Correctly Detecting Intrinsic Type Errors in Typeless Languages Such as MATLAB[1]

Pramod G. Joisha and Prithviraj Banerjee
Center for Parallel and Distributed Computing
Electrical and Computer Engineering Department
Technological Institute
2145 Sheridan Road, Northwestern University, IL 60208--3118.
Phone: (847) 467-4610, Fax: (847) 491-4455
Email: [pjoisha, banerjee]@ece.nwu.edu

Abstract

Among the main impediments that languages such as MATLAB and APL present to a compiler is the lack of an explicit declaration for a variable's type. The determination of this important attribute could allow a compiler to generate more efficient code, and is a problem that has been extensively studied in the past. This paper revisits this problem but unlike prior efforts, the objective is a uniform approach to type estimation that also accommodates type incorrect programs in a way that facilitates stronger type error detection through the exact localization of the type error at run time. We also show how our methodology makes it possible to further reduce the run-time overhead due to type conformability checking. The techniques are clearly demonstrated by applying them to deduce the intrinsic types of program variables in the MATLAB language.

1 Introduction

If a compiler could predict the range of values and shapes that program variables take on during execution, simpler and more efficient code could be generated to carry out the operations in the program. For instance, if the statement[2] $c \leftarrow a + b$ occurred somewhere in a MATLAB[3] program and if the compiler could establish that both a and b are always bound to scalar-shaped integer values in every execution of this statement, then a simple machine instruction that adds the values of a and b could be generated, rather than an invocation to a generalized array addition subroutine. Such an inference would also be beneficial storage-wise because it would permit the compiler to statically allocate a single word for the result c instead of having to overestimate c as being, say a double-word result, or having to bind c to some dynamically allocated storage. Thus, in programming languages such as MATLAB and APL that lack declarations, inferring the attribute of type is clearly desirable from the viewpoint of producing an efficient translation of the source.

This paper presents a scheme to automatically infer the types of program variables in dynamically typed languages such as MATLAB and APL. The presentation centers on the MATLAB language, which was primarily chosen on account of the immense (and still growing) popularity that it enjoys in the programming community.

The difference between our work and previous efforts is its ability to *exactly localize* a type error, when one does occur at run time. By "exactly localizing" a type error, we mean allowing program execution to continue *successfully* until the occurrence of the type error, at which point execution is terminated. Consider

APL01, 06/01, New Haven CT
©2001 ACM 1-58113-419-3 / 00/0008 $5.00

[1] This research was supported by DARPA under Contract F30602--98--2--0144.

[2] The symbol $\leftarrow$ will be used to denote an assignment operation.

[3] MATLAB is a registered trademark of The MathWorks, Inc.

a program P whose type correctness cannot be established statically. Previous approaches will enable static type estimates for all the variables in the program that are bound to be honored at run time along every type-correct execution path in P. However, if a particular type-incorrect execution path is exercised, it is not clear how the static inferences will affect program execution. Will a type error be detected, and if so, does the detection exactly localize the type error in the sense stated earlier? The capability to localize a type error is desirable because it would allow the program to execute exactly as written and thereby facilitate stronger debugging support. This capability would also benefit system-critical software, where premature code abortion on some input data may be an ill-advised option. The focus of this work is to investigate the additions necessary to previously proposed lattice-based techniques so as to encompass type incorrect programs and to also permit the precise localization of run-time type errors. In addition, we also show how our methods enable a reduction in the type conformability checking overhead through the exploitation of the monotonicity property of the type functions involved, and through the application of well-known compiler optimizations such as loop peeling.

The rest of this paper is organized as follows. In section 2, we describe previous research that forms the basis of our work. The necessary extensions to handle type errors correctly, along with a description and an informal justification of the modified type determination process, are provided in section 3. This is followed in section 4 by an actual application of the techniques to the problem of intrinsic type estimation in MATLAB. The fact that previous approaches lack the ability to precisely localize type errors, even if they do manage to detect one, is shown in section 5. We then discuss how the overhead due to type conformability checking can be reduced by our scheme in section 6. Other research efforts in the area of type determination are mentioned in section 7. Finally, we conclude the paper in section 8.

2 Lattices for Type Determination

An oft-cited work in the area of type estimation is that due to Kaplan and Ullman [9]. They proposed a mathematical framework based on the theory of lattices to automatically infer the types of variables in a model of computation that abstracted programming languages such as APL, SETL and SNOBOL. Their framework postulates the existence of a lattice [15] T of types, chosen by the compiler designer. The objective is to determine statically a type that subsumes the actual run-time type *as tightly as possible*. A type t is said to "subsume" a type s if all data representable by s is also representable by t. A case in point is the COMPLEX type in the MATLAB language, which has a value range that numerically contains the value range of the REAL type in the language. The motivation behind using a lattice is that a type lower down in the lattice hierarchy can be safely replaced by a type higher up without compromising program correctness[4]. The need to move down the lattice is because elements higher up tend to be more expensive, in terms of execution time and memory costs, than elements lower down. For example, the COMPLEX type typically requires at least twice the execution time and storage space as the REAL type in MATLAB. By making the inference as close as possible to the actual run-time type, the approach attempts to optimize on the program's operational and storage requirements.

2.1 Overview of the earlier framework

Formally, if s and t are two types in a lattice of types T, the join of s and t, denoted by $s \vee t$, represents the "smallest" type that subsumes both s and t. By this definition, any other type that subsumes s and t also subsumes $s \vee t$. The meet of s and t, denoted by $s \wedge t$, is the "largest" type whose value range is contained within the value ranges of both s and t.

Type determination in the framework begins at the level of assignment statements. Consider an assignment statement

$$Z \leftarrow \Theta(X_1, X_2, \ldots, X_k);$$

where Z and $X_i (1 \le i \le k)$ are program variables, and where Θ denotes a k-ary language operator. From the semantics of Θ, it may be possible to determine the type of Z, at least conservatively, given only the types t_i of the operands X_i. In such cases, the type semantics of Θ could be modeled by a *type function* T_Θ that maps the set T^k into the set T. Type functions such as T_Θ are often characterized as *forward type functions* because they indicate what the type of the result should be, given the types of the arguments. For

[4] Phrases such as "higher up" and "lower down" are with reference to the underlying partial order. For instance, a type t is said to be higher up in the lattice with respect to a type s if $s < t$.

example, if we were to consider the array addition built-in function in MATLAB (in infix notation, this language operator is designated by the symbol +), adding two operands of type INTEGER and REAL yields a result whose type is "at most" REAL. We can thus write T_+ (INTEGER, REAL) = REAL.

It is also possible that, knowing something about the type of an operator's result and the types of its operands, something more may be deducible about the types of its operands. For instance, if we knew that the type of c in the MATLAB statement $c \leftarrow a + b$ was REAL after the assignment, and if the previous best estimates for the types of a and b before the assignment were COMPLEX and REAL respectively, then the new best estimate for the type of a before the assignment becomes REAL. We thus write T_+^1(REAL, COMPLEX, REAL) = REAL, where the notation $T_\Theta^j(t, t_1, \ldots, t_k)$ represents the new type estimate for X_j before the assignment in $Z \leftarrow \Theta(X_1, X_2, \ldots, X_k)$, given that t is the type of Z after the assignment and that the t_i s are the respective previous best type estimates for the X_i s before the assignment. Type functions such as T_Θ^v that model these semantics in the backward direction are usually referred to as *backward type functions*.

Program-wide type determination in [9] consists of a series of steps on the program's control-flow graph (CFG) [12]. In each step, a forward or backward type inference, using the forward and backward type functions, is performed on a node in this flow graph. A sequence of such steps that spans all the nodes in the CFG constitutes a forward or backward pass in the type determination process. The procedure begins with initial conservative solutions for each variable's type; these then get successively refined with every iteration of a forward or backward type inference pass. It has been shown that for lattices in which the *finite chain condition* holds, a finite number of such iterations produces a safe solution that cannot be improved upon further, and that is in some sense an optimal solution to the type determination problem [9].

2.2 Requirements

Crucial to the success of the Kaplan and Ullman approach is the satisfaction of two conditions:

Postulate 1. **Finite Chain Condition**

There are no infinite sequences of elements in T that are related by the lattice's partial order $\leq$. This condition is necessary for the initially chosen conservative solution to converge to a fixed point in a finite number of forward or backward type inference steps.

Postulate 2. **Monotonicity Condition**

The type functions are *monotonic* with respect to the defined lattice T. For the forward type function T_Θ, monotonicity implies that

$$T_\Theta(t_1, t_2, \ldots, t_k) \leq T_\Theta(t_1', t_2', \ldots, t_k'),$$

if $t_i \leq t_i'$ for all $1 \leq i \leq k$

Intuitively, monotonicity for the forward type function means that the more we know about the types of the X_i s, the more we will know about the type of $\Theta(X_1, X_2, \ldots, X_k)$.

The above two conditions are sufficient in that if a lattice of types can be defined that satisfies them, then such a lattice will be a suitable device for the purposes of type estimation.

3 Detection of Type Errors

In the framework presented in [9], the important issue of type incorrect programs was not explicitly dealt with. That is, programs were considered to be type correct to start with, so that the greatest element in the type lattice, which was assumed to comprise only "legal" types, formed a suitable initial solution for the type determination problem. Though the framework in [9] could be used on general programs by assuming them to be type correct and by then inserting code that enforces the static type estimates during program execution, such an approach is assured to work fine only if the program turns out to be type correct at run time. If the program turns out to be type incorrect, it is not clear whether the static inferences will continue to hold, and if they do not, whether the enforcing code will allow program execution to proceed successfully until the point of occurrence of the type error. The extensions described in this section are meant to address this important issue.

3.1 Legal and Illegal Types

The first consideration should be the ability to differentiate between type correct and type incorrect

programs. We do so by regarding two separate sets of types, L_T and I_T, which represent legal and "illegal" types respectively. A legal type is basically the type that a variable can assume after a successful assignment at program execution time. The term type is used here in the most general sense — it represents a *collection* of data that have been organized together for reasons of logical similarity or mere convenience. Illegal types, on the other hand, are abstractions meant to signify type errors. Since the partial order that underlies a type lattice indicates a "subsumes" relationship, legal types should not be comparable by it to illegal types. The reason for this is that relationships such as $l \leq i$ or $i \leq l$, where l denotes a legal type and i denotes an illegal type, would contradict program correctness as we go up the lattice. Consequently, a *bounded* type lattice[5] that is serviceable even in the presence of type incorrect programs should have the general form shown in Figure 1.

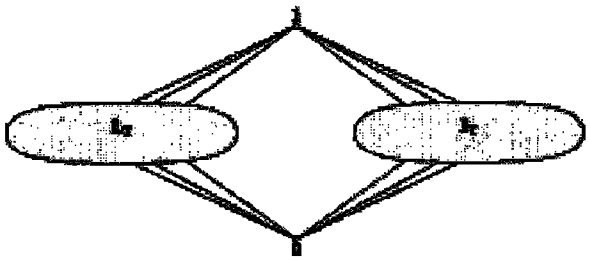

Figure 1: Layout of a Type Lattice Usable in the Presence of Type Incorrect Programs

In Figure 1, the shaded regions represent lattice points in the legal and illegal type sets. The greatest and least elements of the lattice, namely **1** and **0**, do not depict a particular legal or illegal type and are needed only to form a lattice out of L_T and I_T. In a set-theoretic sense, the most general type **1** could be thought of as $L_T \cup I_T$, while the most restrictive type **0** could be thought of as the empty set. In terms of an interpretation, **0** could be regarded as the type of a program variable that is yet to be assigned a value.

Without loss of generality, we can suppose that there exists a single lattice point in L_T , say h , that subsumes every other legal type in Figure 1. This is because, if such an element does not exist, the lattice can always be extended through the introduction of a new legal type h that subsumes all the *maximal* legal types in L_T. (We call a legal type l maximal if there exists no other legal type l' such that $l < l'$.) However, since h may not be mappable to any useful machine representation that permits the efficient manipulation of objects belonging to that type by a computer, we assume the following:

Postulate 3. **A Mappable Largest Legal Type**

The largest among the legal types in L_T enjoys an efficient machine representation.

Though the requirement of a mappable largest legal type is not essential, its fulfillment would be helpful to a compiler writer since it would allow the generation of efficient code whenever the result of an inference is a legal type. This is because even if a particular inferred legal type lacks an efficient machine representation, another legal type that is higher up in the lattice and that does have an efficient machine representation can always be used in its place. Such a substitution is valid because program correctness is preserved as we move up the lattice. What Postulate 3 does is to make the prospect of finding such a legal type a surety. This is not to say that a type lattice in which Postulate 3 does not hold cannot be used for type estimation. In fact, in [3], a lattice of APL legal intrinsic types in which the greatest element does not have an efficient machine representation is used. In such cases, if the static inference is the greatest legal type, the emitted code may have to rely on run-time resolution to carry out the associated operation. At run time, a switch-like construct selects the correct operation to invoke since by then, the exact types of the operands will be known. The difference is of course between a few machine instructions that operate directly on machine-representable data objects — such as integer or floating-point numbers — and the execution of conditional code, along with the maintenance of additional bookkeeping information, to achieve run-time resolution.

3.2 Type assertions

A problem that the lattice in Figure 1 poses to a compiler writer is that a type inference is useful so long as it is **0** or falls onto any of the lattice points in L_T and I_T . This is because a legal type inference could be used to produce an efficient translation of the source, while an illegal type inference could be used to flag program error. However, if the result of a static type inference is **1**, then nothing useful is gained since we are back where we started: that a program variable's type could be either legal or illegal. The possibility of **1** being the result of a type inference cannot be ruled out since without any advance knowledge on program correctness, the process of

[5] Bounded lattices are lattices that have both a greatest and a least element [15]. It is easy to show that any lattice that satisfies Postulate 1 must be bounded.

type estimation would begin with 1 as the initial conservative solution. In fact, in dynamically typed languages such as MATLAB and APL, type correctness may be specific to an execution instance. The question therefore boils down to what needs to be done in order to get around this lack of complete static information in a reasonably efficient manner.

Our solution to this problem involves preceding every assignment statement of the form

$$S: Z \leftarrow \Theta(X_1, X_2, \ldots, X_k);$$

in the original code fragment by a *type assertion:*

$$S': assert(T_\Theta(\bar{\tau}_{s'}(X_1), \bar{\tau}_{s'}(X_2), \ldots \bar{\tau}_{s'}(X_k)) \leq A);$$

$$S'': Z \leftarrow \Theta(X_1, X_2, \ldots, X_k)$$

The function call in statement S' tests at run time whether the condition $T_\Theta(\bar{\tau}_{s'}(X_1), \bar{\tau}_{s'}(X_2), \ldots, \bar{\tau}_{s'}(X_k)) \leq A$ is true; if so, execution is allowed to continue and if not, execution is terminated. The notation $\bar{\tau}_{s'}(X_i)$ stands for the most restrictive type of the program variable X_i at the particular point S' in the program and can be established at run time by inspecting X_i when control is at that point in the program[6]. No changes have to be made to the framework in [9] to handle the inserted *assert* calls since they can be regarded as "dummy" assignments in which the assigned variables are not subsequently used. Other aspects of the framework, such as its treatment of confluence points in the CFG, remain unchanged.

Conceptually, type assertions enable us to assume that the reaching types for every program variable are legal at every point in the program. Thus, instead of 1, we can begin with A as the initial type solution. Physically, a type assertion is the type conformability code of a language operator, in-lined just before its invocation at the call-site. By decoupling an operator's type checking semantics this way, it may be possible to lessen the overall overhead due to type conformability checking (see section 6).

3.2.1 Forward type inference

For the sake of simplicity, we assume that every node in the control-flow graph consists of either a type assertion or an assignment statement. Let $\tau_p(V)$ be a *static estimate* of the type of a program variable V *immediately after* the node p is executed in this flow graph. Let $\rho_q(V)$ be a static estimate of the type of a program variable V when control reaches the node q. If p is the only predecessor of q in the CFG, then $\tau_p(V)$ would be an acceptable pick for $\rho_q(V)$. However, a particular node could have more than one predecessor. To cover such situations, we determine $\rho_q(V)$ to be the maximum of the type estimates at all the predecessor nodes p:

$$\rho_q(V) = \bigvee_{\text{all predecessor nodes } p} \tau_p(V) \qquad (1)$$

Now suppose that the flow graph node q contains the type check $assert(T_\Theta(\bar{\tau}_q(X_1), \bar{\tau}_q(X_2), \ldots, \bar{\tau}_q(X_k)) \leq A)$. We can make a forward type inference through the assertion to arrive at static type estimates $\bar{\tau}_q(V)$ for every V at the node. This is because the *assert* invocation basically establishes constraints on the types of the X_is *after the call*, so that the type of $\Theta(X_1, X_2, \ldots, X_k)$ in the following statement is always legal Thus, such a forward type inference should employ the backward type function . $T^{\nu}_{\Theta}(A, \ldots)$ since new type estimates for the X_is that satisfy the assertion would have to be determined. We arrive at these estimates conservatively:

$$\bar{\tau}_q(V) = \begin{cases} \rho_q(V) & \text{if } V \neq X_j \text{ for all } 1 \leq j \leq k, \\ \bigwedge_{\substack{\text{all } j \text{ such} \\ \text{that } V = X_j}} T^{v}_{\Theta}(A, \mu_1(X_1), \rho_q(X_2), \ldots, \rho_q(X_k)) & \text{otherwise.} \end{cases} \qquad (2)$$

Correctness. We need to show that $\bar{\tau}_q(V) \leq \tau_q(V)$ will hold true at node q, if $\bar{\tau}_p(V) \leq \tau_p(V)$ holds true at each predecessor node p. We show this by first demonstrating that $\bar{\tau}_q(V) \leq \rho_q(V)$ will be true for all V at q. Because control will always reach the node q from one of its predecessor nodes p, the most restrictive type $\bar{\tau}_q(V)$ at q can at most be the maximum of the most restrictive types at each of the predecessors p:

$$\bar{\tau}(V) \leq \bigvee_{\substack{\text{all predecessor} \\ \text{nodes } p}} \bar{\tau}(V) \qquad (3)$$

[6] The most restrictive type is the exact type of the datum in question. It could be at the granularity of the type system at hand, or even lower. For example, the most restrictive intrinsic type for the real number 0:1 could be REAL in the MATLAB type system, or the type "real numbers that lie between 0 and 1." It could also be the type of a *particular* real number (i.e., 0:1).

We are also given that $\bar{\tau}_p(V) \leq \tau_p(V)$ is true at each predecessor node p. Therefore, from the monotonicity of the join operator $\vee$

$$\bigvee_{\substack{\text{all predecessor}\\ \text{nodes } p}} \bar{\tau}_p(V) \leq \bigvee_{\substack{\text{all predecessor}\\ \text{nodes } p}} \tau_p(V)$$

From Equation (1), the last constraint becomes

$$\bigvee_{\substack{\text{all predecessor}\\ \text{nodes } p}} \bar{\tau}_p(V) \leq p_q(V) \qquad (4)$$

Inequalities (3) and (4) thus lead to

$$\bar{\tau}_q(V) \leq p_q(V). \qquad (5)$$

We now need to consider the two scenarios that arise from V appearing or not appearing in the type assertion. If V does not appear in the *assert* call, the claim $\bar{\tau}_q(V) \leq \tau_q(V)$ immediately follows from Inequality (5) and Equation (2). Therefore, suppose that V does appear in the *assert* call. From the semantics of the call, an estimate for $\bar{\tau}(V)$ after the assertion would be

$$\bigwedge_{\substack{\text{all } j \text{ such}\\ \text{that } V=X_j}} T^j_\Theta(k, \bar{\tau}_q(X_2), \ldots, \bar{\tau}_q(X_k)).$$

However, from the definition of $\bar{\tau}_q(V)$, the following should hold after the assertion:

$$\bar{\tau}(V) \leq \bigwedge_{\substack{\text{all } j \text{ such}\\ \text{that } V=X_j}} T^j_\Theta(k, \bar{\tau}_q(X_1), \bar{\tau}_q(X_2), \ldots, \bar{\tau}_q(X_k)). \qquad (6)$$

From Inequality (5), we also have $\bar{\tau}_q(X_1) \leq p_q(X_i)$ for all $1 \leq i \leq k$. Hence, from the monotonicities of $T^j_\Theta(k, \ldots)$ and the meet operator $\wedge$, we obtain

$$\bigwedge_{\substack{\text{all } j \text{ such}\\ \text{that } V=X_j}} T^j_\Theta(k, \bar{\tau}_q(X_1), \bar{\tau}_q(X_2), \ldots, \bar{\tau}_q(X_k)) \leq$$

$$\bigwedge_{\substack{\text{all } j \text{ such}\\ \text{that } V=X_j}} T^j_\Theta(k, p_q(X_1), p_q(X_2), \ldots, p_q(X_k)).$$

Therefore, from Inequality (6) and Equation (2), we get $\bar{\tau}_q(V) \leq \tau_q(V)$.

If the flow graph node q contains an assignment statement, the forward type inference is straightforward and is shown below:

$$\tau_q(V) = \begin{cases} T_\Theta(p_q(X_1), p_q(X_2), \ldots, p_q(X_k)) & \text{if } V = Z, \\ p_q(V) & \text{otherwise.} \end{cases} \qquad (7)$$

Correctness. Once again, we need to show that $\bar{\tau}_q(V) \leq \tau_q(V)$ will hold true at node q, if $\bar{\tau}_p(V) \leq \tau_p(V)$ holds true at each predecessor node p. As before, we can show that Inequality (5) is satisfied at node p. Therefore, if V is not, Z, $\bar{\tau}_q(V) \leq \tau_q(V)$ trivially follows from Equation (7). If V is Z, then from the definition of $\bar{\tau}_q(V)$,

$$\bar{\tau}_q(V) \leq T_\Theta(\bar{\tau}_q(X_1), \bar{\tau}_q(X_2), \ldots, \bar{\tau}_q(X_k)). \qquad (8)$$

Since $\bar{\tau}_q(X_i) \leq p_q(X_i)$ for all $1 \leq i \leq k$, we have from the monotonicity of T_Θ,

$$T_\Theta(\bar{\tau}_q(X_1), \bar{\tau}_q(X_2), \ldots, \bar{\tau}_q(X_k)) \leq T_\Theta(p_q(X_1), p_q(X_2), \ldots, p_q(X_k)).$$

Therefore, from Inequality (8) and Equation (7), we can conclude that $\bar{\tau}_q(V) \leq \tau_q(V)$.

3.2.2 Backward type inference

For the assignment statement $Z \leftarrow \Theta(X_1, X_2, \ldots, X_k)$, a backward type inference arises by considering the context in which Z is *subsequently* used. However, because every such use is preceded by an *assert* call, no useful inferences about the type of Z can be made in the backward direction. The reason for this is that a type check of the form $assert(T_\Theta(\ldots, \bar{\tau}_q(Z), \ldots) \leq k)$ only makes guarantees regarding the type of Z after the call, not before. If those guarantees are not met at run time, execution halts at the assertion.

```
R : a ←...
S : b ←ι a
```

Figure 2. An APL Code Fragment

```
R  : a ←...
S" :▷ Assert that the type of a
      is a scalar nonnegative integer.
S" : b ←ι a
```

Figure 3: An APL code fragment with embedded-Type Checking Code

As an example, consider the APL code fragment shown in Figure 2. By noting that the ' primitive expects a scalar nonnegative integer as its argument, we can deduce that the value of a computed in the first line must be of this type. However, such an

inference is valid only if the code fragment is given to be type correct. Even if the type checking code associated with statement S were considered explicitly as shown in Figure 3, we still cannot hope to deduce useful type information, simply because the assertion may not hold immediately after statement R. That is, a may not be a scalar nonnegative integer between statements R and S' in Figure 3. Thus, if a program is not given to be type correct to start with, no useful type information can be derived via a backward inference. To arrive at a type solution, we must therefore rely on only forward type inferences.

3.2.3 Program-wide type inference

Program-wide type determination begins with conservative type estimates $\tau_r(V)$ for every program variable V at every node r of the CFG. Because of the embedded type assertions, Δ is a legitimate starting estimate. The procedure then performs a forward type inference *pass* by computing type estimates at every node using the type estimates at the predecessor nodes, as discussed in section 3.2.1. It does not matter how the forward pass traverses the CFG to accomplish this type estimation because the type estimates at every node will always be safe. That is, if the forward pass does not process a node, the old type estimates at the node, which are given to be safe, will remain in place, while if a node is visited, the new type estimates computed at that node will be such that they continue to be safe. Therefore, a way in which the forward pass could be implemented is by traversing the CFG in a *depth-first* or *breadth-first* fashion, computing the type estimates at each node along the way. However, revisiting a node may provide opportunities for further refinements in the type estimates.

Because of the finite chain condition and the monotonicity of the type functions involved, this improvement will not occur indefinitely and will eventually reach a stable state. Thus, program-wide type determination could be performed by repeatedly applying the forward pass on the CFG until a fixed-point type estimate is achieved at each node. Program-wide type estimates for the variables may then be determined by taking a join of the respective fixed-point nodal type estimates.

3.3 Type error localization

Consider the assignment statement

$$S: Z \leftarrow \Theta(X_1, X_2, \ldots, X_k);$$

in the original code fragment and the corresponding statement pair

$$S': assert(T_\Theta(\tau_{r'}(X_1), \tau_{r'}(X_2), \ldots, \tau_{r'}(X_k)) \leq \Delta);$$

$$S'': Z \leftarrow \Theta(X_1, X_2, \ldots, X_k);$$

in the translated version. Suppose that control is currently at statement S in the original code fragment. The interpreter would successfully execute this statement if and only if $T_\Theta(\tau_s(X_1), \tau_s(X_2), \ldots, \tau_s(X_k)) \leq \Delta$. If it does execute successfully, then the associated type assertion S' in the translated version would also execute successfully and control would reach the statement S''. The latter statement would then execute correctly because the type of Z in S'' was statically estimated to be at least as large as $\tau_s(Z)$.. If $T_\Theta(\tau_s(X_1), \tau_s(X_2), \ldots, \tau_s(X_k)) \not\leq \Delta$, then a type error would crop up at statement S in the original code fragment while the corresponding type assertion in the translated version would fail. Since this argument applies to every statement in the original code fragment, we thus observe that the translated version will faithfully reproduce the original code's execution behavior.

A key point to note is that the procedure will produce legal static type estimates even when the program is always (i.e., under all execution paths) type-incorrect. The embedded assertions will however intercept the type error at run time, allowing the program to execute successfully until the point of occurrence of the type error. In certain situations, it may be possible to statically verify that a particular assertion will never hold at run time. As an example, this could happen when the assertion involves the types of program constants. In such cases, the type error could be flagged at compile time, giving an opportunity for the programmer to take immediate remedial action.

4 A MATLAB Example

To demonstrate the lattice-based type determination techniques discussed so far, we consider the problem of inferring the intrinsic variable types in a MATLAB program. In particular, we shall examine the simple, contrived MATLAB code fragment shown in Figure 4. The code excerpt consists of a while loop within which reaching values for the program variables a and b are added by using the array addition built-in function. The result is assigned to the program variable c, which is then operated in the next two statements by the colon built-in function. We assume that c and d are not live before the loop.

The construction $m : n$ produces a row vector of elements that form an arithmetic progression with a common difference of 1. The starting and ending values of this progression are m' and $m'+1\lfloor n'-m' \rfloor$, where m' and n' represent the run-time *real values* of m and n respectively. (The construction $1 : c$ is therefore similar to the APL construction ιc.

```
while (...),
        c ← a + b;
        a ← 2 : c;
        d ← 1 : c;
end;
```

Figure 4. A Simple MATLAB Code Fragment

In MATLAB, the notion of logical similarity in the context of types consists of a structural aspect (e.g., a 2X3 matrix) and an arithmetic aspect (e.g., integers, reals, complexes and so on). These two aspects are independent of each other and can be analyzed in isolation; we call the former the attribute of *shape* and the latter the attribute of *intrinsic type*. The discussion in this section will concern itself with the problem of intrinsic type estimation in MATLAB, though the techniques are also applicable to shape estimation using lattices. An implementation could use the methodology of this section to arrive at intrinsic type estimates for the program variables and to translate the source into a version that has the necessary intrinsic type checks; the translation could then be subjected to a separate shape inferring phase that gathers the requisite shape information and then inserts the needed shape checks. When lattices are used for shape determination, rough estimates of shape, such as whether a program variable is a scalar, row vector, column vector, matrix or an "arbitrary" array, can be arrived at in the same way intrinsic types were inferred. These shape estimates may then be used like intrinsic type estimates to emit code with the necessary assertions. However, the shape semantics of language operators in MATLAB and APL often impose requirements at the granularity of array extents. For example, the matrix multiply built-in function in MATLAB (denoted by the infix operator *) expects at least one operand to be a scalar, or both to be matrices such that the extent along the second dimension of the first operand matches the extent along the first dimension of the second operand [10]. Any other combination of shapes produces a run-time error. Figuring out what an array's extents actually are is advantageous because it may sanction important optimizations such as run-time shape conformability check reduction, memory preallocation and scalarization [8]. Alternate approaches that make this possible and that are based on a shape calculus have been discussed in [8].

4.1 The MATLAB Intrinsic Type Lattice

For the given code fragment, an appropriate way of arranging the intrinsic types of MATLAB into a lattice T is shown in Figure 5, where 0 is the least element of the lattice and 1 is its greatest element. The BOOLEAN intrinsic type stands for a 1-bit value — namely, 0 or 1. The UINT8 intrinsic type represents unsigned integer bytes and is primarily intended for image processing applications [11]. The INTEGER, REAL and COMPLEX intrinsic types denote integers, real numbers and complex numbers respectively. The artificial intrinsic type NONREAL designates strictly complex numbers — that is, those with nonzero imaginary parts. The lattice point i indicates the illegal intrinsic type, which, as remarked earlier, is an abstraction meant to express an ill-formed MATLAB expression. For instance, the colon built-in function expects the intrinsic types of both of its operands to be at most REAL; a NONREAL intrinsic type causes the run-time system to complain[1]. On comparing Figures 5 and 1, we observe that BOOLEAN, UINT8, INTEGER, REAL, NONREAL and COMPLEX correspond to the legal intrinsic types. The ascendancy from BOOLEAN to COMPLEX, and from NONREAL to COMPLEX reflects the inclusion of a legal intrinsic type's value range in the value range of the legal intrinsic type above it.

4.2 Intrinsic Type Functions

Our first task is to cast MATLAB's intrinsic type semantics for the array addition and colon built-in functions into the relevant type functions. The resulting forward intrinsic type functions are shown in Tables 1 and 2. In these tables, we use B, U, I, R, N and C as abbreviations for BOOLEAN, UINT8, INTEGER, REAL, NONREAL and COMPLEX respectively.

[7] An invocation such as $1 : 2 + 3i$, where i is the imaginary unit, elicits an alert (**'Warning: COLON arguments must be real scalars.'**)from the MATLAB run-time system.

An important point to note in these tables is that only the intrinsic type semantics are captured without any attention to shape. That is, Tables 1 and 2 tell us how the intrinsic types map, assuming that the arguments are shape conforming. The formulations ensure that the illegal intrinsic type i is either propagated, or generated if the arguments are not intrinsic-type conforming. Technically, the decision to propagate the illegal intrinsic type is one of convenience because reaching types will never be i due to the preceding type assertions. Generation of the illegal intrinsic type is however necessary so as to allow the assertions to detect type errors at run time. When one of the operands to the intrinsic type functions is 0, we have some leeway in choosing the image of the map; the choices shown in Tables 1 and 2 are such that monotonicity is secured. Observe also that whenever one (or both) of the operands to the intrinsic type functions is 1, the outcome is also usually 1 since nothing can be said about the intrinsic type of the result except that it could also be either legal or illegal. When both operands to either of the intrinsic type functions are COMPLEX or lower, we can usually do better.

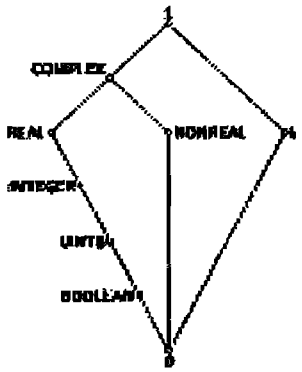

Figure 5. The Lattice $\mathcal{T}$ of Intrinsic Types in MATLAB

	0	B	U	I	R	N	C	i	1
0	0	0	0	0	0	0	0	i	i
B	0	U	I	I	R	N	C	i	1
U	0	I	I	I	R	N	C	i	1
I	0	I	I	I	R	N	C	i	1
R	0	R	R	R	R	N	C	i	1
N	0	N	N	N	N	C	C	i	1
C	0	C	C	C	C	C	C	i	1
i	i	i	i	i	i	i	i	i	i
1	i	1	1	1	1	1	1	i	1

Table 1. $\mathcal{T}_+(s,t)$

	0	B	U	I	R	N	C	i	1
0	0	0		0	0	i	i	i	i
B	0	B	U	I	I	i	1	i	1
U	0	U	U	I	I	i	1	i	1
I	0	I	I	I	I	i	1	i	1
R	0	R	R	R	R	i	1	i	1
N	i	i	i	i	i	i	i	i	i
C		1	1	1	1	i	1	i	1
i	i	i	i	i	i	i	i	i	i
1	i	1	1	1	1	i	1	i	1

Table 2. $\mathcal{T}:(s,t)$

for example, $\mathcal{T}_+$(REAL;COMPLEX)=COMPLEX since when a real number is added to a complex number, the result is a complex number. In the interests of presentation clarity, we do not assume a particular bit width for the INTEGER, REAL and COMPLEX intrinsic types. Thus, adding two INTEGER quantities will always produce an INTEGER quantity as indicated in Table 1. However, because of the stated finite bit widths of the BOOLEAN and UINT8 intrinsic types, adding a BOOLEAN to a UINT8 could produce an overflow; therefore, a safe intrinsic type for the result in this case is INTEGER, as shown in Table 1.

We also need to consider the backward intrinsic type functions for the given built-in functions; these are shown in Tables 3 and 4. The backward intrinsic type function $\mathcal{T}_+^2$(COMPLEX, s,t) can be obtained from Table 3 because of the relation

$$\mathcal{T}_+^2(\text{COMPLEX},s,t) = \mathcal{T}_+^1(\text{COMPLEX},t,s) \qquad (9)$$

Once again, we have some freedom in deciding the maps of the backward intrinsic type functions when one of the operands is i, since such a situation will never occur due to the reaching types being legal at every point in the program. In fact, the backward intrinsic-type maps have been so selected so as to either improve the previous legal estimate or produce COMPLEX or REAL as the new estimate.

Note that the lattice in Figure 5 satisfies Postulate 1. By inspection, the backward intrinsic type functions, along with the forward ones in Tables 1 and 2, can be ascertained to be monotonic with respect to the lattice in Figure 5. Hence Postulate 2 is honored. Since the real and imaginary parts of the COMPLEX intrinsic type can be manipulated separately, COMPLEX has an efficient machine representation thereby enabling

	0	B	U	I	R	N	C	i	1
0	0	0	0	0	0	0	0	0	0
B	B	B	B	B	B	B	B	B	B
U	U	U	U	U	U	U	U	U	U
I	I	I	I	I	I	I	I	I	I
R	R	R	R	R	R	R	R	R	R
N	N	N	N	N	N	N	N	N	N
C	C	C	C	C	C	C	C	C	C
i	C	C	C	C	C	C	C	C	C
1	C	C	C	C	C	C	C	C	C

Table 3. $T_+^1(\text{COMPLEX}, s, t)$

	0	B	U	I	R	N	C	i	1
0	0	B	U	I	R	R	R	R	R
B	0	B	U	I	R	R	R	R	R
U	0	B	U	I	R	R	R	R	R
I	0	B	U	I	R	R	R	R	R
R	0	B	U	I	R	R	R	R	R
N	0	B	U	I	R	R	R	R	R
C	0	B	U	I	R	R	R	R	R
i	0	B	U	I	R	R	R	R	R
1	0	B	U	I	R	R	R	R	R

Table 4. $T_+^2(\text{COMPLEX}, s, t)$

Postulate 3 to hold. We can therefore apply the techniques of the previous section to arrive at useful intrinsic type inferences for the code fragment in Figure 4, even if it is not given to be type correct.

4.3 Program-Wide Type Inference

Figure 6 shows the control-flow graph for the given code excerpt. Execution is assumed to begin at the "start" node *S* and end at the "finish" node *F*. Nodes 1" , 2" and 3" respectively correspond to the three assignment statements in Figure 4, while the associated type assertions, which presumably are automatically inserted by an interpreter or a compiler, respectively correspond to the nodes 1', 2' and 3'. We assume that the forward type inference pass operates by visiting the nodes in the CFG in a depth-first order; Figure 6 displays one such traversal using solid arrows.

The process begins with conservative estimates for the intrinsic types at every node of the flow graph. At the start node *S*, these are $\tau_s(a) = \tau_s(b) = \text{COMPLEX}$ and $\tau_s(c) = \tau_s(d) = 0$ because only a and b are live on entry into the code fragment. That is, whenever control flows through node *S*, we can always be assured that the relations $\bar{\tau}_s(a) \leq \text{COMPLEX}$, $\bar{\tau}_s(b) \leq \text{COMPLEX}$, $\bar{\tau}_s(c) = 0$, and $\bar{\tau}_s(d) = 0$ prevail. However, at the remaining nodes, the most restrictive intrinsic type for each program variable can be any legal value. Therefore, we select COMPLEX as the initial safe estimate at these nodes. All of the initial estimates are exhibited in Figure 6 to the right of the flow graph. Note that UINT8 and BOOLEAN were used as the most restrictive intrinsic types for the program constants 2 and 1 in the assertions at nodes 2' and 3'.

At the right of Figure 6 are shown two applications of the forward type inference pass. A particular step in an application of the pass computes the estimates $\tau_q(V)$ for every program variable V at a particular node q in the CFG. Depending on whether the node contains a type assertion or an assignment statement, the computations happen in accordance with Equation (2) or Equation (7). The reaching estimates $p_q(V)$ are used in these computations and have been calculated using Equation (1); for reasons of brevity these calculations have not been explicitly displayed in the figure. To illustrate one such calculation, the following are the reaching intrinsic type estimates at node *F* on the first application of the forward pass:

$p_F(a) = \text{COMPLEX} \vee \text{INTEGER} = \text{COMPLEX}$,

$p_F(b) = \text{COMPLEX} \vee \text{COMPLEX} = \text{COMPLEX}$,

$p_F(c) = 0 \vee \text{REAL} = \text{REAL}$,

$p_F(d) = 0 \vee \text{INTEGER} = \text{INTEGER}$.

On the second application of the forward pass, we arrive at fixed-point intrinsic type estimates at every node of the CFG; this can be verified by applying the pass for the third time. By taking the join of the respective estimates at all of the CFG nodes, we can arrive at the program-wide intrinsic type estimate of COMPLEX for a, b and c, and a program-wide intrinsic type estimate of INTEGER for d. Given no prior knowledge regarding the type correctness of the original code fragment, these are also the best possible estimates. It should be mentioned here that the

solution obtained is to some extent dependent on the initially chosen estimates at node *S*. For instance, Figure 7 shows the nodal estimates when a different starting solution is picked at *S*. We note that a fixed-point solution is achieved after the first application of the pass and that the nodal estimates of the fixed-point solution are more conservative than that in Figure 6; in fact, COMPLEX becomes the program- wide intrinsic type estimate for all the program variables in Figure 7. Thus, selecting the best initial estimates at node *S* is crucial to arriving at good program-wide type estimates for all the variables.

5 Comparisons

If the code fragment in Figure 4 were type correct, then the most restrictive intrinsic type for c would be REAL. Through the use of a backward type inference, the framework in [9] would determine this. Thus, if compilation were done *assuming* the excerpt to be type correct, the translation would declare c to be REAL. However, such a translation will not correctly localize a type error, when one does occur at run time. For example, if the generated code were then executed with the values $1+2i$ and $1+3i$ for a and b respectively, the first statement in the code fragment will either execute wrongly (because the intrinsic type of c will not be "large enough" to accommodate the expected result $2+5i$) or will not be executed at all (due to preceding type checking code anticipating the type error). In either case, the translated version will not execute

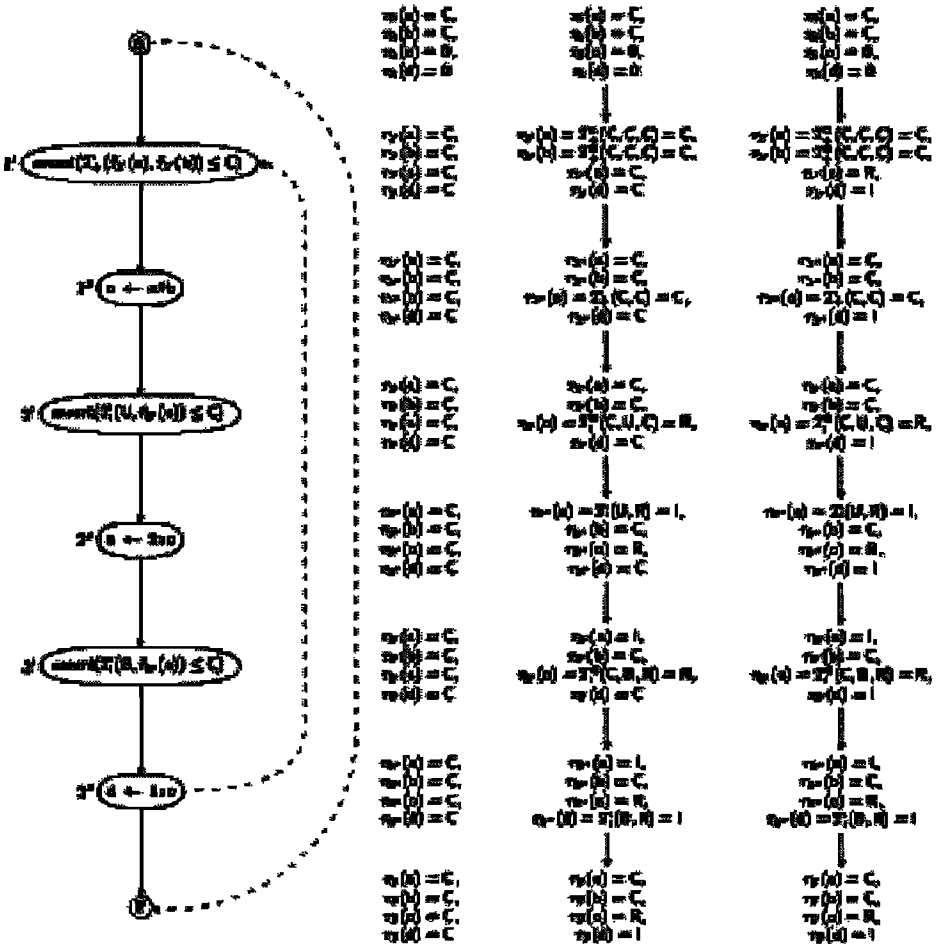

Figure 6. Estimating the Intrinsic Types Using a Depth-First Traversal on the Control-Flow Graph

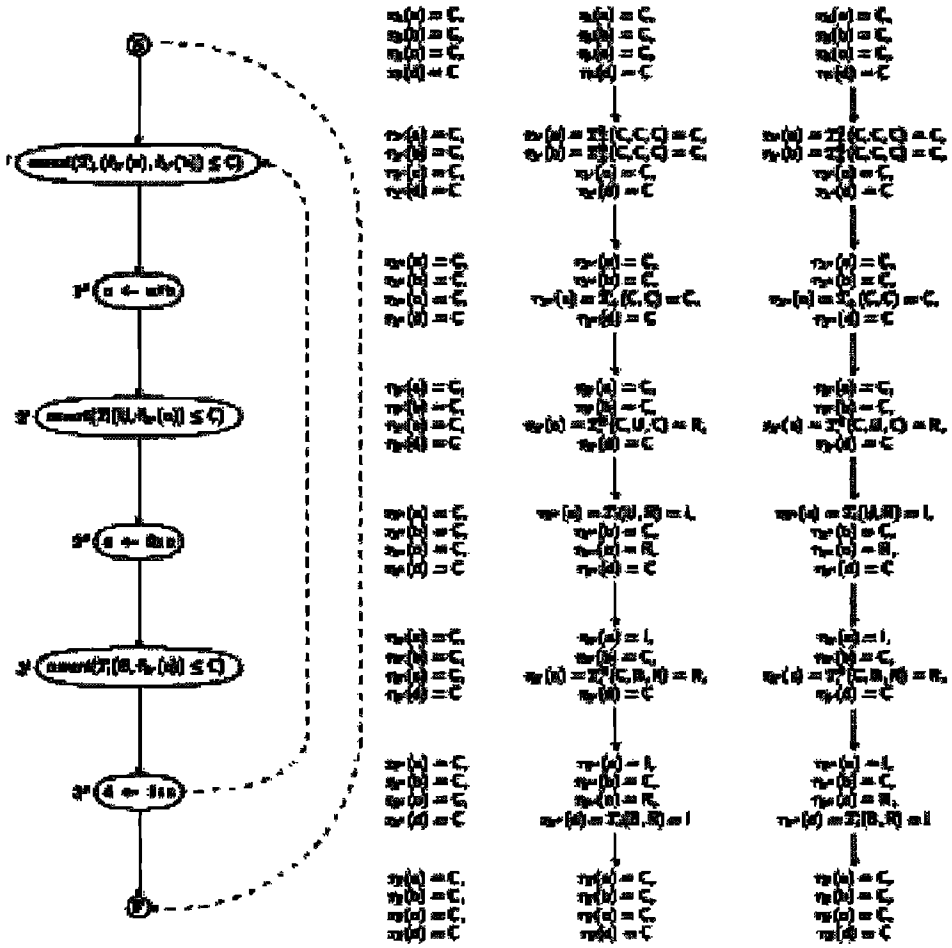

Figure 7. Intrinsic Type Estimation with a Different Starting Solution at S

exactly as the original code fragment.[8] However, if code were generated using our scheme, execution is guaranteed to proceed successfully until the type assertion that immediately precedes the second line in the fragment, thus allowing the type error to be correctly isolated.

6 Lessening the Run-Time Type Checking Overhead

Some of the type assertions may be verifiable at compile time. For instance, consider the type assertions generated for the example in Figure 4; these are reprised in Figure 8 and marked by the ▷ prefix. When control reaches the first assertion, $\bar{\tau}(a) \leq$ COMPLEX and $\bar{\tau}(b) \leq$ COMPLEX are guaranteed to be true. Since T_+(COMPLEX,COMPLEX)=COMPLEX from Table 1, we can conclude from the monotonic property that the first assertion will always hold. Additionally, since $T_:$(BOOLEAN,$\bar{\tau}(c)$) $\leq T_:$ (UINT8,$\bar{\tau}(c)$) from monotonicity, success at the second assertion implies success at the third assertion. Eliminating the redundant assertions will then produce the equivalent code in Figure 9. Next, from Figures 6 and 7, the variable a will be of type INTEGER in the statement $c \leftarrow a + b$ from the second iteration on. Therefore, the single assertion in Figure 9 need not be executed beyond the second iteration because if a type error does occur, it will occur within the first two iterations. We can therefore apply the loop peeling transformation [12] to remove the first two iterations

[8] If the code fragment in Figure 4 were run through the MATLAB interpreter with a and b set to $1+2i$ and $1+3i$ respectively, and if the first iteration of the loop were executed, the run-time system will execute the assignment to c without a hitch but will complain when attempting to compute $2 : c$ in the next statement.

of the loop into separate code. The result, shown in Figure 10, produces a loop body that is free of type checks. Notice that though the type checking overhead has been pared down to a minimal, a type error, when one does occur, will still be correctly localized in Figure 10. The code in Figure 10 can now be subjected to a shape determination phase to gather the necessary shape information for further optimizations of the translated code [8]. It could also be executed by an interpreter as an optimized version of the source program in Figure 4.

```
while (test),
    ▷ assert(T+(τ̄(a),τ̄(b)) ≤ C)
    c ← a + b;
    ▷ assert(T:(U,τ̄(c)) ≤ C)
    a ← 2 : c;
    ▷ assert(T:(B,τ̄(c)) ≤ C)
    d ← 1 : c;
end;
```

Figure 8. With Embedded Type Assertions

```
while (test),
    c ← a + b;
    ▷ assert(T:(U,τ̄(c)) ≤ C)
    a ← 2 : c;
    d ← 1 : c;
end;
```

Figure 9. After Eliminating Redundant Type Assertions

```
if (test),
    c ← a + b;
    ▷ assert(T:(U,τ̄(c)) ≤ C)
    a ← 2 : c;
    d ← 1 : c;
end;
if (test),
    c ← a + b;
    ▷ assert(T:(U,τ̄(c)) ≤ C))
    a ← 2 : c;
    d ← 1 : c;
end;
while (test,
    c ← a + b;
    a ← 2 : c;
    d ← 1 : c;
end;
```

Figure 10. After Loop Peeling

7 Related Work

Previous attempts in the area of type estimation for languages such as MATLAB and APL aimed at determining conservative static estimates under the implicit assumption that the program was type correct [1, 9, 5, 3, 6, 13, 4]. In some of these approaches, a provision for type incorrectness was made by the argument that run-time type checking code would ultimately catch a type error when it did occur, though not necessarily at the point of occurrence (see section 5). It is in that respect that our work chiefly differs from these previous attempts. Work due to Bauer and Saal was among the earliest to recognize that a substantial percentage of the run- time type checks needed for APL could be avoided by static analysis [1]. They showed how even a simple type determination scheme could produce significant improvements in performance. The determination of types through the use of data-flow analysis was first reported in [14]. The work in [9] improved on this by providing more powerful algorithms for type detection. In [5], a type determination mechanism in a production APL compiler that emitted System/370 assembly code was described. The inferring occurred in a front-end compilation process and used a type calculus along with a global data-flow analyzer. However, the process

was limited to some extent because when the back-end compilation started, the system expected user intervention to resolve the type of any variable that was assigned a general type during the front-end compilation process.

In the work due to Budd [3], a partial ordering of intrinsic type and shape was used in the type determination process. Data-flow techniques were then applied to propagate type information across expressions, statements and procedures. A point of pragmatic interest in the context of [3], and that was alluded to in section 3.1, is that the greatest element in the intrinsic type lattice used did not have an efficient machine representation. This impacted the quality of the generated code when the static inference was the greatest element. The observation that backward type inferences can be used with only limited success in real codes was also pointed out in [3].

In the area of type determination for MATLAB, an important effort is the FALCON project [6, 7]. In the FALCON approach, a static inference mechanism attempts to deduce as much of the intrinsic type information as possible at compile time, and treats the intrinsic types of the remaining variables to be COMPLEX. Matters relating to how the compiled code performs in the presence of type errors were not dealt with. Techniques similar to those in [6] have been used in the Menhir project [4], and in other MATLAB compilers such as "Otter" [13]. In the case of Menhir, the system relies on user-provided annotations called directives when sufficient type information is lacking.

8 Summary

In this paper, we presented a scheme using which the types of program variables in typeless languages such as MATLAB and APL can be inferred. The unique advantage of our approach is its ability to also correctly handle type incorrect programs. In particular, our scheme provides a stronger type error detection support than previously proposed methods. In addition, we also showed how our approach may empower further reductions in the type conformability checking overhead. The described techniques are currently being integrated into the MATCH compiler, a translator that aims on converting MATLAB sources into code for embedded processors, DSPs and FPGAs [2].

References

[1] A. M. Bauer and H. J. Saal. "Does APL Really Need Run-Time Checking?". Software---Practice and Experience, 4:129--138, 1974.

[2] P. Banerjee, U. N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Chang, M. Haldar, P. G. Joisha, A. Jones, A. Kanhere, A. Nayak, S. Periyacheri, and M. Walkden. "A MATLAB Compiler for Configurable Computing Systems". Technical Report CPDC--TR--9906--013, Center for Parallel and Distributed Computing, Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208--3118, USA, September 1999.

[3] T. Budd. An APL Compiler. Springer-Verlag New York, Inc., New York City, NY 10010, USA, 1988. ISBN 0--387--96643--9.

[4] S. Chauveau and F. Bodin. "Menhir: An Environment for High Performance MATLAB". Lecture Notes in Computer Science, 1511:27--40, 1998. Proceedings of the 4th International Workshop on Languages, Compilers and Run-Time Systems, Pittsburgh, PA, USA, May 1998.

[5] W.-M. Ching. "Program Analysis and Code Generation in an APL/370 Compiler". IBM Journal of Research and Development, 30(6):594--602, November 1986.

[6] L. A. De Rose. "Compiler Techniques for MATLAB Programs". Ph.D. dissertation, University of Illinois at Urbana-Champaign, Department of Computer Science, May 1996.

[7]http://www.csrd.uiuc.edu/falcon/falcon.html, The FALCON Project Home Page.

[8] P. G. Joisha, U. N. Shenoy, and P. Banerjee. "An Approach to Array Shape Determination in MATLAB". Technical Report CPDC--TR--2000--10--010, Center for Parallel and Distributed Computing, Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208--3118, USA, October 2000.

[9] M. A. Kaplan and J. D. Ullman. "A Scheme for the Automatic Inference of Variable Types". Journal of the ACM, 27(1):128--145, January 1980.

[10] The MathWorks, Inc., 24 Prime Park Way, Natick, MA 01760--1500, USA. MATLAB---The Language of Technical Computing, January 1997. Using MATLAB (Version 5).

[11] The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760--2098, USA. Image Processing Toolbox: For Use with MATLAB, September 2000. User's Guide (Version 2).

[12] S. S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, Inc., San Francisco, CA 94104, USA, 1997. ISBN 1--55860--320--4.

[13] M. J. Quinn, A. Malishevsky, N. Seelam, and Y. Zhao. "Preliminary Results from a Parallel MATLAB Compiler". In the Proceedings of the 12th International Parallel Processing Symposium, pages 81--87, Orlando, FL, USA, April 1998.

[14] A. M. Tenenbaum. "Type Determination in Very High-Level Languages". Ph.D. dissertation, Courant Institute of Mathematical Sciences, New York University, Department of Computer Science, October 1974. Computer Science Report NSO--3.

[15] J. P. Tremblay and R. Manohar. Discrete Mathematical Structures with Applications to Computer Science. Computer Science Series. McGraw-Hill, Inc., New York City, NY 10121, USA, 1975. ISBN 0--07-- 065142--6.