stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 60/76 SEPTEMBER

H.J. BOOM

EXTENDED TYPE CHECKING

Prepublication

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

AMS(MOS) subject classification scheme (1976): 68A30

ACM-Computing Reviews-categories: 4.22, 5.24

Extended type checking *)

by

H.J. Boom

ABSTRACT

It is possible to formulate type checking in such a way that much greater flexibility is obtained, both in allowing greater generality of expression to the programmer, and in allowing the programmer to specify extra type checks to be applied to his programs. These improvements arise indirectly from separating types into "specificational types", which indicate predicates required of input data for proper operation of a program and "representational types", which specify how a type is implemented. It turns out that data-type checking is intimately related to data-flow analysis, and that the two can be fruitfully integrated. This integration has interesting effects on the definition and implementation of programming languages.

KEYWORDS & PHRASES: *data type, mode, specificational data type, representational data type, mvalues, modals, data-flow analysis, programming languages, program correctness, symbolic execution.*

*) This report will be submitted for publication elsewhere.

Algol 68 is extremely bug-resistant. For example, in one program of about 1000 lines, only about four errors penetrated to run-time and these were uninitialized-variable errors. Each of these variables was intended to have been initialized as a side effect of a procedure call. By far the greatest majority of programming errors were caught by the mode checking at compile-time. As has been noticed elsewhere, the reason for this is that a mode corresponds to a number of statically verifiable properties of an object. These properties may not be explicitly written in the program source code, which usually discusses only the representation of the object on the machine, and may exist only in the mind of the programmer. The programmer must himself ensure that the primitive operations he defines preserve the properties he is concerned with. Thereafter when the compiler checks mode compatibility it is implicitly ensuring that these properties hold, even though it knows nothing of the properties themselves.

An Algol 68 mode is thus used by a programmer in two different ways, as a means of indicating how composite values are to be represented on the machine, and as an indication of the operations and properties that are to be acceptable on values of that mode. When a mode performs the first role, we shall call it a "representational type"; in the second role we shall call it a "specificational type". Algol 68 does not distinguish between the two kinds of types, and requires that the mode of every value must be known at compile time. We wish to investigate the consequences of these restrictions. We shall do this by investigating what happens if we relax them.

One constraint on such relaxation is that if a program could benefit from various static checks under the Algol 68 regime, then it should still be subject to them in the relaxed regime; however, if explicit use of indeterminate types is made, static checks may be suppressed. Our new language, Unrestricto, will therefore still allow statically known types, but will not require them. Just as, in the past, procedures have become data items, so do types become data items. The structure of type usage in present-day programs is such that types can nonetheless be statically determined by using constant propagation. We shall therefore discuss several matters in this paper:

- the nature of the type generalizations.
- the nature of constant propagation.
- the way in which the two can be combined to formulate restrictions to promote efficient implementability.

The nature of the type generalizations.

We introduce a new mode into the language, called "type" or "type". It is expressions of mode "type" which can be used in declarations to declare indicators. In particular, "real", "int", "ref int", "type" , "proc type" are all objects of mode "type". "ref" is an operator, which accepts a type and yields another type:

```
op ref = (type a) type: ... ;
```

Consider, as an example, a matrix multiplication routine. Such a routine usually consists of three nested loops which perform various additions and multiplications. For matrix multiplication to make sense, the matrix elements are usually required to come from a ring. The same collection of nested loops can thus be used for matrices of integers, reals, complex numbers, or polynomials, or even matrices whose elements are themselves matrices. However, in normal languages it is necessary to specify the type of matrix elements before the matrix multiplication routine is compiled, eliminating this flexibility. It might be pleasant to give the type of the ring element as a parameter as well. This gives us:

```
proc   mat mul = (type elem, [,] elem a,b) [,] elem:
       begin
       var c:[lwb a: 1 upb a, 2 lwb b: upb b]elem
       for i from 1 lwb a to 1 upb a
       do for k from 2 lwb c to 2 upb c
          do elem s := elem 0;
             for j from 1 lwb b to 1 upb b
             do s := a[i,j] * b[j,k] + s
             od;
             c[i,k] := s
          od
       od;
       c
       end
```

This very much resembles the "modals" proposals of Algol 68, but there are differences. First, there is no pussyfooting about the arbitrariness of the type "elem". Anything at all is permitted, as long as the operations "+" and "*", and the constant "0" are defined on it. The type is not hidden behind refs and procs in order to give it a statically known size. If the compiler wishes to use descriptors or other mechanisms in order to represent it, that is its business, not the user's. We shall discuss such mechanisms later. Second, no elaborately contrived mechanisms are used to enable compile-time operator identification. The identification of the operators "+", "*", and "0" indeed does depend dynamically on the actual types provided.

It might be pleasant to place the various requirements on the parameters explicitly at the procedure heading, thus:

```
proc matmul = (type elem, [,] elem a,b) [,] elem:
     require op + (elem, elem) elem,
             op * (elem, elem) elem,
             const 0 elem,
             2 lwb a = 1 upb b,
             2 upb a = 1 upb b
     eriuqer:
     begin ....
```

These requirements can then be made more detailed than those the compiler could otherwise glean from the program text; for example, the result types of the operations have been specified.

If we look at this program with Pascal eyes, we find that there are several implicit type parameters here, namely, the subscript ranges for a and b. If we make these explicit, we get

```
proc matmul = (elem, r1, r2, r3: type,
                  a: array [r1,r2] of elem,
                  b: array [r2,r3] of elem)
          array [r1:r3] of elem

     require op + (elem,elem) elem,
                * (elem,elem) elem,
```

```
        const 0 elem
    eriuqer:
    begin
    for i in r1
    do for k in r3
       do elem s := elem 0;
          for j in r2
          do s +:= a[i,j] × b[j,k]
          od;
          c[i,k] := s
       od
    od;
    c
    end
```

(the syntax of Unrestricto fluctuates haphazardly between that of Pascal and Algol 68).

In this procedure we have the advantage of full static type checking, even though it appears that all of the data types involved are unknown. The reason is that we have indeed used specific data types, but that these types are specificational, rather than representational. We require that our operands have various properties. We give those properties names and call them data types. For example, the range over which the first subscript can vary is a property of an array. We give this property of "b" the name "r2". Thereafter, we use data type propagation as a method for ascertaining that this name of a property is appropriately associated with the various objects wherever necessary. A single property, such as "r2", may appear in various places in the program, and apply in various ways. It indicates the second subscript range of "a", the first subscript range of "b", and the range of "j". A compile-time subscript check can then easily be performed. The specification data types have been provided, but not the representational ones. If *matmul* were ever to be called, the compiler would have to perform or compile a check to determine whether the parameters did indeed have the correct properties. Seen in this way, the data types and requirements form a "pattern" (in the Snobol sense), which has to be matched against the actual

parameters. This pattern match may occur at compile-time or at run-time, as an integral part of operator identification or as an extra check. If we view matters in the Snobol way, our patterns need not have all these types explicitly provided as parameters, and we may extract them during operator identification or type compatibility checking. Let us use "v : p" as a pattern meaning "an object of type *p* which we will call v. If the pattern matches, it has the side effect of declaring v and defining a value for it. We can now rewrite our procedure heading as follows:

```
proc matmul = (a:array [r1: type, r2: type] of elem:type,
               b:array [r2,r3:type] of elem)
                   array [r1,r3] of elem:
    require
            op + (elem,elem) elem,
               * (elem,elem) elem,
               0 elem

    ensure
    begin ...
```

Execution of a call to matmul would then begin with a pattern match between the actual and formal parameters to determine "a", "r1", "r2", "elem", "b", and "r3". The requirements would further provide "+", "*", and "0".

We shall define a "mode" to be a combination of specificational data type with a representational data type. This involves specifying what the properties of objects of that mode are, and how they are to be implemented in terms of other more primitive objects. It is quite possible to associate one specificational data type with several representational types. For example, one might very well have different means of representation for normal arrays and sparse arrays. These would correspond to different modes, perhaps "array" and "sparse". The two modes would have the same specifications, but different representations. Both of them could be used as arguments to the procedure *matmul*, but they would be implemented differently.

When writing types down, we shall use the notation

$$U(i)x_i$$

for a discriminated union of the types x_i for all applicable values of i. If we wish to mention a restriction on the index i, we shall use notation like U(int i)x_i. We shall include bounds in the mode. The mode denoted ref flex [ ]real in Algol 68 will be written in the following form:

ref U(int i,j) array [i..j] of real.

The mode ref [ ] real becomes

U(int i,j) ref [i..j] real.

In the first case, one single variable can refer to arrays of various sizes; in the second case, the size is built into the variable, although variables of different sizes are acceptable. It will be assumed throughout this paper that operations can be applied on values inside unions provided only that they be defined on each of the possibilities within the union.

Roughly speaking, information must be stored at run-time for

(a) values of primitive modes, and

(b) places where a union tag is needed.

The above notation makes union tags more explicit.

If we assume that there is only one way of representing arrays, the representational types for the types "elem" and [,] elem in our matrix multiplication example could be:

elem:	U(type elem) elem
[,]elem:	U(type elem) [,] elem

Each of these would have to have a type-tag specifying the mode.

It is also possible to make a specialized version of the program for a specific type. The union could then be reduced to fewer modes or only one, with a corresponding gain in possible efficiency. In particular, the union may be sufficiently simplified to permit static operator identification.

The representation at run-time can depend on the information that is available at compile-time concerning the object. The more that is known, the fewer the type-tags that mey be required. In the extreme case where the value

itself is known, no information at all may be required at run-time. At run-time all values are eventually represented by constellations of bits, and these constellations are meaningless without a priori knowledge of the representational type.

Other properties in the mode.

We have just illustrated how properties of values can be included in their modes. The properties used in the example were the sort of properties that compilers have traditionally been concerned with and have traditionally been associated with data types, whether operators had been declared, whether subscripts were in bounds, etc. In principle, the types involved can be constructed by taking a universal type, which we call "any", and applying restrictions to it, such as

it must be an array with two subscripts.
The upper bound must be χ.
An operation "+" must be defined.

These restrictions define properties which a programmer may use in constructing a program. They do not imply that the type is implemented in any special way. But by associating these properties in a data type, we can use well-understood methods of propagating type-information through a program. In the example, the restrictions have been those traditionally associated with data types, because it is then easy to see that such propagation is reasonable. There is no reason, however, for restricting consideration to such properties. Any property capable of expression can be used to define a specificational data type. The compiler may not be able to perform the analysis necessary to verify that any particular value satisfies these properties, but in the worst case, it can check that the property has been explicitly been asserted by the programmer as axiom whenever required. Including such a property in the type enables one to reduce the number of places where it needs to be explicitly asserted; it is automatically propagated through assignments, function calls, and so forth as part of the data type.

It is reasonable to suppose that, if the specifications and the program are both modular and fit well with each other, these properties will not need to be specified often. To prevent deceitful assertions by the user of

a module, it may be possible to

- give the property a name,
- assert it at appropriate points within the module as axiom,
- permit the programmer to mention that property in the rest of the program, but
- prohibit him from asserting it as axiom elsewhere.

Although a user may be aware that the property holds in many parts of the program, and may even mention that fact in data types, he can not assert it as axiom. The only places in the program where human verificaiton may be needed for the property will be those in the module where it is asserted as axiom. The compiler, even if incapable of verifying the properties directly, will then automatically extend the effectiveness of the human verification to the rest of the program as part of normal type checking.

The relation between types and data-flow.

Let us consider why type checking can provide such flow-of-predicates for verification. The properties we are concerned with are properties of values (at run-time), not properties of variables (at compile time). Data types, however, are predicated on names in the source program, and are then indirectly attributed to the values which execution may associate with these names. These data types specify properties of the values (specificational) and the way in which the values are to be interpreted (representational). It may be senseless to speak of properties of values (such as $x > 3$) without knowing the data type. If the value x is represented by some bits on a machine, it is necessary to know the data type before one can determine whether $x > 3$. For the time being, we shall avoid this difficulty by assuming that run-time values are tagged with their representational types, thereby making their meanings intrinsic. We have already seen how to abolish these type tags in practice by encoding them into the representational data type.

A type-secure language is so constructed that whenever a compile-time name N is of some representational type T, then all values which can ever be named by N will have the type-tag T. Therefore, if we can associate a predicate into the type T, type-security will imply that that predicate will

be satisfied by all values which will ever be named by N.

How is a type-secure language usually constructed? Predicates apply (or fail to apply) to run-time data. Data flow (and thus the continuing validity of predicates) is related to the structure of the source program. In Algol 68, for example, data may flow from a defining to an applied occurrence of an indicator, or from the source of an assignment to a subsequent dereferencing. It is, in general, impossible to determine data flow without examining the possible paths of control flow. However, it is possible to place bounds on the possible paths of control and data flow in a program, and careful analysis can reveal fairly restrictive bounds. A crucial property of type-secure languages is that all paths of possible data flow are between program components of the same type. By specifying types for identifiers, we are indicating limits on the possible data flow. These limits are then checked by the compiler, and the predicates associated with a type can be safely predicated on run-time values.

Data flow thus determines the flow of validity of predicates. Association of types with identifiers serves as some indication of the data flows not intended by the programmer, and thus helps catch errors.

We shall now consider an example in which it appears unwise to associate a type with a variable. The example is a Fortran program, but we shall use the terminology of this paper instead of that of the Fortran standard.

```
EQUIVALENCE (X,I)
X = 3.0
X = X + 1.0
I = ITRUNC (X)
I = I - 1
WRITE (...) I
:
```

Here we see a single variable with two names, "X" and "I". When one uses the name "X" one implicitly asserts that the value being fetched or stored is real; when "I" is used, integral. "X" and "I" name one single variable which is capable of containing either a real or an integral value. To per-

form static type checking, it does not suffice to associate date types with the identifiers, as the following code shows:

```
x = 3.4
I = I - 1
```

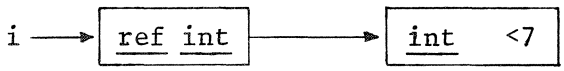
We must associate data types with possible paths of data flow. There is a data flow from a definition of a variable V along a possible path of control to (a) use(s) of V, assuming that there are no other definitions of V along that path. We associate a type with such a data transmission, and then we can check that the definitions and uses of the variable are properly matched. Associating types with variables corresponds to a less refined data flow analysis, namely, that any definition of a variable can be associated with any use. A more refined analysis takes account of the possible paths of flow-of-control. Ultimately, one wants to consider only the data transmissions and control paths actually involved in program execution, but these cannot in general be determined statically.

It is reasonable for a programming language to define a standard class of possibly possible data transmissions and to use this class of transmissions for type checking. The type-checking done in conventional type-secure languages can usually be determined by connecting all definitions of a variable-identifier to all uses by possible transmissions and then extending this set by using tricks for procedure-calling, values of expressions, etc. If type checking is extended to include arbitrary user-specified predicates, a more refined view of data flow is necessary. To this end we want our programming language to be such that the data flow is as explicit as possible. Features that make data flow explicit, such as identity declarations and value-result parameter passing are to be encouraged; others, such as call-by-name and pointer variables, should be discouraged, so that the programmer will use them only when explicitly necessary. It would not be unreasonable to determine the scope of an identifier to extend only along those control paths where it has a value -- a variety of definition-before-use in which "before" relates to flow-of-control instead of to textual order. Rigorous checking of such paths of data flow may cause great reduction in the number of undefined-variable errors that can penetrate to run-time.

The nature of variables at compile-time.

A compiler needs certain information about variables, and may spend much effort discovering other information which it may not strictly need, but may improve efficiency. This information usually contains the data type and the run-time access algorithm, and sometimes also the pattern of use and disuse of the variable, whether it is constant, or other such optimization data. For convenience in speaking of this information about a variable, we shall gather it together into a single package and call it an "mvariable". In general, if "foo" is any concept which may have a significance in the abstract run-time machine, "mfoo" will designate the analogous concept at compile time, and will usually contain predicates which the run-time object will satisfy. We say that the mfoo "represents" the foo. "mvalues" represent run-time values. The mvalues necessary for fairly normal compilation are those representing run-time values which can be named directly in the source program (values of variables), and those on the temporaries stack. It is possible for mvalues to have relations with each other; in Algol 68, for example, an mvalue for a structured value may very well have two other mvalues for its fields, and an mvalue for a reference-to-real value may "m"refer to an mvalue for a real value.

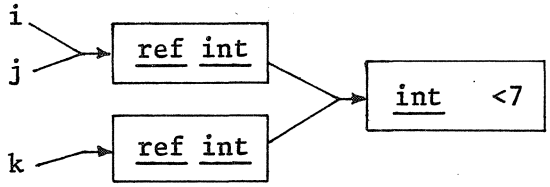
We shall draw mvalues as boxes, and the relations between them as lines and arrows. Suppose it is known that the value of a variable, called "i", is less than 7. In conventional notation one would write "i < 7". We, however, shall draw the following diagram



Proceeding from the above predicate, let us assume that the following lines of code are:

```
ref int j = i;
int k;
k := i;
```

After these lines we have the mstate:

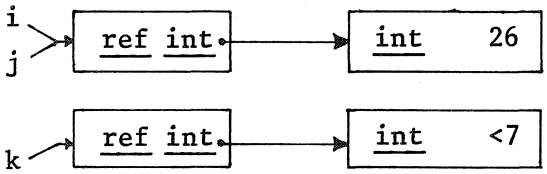


We notice several things. First, the identifiers themselves appear as things to talk about, not, as in traditional predicate-calculus notation, as components of the descriptive formalism. This enables us to distinguish clearly between the identifier and the variable or value it possesses.

Second, it is not necessary to postulate an unbreakable bond between a variable and its identifier, as is done in many formalisms for proving program correctness. Third, it is possible to speak of values without necessarily doing so in conjuction with particular identifiers in the source program. If the compiler were to process yet another statement,

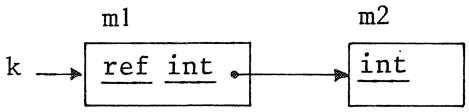
```
i := 26,
```

it would end up with the description:



The corresponding change in j is implicitly performed, and does not require specific proof rules. Notice that the mvalue "int <7" has now been completely shifted from "i" and "j" to "k". Any properties formerly involving this mvalue still involve it, without any need to alter them, provided they dealt with the mvalue directly. This suggests that the proper objects to use in predicates are not identifiers (which are traditional) but mvalues. A human being might have to code identifiers in order to write an mvalue as a string of characters, but a computer need feel itself under no such constraint. The notation used to represent information depends on the medium, and conventional data-structuring techniques can be a great help in the representation of predicates within a machine.

Let us consider another example. Let "k" be of type "ref int", having been assigned a value previously in some unscrutable manner:



(For convenience, we have labelled the mvalues "m1", "m2", etc.). We now declare an array:

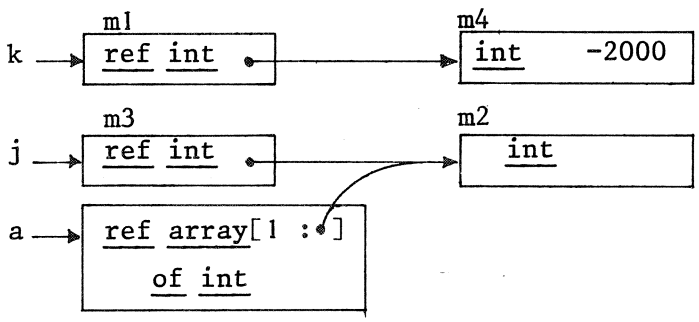
```
array [1 : k] of int a;
```

This array will have type "ref array [1 : m2] of int". This type contains the mvalue "m2", and is therefore independent of any subsequent actions on k.

In particular, it is possible to execute

```
j := k;
k := -2000;
```

without altering the type of "a". In fact, after these two assignments, "ref array [1 : j] of int" will be recognized as the type of "a", since "j" is now used to denote the same mvalue:



The reasons why this scheme can work so simply is that mvalues reflect data flow more precisely than variables. In fact, the life history of an mvalue consists of a creation (in some construct which computes or obtains a value) followed by a number of uses (at least within straight-line code). If one has some mechanism for deleting "useless" mvalues (i.e., those whose value will no longer be needed), the mvalues correspond to the data transmissions discussed earlier. Now, however, we have made objects out of them.

This makes it easier to discuss their properties. Since mvalues are static objects corresponding to values, we can replace dynamic types involving values by static types involving mvalues.

An algorithm for constructing mvalues to be associated with a program despite the presence of awkward features such as pointer and procedure variables and recursion has been discussed in [BOOM 1]. A rather elegant mathematical formulation of some related ideas (using quite different notation) appears in [COUSOT 1].

One property of mvalues that has been implicitly assumed in the above discussion is "individuality" [BOOM 1]. An "individual" mvalue is one for which the compiler can unambiguously determine where in the program its corresponding value is used or altered. If the program is sufficiently convoluted, an mvalue (and its values) may "escape" from the compiler's ken. To perform data-flow and side-effects analysis, escaping must be explicitly recognized and treated as a side-effect, just like assigning to a variable or reading its value. Data-flow analysis can be done easily only if the mvalues involved are individual. In practice, the vast majority of variables in programs can indeed correspond to individual mvalues, if the compiler is willing to do proper analysis. Most do not. It is important that language features be designed so as to make individuality easy to detect. For example, parameter passing by reference is more difficult to analyze than by value-result, because there are much greater possibilities for abuse (such as saving a pointer to the variable for later misuse). If the language were to be so constructed that individuality is easy to preserve and nonindividuality explicitly indicated in the syntax or the type, data flow analysis would become much easier.

Effects on language.

These considerations suggest the following approach to language difintion and implementation.

First, a superlanguage is defined using a grammar (a precedence grammar with kludges may suffice) and an interpreter for semantics. This superlanguage should be so designed that most data flow can be easily determined at compile-time.

Next, a series of analysis procedures are defined to determine mvalues and perform elementary transformations on the program, such as making specialized copies of procedures or deleting redundant code, where necessary or where requested by the programmer. This phase of the definition (and probably of the implementation) may be analogous to optimization and macro-processing.

Finally, restrictions are placed for efficient implementability. These restrictions define the official level of language, and will usually consist of requirements that certain information be known to the compiler after the above-mentioned analysis. Conventional code generators may wish to know the types of values at compile time, instead of generating run-time type tests.

The above approach may also be useful on some existing languages; for example, APL could benefit from the analysis suggested above.

[BOOM 1] H.J. Boom, *Optimization Analysis of Programs in Languages with Pointer Variables*, Phd. thesis, Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada.

[COUSOT 1] Patrick & Radhia Cousot, *Static Verification of Dynamic Type Properties of Variables*.