Type Checking in an Imperfect World

Terrence C. Miller

Applied Physics and Information Sciences Dept.
C-014
University of California, San Diego
La Jolla, CA 92093

We present an algorithm for the determination of run-time types which functions in the presence of errors, and show that it provides more information than that obtained using a previously published algorithm.

In Section 1 we define the problem and state the requirements for a practically useful type prediction algorithm. In Section 2 we introduce a model programming language and in Section 3 define type inference rules for that language. Section 4 presents a type prediction algorithm and Section 5 describes how to apply the results to solve the problems stated in Section 1. Section 6 presents an example of our procedure and demonstrates how previous work does not satisfy all requirements.

1 - PROBLEM

A requirement for the efficient translation and execution of many programming languages is the ability to predict at compile-time the "types" of variables and expressions. A type is defined to be some subset of all possible quantities that may be represented in the language in question. Factors which can be used to differentiate types include rank (number of dimensions), size, storage representation, and the intended meaning of the data. Type predictions can be used in three ways:

1) To permit compilation instead of interpretation of a language with insufficient typing mechanisms. The work of Miller [M] describes the necessity of this facility as part of an APL compiler. In addition, even languages with extensive declaration facilities such as Pascal contain problem areas which can not be reasonably compiled without type prediction. An example is:

```
VAR
   I,J: INTEGER;
   K: SET OF INTEGER;

BEGIN
  ...........
  K:=[I..J]
  ...........
END;
```

in which the set K is a set over the integers. This set poses a considerable problem to any Pascal compiler which (as most do) represents sets as a bit vector with a position for each possible element [P].

2) To permit generation of more efficient code.

3) To permit elimination of redundant run-time type checking. Bauer and Saal [B] demonstrated the importance of this for APL, and a similar situation holds for value sensitive type checking in Pascal [P].

We claim that to be practically useful a type prediction algorithm must work in the presence of errors, and generate additional information as follows:

1) The type determination system must identify type conflicts and locate their source so that an appropriate diagnostic can be issued.

2) For each type requirement identified, the system must also locate the earliest point in the execution of the program (or possibly before) at which the satisfaction can be verified.

3) The system must differentiate between requirements for legal execution and predictions of the result of any execution.

We claim that the algorithm described below meets these requirements. It is derived from an earlier version presented in [M]. The form in which the algorithm is described is derived from that used by Kaplan & Ullman in [K], and we use their results.

2. - MODEL PROGRAMMING LANGUAGE

The algorithm is stated in terms of a model programming language. The basic operations are an assignment statement with the form:

$$X_m := func(X_1, .., X_d)$$

where

X_i's are variable names.
func is a function of degree d >= 0

and a predicate of the form:

$$pred(X_1, ..., X_d)$$

where

the X_i's are variable names
pred is a boolean function of degree d >= 0

A program is a directed graph with the following types of nodes:

1) Assignment nodes - which have in-degree and out-degree equal to one and which are labeled with an assignment statement.

2) Start node - (one and only one) which has in-degree zero and out-degree equal to one.

3) Stop nodes - which have in-degree one and out-degree zero.

4) Fork nodes - which have in-degree one and out-degree two and which are labeled with a predicate.

5) Join nodes - which have in-degree two and out-degree one.

A "program execution" is a path through the program graph which begins at the start node and ends at a stop node.

Expressions in a real programming language are represented by creating additional variables to hold the result of each single operation (except for assignment operations in the real language, which use the variable specified), and modeling the expression as a sequence of assignment nodes.

Control structure in a real language is modeled by converting it to an equivalent IF...THEN....ELSE... form which can be modeled using the Fork and Join nodes. We associate each of the edges leaving a Fork node with a particular result of the predicate (thus the predicate supplies information for type predictions), but we consider all paths as possible.

3. - TYPE INFERENCE

We associate with each edge of the program graph information about the type of each variable. The type information at a given position is that information which is valid when execution of the program takes that edge. For each edge j and variable name X_k we have two constant functions - $P_j(X_k)$ and $R_j(X_k)$ - giving an element t in the set T of all types.

The set T is the Cartesian product of the power sets of the sets of all possible values for each measure which may be applied to a variable in the real programming language in question. The type t containing the empty set as the possible values of each measure is written as $\emptyset$, and the type containing all possible values is written as $\mathcal{I}$. However, in many cases we will be concearned only with a small number of the possible elements of a given power set and will give those elements names (for example the value of a variable may be described as Integer, Real Number, or Character without finer distinctions). The set of values for a particular measure included in a type t is specified by <name of measure>(t).

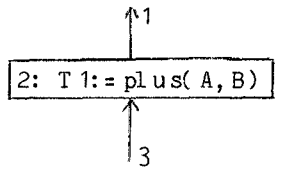
P specifies predictions that a given variable will have the specified type each time program execution takes the given edge. We use p to refer to the vector whose elements give the predictions at each edge. R specifies requirements on the type of a given variable which must be met each time program execution takes a given edge. We use r to refer to the vector whose elements give the requirements at each edge.

The set expressions for a measure may contain property variables. These variables are used as place holders for type information not yet available. The expressions are manipulated symbolically. Associated with each variable Z_k is a set expression A_k giving all allowable values for Z_k.

3.1 - INFERENCE WITHIN STATEMENTS

Using property variables, it is possible to specify the requirements and

guaranteed results associated with an individual assignment or fork node by specifying types on the adjacent edges. For example, the model and the Rank conformability rules for the APL expression A+B are:



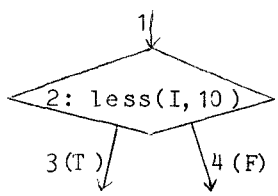
Rank(P_1(T1)) is Z_1
Rank(R_3(A)) is $0 \vee Z_1$
Rank(R_3(B)) is $0 \vee Z_1$

where T1 is a unique, created variable name and where Z_1 is a unique property variable whose value may range over all legal array ranks. A second example is the Pascal fragment:

```
VAR
  I: 0..19;
  J: 0..9;
  K: 10..19;

BEGIN
  READ(I);
  IF I<10 THEN J:=I ELSE K:=I;
END
```

where the IF statement is modeled as:



Value(R_1(I)) is [0..19]
Value(P_2(I)) is [0..9]
Value(P_4(I)) is [10..19]

.2 - INFERENCE ACROSS STATEMENTS

For each node 1 containing an assignment statement of the form:

$$X_m := func(X_1, \ldots, X_d) \quad d > 0$$

we define 2*d functions:

$$TF_1(X_k)(t) \quad 1 <= k <= d$$

and

$$TB_1(X_k)(t) \quad 1 <= k <= d$$

where the TF's specify predictions which can be made about the type of the result of the function based on predictions of the type of an operand, and the TB's specify the requirements which must be placed on operand k in order to satisfy requirements placed on the result.

3.2.1 - FORWARD INFERENCE

Given a prediction of the types of the variables before execution of a node 1, it may be possible to refine the prediction of the types of variables after execution of the node. The new prediction f_1 is the union of the predictions f_1^j derived from each edge j entering that node, and is applied (intersection with the old prediction) to all edges leaving the node. The forward inference function giving a new prediction for the type of variable Xk is given as $f_1^j(Xk)$ which is defined by:

$P_j(X_k)$ if node 1 is not an assignment to X_k

the intersection of all $TF_1(Xm)(P_j(X_m))$'s if node 1 is an assignment to X_k with operand X_m

For convenience, we will define f_1^k to be $\emptyset$ for all edges k which do not enter node 1.

3.2.2 - BACKWARD INFERENCE

Given a requirement on the types of variables which must hold after the execution of a node 1, it may be possible to refine the requirements which must be imposed before the node executes. The new requirements b_1 are the intersection of those (b_1^j) generated from each edge j leaving node 1 and are applied on each edge entering node 1. The backward inference function giving a new requirement for the type of variable X_k is given as $b_1^j(X_k)$ which is defined by:

$R_j(X_k)$ if node 1 is not an assignment or reference to X_k

$TB_1(X_k)(R_j(X_k))$ if node 1 is an assignment to Xk and Xk is an operand

$R_j(X_k)$
$\cap\, TB_1(X_k)(R_j(X_0))$
if node 1 is an assignment to X_0

with operand X_k and $k \neq 0$

1 otherwise

For convenience we define b_1^k to be 1 for all edges k not leaving node 1.

3.3 - INFERENCE ACROSS THE PROGRAM

3.3.1 FORWARD

Given the functions f_l^k we can construct using the method given in [K] a function F(x) mapping p to p which generates the new inferences resulting from a single parallel execution of every step of the program. F(x) is the inner product using set union and function application of the matrix F (f_{mj} is f_l^j where edge m leaves from node l) and x. Given a safe prediction s, we can generate a new prediction $F_s(x) = s \cap F(x)$. Starting execution of the program with every variable undefined ($x = \emptyset$) we generate final predictions under the constraint of known information s by applying $x := F_s(x)$ until no change results ($\Phi_s(x)$) (least fixed point of F_s). Kaplan & Ullman [K] prove that this procedure will terminate.

3.3.2 BACKWARDS

In a similar fashion, given the functions b_l^k we can construct a function B(x) mapping r to r which generates the new inferences resulting from a single parallel execution of every step of the program. B(x) is the inner product using set intersection (not union) and function application of the matrix B (b_{mj} is f_l^j where edge m enters node l) and x. Given a sufficient requirement s, we can generate a new requirement $B_s(x) = s \cap B(x)$. Starting execution of the program with every variable undefined ($x = \bar{1}$ Note: when the program starts executing we have established no requirements) we generate final predictions under the constraint of known information s by applying $x := B_s(x)$ until no change results ($\Psi_s(x)$).

4. - TYPE PREDICTION ALGORITHM

The steps of the algorithm for generating type predictions and identifying necessary run-time checks are:

1) Translate the real program into the model programming language.

2) Initialize all type predictions and requirements to .

3) Apply (set intersection) all inferences generated within a single node.

4) Apply any information available from declarations.

5) If, after steps 3 and 4, there exist predictions or requirements which have not been fully specified, generate new property variables to represent the eventual value for each such location. (In systems such as that described in [M] which compile at first execution, the values of some property variables may be available from inspection of the values of the actual variables.) Propagation of these place holders to other locations will permit identification of the locations for run-time checks and the point at which information needed to compile parts of the program is available.

6) Apply the functions Φ and Ψ to the program. Since forward and backwards propagation do not interact, only one application of each is necessary.

7) Simplify the set expressions representing type predictions and requirements, and assign values to property variables. The property variables connect the requirements and predictions which were propagated separately.

5. - INTERPRETATION OF TYPE INFORMATION

The distinction made between predictions and requirements allows the identification of necessary run-time checks and of type errors.

5.1 - TYPE INCOMPATIBILITY

A type incompatibility at an operation of the real program will be revealed at the assignment node of the model corresponding to that operation by:

1) The predicted type of the variable assigned to after execution of the node contains the empty set as the possible values of some measure.

2) The required type of one of the operand variables intersected with its predicted value before execution of the node contains the empty set as the possible values of some measure.

5.2 - TYPE CHECKING

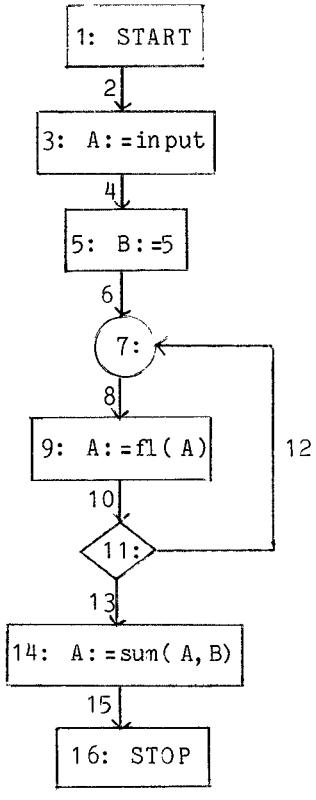
Correct execution of the model program requires that at each edge and for each variable the actual value be a member of the required type. If the type prediction (making use of all information known about the value of property variables) does not guarantee this, run-time type checking is necessary. The test can be specified as a requirement on the value of a property variable. The location(s) at which the test is required is the location(s) at which the property variable appears and which dominates (as defined by in [H]) the node generating the requirement.

6. - EXAMPLE

We now present a simple program in the model programming language and apply our algorithm to it. The example chosen is the same as that used in an earlier paper [K] presenting a type prediction algorithm, and we will compare our results to theirs.

6.'1 - THE PROGRAM

We analyze the following program:



where the variables A and B may be of type Real, Integer (a subset of Real), or Character and the properties of functions f1 and sum are defined by:

sum(x,y)

x\y	Real	Integer	Character
Real	Real	Real	Error
Integer	Real	Integer	Error
Character	Error	Error	Character

x	f1(x)
Real	Integer
Integer	Integer
Character	Character

We also demonstrate the effect of introducing type incompatibility into the example program by changing the definition of the function sum to be:

sum(x,y)

x\y	Real	Integer	Character
Real	Error	Error	Error
Integer	Error	Error	Error
Character	Error	Error	Character

6.'2 - TYPE PREDICTION

The type inferences generated within statements and the place holder variables are:

$P_4(A) = Z'_1$

A_1 is $Z_1 \neq \emptyset$

$P_6(B) = \text{Integer}$

$P_{10}(A) = Z_2$

$R_8(A) = Z_2 \cup \text{Real}$

A_2 is $Z_2 \cap (\text{Real} - \text{Integer}) = \emptyset$

$P_{15}(A) = Z_3 \cup Z_4$

$R_{13}(A) = Z_3$

$R_{13}(B) = Z_4$

A_3 is A_4 is $Z_3 \cap Z_4 \neq \emptyset$

The across statement inference functions are given by:

$TF_{14}(A)$ and $TF_{14}(B)$:

x	TF(x)
Real	Real
Integer	Real
Character	Character

$TB_{14}(A)$ and $TB_{14}(B)$

x	TB(x)
Real	Real
Integer	Integer
Character	Character

$TF_9(A)$:

x	TF(x)
Real	Integer
Integer	Character
Character	Integer
	Integer
	Character

$TB_9(A)$:

x	TB(x)
Real	Real
Integer	Real
Character	Character

$TF_3(A) = TB_3(B) = TB_5(A) = 1$

$TF_5(B) = \text{Integer}$

Applying the above information yields:

edge	P(A)	R(A)
2	$\emptyset$	1
4	Z_1	$Z_2 \cup$ Real
6	Z_1	$Z_2 \cup$ Real
6	Z_1	$Z_2 \cup$ Real
8	(Integer $\cup$ Character) $\cap$ Z_2	($Z_2 \cup$ Real)
10	(Integer $\cup$ Character) $\cap$ Z_2	($Z_2 \cup$ Real) $\cap Z^3$
12	(Integer $\cup$ Character) $\cap$ Z_2	$Z_2 \cup$ Real
13	(Integer $\cup$ Character) $\cap$ Z_2	Z_3
15	$Z_3 \cup Z_4$	1

edge	P(B)	R(B)
2	$\emptyset$	1
4	$\emptyset$	1
6	Integer	Z_4
8	Integer	Z_4
10	Integer	Z_4
12	Integer	Z_4
13	Integer	Z_4
15	Integer	1

Type compatibility together with the requirements A_K generate the following values for property variables:

Z_2 is Integer

Z_3 is Real

Z_4 is Integer

using these values we obtain:

edge	P(A)	R(A)
2	$\emptyset$	1
4	Z_4	Real
6	Z_4	Real
8	Z_4	Real
10	Integer	Real
12	Integer	Real
13	Integer	Real
15	Integer	1

edge	P(B)	R(B)
2	$\emptyset$	1
4	$\emptyset$	1
6	Integer	Real
8	Integer	Real
10	Integer	Real
12	Integer	Real
13	Integer	Real
15	Integer	1

One run-time check is required to verify that A is of type Real at edge 4.

When our algorithm is applied to the erroneous version of this program, the results are:

edge	P(A)	R(A)
2		
4	Z_4	Character
6	Z_4	Character
8	Z_4	Character
10	Character	Character
12	Character	Character
13	Character	Character
15	Character	

edge	P(B)	R(B)
2		
4		
6	Integer	Character
8	Integer	Character
10	Integer	Character
12	Integer	Character
13	Integer	Character
15	Integer	

A type conflict exists because of the assignment made in node 5.

5.3 - PREVIOUS WORK

The work of Kaplan & Ullman [K] presented an algorithm for making type predictions. It assumes a correctly executing program, and does not make the distinction between requirements and predictions. We feel (as shown in the example) that there exist circumstances in which useful information is developed by our algorithm and not by theirs.

The Kaplan & Ullman algorithm indicates that:

loc.	A	B
8	Real	Integer
13	Integer	Integer

There is no indication that the type Integer assigned to the variable B is a prediction which will always be true, but that the types assigned to variable A in positions 8 & 13 are requirements which must be verified. Further, there is no indication that satisfying the requirement at 8 makes checking at position 13 unnecessary.

In the case of the errouneous program, the Kaplan & Ullman algorithm will predict:

loc.	A	B
8	Error	Error
13	Error	Error

which does not indicate the source of the error.

7. - REFERENCES

[B] Bauer, A.M., Saal, H. J., "Does APL Really Need Run-time Checking", Software-Practice and Experience, 4()129-138, (1974).

[H] Hecht, M. S., Flow Analysis of Computer Programs, Elsevier North Holland, New York, (1977).

[K] Kaplan, M. A. and J. D. Ullman, "A General Scheme for the Automatic Inference of Variable Types", Fifth ACM Symposium on Principles of Programming Languages, 1978.

[M] Miller, T. C., "Tentative Compilation: A Design for an APL Compiler", Ph.D thesis, Dept. of Computer Science, Yale University, 1978.

[P] conversations with several Pascal compiler implementers, Workshop on Systems Programming Extensions to Pascal, Institute for Information Systems, University of California, San Diego, 1978.