languages.

2 Overloading in SETL, Ada, and ML

With the advent of very-high-level languages like SETL [SDDS86], polymorphic languages like ML [Har86], and freely overloadable languages like Ada [Uni83], overloading has received increased attention, but in each case for different reasons.

2.1 Overloading in SETL

In SETL, some operators can take arguments of different types and perform type dependent operations. The simple expression

$$a + b$$

will yield an integer, real, string, tuple, or set, depending on the *run-time* types of a and b. Since SETL is weakly typed these run-time types may well change over the course of a program execution or over different executions. This kind of overloading, in which "different" code, dependent on the run-time type of the arguments, is executed, has usually not been considered a form of polymorphism We will argue that, instead, it should be considered a form of polymorphism. Consequently, we will refer to SETL's overloading as *polymorphic* (or *dynamic*) overloading.

2.2 Overloading in Ada

Ada does not only have a basic set of overloaded operators, like `+`, but its visibility rules also provide a mechanism for the user to overload operator symbols and subprogram names even further. Every occurrence of an operator symbol or function name is considered to designate exactly one out of several possible functions. The syntactic context, precisely defined in [Uni83], is used to disambiguate which one of the candidate functions an operator symbol or function name actually stands for. This process is called overload resolution. We will call Ada's kind of overloading *syntactic* (or *classical* or *static*) overloading since the true identity of an operator, respectively subprogram, can be determined from its syntactic context alone; specifically, no dynamic context — as in the case of SETL — is necessary.

This difference is not merely cosmetic, but leads to profound differences, as exemplified a little bit later.

2.3 Overloading and ML

The type information of the arguments (*signature*) is used in Ada to resolve operator/subprogram overloading. This is possible since all arguments have to be declared with a type. In contrast, in a largely declaration free language like ML the signature of an operator is used to infer the types of the arguments. It is clear that overloading makes type inference harder, and the need for type inference interferes with overload resolution. This is probably the reason why ML stays away from overloading as far as it can. It also adopts the syntactic (classical) overloading model in order not to have to integrate polymorphic overloading into its type model and type inference algorithm.

For example, the function declaration

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

is not accepted in ML, since the `+` cannot be disambiguated within the given context (it stands for both integer addition and real addition). Note that this problem cannot occur in Ada, because the (monomorphic) types of `x` and `y` would have to be available for the function to be a legal program unit. Note also that the equivalent declaration in SETL

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

is perfectly legal.

3 Enlarging the context does not resolve polymorphic overloading

It might seem like the main problem for

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

to be illegal in ML, but for

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

to be legal in SETL, is the limited syntactic context in ML that is taken into account for resolving overloading. In particular, applications of a function `f` are not taken into account to resolve overload ambiguities *inside the body* of `f` (of course they are used to resolve which of possibly many `f`'s is actually denoted at a particular call site).

For example, if we had

```
fun add (x,y) = x + y;
val sum = add(5,8);
```

we could conclude that `add` must have type **integer** $\times$ **integer** $\rightarrow$ **integer**. Since ML is an interactive language it cannot take the right context ('right' as opposed to 'left' and not 'wrong') into account, and so this approach is not feasible for ML (function applications are in the right context of the corresponding function declarations). But is this a possible approach to type inference and overload resolution for SETL, which is noninteractive? The answer is no, and as we think the reason for this provides some justification for calling SETL's overloading polymorphic.

Consider the following legal SETL program.

```
program addtest;

print(add('hel', 'lo'));
print(add(5, 8));

proc add(x,y);
     return x + y;
end proc;

end program;
```

If we treated overloading classically as suggested above, that is, if we had to find out what the "single, true meaning" of the occurrence of `+` above is (exactly one of integer or real addition, string or tuple concatenation, or set union), we would end up with a type error. After the first two lines `+` would already unambiguously stand for string concatenation, but the third line

contradicts this interpretation. Note however, that `addtest` is a perfectly legal SETL program that prints 'hello' and 13.

The crucial difference between syntactic overloading and SETL's overloading is that in syntactic overloading there is one single type for every operator/function occurrence while in SETL every such occurrence can have several such types *simultaneously*. The types of the actual arguments are checked to see if they are legal instances of the general simultaneous types (that is, one of possibly many simultaneous types), and the result type is determined by the instances.

4 SETL's overloading doesn't fit the polymorphic britches

SETL's overloading is obviously a form of polymorphism, but it doesn't fit any of the "established" categories of polymorphism as they are presented in [CW85]. Cardelli and Wegner describe two forms of *ad-hoc* polymorphism, overloading (of the syntactic sort) and coercion, and two forms of universal polymorphism, parametric polymorphism and inclusion polymorphism. But, as we pointed out in the previous section, SETL overloading is distinct from syntactic overloading, it is not as general and regular in its legal type instances as parametric polymorphism, and certainly it is not a form of coercion or inclusion polymorphism.

Ad-hoc polymorphism and universal polymorphism are distinguished, in one of two ways, in terms of their implementation. A "universally polymorphic function will execute the same code for arguments of any admissible type, while an *ad-hoc* polymorphic function may execute different code for each type of argument" (see [CW85, top of page 6]).

> *Parenthetical remark*:
> Surely this is not a very robust criterion. Even the prototypical polymorphic function `length` in ML
>
> ```
> fun length nil = 0 |
>           x::y = 1 + length(y);
> ```
>
> would have to be provided type information *if* lists were implemented by linear array structures (it would need to know the size of the element type). We think that certain implementation techniques (mostly linked structures with pointers) should not serve as main guidelines for the taxonomy of abstract concepts (universal vs. *ad-hoc* polymorphism).

The same code is executed for an overloaded SETL operator, but the code makes critical usage of the types of the arguments, mostly in the form of a case statement, where the separate branches correspond to the *different* pieces of code characteristic for *ad-hoc* polymorphism. In this curious way, SETL's overloading combines features from universal and *ad-hoc* polymorphism. Since, furthermore, overloading in SETL cannot be handled separately (in a preprocessing step) from type inference, but instead is part of type inference and the type model, we feel justified in calling SETL's overloading "truely" polymorphic, not just *ad-hoc* polymorphic. This is not just a matter of nomenclature, but also of technical importance (see forthcoming newsletter on the type model of SETL that incorporates polymorphic overloading).

In summary, SETL overloading can best be described as combining features from syntactic overloading and parametric polymorphism; hence the name polymorphic overloading.

5 Typefinding and overloading in SETL

It is interesting to see how overloading in SETL has been dealt with in previous work on type finding[1]. There are essentially two approaches in this area: Tenenbaum's and Weiss's theses on typefinding [Ten74, Wei85], and the initial work done by the SED group in Europe [DGDFJ87].

5.1 Overloading in Tenenbaum's and Weiss's work

In the work by Tenenbaum and Weiss typefinding is performed globally for a complete program. Procedure calls are modelled by simple assignments of the actual parameters at a particular call site to the formal parameters of the procedure called. This approach works well for type finding for all practical purposes (see, e.g., [Wei85, section 1.3]), but not for type checking, since it does not take the polymorphic character of procedures into account. We will demonstrate this deficiency only for polymorphic overloading here and not for universal polymorphism, although it applies there just as well.

Consider the previous example again.

```
program addtest;
```

[1] Type finding is the *descriptive* counterpart to the *prescreptive* discipline of type inference and type checking.

```
print(add('hel', 'lo'));
print(add(5, 8));

proc add(x,y);
     return x + y;
end proc;

end program;
```

In the Tenenbaum/Weiss approach the procedure calls to `add` are modelled by two assignments to `x` and two assignments to `y`:

```
x := 'hel';
y := 'lo';
```

and

```
x := 5;
y := 8;
```

The type of both `x` and `y` would be **integer|string**, i.e. a union type. The presence of a union type would normally indicate that `x` can take on integer and string values possibly flipping back and forth between the two types. Note, however, that both `x` and `y` in any single incarnation are either integer valued or string valued, but never change their (monomorphic) type. Modelling this classically polymorphic phenomenon by union types, however, leads to loss of information and failure to detect possible type errors. For example, in the above program the result returned by the procedure `add`, and thus the arguments of both print statements, have type **integer|string**, although the type of `add('hel','lo')` should be bf string and the type of `add(5,8)` should be **integer**. Furthermore, if we had an additional print statement

```
print(add('hel', 8));
```

the type error in it would go undetected at compile time.

Note also that the type information obtained in the Tenenbaum/Weiss approach is not invariant with respect to a simple program transformation like inline expansion or partial inline expansion of procedures: Expansion always results in more specific type information. If we expand the definition of `add` in the calls of the example program above we obtain the following program.

```
program addtest;

print('hel' + 'lo');
print(5 + 8);

end program;
```

Here the typefinder correctly determines the arguments of the print statements to be of type **string** and **integer**, respectively. It also catches the type error in the expanded additional print statement

```
print('hel' + 8);
```

The reason for this is that the single occurrence of + in the original programs is replaced by two, respectively three, occurrences of + thus permitting the typefinder to find different types for each one of them, whereas it has to determine one single type for + in the original program. This insight motivated Hyun and Doberkat [HD85] to advocate macro expansion (i.e. inline expansion of what one might otherwise write as a procedure) as a remedy for limited typefinding on a procedural basis.

We can conclude that the nonpolymorphic, simple assignment approach is inappropriate for a strong typing discipline.

5.2 Overloading in the work of the SED group

Donzeau-Gouge *et al.* present a proposal for a programming environment for SETL, in which they outline their approach to typefinding based on the logic programming language Typol [Des84, CDD+85]. Although it is not completely clear from their report [DGDFJ87], it can be expected that due to the extensive use of unification in Typol the SED approach to type finding will correctly deal with universally polymorphic procedures such as

```
proc cons(x,l);
     return [x,l];
end proc;
```

Specifically, it can be expected that `cons` will be determined to have type $\alpha \times \beta \rightarrow [\alpha, \beta]$ (although $[\ldots]$, called *sequence* in [Wei85], is not amongst the enumerated type constructors on page 8 of [DGDFJ87]), where α and β are uninstantiated logical (type) variables. Different applications of `cons` will result in different instantiations of `cons`. The type of `a` and `b` in

```
program nonsense;

a := cons(5,[8]);
b := cons([5.0],[[8.0]]);

proc cons(x,l);
     return [x,l];
end proc;

end program;
```

will correctly be determined as [**integer**, [**integer**]] and [[**real**], [[**real**]]], respectively. Note that, in contrast, the type of both `a` and `b` would be [**integer**, [**integer**]]|[[**real**], [[**real**]]] in the Tenenbaum/Weiss approach.

The SED approach is still bound to fall short of dealing correctly with polymorphic overloading if the method exemplified on pp. 9–11 in [DGDFJ87] is employed without special attention to SETL's dynamic overloading, since it implicitly adopts static overloading as its overload resolution model.

The SED approach can be roughly described in operational terms as follows. In step one every leaf in the abstract syntax tree is assigned a unique logical (type) variable. There is a Typol rule that indicates how the type information of subtrees can be combined to determine the type at a (sub)tree root. This might necessitate unification. The environment keeps track of any variable substitutions that occur in such a unification step. If the operator at the root is overloaded (such as `+`), the root is assigned a new type variable and a type constraint (e.g., `plus(T1,T2,T)`) is added to the type environment. In step two all the type constraints accumulated in step one are resolved by checking them against the possible "ground" constraints for the overloaded operators.

> *Parenthetical remark*:
> To really understand what sounds so confusing in the paragraph above the reader is kindly advised to consult [DGDFJ87]. Since the distribution of that report is very limited, a copy of it can be requested from the author of this paper.

Since there is only one type constraint generated for every occurrence of an overloaded operator and since these constraints are resolved globally for a whole program, this method — implicitly — adopts overload resolution for syntactic overloading, which is imprecise or even incorrect in view of

SETL's polymorphic overloading. This can be exemplified in our running example

```
program addtest;

print(add('hel', 'lo'));
print(add(5, 8));

proc add(x,y);
     return x + y;
end proc;

end program;
```

The first step would proceed as follows. Processing `add` first, which is necessary to deal with universal polymorphism conveniently, the types of `x` and `y` would be the variables `T1` and `T2`, respectively, and `x + y` would have as its type the new type variable `T`. Since `+` is overloaded, the type constraint `plus(T1,T2,T)` would be generated. The procedure `add` would be assigned type `T1` $\times$ `T2` $\rightarrow$ `T`. Finally, both `add('hel','lo')` and `add(5,8)` would have type `T`.

In the second step the generated type constraint `plus(T1,T2,T)` would be resolved against the "operator overloading database". Since `plus(T1,T2,T)` matches all of

```
plus(integer,integer,integer).
plus(real,real,real).
plus(string,string,string).
plus(set(X),set(Y),set(Z)) :=
  union(X,Y,Z).
plus(tuple(X),tuple(Y),tuple(Z)) :=
  union(X,Y,Z).
```

`T` would have a set of possible substitutions (with additional constraints!). Even if we could form a single substitution from this, such as **integer**|-**real**|**string**|**set**(α)|**tuple**(α) (with additional union constraints generated), we could never find the correct types of the arguments in the two print statements, since step one already mandates that they have the same type `T` no matter what `T` may eventually be substituted with in step two. Note

however that, if SETL had syntactic instead of polymorphic overloading, the SED approach would work well: The fact that T has no unique substitution in the overload resolution step (step two) would indicate that the overloaded occurrence of + cannot be resolved, and an appropriate error message could be generated.

6 Conclusion and outlook

We have demonstrated that the kind of operator overloading employed in SETL is distinctly different from overloading in Ada or ML; in fact, we consider it a form of "true" polymorphism (as opposed to *ad-hoc* polymorphism), which is reflected in the term we propose for it: polymorphic overloading.

We show that the approaches to typefinding in SETL have ignored this aspect of overloading resulting in loss of precision in type determination and type errors going undetected at compile time.

A type model we have developed that incorporates SETL's polymorphic overloading will be described in a forthcoming newsletter.

References

[CDD+85] D. Clement, J. Despeyroux, T. Despeyroux, L. Hascoet, and G. Kahn. Natural semantics on the computer. Technical Report RR 416, INRIA, June 1985.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Des84] T. Despeyroux. Executable specification of static semantics. Technical Report RR 295, INRIA, June 1984.

[DGDFJ87] V. Donzeau-Gouge, C. Dubois, P. Facon, and F. Jean. Development of a programming environment for SETL. SED Project Report, 1987.

[Har86] R. Harper. Introduction to Standard ML (preliminary draft). Technical report, University of Edinburgh, February 1986.

[HD85] K. Hyun and E. Doberkat. Inline expansion of Setl procedures. *SIGPLAN Notices*, 20(12):33–38, December 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.

[Uni83] United States Department of Defense. *Reference Manual for the ADA Programming Language*. Springer-Verlag, 1983.

[Wei85] Gerald Weiss. Recursive data types in SETL: Automatic determination, data language description, and effiicient implementation. Ph.D. Thesis 201, Courant Institute of Mathematical Sciences, New York University, October 1985.