Type Checking in the Large

Michael R. Levy
Department of Computer Science
University of Victoria
P.O. Box 1700
Victoria, B.C.
Canada V8W 2Y2

1 Introduction

The larger a software system, the more important it is to maintain consistency between module interfaces and the use of modules. Type checking is an important technique for ensuring this consistency. Static type checking (that is, type checking prior to run-time) is especially attractive because it allows errors to be detected with no run-time cost. For this reason, most program-language designers pay very careful attention to their type system. The ideal is to design a type system which is flexible enough to support algorithm development but which is secure. Secure means that it is possible to guarantee that a procedure will not be given an invalid type of data.

Type checking is straight-forward for most reasonable constructs if the entire program source is available at type-checking time. However, in the presence of separate compilation, this requirement is not reasonable. Some languages therefore abandon any attempt to type-check across module boundaries.

An alternative to supplying the entire source is to provide at least the interface (and perhaps some type definitions) to the type-checker of the user of a module. One of the disadvantages to this approach is the addition to the system of an extra object to be maintained (the interface). Care has to be taken to ensure that the user and the module itself agree on the interface. Polymorphism introduces an extra complications which will be described later in the paper.

We present here a new strategy for static type checking. This strategy ensures type security but removes from the programmer the need to include interface source in user programs.

2 A brief description of the strategy

A large piece of software can be represented as a directed, acyclic graph where nodes represent modules and edges represent a relationship between modules. The relationship is called 'uses' and the the edge is from the user to the used module. For example, if a module A contains calls to procedures in a module B, then this can be represented by the simple graph

$$A \rightarrow B$$

We say "A uses B". Objects such as variables within a module can be classified as being *free* or *bound*. Objects are bound in a module if there is a declaration of the type of the object within the module. All other objects are free. Given a module A it is simple to determine the set of free objects in the module. The types of the bound objects and the use made of the free objects impose constraints on the possible types of the free objects. For many programming languages there exists an algorithm that can infer the set of free objects in a module together with constraints on the types of each free object in the set. The output of this algorithm is called the **imports list** of the module. The type information about all bound objects within a module is called an **exports list**. Type checking is performed by attempting to unify the imports list of a module with the union of the exports list of all the modules it uses. Failure to unify is a type error.

This strategy is useful because it does not require the maintenance of common include files. It is also makes it possible to use type information for the purpose of creating software versions. It diminishes compile time because it can reduce the need to recompile modules that use modules that have been changed.

An algorithm for creating imports lists for Pascal has been previously reported[1]. Interesting questions are raised, however, by programming languages that support generic modules or polymorphic data types and modules.

It is not possible to compile generic modules efficiently because the size of the parameters is unspecified. However, it is reasonable to compile instances of a module, since any particular project is likely to use a manageable number of instances. The type checking algorithm therefore assumes that all 'imported' generic types are known before unification is attempted. In practice this is done by proceeding top-down from the top-level program.

3 A type description language

A **type description** is defined as follows:
There is a set of objects B called *base types* , a set of objects V *called Variables* and a set of objects U called *unknowns*. Each $b \in B$ and each $u \in U$ is a type description. If T_1 and T_2 are type descriptions and $t \in$ V then

$$T_1 \times T_2 ,\ T_1 \rightarrow T_2 ,\ \lambda t . T_1 ,\ bt$$

are type descriptions.

λ-notation is used to denote the types of objects if they are generic (or polymorphic). For example,

$$f : \lambda t.\ list\ t \rightarrow t$$

is a proposition saying that f is a polymorphic function. *list* in this example is a type constructor.

4 Separate type checking without polymorphism

Type checking is performed at link time. Associated with each module is an imports list and an exports list. Type checking succeeds if each module's imports list can be unified with the exports list of all the modules it uses.

Example

Suppose that

$$A \to B$$

and that A has imports list

$$f: \alpha \times \beta \to char$$
$$g: \text{int} \to \alpha$$
$$g: \text{int} \to \beta$$

Also, suppose that the exports list of B is

$$\text{f: int} \times \text{int} \to \text{char}$$
$$\text{g: int} \to \text{real}$$
$$\text{g: int} \to \text{char}$$
$$\text{g: int} \to \text{int}$$

Clearly the unifier is

$$\alpha \Leftarrow \text{int}$$
$$\beta \Leftarrow \text{int}$$

This algorithm is a simplified version of Milner's type-inference algorithm[2]. The differences are that

1. No inference is taking place - we assume that the types of routines are fully declared; and
2. The variables in Milner's algorithm are type variables. Here they are more like 'unknowns'. This distinction becomes more obvious when polymorphic procedures are allowed.

A key question is this: Can modules be compiled without access to complete type information of the imported procedures? If the answer to the question is no, then the language cannot properly support separate type-checking. For the programming language C, the answer to the question is yes. One construct in the programming language Pascal does create a problem, namely *var* parameters. The difficulty is that in expressions like *f(x)* it is not possible to determine whether *x* denotes an expression or a variable without access to the formal heading of *f*. One possible solution to this problem is to modify the syntax of Pascal so that call-by-variable is noted at the actual call, by using the keyword **var**, for example. This would in any case increase program readability.

The algorithm for determining the requirement is dependent on the grammar of expressions. For our purposes, assume

$$e ::= c \mid v \mid f(\,) \mid f(e) \mid f(e,e)$$

where c is a literal constant, v an identifier and f a procedure name. Then we can define a procedure Req(e) as follows:

1. If e is c or v, then
   $Re\,q(e) = \{e : \alpha\}$

   where a is a variable ranging over type descriptions. The **result type** of e is a.

2. If e is $f()$, then
   $Re\,q(e) = \{f : void \to \alpha\}$

   The result type of e is a.

3. If e_1 is $f(e_2)$ then

   $Re\,q(e_1) = \{f : \alpha_0 \to \alpha_1\}$

   where a_0 is the result type of e_2, and a_1 is a new unknown. The result type of e_1 is a_1.

4.

$$Re\,q(f(e_1,e_2)) = \{f : \alpha_0 \times \alpha_1 \to \alpha_2\} \cup Re\,q(e_1) \cup Re\,q(e_2)$$

where a_0 is the result type of e_1, a_1 is the result type of e_2 and a_2 is new. The result
type is a_2.

The 'accuracy' of the type information determined by Req can be improved by using symbol table information.

5 Separate type checking with polymorphism

Our approach to polymorphic procedures is to regard them as templates for a family of possible instantiations. If $f{:}\lambda t.T$, and u is a type, them we will denote by f_u the instantiation of f of type [u/t]T. Only instances of polymorphic procedures are compiled. When a module is compiled, it is possible to determine which instances of its imported polymorphic procedures are required provided that the module being compiled has been fully instantiated. This suggests that initial compilation must proceed top-down, with instance requests being generated after each module compilation. Whenever a module is recompiled, new instance requests must be generated. Further instances may then need to be compiled. In practise, type parameter changes are not very common during software development, so there is not a frequent need for instance compilation.

If the requirement that only instantiated modules be compiled is met, then imports lists can be built using the algorithm given in the previous section. The unification step is, however, more complicated, because exported types may be polymorphic. For example, suppose that a module A has this imports list:

$$f : list\ \alpha \rightarrow int$$
$$g : char \rightarrow \alpha$$

and that a module B, used by A, has this exports list:

$$f : \lambda t\,.list\ t \rightarrow t$$
$$g : char \rightarrow char$$
$$g : char \rightarrow int$$

In this case, f must be instantiated to

$$f_{int} : list\ \ int \rightarrow int$$

before unification can be applied (getting $\alpha \Leftarrow int$).

The type-checking process can be described in the following way:

The **order** of a procedure p of type T is the number of outer-most λ-bindings of T. A non-polymorphic procedure has order 0. If p:T and p has order n, then it is possible to find an instance p' of order 0 of type $T(a_1,a_2,...,a_k)$ where each a_i is a type description.

If $E = (p_1 : T_1 ,...,p_n : T_n)$ is an exports list, then there is an obvious way to extend the notion of instantiation to the entire list. We will denote the instantiation of E that uses $(a_1,a_2,...,a_n)$ by $E(a_1,a_2,...,a_n)$.

A module is **type correct** with respect to an export list E iff there is an instance $I=E(a_1,a_2,...,a_n)$ of E such that the imports list of A unifies with I.

6 Abstractions

If a programming language supports an abstraction mechanism, such as **type** in Pascal, then each variable in a program has an abstract type and a concrete type. The above algorithms can be modified to deal with concrete types in a fairly simple way. The details are somewhat tedious, so the algorithm will only be outlined here. Firstly, the definition of type expressions is extended so that base types have two parts: an abstract type and a concrete type. For example

A: Matrix{Sparse}

Then we define a 'concretize' function on type expressions as follows: If $\rho : R \rightarrow T$ is an abstraction and t is a type description, then C_ρ is the function that extends

$$C_\rho (T \{ R \}) = R$$

and

$$C_\rho (U \{ S \}) = U \{ S \}$$

to arbitrary type descriptions.
Before seeking the match between a requirement and an environment, all procedures supplied by a module are concretized with respect to the abstractions they provide. The rest of the algorithm proceeds as before. It is also possible to look for an abstract match if the concrete one fails, thereby allowing a module to provide procedures that work at either the concrete level or the abstract level.

7 Conclusions

The algorithm presented here belongs to the class of type-inference algorithms. It is a very generous algorithm in the sense that it will allow type matches where most other systems would not. Consider the following example[3]: Suppose that

$$G : \lambda t\,.\lambda u\,.t \times (\,u \rightarrow int\,) \rightarrow int$$

Furthermore, suppose that the body of G contains the expression f(x) where x and f are the formal parameters of G. A system that validates the types of procedures independently of their use must fail because of a call such as G(3,g) where

$$g : char \rightarrow int$$

This call appears to satisfy the type requirement of G, but execution of G will fail since the actual call g(3) is not correct. In the system presented here, an instance of the module containing G is checked for each actual call of G. The instance for G(3,g) will fail because

$$f : char \rightarrow int$$

does not unify with

$$f : int \rightarrow int$$

However, if the type of the actual parameter g was

$$int \rightarrow int$$

the corresponding module instance would pass the type-checking test.

This strategy presented here is useful because it does not require the maintenance of common include files. It also diminishes compile time because it can reduce the need to recompile modules that use modules that have been changed.

Acknowledgement

This research was supported by a grant from The Natural Sciences and Engineering Research Council of Canada.

References

[1]Michael R. Levy Type checking, separate compilation and reusability. In *Proc. of the SIGPLAN '84 Conf. on Compilers*, pp 285-289, June 1984,Montreal.

[2]R. Milner. A theory of type polymorphism in programming. *J. Computer and Systems Sciencces, 17:348-375, 1978.*

[3]Sophia Drossopoulou. *Private Communication.*