Static and Dynamic Type-Checking*

David C.J. Matthews

Abstract

The purpose of a type checker is to prevent an incorrect operation from being performed. A static type checker does this by stopping the compiler from generating a program with type errors, a dynamic type checker halts the program as it is about to make a type error. It is clearly useless to have a dynamic type checking system for a program which is to be produced, distributed and used by anyone other than the original authors since any type errors that occur would be meaningless to the user of the program.

On the other hand, where a user is guiding a program through some data, a dynamic type-checking system is reasonable. Examples are browsing through a database or structure-editing. Here type-errors have meaning to the user.

The ideal language would be basically statically type-checked but would allow dynamic type-checking when necessary. While this is possible with certain type systems there are others for which it is difficult. The implementation of dynamic type checking in various type systems is considered.

*This paper appeared in *Advances in Database Programming Languages,* Bancilhon F., and Buneman P. (eds), ACM Press, 1990.

1 Type-Checking

Type checking is an effective way of reducing programming errors. Its function is to identify those values to which an operation can be "sensibly" applied. The definition of "sensible" depends on the type system but usually operations like adding together two functions are not regarded as sensible while adding two numbers is.

1.1 Static and Dynamic Checking

One way of doing the type checking is to tag each value with a few bits which describe its type. Each operation checks the tag bits and gives some sort of failure if the values have the wrong type. *Dynamic* type checking will prevent some of the more obscure errors but has the disadvantage that the failure is only generated when the program is run.

A better method involves placing some restrictions on the programs that can be written so that the compiler can decide *statically* whether a program could possibly generate type failures when it is run. If the program can be shown to be type-correct there is no need for the tags and we know before the program is ever run that type errors will not occur.

1.2 Binding

Related to this is the question of binding. Declarations bind names to values and so have types. When an identifier is looked up the value with its type is returned. If there are several identifiers with the same name there must be rules for deciding which one is meant in a particular context.

One of the restrictions for static type checking to be possible is that the compiler must know the type of all the identifiers in the program. This requires *static binding* to identifiers, that is the identifiers are matched up with their declarations when the program is compiled and does not depend on the execution paths.

It is possible to have *dynamic binding* where an identifier is looked up when the program is run, but static type checking is possible only if all the identifiers with the same name have the same type. General dynamic binding requires dynamic type checking.

2 Static Type-Checking

Testing a program to try and find errors is difficult, and can never guarantee correctness. The ideal programming language would be one which imposed no restrictions but where the compiler could decide whether the program was correct. Unfortunately that is impossible and so we must accept some restrictions and even then we only have a limited form of correctness. However type-correctness is sufficiently useful that paying the penalty in terms of accepting some restrictions is reasonable. Recent developments in the design of type systems, particularly polymorphism[3], have extended the range of static type checking into areas where traditionally dynamic type checking was thought necessary.

2.1 Type Equivalence

At the lowest level a type must describe the structure of its values in terms of type constructors such as records and unions and the primitive types of the language, in order that the primitive operations can be type checked. In one form of type checking, if a type can be given a name it is treated as an abbreviation for the structure and two values are treated as the same type if they describe the same structure. This is *structural type equivalence* used for example in Algol 68[9].

An alternative is to define that two values have the same type only if they have the same type name. If the type names are different the types are incompatible even if the structures of the types are the same. *Name equivalence* does not mean that the structure of the type is not visible, only that it is not used for type equivalence. *Abstract types* are a variation of name equivalence where the association between a type name, the *abstraction* and its structure, the *implementation* is only visible within the abstract type definition[4]. Outside that it has no structure and name equivalence is used. When types can be returned as a result of functions name equivalence or a variation of it is needed to ensure that type checking is decidable[2][6].

3 Dynamic Binding and Type-Checking

Static type checking is useful when a program is being produced which is to be executed later. If the program is to be executed immediately, and particularly if it just consists of a single command, there is really very little difference between static and dynamic type checking. Command line interpreters, or "shells" are an example. The command

```
edit afile
```

typed to a command interpreter would probably involve a search for the files `edit` and `afile` and checks that `edit` was an executable file and `afile` was a text file. The search and type checking for `afile` might well be done from within the `edit` program. There is no advantage in treating type checking separately from execution.

However, if several commands are put together in a command script it starts to look more like a writing a program. Because the individual commands are dynamically bound and type checked it is not possible to statically type check the completed script even though it resembles a programming language procedure. Apart from command interpreters the Mentor programming environment[7] is another example where structure editing commands in the Mentol language can be put together into procedures.

Apart from the fact that there is no advantage in statically checking a command which is to be executed immediately, there are other reasons why dynamic type checking is used. Programs often create file names, for instance by appending standard suffixes onto a name to make a set of related file names. Since the files are dynamically bound they must be dynamically type checked.

Dynamic binding may not only be by name but by other mechanisms as well. In a structure editor a user may select an item by pointing to it with a mouse. Different parts of the structure will have different types so that changes to the structure are constrained, but the function that returns a selected value must be able to return a value of any type. Dynamic type checking must be used if this value is to be copied somewhere else in the structure.

4 Combining the Two

Static type checking is needed for programs which are to be executed in the future, but dynamic type checking is needed for interactive operations. If we have a system where both of these activities can occur we really need both mechanisms.

The obvious way to do this is to take a static type system and add some additional syntax and a new type *dynamic*.

$$dynamic\ x$$

constructs a value of type *dynamic* by packaging up the value with information describing its type. The inverse operation

$$coerce\ d\ to\ t$$

checks that `d` is a dynamic value with the type information appropriate to the type `t` and returns the original or raises an exception. The syntax is taken from Amber[1]. Dynamic binding can be done by returning values of *dynamic* type and then coercing them to the appropriate type.

A dynamic value contains a value and a representation of the type. The type representation must contain enough information for the **coerce** operation to do the same kind of checking at run-time as the compiler would do at compile-time. We do not want the dynamic type mechanism to subvert the static type system. This may be more complicated than it appears. The rules for static type equivalence which are applied at compile-time may not be reproducible at run-time. To see how the static type system influences the dynamic typing some static type systems will be examined, both from languages which have dynamic types and those that do not.

4.1 Structural Equivalence

The simplest type systems for this purpose are those such as Amber that have a fixed number of primitive types and use structural equality between types. Each primitive type can be assigned a unique identifier and data structures used to describe the structured types. Because names for types are just synonyms for the structure we can always use a representation of the structure for the type representation. Although the name may be declared locally the structure representation is valid anywhere so dynamic type checking is safe. Type inheritance in Amber does not have any serious effect on this.

4.2 Polymorphism

If the language allows polymorphic operations, as in ML, dynamic type checking has to be arranged more carefully. Consider the following two functions.

> ***fun*** *get_dynamic d* = ***coerce*** *d* ***to*** α*;*
> ***fun*** *make_dynamic x* = ***dynamic*** *x;*

get_dynamic takes a dynamic value and coerces it to the type variable α. If the dynamic type matching rules follow the static type rules these should match for any type since unification of a type variable with any type would succeed. Clearly this would allow the type system to be broken because we could make a dynamic value out of, say, an integer value, pass it into *get_dynamic* and treat the result as a string.

make_dynamic will take any value and make a dynamic value from it. This again could break the type system. A solution to both of these problems is simply to forbid polymorphic types in **coerce** or **dynamic** operations.

4.3 Abstract Types

If the static type system allows the user to create abstract types we have to produce a unique identifier for each abstract type and use those in type representations. A dynamic value created from the abstract type will contain different type information to a dynamic value created from the representation. This may be difficult if the dynamic value is created inside the abstract type package since there the distinction between the values of the abstract type and the implementation type is blurred.

4.4 Parameterised Types

Parameterised abstract types create another problem. If we have a type which can be parameterised by other types or values we need to ensure that any dynamic values created from values of the result type or the argument types have the correct type representation. If the parameterisation simply involves macro-expansion this is relatively easy but if it is done at run-time the type representations will have to be passed as run-time values. The type identifier for the resultant type may have to be created dynamically when the parameterisation is done.

For example, we may define a type *tree* which is a binary tree parameterised by the type of the leaves. Inside the type definition we write an operation to walk over the tree in response to commands from the user and return either a leaf or a piece of tree as a dynamic type.

```
abstype α tree = Leaf of α | Tree of α tree * α tree
with
  fun move "value" (Leaf l) = dynamic l (* Return the leaf *)
    | move "value" (Tree t) = dynamic t (* Return the tree *)
    | move "left" (Tree (l, _)) =
          move (nextcommand()) l (* Move left *)
    | move "right" (Tree (_, r)) =
          move (nextcommand()) r (* Move right *)
    | move _ _ = raise bad_command (* Anything else *)
```

end

In this example Standard ML[8] has been used with the addition of the operation to create dynamic types. The **dynamic** operations have been applied to polytypes which would allow the type system to be broken, so clearly this cannot be allowed as it stands. In any case it is difficult to see how it would achieve what is wanted, which is for leaves of *int* trees to be returned as dynamic values coercible to values of type *int.*

However if we treat the parameterised type more like a function so that the parameterised type is always used in its parameterised form we can safely allow dynamically typed values to be created and probably get the required behaviour. In Standard ML this could be done using a parameterised module, called in ML a *functor*[5].

```
functor Tree(Elem: sig type t end) =
  struct
  datatype tree = Leaf of Elem.t | Tree of tree * tree;
  fun move = ... (* As before *)
  end
```

The type *tree* is only available when the functor *Tree* has been applied to a module, in ML a *structure,* containing a type. Other rules in ML ensure that this is a monotype.

```
structure IntTree = Tree(struct type t = int end);
```

This creates a tree whose leaves are integers. In order to get the effect we want the type representation for *int* must have been passed into the functor so that the dynamically typed values returned from leaves are recognisably integers. The tree type *IntTree.tree* itself is a new atomic type so the representation for the type can be created dynamically when the functor is applied.

In CLU, which has parameterised clusters and dynamic types, this is rather more difficult. A cluster parameterised by the same parameter values denotes the same type wherever it appears in a program. Since the type may appear in different segments the CLU linker must examine all the types, construct unique identifiers for each different type, and then pass the identifier to be used for the result of each parameterised type into the type as an additional argument.

4.5 Types as Values

In languages such as Russell and Poly types can be treated as first class values. A type and operations associated with it are packaged together and treated as a run-time value. To ensure decidability name equivalence has to be used. An expression such as

let *atype* == ***if*** ... ***then*** *type1* ***else*** *type2;*

declares *atype* to be a type which is not the same as either *type1* or *type2,* since it is in general not decidable which is actually being returned.

This causes problems if we try to use the dynamic type scheme suggested above for ML. Suppose *type1* and *type2* are different implementations of trees as in the ML example. They both have *move* functions which return dynamically typed values. If we make a tree of type *type1* we expect the dynamically typed values to be coercible to *type1* but not to *type2* or to *atype.* Similarly values from *atype* should not be compatible with either *type1* or *type2.* Unfortunately if the type representation is put into the dynamic values inside the abstract type declaration the dynamic values returned from *atype* trees will be either *type1* or *type2* depending on the actual type returned by the **if**. There seems to be no way to avoid the dynamic type checking behaving differently to the static type checking.

5 Conclusions

Certain applications require dynamic type checking in an otherwise statically typed language. For some type systems this is relatively easy to arrange, but others require considerable thought if the security of the static type system is not to be undermined.

References

[1] Cardelli L. *Amber* AT&T Bell Labs Technical Report 1984.

[2] Demers A. and Donahue J. *Revised Report on Russell* TR79-389, Department of Computer Science, Cornell University, 1979.

[3] Gordon M. et al. *A Metalanguage for Interactive Proof in LCF* Fifth Annual Symposium on Principles of Programming Languages, Tucson 1978.

[4] Liskov B. et al. *CLU Reference Manual* Springer-Verlag, Berlin 1981.

[5] MacQueen D.B. *Modules for Standard ML* AT&T Bell Labs Technical Report 1985.

[6] Matthews D.C.J. *Poly Manual* SIGPLAN Notices. Vol.20 No.9 Sept. 1985.

[7] Mélèse B. et al. *The Mentor-V5 Documentation* Technical Report 43, INRIA 1985.

[8] Milner R. *A Proposal for Standard ML* in "Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming", Austin, Texas 1984.

[9] van Wijngaarden A. et al. *Revised Report on the Algorithmic Language Algol68.* Springer-Verlag, Berlin 1976.