Deriving declarations from programs

Henning Christiansen
Department of Computer Science, Roskilde University,
P.O.Box 260, DK-4000 Roskilde, Denmark
E-mail: henning@dat.ruc.dk

Extended abstract

Abstract. A constraint-based method for deriving type declarations from program statements is described. The relation between declarations and a correctly typed program can be characterized by means of instance constraints and the present paper describes experiences using constraint solving techniques to synthesize the type declarations inherent in a given program.

Such a facility may turn out to be useful from the viewpoint of programming methodology. It may suggests an integration of an experimental and "untyped" style with the higher degree of robustness gained with typed languages.

1 Introduction

Type declarations in programming languages are praised because it leads to more robust programs and it is a common observation that many programming errors are detected by the type checker or even by the programmer himself, when he is designing the declarations. On the other hand, it can be argued that the additional burden of writing type declarations does not fit with the fast and experimental programming style of an elegant programming language such as Prolog, slowing down the creativity of the programmer.

There is an ongoing discussion whether logic programming languages should have type declarations. To us, the arguments from both camps appear to be equally sound. For some applications, it can be more relevant to start out experimenting writing and testing clauses and after a number of iterations, spending the effort of adding type declarations in order to produce a more robust version — which could be enforced as a requirement for adding a module to a larger program.

Most often, the programmer has applied the functions and predicates in some systematic way implying an implicit typing anyway. This can be based on an initial intuition about the problem domain or being part of the understanding gained during an experimental development process.

We suggest that the programming environment assists the programmer by automatically deriving type declarations from program statements, suggesting a "reasonable" set of declarations that matches the program, perhaps guided by partial declarations provided by the programmer. This may provide an incremental style of working where the programmer can switch back and forth between editing the program and the declarations, the system continually reflecting the implied consequences for the declarations, and the other way round, indicating type conflicts in the program.

In this paper, we consider the abductive problem of automatically providing a qualified suggestion for a set of type declarations which makes the type checker succeed on the given program. We consider polymorphic types as applied in the Gödel language and we refer to (Hill, Lloyd,

1994) for definitions and an introduction.

A program being correctly typed means that the actual types assigned to each subphrase must be an instance of the type pattern assigned for its primary symbol. In other words, this about being an instance of something is a fundamental, defining property for the declarations sought and we suggest constraint solving as a way to resolve the set of instance conditions that arise in the particular case.

A constraint solver for instance constraints is described in (Christiansen, 1996a, 1996b), where it is used in order to have a general proof predicate to "reason backwards" as a way to create programs. This approach can be used for a variety of abductive and inductive problems.

In section 2 we show how given type declarations can be written as a Prolog program which will do the type checking and in section 3, we generalize to a constraint-based type checker parameterized by the declarations. The problem of choosing the intuitively correct or best solutions contained in a set of final constraints is discussed in section 4. At this point, however, our work is preliminary and we still need to work out the theory in order to characterize a "best" set of declarations or "a most general prototype"; we hope to include this in the final version of the paper. In section 5 we discuss the possible integration of an automatic type-declaration-generator in a programming environment and possible other applications.

2 Type checking as a logic program

We consider object programs in a Prolog-like language together with polymorphic type declarations as described by (Hill,Lloyd, 1994). We will use the following sample program[1] throughout the paper; we ignore the need for at least one base type and a constant at the moment.

```
append(nil,X,X).
append(cons(U,X),Y,cons(U,Z)):- append(X,Y,Z).
```

One possible set of declarations associated with this program is the following.[2] The capital letter `A` is a parameter, also called a type variable.

```
constant nil : list(A).
function cons : A * list(A) -> list(A)
predicate append : list(A) * list(A) * list(A)
```

We do not allow overloading of symbols in the sense that two different functions (predicates, constants) can have the the same name, but the same symbol can be used simultaneously in different categories.

We are not interested in executing object programs, but only analyzing them for type information, and we can choose whatever representation is practical for our purpose. The program above will be represented by the following data structure when given to our type checker.

```
[(append(nil,var(X1), var(X1)):- true),
 (append(cons(var(U2),var(X2)),var(Y2),cons(var(U2),var(Z2))):-
       append(var(X2),var(Y2),var(Z2)))]
```

Each object variable is represented by a Prolog variable enclosed by the marker `var(_)`; for simplicity we assume the variables for each clause have been standardized apart. The variables will be used for representing whatever-type-the-object-variable-is-assigned-during-type-checking.

[1] When referring to a "program" we deliberately refer to a set of program statements and consider the declarations as something external wrt. the program.

[2] The meaning is as in Gödel, however, we have switched the use of upper- and lower-case letters. For simplicity, we have left out explicit declarations of base types and type constructors.

The following Prolog program accepts exactly all programs represented in this way that are correctly typed wrt. to the type declarations above.[3]

```
%% tc(P) - P is correctly typed program/clause/body/atom

tc([]).

tc([C|Cs]):- tc(C),tc(Cs).

tc((H:-B)):- tc(H), tc(B).

tc((A,B)):- tc(A), tc(B).

tc(true).

%% tc(P,T) - P is a term with type T

tc(nil, list(_)).

tc(cons(Z1,Z2), list(A)):-
    tc(Z1,A),tc(Z2,list(A)).

tc(append(Z1,Z2,Z3)):-
   tc(Z1,list(A)),  tc(Z2,list(A)),  tc(Z3,list(A)).

tc(var(X), X).
```

The last clause implements the fact that a given object variable should be assigned the same type throughout a given clause.

The rule for `cons` expresses that types for the arguments of a `cons` term must be an *instance of* the pattern `[A,list(A)]`. The type of the whole term is given as the actual instance of `list(A)`.

Furthermore, (Hill, Lloyd, 1984) requires a *head condition* to be satisfied, saying that the tuple of types for the head of a clause must be a variant of the type declared for the predicate in the head. Declarations of functions must satisfy a condition of *transparency*, which means that every parameter in a declaration also must appear in the range type (to the right of "`->`").

3 The abductive problem, type declarations from program statements

In order to have a program to work with declarations as arguments, we must devise a suitable data structure. We need a ground representation, i.e., parameters must be recognized as ground structures, in order to have a sound treatment in a constraint solver. We have chosen the following easy-to-read and *ad hoc* representation in Prolog using a few operator declarations, here shown for the type declarations of the previous section.

```
[constant   nil     : list('A'),
 function   cons    : ['A', list('A')] ->> list('A'),
 predicate  append  : [list('A'), list('A'), list('A')] ].
```

[3] We must assume occur-checking unification, otherwise it is easy to produce examples that fool this type checker.

Instance constraints have the following form,

`instance(` T_1`,` T_2`,` S`)`

satisfied whenever T_1 and T_2 are representations of types τ_1 and τ_2, S a representation of a substitution σ with $\tau_1\sigma = \tau_2$. A similar condition on representations of terms has been used for implementing the binary proof predicate `demo`. (Hill, Gallagher, 1994) describes a direct implementation in Prolog, which is perfect when the first argument is completely specified (i.e., ground at the metalevel), but as argued by (Christiansen, 1996a) such an implementation is condemned to flounder in case of uninstantiated metavariables in the first argument — which could stand for an unknown fragment of an object program or, as in the present context, for an unknown type declaration. (Christiansen, 1996a) describes a constraint solver for instance constraints, which is sound and complete whenever it can be shown that no metavariable can occur in a first as well as a second argument of an instance constraint; this condition obviously holds for the use of instance constraints we make in the present paper. Without this requirement, satisfiability of instance constraints is equivalent with the undecidable semiunification problem. The constraint solver is implemented in Sicstus Prolog (SICS, 1995) using its notion of attributed variables, see (Christiansen, 1996a, 1996b).

We generalize the type checker program in section 2 with an additional argument to hold a representation of the type declarations used. The clause for correct typing of a structural term is as follows; for simplicity, we show a clause for structures with exactly two arguments.[4]

```
tc(Term, Type, Decls):-
    Term =.. [Fun, Term1, Term2],
    tc(Term1, Type1, Decls), tc(Term2, Type2, Decls),
    member( function Fun : DefType, Decls),
    instance( DefType, [Type1, Type2] ->> Type, _).
```

Programs and clauses are traversed recursively as in the previous section, the clause for atoms is as follows (again, a binary predicate for simplicity).

```
tc(Atom, Decls):-
    Atom =.. [Pred, Term1, Term2],
    tc(Term1, Type1, Decls), tc(Term2, Type2, Decls),
    member( predicate Pred : DefType, Decls),
    instance( DefType, [Type1, Type2], _).
```

However, for the head of a clause, we use at alternative predicate called `tc_head` defined by a clause identical to the one above except that the `instance` is replaced by `variant`, which is a new constraint

`variant(` T_1`,` T_2`,` S`,)`

whose meaning is as for `instance` with the additional requirement that S must represent a renaming substitution; this is easily added to the constraint solver. This implements the head condition described in the previous section.

Transparency of function declarations can easily be implemented using the delay mechanisms of Sicstus Prolog (SICS, 1996) as a "lazy" test that wakes up from time to time as a function declaration is gradually being instantiated. We assume such a predicate `transparent(_)` and consider the following query.

`transparent(D), tc(D,` *program* `)`

[4] The general clause for arbitrary terms iterates the `tc` predicate over the argument list in a straightforward way.

The answer returned consists of a partially instantiated pattern for the type declarations together with a large bunch of accumulated constraints C.

It is practical to add a pre-analysis of the program text in order to build a template for type declarations which imposes some fixed order for the declarations of the symbols used. This will make the program execute in a deterministic way, a straightforward recursive traversal of the object program. This means that no alternative answer is possible by backtracking and, thus, soundness and completeness of the constraint solver implies that C together with the possible instantiations of `D` contain *all* possible type declarations under which the program is correctly typed.

On other words, we can put in a Prolog cut (`!`) before going to the next section where we consider the question of how to present concrete and instantiated answers.

4 Which solution to choose?

A bunch of constraints is a very indirect way to display the possible type declarations and it is definitely not very interesting to the user. Ideally, the system should display one, in some sense, best prototypical representative from which all possible solutions could be derived by some intuitively simple transformations.

For example, a metavariables standing for some type or constructor can be displayed as a dummy name, distinguishable from the names supplied by the user. The kind of transformations we have in mind could be renaming of symbols, specialization by instantiation of parameters and identification of types (e.g., if the system has separated a collection of symbols in two different base types, the user should have the right to state that they *are* of the same type).

To describe this in a proper way, we need to develop a suitable ordering on sets of type declarations and devise a collection of such transformations that can be shown sound and complete. We have no suggestion for this at present. Instead we will discuss some examples.

Clearly there are many uninteresting type declarations that satisfy a set of constraints generated for a given source program. One obvious, which always works, is the one which degenerates everything to one base type, namely `term`. In the other end of the spectrum, we have versions with intuitively too many variables and too few constructors such as the following found for the `append` program.

```
[constant nil : 'Par1',
 function cons : ['Par1','Par1'] ->> 'Par1',
 predicate append : ['Par1','Par1','Par1']]
```

This answer has been generated by means of a heuristic device for instantiating the remaining metavariables. It enumerates all possible instantiations (qua renaming of symbols) by trying out combinations of already used and new symbols.[5] The parameter `'Par1'` is created by the system.

There are other and more weird answers; `c1` is a constructor invented by the system:

```
[constant nil : 'Par1',
 function cons : ['Par1','Par1'] ->> 'Par1',
 predicate append : [c1(c1('Par1')),c1(c1('Par1')),c1(c1('Par1'))]]
```

It may be a bit surprising that it is correct, but a little work with the formal definition may convince. The "correct" solution appears as the ninth one from our enumerator.

[5] This is not as inefficient as may sound. Whenever a commitment violates the constraints, the constraint solver will report this immediately as a failure and the enumerator tries another choice. However, a more intelligent strategy for instantiation should of course also take into account the information embedded in the constraints.

```
[constant nil : 'Par1',
 function cons : ['Par1',c1('Par1')] ->>c1 ('Par1'),
 predicate append : [c1('Par1'),c1('Par1'),c1('Par1')]]
```

These examples do not seem very flattering for our approach, but on the other hand, the flaw may rather be in the choice of the sample program, consisting of two small clauses including almost no ground data.

If we provide the following partial declarations in advance,

```
[constant nil : list('A'),
 constant monday : weekday,
 ...
 constant sunday : weekday ].
```

and let the typing predicate also analyze the following sample query,

```
append(cons(monday,cons(tuesday,nil)),
       cons(wednesday,cons(thursday,cons(friday,
                    cons(saturday,cons(sunday,nil))))),
       cons(monday,cons(tuesday,cons(wednesday,cons(thursday,
          cons(friday,cons(saturday,cons(sunday,nil))))))) )
```

we get the desired answer immediately.

5 Discussion

From a practical point of view, it is important to notice that the editing made by the user in the generated declarations easily can be reused. This follows from the fact that the typing predicate (as well as the constraint solver) works equally well starting from a partially instantiated representation of the declaration argument.

This allows an incremental style of working where the user switches back and forth between editing the program and the declarations, with the system continually reflecting the implied consequences for the declarations, and the other way round, indicating type conflicts in the program.[6] It could be interesting to develop an interface where what the user has written (or confirmed) appears with a normal font and the stuff filled in by the system in a shaded or italic font.

Our approach does not seem to be restricted to logic programming languages as the relation at the syntactic level between declarations and program statements appears to be quite independent of the underlying semantics. Especially for object-oriented programming, it would be very useful with some automatic way of creating declarations.

It will also be interesting to consider the use of our approach in an optimizing compiler for an untyped language, as an alternative method to abstract interpretation and dataflow analysis.

References

Christiansen, H., Automated reasoning with a constraint-based meta-interpreter. *Submitted* 1996a.
Paper and implemented system available at `http://www.dat.ruc.dk/software/demo.html`.

[6] This needs a little error recovery, which can be added to our little typing predicate as a default rule at the bottom saying, if anything else fail, the offending subterm can have can have any type.

Christiansen, H., On solving instance constraints. *Proc. META96*, ed. Barklund, J, 1996b.

Hill, P.M., Gallagher, J.P., Meta-programming in Logic Programming. To be published in Volume V of *Handbook of Logic in Artificial Intelligence and Logic Programming*, Oxford University Press. Currently available as *Research Report Series* 94.22, University of Leeds, School of Computer Studies, 1994.

Hill, P.M. and Lloyd, J.W., *The Gödel programming language*, MIT press, 1994.

SICStus Prolog user's manual. Version 3.5, SICS, Swedish Institute of Computer Science, 1996.