Static Type Inference for Specialization in a Telescoping Compiler

Cheryl McCosh
Rice University

Eric Allen
Sun Microsystems

Ken Kennedy
Rice University

Walid Taha
Rice University

Abstract

The telescoping languages approach achieves high performance from applications encoded as high-level scripts. The core idea is to pre-compile underlying libraries to generate multiple variants optimized for use in different possible contexts including different argument types.

This paper proposes a type inference algorithm that enables this kind of specialization. The algorithm infers types over untyped library procedures before actual inputs are known. For our MATLAB compiler, the notion of type captures matrix properties such as size, sparsity pattern, and data type. Type inference is necessary both to determine the minimum number of variants needed to handle all possible uses of the library procedure as well as to statically determine, for each variant, which optimized implementations should be dispatched at each call location.

A key contribution that arose from this work is a notion of *mutually exclusive types*. To illustrate these types, we formalize the underlying type system, constraint-collection, and solution. Finally, we prove that the algorithm is polynomial under practical conditions.

1 Introduction

The productivity of the scientific computing community could be dramatically improved if it were possible for application developers to produce high-performance code by integrating components in high-level, domain-specific scripting languages such as MATLAB(TM). To this end, we have developed a framework, called telescoping languages, that supports high-level programming while achieving application performance comparable to code written in lower-level languages such as Fortran or C [21]. The framework, shown in Figure 1, accomplishes this by preprocessing the libraries that define the language primitives to produce efficient variants

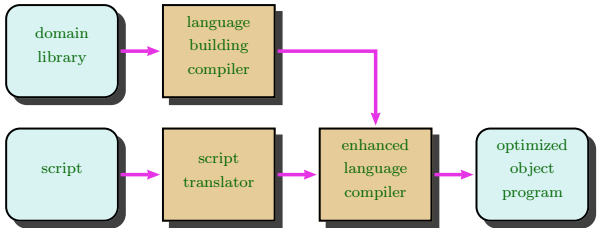

Figure 1: Overview of the telescoping languages approach.

specialized for specific calling contexts. The compiler for user-developed scripts can then replace calls to the general library subroutines with calls to the appropriate variants given the specific calling contexts.

The telescoping-language strategy can also be applied to the task of library generation and maintenance. Library writers can develop and maintain a single version of their code in a high-level language. The library compiler, which we call LibGen, is then responsible for achieving high performance for the multiple possible uses of the library. It generates several Fortran or C variants specialized for the different uses of the library procedure. We have previously demonstrated the effectiveness of this strategy by showing that Fortran variants generated by LibGen from a single MATLAB prototype script had comparable performance to the hand-coded variants in the ARPACK library [5]. The specializations performed were based on information about matrix properties, including size and element pattern, as well intrinsic types. This paper develops the type inference algorithm used to achieve these results.

We use MATLAB as our example language due to its popularity within the scientific community.[1] MATLAB allows scientists to develop algorithms without regard to low-level details. Because MATLAB is weakly-typed, scientists can use a single script for multiple problem

[1] Mathworks quoted the number of licenses after 2001 to be 500,000

domains. Unfortunately, MATLAB does not achieve the performance needed for large-scale scientific applications.

1.1 Problem Statement

Type inference provides information useful for specialization, and is necessary to translate MATLAB into more efficient languages such as Fortran or C. However, pre-compiling libraries, when the calling contexts are unknown, has traditionally led to limited specialization opportunities. Telescoping languages addresses this problem by generating several variants for a given script, each variant intended for a different possible configuration of types over the arguments. Therefore, telescoping languages achieves the benefit of specializing for actual input types without having to wait until the calling routines are available. In order to avoid generating superfluous variants, LibGen must infer which type configurations over the inputs are valid.

The type inference algorithm we present in this paper serves two purposes. First, it determines the minimum number of type configurations that could be legal in practice, since we are inferring types over a dynamically-typed language. Second, type inference is used to statically determine for each variant, which optimized implementations should be dispatched at each call site.

The type inference we use allows us to constrain the types of the variables at each point based on how the variables are defined and used throughout the procedure. This contrasts with traditional type inference systems that evaluate the types at each expression. Type inference thus gives the information necessary to avoid dispatching to semantically invalid variants at the call sites.

1.2 Contributions

This paper defines and formalizes our solution to type inference in telescoping languages. Type inference over a procedure produces a *type jump-function*, which lists every possible type configuration over the variables in terms of the input types. In order to represent this information, we introduce a new notion of types that we term *mutually exclusive types*. These are types over functions that list possible type configurations over the arguments. They allow us to express properties such as, one variable may be scalar only if the input is non-scalar and conversly, that are necessary to accurately infer types in a dynamically-typed language such as MATLAB.

1.3 Organization

We first discuss the type inference problem for telescoping languages using MATLAB as the scripting language in Section 2. We informally describe our solution in Section 3. We then formalize the type inference problem for a subset of MATLAB in Section 3 in order to give a deeper understanding of both the problem and solution. We then give a provably efficient implementation in Section 5. In Section 6 we show how the inference can be extended to handle a broader range of problems. We then discuss related work and conclude.

2 The Type Inference Problem

We first discuss telescoping languages, and describe what is required for the type inference solution. We then discuss MATLAB as an example scripting language for the libraries.

2.1 Type Inference for Telescoping Languages

The telescoping languages strategy requires type inference in order to generate specialized variants from MATLAB scripts. What is needed is a set of valid type configurations that assign types to each variable in the procedure in terms of the input types.

Note that telescoping languages does not distinguish between procedure calls and operations, since one of the goals of the library generation phase is for procedure calls to behave like primitive operations in a higher-level language, hence the name *telescoping languages*.

Type Jump-Functions The LibGen compiler has the burden of inferring types when specific calls to the procedure are not given. Type-inference must result in types defined in terms of the inputs so that for each possible configuration of types over the inputs, a variant is generated with the correct corresponding local types. To handle this, we define *type jump-functions* akin to those used in interprocedural analysis [3, 15]. We use a tabular representation of the type jump-function so that each entry represents one possible type configuration.

Return type-jump-functions, which define the types of the outputs in terms of the types of the inputs, handle the propagation of type information across procedure calls. We also call these mutually exclusive types.

Code Generation For the purposes of this paper, one variant is generated for each entry in the type jump-function table. We specialize each variant with respect to the types given in the entry by replacing each operation or procedure call with a call to a procedure from

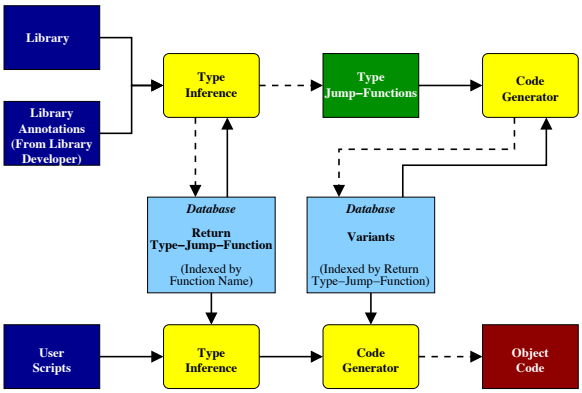

Figure 2: Type-Based specialization in a telescoping compiler.

the table of variants that is specialized for the given types.

Interprocedural Problem In order to infer types for a library procedure, LibGen must have return type-jump-functions for all called procedures. To ensure this, LibGen should perform type inference on the library procedures in reverse post-order on the call graph. If there is a cycle in the call graph, LibGen iterates over the cycle until a fixed point is reached. When source code is unavailable, as in the case of primitive operations, the return type-jump-functions may be entered by hand.

Putting It Together Figure 2 describes how these pieces work together in a telescoping compiler. For library generation, the input to the type inference engine is a library procedure and optional annotations that describe its possible uses. The type inference engine uses information about called procedures and operations, summarized in the return type-jump-function table, to determine the legal types of variables at every point in the procedure. This information is used to produce both a type jump-function and a return type-jump-function for the procedure. Code is generated using the type jump-function as well as a table of previously generated, specialized variants of called procedures.

Script compilation is similar to library generation, but information does not need to be stored back to the tables, since the scripts are only used for a single application. This paper will describe the type inference in terms of library generation. However, the same type inference strategy can also be used during script compilation.

2.2 Type Inference for MATLAB

To describe our type inference mechanism in the context of a concrete language, we consider a subset of MATLAB and ignore some of its rarely-used features such as dynamic evaluation of strings as code, object oriented features, and structures, although the last two can be handled by extensions to the type inference algorithm. This is the subset of the language most commonly used by scientific application developers.

MATLAB's simplicity makes it popular among programmers, as is evidenced by the large number of licenses. However, some of the very features that make MATLAB a desirable language for programming make it difficult for the compiler to translate to lower-level languages. Some of these features include:

1. MATLAB is dynamically typed. This makes inferring primitive types necessary to generate code in lower-level languages such as Fortran or C, both of which require explicit typing. One benefit to dynamic types is that the library developer can write a single script that can be applied to multiple contexts, without extra effort from the library developer. For example, ArnoldiC (the MATLAB prototype script for the core routine from ARPACK) works for both complex and floating-point as well as symmetric and non-symmetric inputs. For the Fortran ARPACK, separate versions are required.

2. Variables can change types in the middle of the program, including arrays growing in the middle of a loop. Inferring the maximum size of a loop would help avoid reallocating the array at every iteration.

3. Operators are heavily overloaded. For example, the $*$ operation can refer to both matrix-matrix multiplication and matrix scaling depending on whether an operand is scalar. Therefore, determining whether a variable is a scalar or not is important for correctness.

4. All variables are treated as arrays, including scalars, which are 1×1 arrays. Therefore, all variables have array properties that must be inferred.

These features not only make type inference essential for correctness and high-performace, they also make it difficult to statically determine the types of the variables. In many cases, the library writer may have intended multiple interpretations of the same MATLAB code. The overloaded operators and weak typing are, in fact, one of the reasons why MATLAB is a popular language in which to program. The compiler must account for all intended possibilities.

MATLAB has call-by-value semantics. Even matrices that are passed into a procedure are copied. The compiler must model the behavior in the generated variants.

Information about the type of a variable in a MATLAB procedure depends on:

1. the operation that *defines* the variable and
2. the operations that use the variable, since operations impose type restrictions on their inputs.

The first causes *forward propagation* of variable properties along the control flow, while the second causes *backward propagation*. The type-inference system must infer types in both directions for the tightest outcome.

We perform type inference over an intermediate representation of the MATLAB code in which all expressions have been expanded so that the results of each operation or function call are assigned to variables. Also, to handle the fact that a variable may change type in the middle of a procedure, we assume that the procedures have been converted to SSA form [9] so that each use of a variable refers to a single definition. Redefinition of a section of an array results in a new array, since this can cause a change in type. The arrays and variables will be re-merged during code generation if possible. Finally, we assume for simplicity that all global variables have been converted to inputs and outputs to the procedures.

Type Problems In order to carry out specialization based on variable types, we first define the variable properties that are of interest. In general, the properties should be such that they can be encoded in the target language and the compiler for that language can leverage the information for optimization. They should also reflect the power of the source language. Since MATLAB is an array-based language, we use the 4-tuple definition of a variable type based on work by deRose [11]. We define a type to be a tuple $\mathcal{T} = <\tau, \delta, \sigma, \pi>$ where,

- τ is the intrinsic type of the variable (e.g., integer, real, complex).
- δ is the upper bound on the number of dimensions for an array variable, also called the rank. A tighter bound can be reached when the type inference system determines that the variable has a size of 1 in one or more dimensions. δ will always be greater than or equal to 2.
- σ is a tuple showing the maximum size of an array variable in each possible dimension. $\sigma^A = <1, 1>$ means that A is a scalar.
- π is the "pattern" of an array variable (e.g., dense, triangular, symmetric, etc.).

This list can be extended as needed for other languages and other problems.

To describe our algorithm, we will focus on the size inference problem and describe how to extend the algorithm to the other type problems in Section 6.

Existing Solutions Telescoping languages proposes pre-compiling libraries before the calling-context is known. Therefore, MATLAB type-inference systems such as FALCON and MaJIC will not suffice, since FALCON relies on inlining to exactly determine types, and MaJIC determines types, in part, during a just-in-time compilation step. Moreover, both systems rely on dataflow analysis that converges on a single type for each variable.

There are two main difficulties to using a dataflow analysis framework for our purposes.

1. It is difficult, if not impossible, to determine that the analysis halts for a given subroutine. This is partially due to the fact that information flows in both directions, but also, the lattices for some components of $\mathcal{T}$ do not meet all the requirements for provable termination.
2. The compiler must find all type solutions allowed by the procedure. Therefore, dataflow analysis is ill-suited for the problem, since if the analysis converges, it converges to a single solution (as in FALCON) or to a set of types. In the first case, there is not enough specialization opportunity. In the second case, the compiler generates more variants than necessary for calling contexts that would never occur. For example, one variable may prove to be complex only if the input is real. Therefore, a variant would not be needed for the case where the input is complex and the variable is complex.

3 Type Inference Solution for Size Problem

We developed an alternative to the dataflow solution that performs analysis over the procedure as a whole rather than iteratively over the control flow. The compiler determines the information that each individual operation or procedure call gives about the types of the variables involved and then combines that information over the entire procedure to find types. Thus, forward and backward inference occur simultaneously.

3.1 Statement Information

The information from the operations is given in the form of propositional constraints on the types of variables involved in the statement. MATLAB operations, and typical library procedures, are heavily overloaded

```
o1 = i1 * i2

σ^o1 =<1,1>   ∧ σ^i1 =<1,1>   ∧ σ^i2 =<1,1>   XOR
σ^o1 =<$1,$2> ∧ σ^i1 =<1,1>   ∧ σ^i2 =<$1,$2> XOR
σ^o1 =<$1,$2> ∧ σ^i1 =<$1,$2> ∧ σ^i2 =<1,1>   XOR
σ^o1 =<$1,$3> ∧ σ^i1 =<$1,$2> ∧ σ^i2 =<$2,$3> XOR
σ^o1 =<1,1>   ∧ σ^i1 =<1,$1>  ∧ σ^i2 =<$1,1>
```

Figure 3: Example of a return type-jump-function for the size inference problem on the MATLAB "*" operator. Each clause gives a possible size configuration over the sizes of the variable. The first clause states that the operation could be scalar multiplication. This second and third clauses state that "*" could be a scaling operation. The fourth clause shows the operation could be matrix-matrix or matrix-vector multiplication. The last clause gives the possibility that the operation is the multiplication of two vectors. This last case is necessary to keep track of the fact that $o1$ may be scalar.

based on the types of the inputs. Therefore, statement constraints needs to allow all possible valid type configurations.

The constraints are formed using a database of return type-jump-functions containing one entry per procedure or operation. Figure 3 shows an example of the return type-jump-function for the size problem on the MATLAB multiplication operation,"*". The possible type configurations in the return type-jump-function are composed through logical disjunction and are called *clauses*. These clauses are defined to be mutually exclusive, thereby imposing the property that each clause represents a distinct type configuration.

The $-variables are simply place holders for integer values representing sizes. Since each $-variable can be used for multiple variable sizes within a single statement constraint, they capture the size relationships between the variables in a single operation. The fields in σ can be defined to be linear expressions over the $-variables. Thus far, these expressions are sufficient to represent the size relationships between the variables. The constraints also keep track of whether a variable is a scalar or an array. For each clause, if a $-variable or any constant not equal to 1 appears in a size tuple, it is assumed that the corresponding variable cannot be a scalar for that clause to hold, although some $-variables may evaluate to 1. This is necessary to maintain the property that the clauses are mutually exclusive.

The constraint formed from this return type-jump-function at a particular application site is identical to the return type-jump-function except that the variables are replaced by the actual input and output parameters. Since the statement constraints should be formed in isolation from each other, $-variables in a statement constraint should be interpreted as though they are existentially quantified. Rather than including binding constructs, we ensure that $-variables are not shared accross statement constraints. In other words, the $-variables are replaced by $-variables that have not been used before in any other statement's constraints.

$-variables may be replaced by constants in a particular statement constraint, when it can be determined that the matrix size is constant in a particular dimension at that statement. This can cause a reduction in the number of clauses in the statement constraint, since the constant may guarantee that certain clauses can never hold. The size of a matrix may also depend on a procedure variable. These procedure variables are treated like constants by the compiler. However, unlike the case of constants, since the compiler may not be able to determine that a program variable is never 1, the clauses stating the possibility that the size may be 1 must be maintained.

3.2 Combining Statement Information

	config A	config B	config C
σ^{A_1}	<1,1>	<1,1>	<$1,$1>
σ^{v_1}	<1,1>	<$1,1>	<$1,1>
σ^{k_1}	<1,1>	<1,1>	<1,1>
σ^{v_2}	<1,1>	<$1,1>	<$1,1>
σ^{w_1}	<1,1>	<$1,1>	<$1,1>
σ^{α_1}	<1,1>	<1,1>	<1,1>
σ^{f_1}	<1,1>	<$1,1>	<$1,1>
σ^{c_1}	<1,1>	<1,1>	<1,1>
σ^{f_2}	<1,1>	<$1,1>	<$1,1>
σ^{α_2}	<1,1>	<1,1>	<1,1>
σ^{V_1}	<1,>	<$1,>	<$1,>
σ^{f_3}	<$1,$1>	<$1,$1>	<$1,$1>
σ^{β_1}	<1,1>	<1,1>	<1,1>
σ^{v_3}	<$1,1>	<$1,1>	<$1,1>
σ^{V_2}	<$1,>	<$1,>	<$1,>
σ^{w_2}	<$1,1>	<$1,1>	<$1,1>
σ^{h_1}	<j,1>	<j,1>	<j,1>
σ^{f_4}	<$1,1>	<$1,1>	<$1,1>
σ^{c_2}	<j,1>	<j,1>	<j,1>
σ^{f_5}	<$1,1>	<$1,1>	<$1,1>
σ^{h_2}	<j,1>	<j,$1>	<j,1>

```
function[V, H, f] =
  ArnoldiC(A_1, k_1, v_1);
v_2 = v_1/norm(v_1);
w_1 = A_1 * v_2;
α_1 = v_2' * w_1;
tmp_1 = v_2 * α_1;
f_1 = w_1 − tmp_1;
c_1 = v_2' * f_1;
tmp_2 = v_2 * c_1;
f_2 = f_1 − tmp_2;
α_2 = α_1 + c_1;
V_1(:, 1) = v_2;
H_1(1, 1) = α_2;
for j = 2 : k_1,
  f_3 = φ(f_2, f_5);
  β_1 = norm(f_3);
  v_3 = f_3/β_1;
  H_2(j, j−1) = β_1;
  V_2(:, j) = v_3;
  w_2 = A_1 * v_3;
  h_1 = V_2(:, 1:j)' * w_2;
  tmp_3 = V_2(:, 1:j) * h_1;
  f_4 = w_2 − tmp_3;
  c_2 = V_2(:, 1:j)' * f_4;
  tmp_4 = V_2(:, 1:j) * c_2;
  f_5 = f_4 − tmp_4;
  h_2 = h_1 + c_2;
  H_3(1:j, j) = h_2;
end
```

Figure 4: The resulting type jump-function and the corresponding pruned SSA form of ArnoldiC.

In a valid procedure, the type of a variable must satisfy all the constraints imposed by all the operations that can feasibly execute in any run of the program. Therefore, the type jump-function can be determined by taking the conjunction of the statement constraints over the function, called the procedure constraint, and finding all possible type configurations of the variables that satisfy the resulting boolean expression. Solving typical propositional constraints of this form is NP-hard.[2] However, we show in Section 5 that under certain conditions that occur most frequently in practice, we can devise an efficient algorithm using the specific properties of the problem.

The type jump-function inferred by our type inference algorithm from the MATLAB function, ArnoldiC,[3] is shown in Figure 4. Each column represents a different possible type configuration for the SSA form of ArnoldiC.

In this example, the types of the variables can be exactly inferred from the type jump-function given the inputs. For example, if A_1 is non-scalar, then the last type-configuration should be used. This last configuration is, in fact, the only configuration intended by the library writers. An annotation stating that the input A is never scalar would have made the last type configuration the only result of the type inference. Also, the sizes of the variables will be known on input, when the size of A_1 and the value of k_1 are known, since all the sizes are written in terms of \$1, which can be inferred from the size of A_1, or j, which has a greatest possible value of k_1. Note that the entries in the type jump-function are mutually exclusive and jointly exhaustive, and they involve the fewest number of \$-variables possible, so that the fewest values need to be known for all the sizes to be known. Note that the type inference algorithm is powerful enough to infer that A_1 must be square if it is a matrix, as well as the fact that many of the variables are vectors.

Before we describe the implementation, we formalize the type inference problem and solution for sizes.

4 Formal Description

In order to describe the type inference problem and solution more precisely, we formalize the size problem over a subset of MATLAB, which we call CORE-MATLAB, consisting of only the features of interest to the type inference algorithm. We first describe the subset of MATLAB over which we infer types. We then give the type system for our problem and formalize the solution we informally described in the previous section. We show that our solution is sound and complete with respect to the type system.

[2]The well known 3-SAT problem can be reduced to this problem.

[3]This function is from the ARPACK development code.

Formal Args	$p \in Args$
Array Vars	$x \in X \geq Args$
Integer	$n, m, k, q, r, y, z \in Int$
Functions	$f \in F$
Proc Defs	$Defs ::= fn\ [p_0] := f(p_1, \ldots, p_n)P$
Proc Bodies	$P ::= stop \mid C; B$
Statements	$c \in C ::= X := E$
Exprs	$e \in E ::= V \mid X \mid X(E, E) \mid f(\overline{E}) \mid E + E \mid E * E$
Matrix	$v \in V ::= [k_{11}, \ldots, k_{1m}; \ldots; k_{n1}, \ldots, k_{nm}]$
Types	$t \in T ::= <s_1, s_2>$
Size Exprs	$s \in S ::= nd \mid n \mid d + s$
\$-Vars	$d \in D ::= \$1, \$2, \ldots$
Return TJF	$r \in R ::= R \text{ XOR } cl \mid \epsilon$
Clause	$cl \in CL ::= CL \wedge \sigma^x = t \mid \sigma^x = t$
RTJF Tables	$J : F \rightarrow R$ (contains return type-jump-functions for all called procedures).
FN Tables	$T : F \rightarrow Defs$(contains headers for all called procedures).
Environs	$\Gamma \in G ::= [] \mid x : t \wedge \Gamma$

$$\frac{t' = <\mu(s_1), \mu(s_2)> \quad t \neq <1,1> \Rightarrow t' \neq <1,1>}{\mu(t) = t'}[\text{TMAP}]$$

$$\frac{\{\mu(d_i) = s_i\}^{i \in \{1,\ldots,n\}}}{\mu(k_1 d_1 + \cdots + k_n d_n) = k_1 s_1 + \cdots + k_n s_n}[\text{SMAP}]$$

$$\frac{d \mapsto s \in \mu}{\mu(d) = s}[\text{DMAP}]$$

$$\frac{\begin{array}{c} T(f) = fn\ p_0 := f(p_1, \ldots, p_n) \\ \exists cl \in J(f)\ s.t. \\ \{cl(p_i) = t_i' \exists \mu\ s.t.\ t_i = \mu(t_i')\}^{i \in \{1,\ldots,n\}} \end{array}}{t_1 \cdots t_n \rightarrow t_0 \in Types(f)}[\text{TYPES}]$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t}[\text{VAR}] \quad \frac{\Gamma \vdash e_1 : <1,1> \quad \Gamma \vdash e_2 : <1,1>}{\Gamma \vdash x(e_1, e_2) : <1,1>}[\text{ELEM}]$$

$$\frac{}{\Gamma \vdash [k_{11}, \ldots, k_{1m}; \ldots; k_{n_n 1}, \ldots, k_{nm}] : <n, m>}[\text{VALUE}]$$

$$\frac{t_1 \cdots t_n \rightarrow t_0 \in Types(f) \quad \Gamma \vdash e_1 : t_1 \cdots \Gamma \vdash e_n : t_n}{\Gamma \vdash f(e_1, \ldots, e_n) : t_0}[\text{APP}]$$

$$\frac{\Gamma(x) = t \quad \Gamma \vdash e : t}{\Gamma \vdash x := e}[\text{STMT}] \quad \frac{\Gamma \vdash c \quad \Gamma \vdash P}{\Gamma \vdash c; P}[\text{PROC}]$$

$$\frac{}{\Gamma \vdash stop}[\text{STOP}]$$

Figure 5: Simple Type System for Size Problem

$$\text{Stores } \Sigma \in ST ::= [] \mid x := v, \Sigma$$
$$Sizeof([k_{11}, \ldots, k_{1m}; \ldots; k_{n1}, \ldots, k_{nm}]) = <n, m>$$
$$Add(v_1, v_2) = v, \text{ where } v \text{ is result of matrix addition on } v_1 \text{ and } v_2.$$
$$Mult(v_1, v_2) = v, \text{ where } v \text{ is result of matrix multiply on } v_1 \text{ and } v_2.$$

$$\frac{\{x := v\} \in \Sigma}{\Sigma, x \hookrightarrow v}[\textsc{Var}] \quad \frac{\{x := v\} \notin \Sigma}{\Sigma, x \hookrightarrow error}[\text{E-}\textsc{Var}]$$

$$\frac{\{\Sigma, e_i \hookrightarrow v_i\}^{i \in \{1,\cdots,n\}} \quad T(f) = fn\ [p_0] := f(p_1, \ldots, p_n)\ P \quad \Sigma, [p_1 \mapsto v_1, \ldots, p_n \mapsto v_n]P \hookrightarrow \Sigma' \quad \Sigma'(p_0) = v}{\Sigma, f(e_1, \cdots, e_n) \hookrightarrow v}[\textsc{App}]$$

$$\frac{T(f) = fn\ [p_0] := f(p_1, \ldots, p_m)\ P \quad m \neq n}{\Sigma, f(e_1, \cdots, e_n) \hookrightarrow error}[\text{E1-}\textsc{App}] \quad \frac{f \notin T}{\Sigma, f(e_1, \cdots, e_n) \hookrightarrow error}[\text{E2-}\textsc{App}]$$

$$\frac{\Sigma, e_1 \hookrightarrow v_1 \quad \Sigma, e_2 \hookrightarrow v_2 \quad \Sigma(x) = v \quad Sizeof(v_1) = <1,1> \quad Sizeof(v_2) = <1,1> \quad Sizeof(v) = <n_1, n_2> \quad v_1 \leq n_1 \quad v_2 \leq n_2}{\Sigma, x(e_1, e_2) \hookrightarrow v(v_1 v_2)}[\textsc{Elem}]$$

$$\frac{\Sigma, e_1 \hookrightarrow v_1 \quad \Sigma, e_2 \hookrightarrow v_2 \quad \{x := v\} \notin \Sigma \text{ or } Sizeof(v_1) \neq <1,1> \text{ or } Sizeof(v_2) \neq <1,1>}{\Sigma, x(e_1, e_2) \hookrightarrow error}[\text{E-}\textsc{Elem}]$$

$$\frac{\Sigma, e_1 \hookrightarrow v_1 \quad \Sigma, e_2 \hookrightarrow v_2 \quad Sizeof(v_1) = Sizeof(v_2)}{\Sigma, e_1 + e_2 \hookrightarrow Add(v_1, v_2)}[\textsc{Add}]$$

$$\frac{\Sigma, e_1 \hookrightarrow v_1 \quad \Sigma, e_2 \hookrightarrow v_2 \quad Sizeof(v_1) = <n_1, n_2>, Sizeof(v_2) = <m_1, m_2> \quad n_2 = m_1}{\Sigma, e_1 * e_2 \hookrightarrow Mult(v_1, v_2)}[\textsc{Mult}]$$

$$\frac{\Sigma, e_1 \hookrightarrow v_1 \quad \Sigma, e_2 \hookrightarrow v_2 \quad Sizeof(v_1) \neq Sizeof(v_2)}{\Sigma, e_1 + e_2 \hookrightarrow error}[\text{E-}\textsc{Add}]$$

$$\frac{\Sigma, e_1 \hookrightarrow v_1 \quad \Sigma, e_2 \hookrightarrow v_2 \quad Sizeof(v_1) = <n_1, n_2>, Sizeof(v_2) = <m_1, m_2> \quad n_2 \neq m_1}{\Sigma, e_1 + e_2 \hookrightarrow error}[\text{E-}\textsc{Mult}]$$

$$\frac{\Sigma, c \hookrightarrow \Sigma' \quad \Sigma', P \hookrightarrow \Sigma''}{\Sigma, c; P \hookrightarrow \Sigma''}[\textsc{Proc}] \quad \frac{}{\Sigma, stop \hookrightarrow \Sigma}[\textsc{Stop}] \quad \frac{\Sigma, e \hookrightarrow v}{\Sigma, x := e \hookrightarrow \Sigma + \{x := v\}}[\textsc{Stmt}]$$

Figure 6: CORE-MATLAB Operational Semantics

4.1 CORE-MATLAB

Many of the MATLAB features that are trivially handled by the type inference algorithm are omitted in CORE-MATLAB for brevity. For example, in CORE-MATLAB, we assume only a single output to any function call or operation and that subscripted array accesses only access single elements of an array. We also assume all numbers are integers, since we have isolated the type inference to the size problem, and will not infer primitive types, which would be necessary to properly handle non-integer numbers. Similarly, CORE-MATLAB only allows two-dimensional matrices, since we are not formalizing the dimension problem.

A more difficult feature of MATLAB, which we handle in our implementation, but do not include in CORE-MATLAB is support for complex control flow. Similarly, CORE-MATLAB does not support recursion

We use":=" to represent assignment in the formalization to distinguish from equality, although this is not MATLAB syntax.

CORE-MATLAB does not support direct assignment to a section of an array. We choose not to support this because we are operating on an SSA form that creates a new variable when ever a section of an array is assigned and copies the new array into the new variable. In the actual implementation, we handle assigning to array sections directly.

Type System Figure 5 describes the CORE-MATLAB syntax along with a simple type system for the size inference problem in telescoping languages. A procedure consists of a header and a fragment of code. Of primary interest is the *Principal*, which is a set of Γ's, where a Γ is a type environment corresponding to a single entry in the type jump-function table for a procedure, P.

$Princ(P) = \{\Gamma \mid \Gamma \vdash P \quad FV(P) = dom(\Gamma)\}$

It is assumed that we keep a table, J, mapping function names to the corresponding return type-jump-function. $Types(f)$ is the set of type assignments that are acceptable for any call to f. This set is formed from the return type-jump-function for f using a mapping, μ, from $-variables to linear expressions of $-variables.

In order to have a valid type derivation of a procedure P it must be assumed that the return type-jump-functions for any procedure f called in P must give only the input and output type configurations that allow f

to be well-typed. Since the type inference process infers these entries, we must show that given correct entries for J on the primitive operators, we infer the correct return type-jump-function for any procedure that is built up from the primitive operations. Therefore, any entry to J entered from results of type inference is correct.

To show all of this, we first prove that the type system we present is type safe with respect to CORE-MATLAB, given that J is correct. We then show that we can infer exactly the set of Γ that allow for a type derivation over P, which we say is the correct *Principal*. It is trivial to show that by inferring *Principal* for P, we can extend J to include a new entry for P, that takes only the information from the input and output parameters from each Γ in *Principal*, and combines them using XOR to produce the return type-jump-function. Since $J(P)$ is based on a correct *Principal* for P, the new J is still correct.

Note that the type system uses XOR types over functions to represent the different possibilities. We term these types mutually exclusive types. These types are necessary both to define the smallest number of variants and for efficiency purposes as we will show in Section 5.

Operational Semantics The evaluation rules, as shown in Figure 6, describe the CORE-MATLAB semantics. We evaluate these rules over procedure bodies with a given set of input types to the procedure.

We add syntax and evaluation rules for two primitive operators *, and +, to show the types of errors that can be caught from static analysis, namely, that certain size constraints hold on inputs to the operations and procedure calls. These primitive operations have return type-jump-function entries in J and are therefore not handled separately in the type system. In telescoping languages, the distinction between primitive operations and calls to library procedures is removed.

Safety We define the following judgment to relate type environments to stores in the evaluation rules.

$$\frac{\{x := v\} \in \Sigma \quad \Gamma \vdash x : t \quad \cdot \vdash v : t}{\Gamma \vdash \Sigma}$$

Appendix A provides a proof for the following type safety theorem.

Theorem 4.1 *Type Safety:*
Given J, if $\Gamma \vdash \Sigma, \Gamma \vdash e : t$ and $\Sigma, e \hookrightarrow v$ then $\Gamma \vdash v : t$.
Also, if $\Gamma \vdash \Sigma, \Gamma \vdash P$, and $\Sigma, P \hookrightarrow \Sigma'$, then $\Gamma \vdash \Sigma'$.

Note that this theorem ensures that no well-typed procedure results in an error.

4.2 Deriving Principal

The typing rules only state that a procedure is correct with respect to a given *Principal*. What it does not state is how we derive *Principal* for the procedure. We now formally describe the process by which we derive all valid Γ's.

Principal can be derived in two steps. First, the procedure constraint must be formed from the statement constraints. Then the procedure constraint must be solved to form the type jump-function. We show that these two steps are sufficient to find *Principal*.

Building the Procedure Constraint Before procedure constraints can be built, the procedure must be transformed so that each statement contains only one operation or procedure call, since constraints are formed over statements and not operations. This normalization is described in Figure A in the Appendix, as well as that normalization preserves types.

As shown in Figure 7, statement constraints are formed from return type-jump-functions, with the formal parameters replaced by the actual parameters. The $-variables are replaced by $-variables that are fresh in the procedure constraint to isolate the statement constraints from each other. If one of the actual parameters involved in the statement is a subscripted array access, an extra atom stating that the parameter must be a scalar is appended to the statement constraint. This has the effect of negating any clause that treats that parameter as a non-scalar. In the actual implementation, the compiler just eliminates these clauses.

Solution for the Procedure Constraint The procedure constraint merely states what properties must hold over the variable types for the procedure to be valid. It does not give the possible type configurations, or Γ, necessary for variant generation. The program constraint must be solved to determine all possible type configurations. The solution to a procedure constraint is described in Figure 8. A Γ satisfies the constraints if for all the program constraints, there exists a mapping, M, from $-variables to linear expressions over $-variables that match the constraints in Γ, and this mapping does not cause scalars to be mapped to non-scalars.

Soundness and Completeness of Solution We show that our solution is both sound and complete with respect to the type system given in Figure 5.

Theorem 4.2 *Soundness: Given J, if $M, \Gamma \vdash pc$ and $tr[P] : pc$ then $\Gamma \vdash P$.*

Proc Constraints $pc \in PC ::= PC \wedge SC \mid \epsilon$
Stmt Constraints $sc \in SC ::= SC \text{ xor } CL \mid CL$
Expressions $e \in E ::= V \mid X$
$\$-vars(r) = \overline{d}$, where $\overline{d}$ is the set of \$-vars used in r.

$$\frac{}{\substack{[k_{11},\ldots,k_{1m};\ldots;k_{n1},\ldots,k_{nm}]: \\ \sigma^{[k_{11},\ldots,k_{1m};\ldots;k_{n1},\ldots,k_{nm}]} =<n,m>}}[\textsc{Value}]$$

$$\frac{}{x:\epsilon}[\textsc{Var}] \quad \frac{}{\substack{x(e_1,e_2):\sigma^{x(e_1,e_2)} =<1,1> \wedge \\ \sigma^{e_2} =<1,1> \wedge \sigma^{e_1} =<1,1>}}[\textsc{Elem}]$$

$$\frac{}{stop:\epsilon}[\textsc{Stop}] \quad \frac{c:sc \quad P:pc}{c;P:sc \wedge pc}[\textsc{Proc}]$$

$$\frac{\substack{e_1:t_1,\ldots,e_n:t_n \\ J(f)=r \quad T(f)=fn\ p_0 := f(p_1,\ldots,p_n)\ P \\ \$-vars(r)=\overline{d} \quad \overline{d'}\ fresh \quad |\overline{d'}|=|\overline{d}|}}{\substack{x=f(e_1,\ldots,e_n): \\ ([p_0 \mapsto x, p_i \mapsto e_i][\overline{d} \mapsto \overline{d'}]r) \wedge t_1 \wedge \cdots \wedge t_n}}[\textsc{Stmt}]$$

Figure 7: Simple Constraint System for Size Problem

$$\frac{t' =<M(s_1),M(s_2)> \quad t \neq<1,1> \Rightarrow t' \neq<1,1>}{M(t)=t'}[\textsc{TMap}]$$

$$\frac{\{M(d_i)=s_i\}^{i\in\{1,\ldots,n\}}}{\substack{M(k_1d_1+\cdots+k_nd_n)= \\ k_1s_1+\cdots+k_ns_n}}[\textsc{SMap}] \quad \frac{d \mapsto s \in M}{M(d)=s}[\textsc{DMap}]$$

$$\frac{\Gamma(x)=M(t)}{M,\Gamma \vdash \sigma^x = t}[\textsc{Atom}] \quad \frac{cl = at \wedge cl' \quad M,\Gamma \vdash at \quad M,\Gamma \vdash cl'}{M,\Gamma \vdash cl}[\textsc{Clause}]$$

$$\frac{sc = cl \text{ xor } sc' \quad M,\Gamma \vdash cl \text{ or } M,\Gamma \vdash sc'}{M,\Gamma \vdash sc}[\textsc{Stmt-Constr}]$$

$$\frac{pc = sc \wedge pc' \quad M,\Gamma \vdash sc \quad M,\Gamma \vdash pc'}{M,\Gamma \vdash pc}[\textsc{Proc-Constr}]$$

Figure 8: Well-formed Type Jump-Functions

Theorem 4.3 *Completeness: Given J and P, if* $\Gamma \nvdash P$ *then there does not exist a M such that* $M,\Gamma \vdash pc$*, where* $tr[P] : pc$.

5 An Implementation for Solving Constraints

Now that we have formally defined the type-inference problem and desired solution for CORE-MATLAB, we describe an efficient implementation over the whole subset of MATLAB described in Section 2.

Solving the constraints can be broken down into three phases - building a graphical representation of the constraints, finding n-cliques over the graph, and solving the cliques to produce the type jump-function.

First, there are a number of assumptions necessary for this algorithm to perform efficiently.

1. The type-inference engine has valid code on input (i.e., all variables are defined before being used).[4]
2. All global variables have been converted to input and output parameters.
3. The number of input and output parameters in each operation or procedure is bounded by a constant. This property is important to limit the complexity and is common in practice since parameter lists do not tend grow with the size of the procedure [7]. Also, few programmers use global variables excessively.

5.1 Reducing to Clique-Finding Problem

After the constraints for each operation have been determined, the compiler must reason about them over the whole procedure. That is, it must find all possible type configurations that satisfy the whole-procedure constraint. By representing the operation constraints as nodes in a leveled-graph, the problem is reduced to finding n-cliques, where n is the number of operations in the procedure, and an n-clique is a complete subgraph involving n nodes.

Figure 9 shows a simple example of how the graph is constructed. Each clause, or possible type configuration for that operation, is represented by a node at the level corresponding to its statement. There is an edge from one node to another if the expressions in the nodes do not contradict each other. For example, there are no edges from $1b$ to $2b$ since c cannot be both scalar and non-scalar. Note that since each clause is mutually exclusive, there is no edge between nodes on the same level.

The final graph has n levels, where n is the number of operations or procedure calls (or statements in the expanded form). Each level is bounded by 2^v nodes, since 2 is the number of possible types for each variable (scalar or non-scalar), and v is the number of variables involved in the statement. v is assumed to be small by the third assumption. 2^v is the number of possible type configurations, or entries in the type-jump-function table, for the variables in the operation corresponding to

[4]This is a reasonable assumption for MATLAB programs since users can develop and test their code in the MATLAB interpreter before giving it to the optimizing compiler.

```
A = b + c
```
1a $\sigma^A = <1,1> \wedge \sigma^b = <1,1> \wedge \sigma^c = <1,1>$ XOR
1b $\sigma^A = <\$1,\$2> \wedge \sigma^b = <1,1> \wedge \sigma^c = <\$1,\$2>$ XOR
1c $\sigma^A = <\$1,\$2> \wedge \sigma^b = <\$1,\$2> \wedge \sigma^c = <1,1>$ XOR
1d $\sigma^A = <\$1,\$2> \wedge \sigma^b = <\$1,\$2> \wedge \sigma^c = <\$1,\$2>$

```
E = c - d
```
2a $\sigma^E = <1,1> \wedge \sigma^c = <1,1> \wedge \sigma^d = <1,1>$ XOR
2b $\sigma^E = <\$3,\$4> \wedge \sigma^c = <1,1> \wedge \sigma^d = <\$3,\$4>$ XOR
2c $\sigma^E = <\$3,\$4> \wedge \sigma^c = <\$3,\$4> \wedge \sigma^d = <1,1>$ XOR
2d $\sigma^E = <\$3,\$4> \wedge \sigma^c = <\$3,\$4> \wedge \sigma^d = <\$3,\$4>$

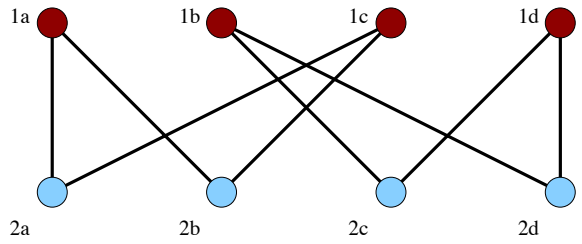

Figure 9: Example graph.

that level, since there are two possibilities for each variable. Since v is assumed to be bounded by a small constant, 2^v must be bounded by a constant.

Finding possible constraints over the entire procedure corresponds to finding sets of clauses that do not contradict each other such that there is one clause from each operation or procedure call. Having at least one clause from each operation constraint is necessary since otherwise, the resulting types will not hold over the entire procedure. On the graph, this is exactly the problem of finding all n-cliques (where n is the number of levels in the graph, and a clique is a complete subgraph) such that each n-clique has exactly one node from each level.

5.2 Finding Cliques

In order to show that using cliques to determine types is viable for our problem, we must first show that the number of type configurations is bounded by a small number, and from this, that the total number of n-cliques is bounded by a small number.

Let *u-vars* be defined to be the smallest set of variables such that all other variable types can be determined from the types of the u-vars. U-vars represent the set of variables that are not guaranteed to be statically determinable in terms of the types of the other variables (i.e., input types). The algorithm may be able to infer exact types for some or all of the u-vars by their uses. For the simple case where all operations are input-dependent, u is bounded above by the number of input parameters and is therefore small by assumption 3.

We define a *valid* procedure to be a procedure with the property that all definitions of variables occur lexically before any of their uses.

Theorem 5.1 *The number of possible type configurations is bounded by 2^u, where u is an upper bound on the number of u-vars.*

Proof: By the definition of u-vars, all other variable types excepting the u-vars can be statically determined in terms of the types of the u-vars. Therefore, the total number of possible configurations over all the variables is just the number of possible configurations of the u-var types. This is 2^u since each u-var could potentially take on two types (scalar or non-scalar). □

Theorem 5.2 *The number of n-cliques is bounded by 2^u.*

Proof (by contradiction): Since each clique represents a possible type configuration, we need to show that no two cliques represent the same type configuration. We start by assuming there are two distinct cliques that represent the same type configuration over the variables. The cliques must differ at least at one level. Since expressions in nodes of the same level contradict each other, at least one variable must have a different type. Therefore, the two cliques cannot have the same type configuration over the variables. □

Finding n-cliques in general is $\mathcal{NP}$-Complete. However, we claim that given the structure of the problem, we are solving a subset of the n-clique problem that has a polynomial time solution given a bound on u.

The solution must be able to take advantage of the specific properties of the problem. Figure 10 gives a simple iterative algorithm to achieve this. The algorithm starts with one level and puts each node in that level in its own clique. For each subsequent step, it compares each node in the current level with each already formed clique. If the node has an edge to every member of the clique, it forms a new clique with the old clique. It does this until it reaches the last level.

The loop starting on line 4 in Figure 10 iterates over the cliques from the previous step. Because the bound of 2^u only holds for the final number of cliques, we need to find a bound on the number of intermediate cliques to limit the complexity.

Theorem 5.3 *The number of cliques at each step of the iterative n-clique-finding algorithm is bounded by 2^u if the levels are visited in program order.*

Proof: We have from above that on valid procedures there is an upper bound of 2^u on the number of cliques. At any operation or procedure call, the rest of the code

```
input:  graph G
output:  CurrCliques
initialize CurrCliques to first-level nodes
1 for every level r in G - first row
2   newCliques=empty
3   for every node n in r
4     for every clique c in CurrCliques
5       candidate=true
6       for every node q in c
7         candidate=candidate & edge?(n,q)
8       end for
9       if (candidate)
10        newCliques=newCliques + clique(c,n)
11    end for
12  end for
13  CurrCliques=newCliques
14 end for
```

Figure 10: Iterative n-clique finding algorithm.

can be left off and the remaining (beginning) code is still valid.[5] Therefore, since every iteration of the algorithm has processed valid code, if the levels are in program order, after every iteration, the algorithm will have produced cliques on valid code. The number of cliques after every iteration must be bounded by 2^u. □

With 2^u cliques after every iteration, the n-clique-finding algorithm takes $2^v 2^u n^2$ steps, where n is the number of operations, 2^v is the maximum number of nodes in a level (always bound by a small constant), and u is the number undeterminable variables. Therefore, the overall time complexity is $O(n^2)$ if 2^u is bounded by a constant, which is true when all operations are input dependent.

Of course, in MATLAB, not all operations are input dependent. For some operations, the types of the outputs could depend on the values of the inputs rather than the types. This means that an operation could produce multiple output types on a given set of input types. Variables defined by such operations are u-vars. u should still remain small, since few operations are not input-dependent. Note that the algorithm works even without this assumption, but the complexity could become exponential in the worst case.

5.3 Handling Control Flow

Loops as well as branch statements use control-flow constructs. In the SSA representation, ϕ-functions represent a merger of variable values under the assumption that every control-flow branch can be taken. As a result, ϕ-functions also represent the *meet* operation for types. The ϕ-nodes themselves do not need to be dealt with explicitly by the algorithm, but the introduction of a new variable defined by the ϕ-node causes an u-var if used later in the program.[6]

Although u could now be as large as the number of operations and procedures in the function, in actuality, this number should still remain small, since the amount of control flow is usually limited. If the amount of control flow does become large, the graph can be split at statements corresponding to join points, and separate cliques can be found for each piece of the graph. The cliques can be merged when the all the pieces have been analyzed. This can greatly reduce the complexity.

The compiler is not able to generate specialized variants for the added cliques corresponding to decisions from ϕ-nodes and non-input-determined operations. However, the compiler can generate specialized paths from the control-flow points and operations if it can determine that optimizing would be beneficial. For example, the compiler would want to have separate paths if the outcome of the ϕ-node could either be real or complex. Since this could cause an increase in the size of the code, the compiler must be careful about how many specialized paths are generated.

Ultimately, the compiler could allocate the meet of the types to the variable on all paths.

5.4 Subscripted Array Accesses

MATLAB has support for using and assigning to pieces of arrays. When only a piece of an array is accessed in a statement, there are no constraint on that array for that dimension. The sizes of the other variables will, however, be constrained by the size of the piece of the array accessed. This includes subscripted arrays on the left-hand side of an assignment statement.

If the size of the array access is defined in terms of the value of another variable the compiler needs to account for the fact that the value could make the access scalar. Figure 11 illustrates how the constraints are written to handle this situation. The first fields of σ^v do not appear in the actual constraint, but are left in to show the relationship of the other variables to the piece of v accessed.

Array accesses do give us some information about the size of the actual variable, however. The variable must be at least the size of the portion accessed in

[5] Since valid only refers to the fact that every variable is defined before being used, SSA gives us this property in the presence of control-flow constructs.

[6] We can get the benefit of working with pruned SSA without requiring the code to be in the pruned form, since if a variable is never used, it never appears in a constraint.

```
t = V(:,1:j)* h
```

$\sigma^{\mathbf{t}} = <1,1>$ $(\wedge\ \sigma^{\mathbf{V(:,1:j)}} = <1,1>)$ $\wedge\ \sigma^{\mathbf{h}} = <1,1>$ XOR
$\sigma^{\mathbf{t}} = <\$1,\$2>$ $(\wedge\ \sigma^{\mathbf{V(:,1:j)}} = <1,1>)$ $\wedge\ \sigma^{\mathbf{h}} = <\$1,\$2>$ XOR
$\sigma^{\mathbf{t}} = <\$1,j>$ $(\wedge\ \sigma^{\mathbf{V(:,1:j)}} = <\$1,j>)$ $\wedge\ \sigma^{\mathbf{h}} = <1,1>$ XOR
$\sigma^{\mathbf{t}} = <\$1,\$3>$ $(\wedge\ \sigma^{\mathbf{V(:,1:j)}} = <\$1,j>)$ $\wedge\ \sigma^{\mathbf{h}} = <j,\$3>$ XOR
$\sigma^{\mathbf{t}} = <1,1>$ $(\wedge\ \sigma^{\mathbf{V(:,1:j)}} = <1,j>)$ $\wedge\ \sigma^{\mathbf{h}} = <j,1>$

Figure 11: Dealing with Subscripted Array Access

that dimension.[7] If the compiler can determine that the subscript size is greater than one, then it can add a constraint that forces the variable to be non-scalar, reducing the number of cliques.

In order to take advantage of subscript information when the subscripts are written in terms of procedure variables, we must first perform a variation on constant propagation to determine the maximum value of the variables and, therefore, the maximum array sizes.

5.5 Using Annotations

One of the key ideas in telescoping languages is allowing the compiler to utilize the knowledge of the library writer through annotations. This information is important in knowing what cases can and cannot occur in practice, which the compiler alone may not be able to infer.

The library writer can provide type information on the parameters of the procedure to be analyzed. In the absence of source code, these annotations can be transformed into the return type-jump-function. The compiler treats these annotations as constraints on the procedure header, which corresponds to the zeroth level in the graph. Any cliques occurring in the graph must have part of the user-defined annotations as one of their nodes. Annotations can greatly reduce the number of possible cliques and, therefore, specialized variants. They can also reduce the runtime of the algorithm. If there are no provided annotation, the header is assumed to be unconstrained.

5.6 Interprocedural Type Inference

When the algorithm encounters a procedure call or a built-in operation, it looks in the database for the appropriate return type-jump-function to build the constraints at that statement. If the compiler encounters a procedure call for which there is no return type-jump-function, it simply considers the variables unconstrained by that statement. This degrades the analysis of the algorithm in that it may infer more type configurations than are legal. However, all legal configurations will still be inferred.

However, in the case of a cycle in the call graph or recursion, tighter information can be gained by iterating over the cycle until a fixed-point is reached. A fixed-point can be reached in a constant number of iterations for each procedure involved. At each iteration, the type inference algorithm tries to reduce the number of clauses in the return type-jump-function for the procedure (initially the return type jump-function contains all possible type configurations over the input and output arguments). Since there are only a constant number of clauses, this number can only be reduced a constant number of times.

5.7 Solving the Cliques

Each clique represents a different possible type configuration. However, the equations in each clique still need to be solved to determine a more succinct representation of the type configurations (i.e., the minimal set of types that satisfy the equations in the cliques). The compiler solves the cliques using a variant of techniques used for solving linear Diophantine equations. This step involves a linear pass over each equation in the clique, and can be performed in linear time with respect to the number of equations.

All the variable sizes are computed in terms of the sizes of the u-vars. The u-vars themselves may be statically inferred. In some cases, it is determined that a clique is invalid if there is no solution from the solver.

6 Extending Type Inference

The algorithm described in Section 5 can be applied to the type problems discussed in Section 2.2. The compiler infers each element of $\mathcal{T}$ in a separate pass and then takes the cross product of the different types to determine which variants are necessary. It can handle types separately since, except for dimensionality and size, the types are independent of each other, although knowing that a variable is scalar makes pattern inference unnecessary.

6.1 Handling Multiple Dimensions

The type inference algorithm described in this paper can easily be extended to handle inferring sizes over multiple dimensions by simply increasing the number of fields in σ.

In order to determine the number of fields needed, an upper bound on the number of dimensions of an array is needed,before size inference can occur. Only

[7] This is only true because it is assumed that redefining parts of an array create an entirely new array in SSA.

an upper bound is needed since the size inference will determine the actual dimensionality by inferring that some dimensions have size 1. Inferring dimensionality involves a single pass over the code to determine which dimensions of each variable are accessed explicitly or may be used by an operation. If an upper bound cannot be determined (some operations have no limit on the number of dimensions), a dummy field in the size tuple is used to represent the sizes of dimensions beyond what is explicitly referenced.

Inferring sizes of arrays with dimension over two should not increase the complexity, since the number of possible types for each variable is still only two - scalar or non-scalar. If the number of explicitly accessed dimensions grows arbitrarily this could increase the time to build the constraints, build the graph, and solve the constraints. However, it is unlikely that the procedure will explicitly access an arbitrary number of dimensions.

6.2 Pattern and Intrinsic Type Problems

The problems of inferring intrinsic types and patterns differ from inferring size in that the algorithm only operates on finite lattices. Therefore, the constraints are formulated differently. The constraints must restrict variables to a range of types on the respective lattice. Using ranges allows for values of types lower in the lattice than those defined by the parameter type specifications to be accepted as input. For example, an input argument that is defined as type `real` could actually be of type `int` when called. Using ranges also reduces the number of cliques, since each node can represent multiple possibilities.

The constraints for intrinsic types on the *mean* operation are shown in Figure 6.2.

```
o=mean(i)

(real ≤ τ^o ≤ real) ∧ (⊥≤ τ^i ≤ real)      XOR
(comp ≤ τ^o ≤ comp) ∧ (comp ≤ τ^i ≤ comp)
```

Figure 12: Intrinsic type constraints on mean operation.

Two constraint clauses are not compatible if a variable appears in both clauses and the corresponding ranges do not intersect. The compiler still needs mutual exclusivity for the algorithm to run efficiently. Also, since maintaining the input dependence property is important in reducing the complexity, when possible, the constraints are formulated so that the same input configuration should give only one type of output.

Once the compiler has found the cliques, solving the equations corresponds to taking the intersection of all ranges for each variable over the clique.

The lattices for the intrinsic type and shape problems include a topmost element which is the most general case, a $\perp$, which represents invalid types, and intermediate elements. The meet between two elements is the top-most intersection of their paths from the bottom element.

The number of steps for finding cliques for these problems is bounded by $l^v l^u n^2$ steps, where l is the number of lattice elements, which should be bounded by a small constant. Therefore, the overall time complexity is still $O(n^2)$ if l^u is bounded by a constant, which is true when operations are primarily input dependent.

LibGen allows the library writer to extend the type inference problem with new type problems not handled by size, shape and intrinsic type. The new problems, like intrinsic type and shape, must work on a single finite lattice. The library writer extends the inference problem by including a new base lattice for the new type problem.

We envision future work that allows programmers to perform type inference in the presence of user-defined types as well.

7 Related Work

MCC is a Matlab to C compiler provided by Mathworks. The output from MCC does not optimize library calls; every call is identical to what would have been made if the source code were simply evaluated by the Matlab interpreter. Furthermore, MCC performs no type inference; all variables are simply treated as arrays of a general type [22].

Type inference in the FALCON and MaJIC compilers from the University of Illinois at Urbana-Champagn is based on dataflow analysis [11, 10, 2, 1]. FALCON relies on inlining to exactly determine types, which can result in massive code bloat and script compilation times for even moderately large libraries. In contrast, MaJIC performs just-in-time compilation, incurring additional run-time costs. Additionally, as discussed in Section 2 dataflow analysis is inadequate for specialization of library calls because provable termination is difficult to guarantee and does not provide adequate precision.

MAGICA, a MATLAB type inference system developed at Northwestern, is able to infer array sizes for multiple dimensions [20] by solving algebraic systems of equations by dispatching to Mathematica. MAGICA infers only a single set of explicit types that it can guarantee for all contexts, and is therefore ill-suited for specialization of library calls. Furthermore, we are not aware of any complexity results for this system.

Recently, Elphick et al implemented a type inference system for Matlab that relied on partial evaluation [13].

Partial evaluation is insufficient for the purposes of telescoping languages because it performs type inference before specific calling contexts are known.

In [32], Xi and Pfenning used dependent types to perform array bounds checks in Matlab programs. However, their system accepts fewer valid programs than the system described in this paper, and they do not address the more general problem of type inference for library specialization.

Hindley-Milner type inference is a common technique for performing polymorphic type inference in functional programming languages [23]. But it is ill-suited for typing scripts in languages designed for scientific computation because these languages provide much more limited constructs for variable binding, datatype definition, and control flow. This lack of expressiveness allows us to form much more powerful constraints over program variables. Unlike Hindley-Milner, our system determines logical constraints in several program variables at once based on the use of those variables in all program statements in a procedure. Hindley-Milner determines a (polymorphic) type for each expression in a program independently. Limited dependency between types is expressed through the use of type variables. But complex logical relationships between variable types expressed as a disjunction of conjunctions of variable type judgements are not expressible. Furthermore, Hindley-Milner does not support types such as matrix sizes that are parameterized by integer values.

Subtyping with union and intersection types express properties similar to those needed in telescoping languages [26, 24, 4, 8, 28, 27, 25]. However, because the various clauses formed by our system in a statement constraint over the procedures are disjoint, subtyping with intersection types does not apply. The return type-jump-functions are similar to disjoint-union types for procedures, but the relationships between the procedure variables is not expressible using disjoint-union types because disjoint-union types are types of expressions; they are not logical constraints expressing dependencies over multiple program variables.

Techniques proposed in this paper can produce code that works with specialized libraries that are automatically tuned for specific platforms or problems such as ATLAS and FFTW [31, 14]. For example, specialized BLAS routines can be called directly if the input matrix size is known.

Constraint Logic Programming (CLP) [29, 17, 18] extends the purely syntactic logic programming (typified by linear unification) by adding semantic constraints over specific domains. Some of the well-known CLP systems include CHIP [12], CLP($\mathcal{R}$) [19], Prolog-III [6], and ECL^{i}PS^e [30]. While a general purpose CLP system could be employed in solving the constraints within our type-inference system, our algorithm utilizes the properties of the problem to operate within a provably efficient time complexity.

Guyer and Lin have developed an annotation language for guiding optimizations on libraries [16]. An important direction for future work is to determine whether this language could be used to express annotations adequate for our system.

8 Conclusions

This paper develops a type inference algorithm used for library generation in a telescoping compiler. The algorithm infers all possible type configurations in order to determine the minimum number of variants that should be generated to handle all valid types of inputs efficiently as well as the types of the procedure variables in terms of input types and values. The latter information allows the compiler to replace the operations and procedure calls with calls to variants optimized for the given types.

The type inference algorithm developed in this paper provides a basis for future work in other languages and type problems, including object oriented features.

References

[1] G. Almási. *MaJIC: A Matlab Just-in-time Compiler*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[2] G. Almási and D. Padua. MaJIC: Compiling MATLAB for speed and responsiveness. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 294–303, June 2002.

[3] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. *SIGPLAN Notices*, 21(7):162–175, July 1986.

[4] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, Feb./Mar. 1988.

[5] A. Chauhan, C. McCosh, K. Kennedy, and R. Hanson. Automatic type-driven library generation for telescoping languages. In *ACM/IEEE SC2003 Conference on High Performance Networking and Computing (Supercomputing)*, Nov. 2003.

[6] A. Colmerauer. An introduction to Prolog-III. *Commun. ACM*, 33(7):69–90, July 1990.

[7] K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of*

the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, pages 57–66, Atlanta, GA, June 1988.

[8] M. Coppo, M. Dezani-Ciancaglini, B. Venneri, c of, and t Zeitschrift. ur mathematische logik und grundlagen der mathematik, 1981.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[10] L. DeRose and D. Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Transactions on Programming Languages and Systems*, 21(2):286–323, Mar. 1999.

[11] L. A. DeRose. *Compiler Techniques for Matlab Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[12] M. Dincbas, P. V. Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 693–702, Dec. 1988.

[13] D. Elphick, M. Leuschel, and S. Cox. Partial evaluation of MATLAB. In *Generative Programming and Component Engineering (GPCE'03)*, Lecture Notes in Computer Science, pages 344–363, 2003.

[14] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, May 1998.

[15] D. Grove and L. Torczon. Interprocedural constant propagation: A study of jump function implementations. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 90–99, June 1993.

[16] S. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Proceedings of the Second Conference on Domain-Specific Languages*, Mar. 1999.

[17] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 111–119, 1987.

[18] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[19] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992. Also available as Technical Report, IBM Research Division, RC 16292 (#72336), 1990.

[20] P. Joisha and P. Banerjee. Implementing an array shape inference system for MATLAB. Technical Report CPDC-TR-2002-10-003, Center for Parallel and Distributed Computing, Department of Electrical and Computer Engineering, Northwestern University, Oct. 2002.

[21] K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnson, J. Mellor-Crummey, and L. Torczon. Telescoping Languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61(12):1803–1826, Dec. 2001.

[22] http://www.mathworks.com/. MATHWORKS.

[23] R. Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 17(2):348–375, 1978.

[24] J. Mitchell. Type inference with simple types. *Journal of Functional Programming*, pages 245–285, 1991.

[25] B. C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, 1991.

[26] J. Reynolds. Using category theory to design implicit conversions and generic operators. In *Semantics-Directed Compiler Generation*, Springer-Verlag Lecture Notes in Computer Science, pages 211–258, 1980.

[27] J. Reynolds. Design of the programming language forsythe. Technical Report CMU-CS96 -146, Carnegie Mellon University, June 1996.

[28] P. Salle. Une extension de la theorie des types en -calcul. In *Springer-Verlag*, volume 62 of *Lecture Notes in Computer Science*, pages 398–410, 1982.

[29] G. L. Steele. *The Definition and Implementation of a Computer Programming Language based on Constraints*. PhD thesis, M.I.T., 1980. AI-TR 595.

[30] M. Wallace, S. Novello, and J. Schimpf. *ECL^{i}PS^e: A Platform for Constraint Logic Programming.* William Penney Laboratory, Imperial College, London, 1997.

[31] R. C. Whaley and J. J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proceedings of SC: High Performance Networking and Computing Conference*, Nov. 1998.

[32] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, June 1998.

A Appendix

$$\frac{\Sigma, e_1 \hookrightarrow error \text{ or } \Sigma, e_2 \hookrightarrow error}{\Sigma, x(e_1, e_2) \hookrightarrow error} \text{ [EP-Elem]}$$

$$\frac{\Sigma, e \hookrightarrow error}{\Sigma, x := e \hookrightarrow error} \text{ [EP-Stmt]}$$

$$\frac{\Sigma, c \hookrightarrow error}{\Sigma, c; P \hookrightarrow error} \text{ [EP-Proc]}$$

$$\frac{\begin{array}{c}\exists i \in \{1, \cdots, n\} \text{ s.t. } \Sigma, e_i \hookrightarrow error \text{ or} \\ T(f) = fn\ [p_0] := f(p_1, \ldots, p_n)\ P \\ \Sigma, [p_1 \mapsto v_1, \ldots, p_n \mapsto v_n]P \hookrightarrow error\end{array}}{\Sigma, f(e_1, \cdots, e_n) \hookrightarrow error} \text{ [EP-App]}$$

$$\frac{\Sigma, e_1 \hookrightarrow error \text{ or } \Sigma, e_2 \hookrightarrow error}{\Sigma, e_1 + e_2 \hookrightarrow error} \text{ [EP-Add]}$$

$$\frac{\Sigma, e_1 \hookrightarrow error \text{ or } \Sigma, e_2 \hookrightarrow error}{\Sigma, e_1 * e_2 \hookrightarrow error} \text{ [EP-Mult]}$$

$$\frac{\Sigma, e \hookrightarrow error}{\Sigma, x(q, r) := e \hookrightarrow error} \text{ [EP-ElemStmt]}$$

Figure 13: Error Propagation Rules

Proof of Theorem 4.1: By structural induction on evaluation derivation.

Case [Var]: The result is immediate because $x := v \in \Sigma$, since $\Gamma \vdash \Sigma$, and therefore, $\Gamma \vdash x : t$ means that $\Gamma \vdash v : t$

Case [E-Var]: If $x; = v \notin \Sigma$, then x is not well-typed.

Case [Elem]: By the induction hypothesis $\Sigma, e_1 \hookrightarrow v_1$, $\Sigma, e_2 \hookrightarrow v_2$, and $\Sigma, x \hookrightarrow v$, and $\Gamma \vdash v_1 : t_1$, $\Gamma \vdash v_2 : t_2$, and $\Gamma \vdash v :< s_1, s_2 >$. We know that the sizes of v_1 and v_2 are both $<1, 1>$. Since we know that $v_1 \leq s_1$ and $v_2 \leq s_2$, the evaluation results in $v' :<1, 1>$, since we are only accessing a single element of the matrix.

Case [E-Elem]: By induction hypothesis $\Sigma, e_1 \hookrightarrow v_1$, $\Sigma, e_2 \hookrightarrow v_2$, and $\Sigma, x \hookrightarrow v$, and $\vdash v_1 : t_1$, $\Gamma \vdash v_2 : t_2$, and $\Gamma \vdash v :< s_1, s_2 >$. If $\{x := v\} \notin \Sigma$, then this fails an assumption of the theorem and is therefore not well-typed. Also, if either $t_1 \neq <1, 1>$ or $t_2 \neq < 1, 1 >$, then this expression is not well-typed.

Case [App]: We know by the induction hypothesis $\Sigma, e_i \hookrightarrow v_i$ that $v_i : t_i \forall i \in 1, \ldots, n$. We also know that $t_1, \ldots, t_n \rightarrow t_0 \in Types(f)$. Since $Types(f)$ depends on the return type-jump-function for f, we kinow that there is a valid evaluation of f with the given input types, where p_0 has type t_0. Therefore, since v gets the value and and therefore type of p_0, $v : t_0$.

Case [E1-App]: If the wrong number of arguments are given then no typing assignment to arguments will be in $Types(f)$. Therefore, this expression is not well-typed.

Case [E1-App]: If $f \notin T$, then there is no entry for f in table J. By the definition of $Types(f)$, this means that the expression is not well-typed.

Case [Add]: By the induction hypothesis $\Sigma, e_1 \hookrightarrow v_1$, $\Sigma, e_2 \hookrightarrow v_2$, and $v_1 : t_1$ and $v_2 : t_2$. Then we assume there is an entry in the type jump-function table J, that shows that $t, t \rightarrow t$, for the $+$ operation. Then t_1 and t_2 must be the same if this is well typed. Therefore, $v : t$, by the definition of *Add*.

Case [E-Add]: By the induction hypothesis $\Sigma, e_1 \hookrightarrow v_1$, $\Sigma, e_2 \hookrightarrow v_2$, and $v_1 : t_1$ and $v_2 : t_2$. There is an entry in the type jump-function table J, that shows that $t, t \rightarrow t$, for the $+$ operation. Since t_1 and t_2 are not the same, this expression is not well typed.

Case [Mult]: By the induction hypothesis $\Sigma, e_1 \hookrightarrow v_1$, $\Sigma, e_2 \hookrightarrow v_2$, and $v_1 :< s_1', s_1'' >$ and $v_2 :< s_2', s_2'' >$. Then we assume there is an entry in the type jump-function table J, that shows that $< s_1', s_1'' >$, $< s_1'', s_2'' > \rightarrow < s_1', s_2'' >$, for the $*$ operation. Then s_1'' and s_2' must be the same if this is well typed. Therefore, $v :< s_1', s_2'' >$, by the definition of $Mult$.

Case [E-Mult]: By the induction hypothesis $\Sigma, e_1 \hookrightarrow v_1$, $\Sigma, e_2 \hookrightarrow v_2$, and $v_1 :< s_1', s_1'' >$ and $v_2 :< s_2', s_2'' >$.

$$\frac{}{nrm[x := v] \rightarrow x := v}[\textsc{Val}] \quad \frac{}{nrm[x := x'] \rightarrow x := x'}[\textsc{Var}]$$

$$\frac{g_1, \ldots, g_n \ fresh \ in \ P}{nrm[x := f(e_1, \ldots, e_n)] \rightarrow nrm[g_1 := e_1]; \ldots; nrm[g_n := e_n]; x := f(g_1, \ldots, g_n)}[\textsc{App}]$$

$$\frac{}{nrm[c; P] \rightarrow nrm[c]; nrm[P]}[\text{I-}\textsc{Proc}]$$

Figure 14: Normalization Rules.

Then we assume there is an entry in the type jump-function table J, that shows that $< s_1', s_1'' >, < s_1'', s_2'' > \rightarrow < s_1', s_2'' >$, for the $*$ operation. Since s_1'' and s_2' are not the same, this expression is not well-typed.

Case [Stmt]: By the induction hypothesis $\Sigma, e \hookrightarrow v$ we have $\Gamma \vdash v : t$. Since $\Gamma \vdash x := e$ by assumption on the theorem, $\Gamma(x) = t$ and $\Gamma \vdash e : t$. Therefore, since $\Gamma \vdash \Sigma$, then $\Gamma \vdash \Sigma + \{x := e\}$.

Case [Stop]: Trivially true.

Case [Proc]: By the induction hypothesis $\Sigma, c \hookrightarrow \Sigma'$, we have that $\Gamma \vdash \Sigma'$. We also have by the induction hypothesis $\Sigma', P \hookrightarrow \Sigma''$ that $\Gamma \vdash \Sigma''$.

Case Error propagation: The error propagation rules are given in Figure 13. These rules state that if an evaluation in the antecedent results in an error, then the evaluation of the expression results in an error. Since we have by induction hypothesis that any evaluation that results in an error will not occur in a well-typed expression, the evaluations in the antecedents must evaluate expressions that are not well-typed. Therefore, the expression in the error propagation rule must not be well-typed.

□

We show that normalization preserves types.

Theorem A.1 *If $\Gamma \vdash P$, then there exists $\Gamma' \supseteq \Gamma$ such that $\Gamma' \vdash nrm[P]$. Furthermore, $\Gamma' \vdash P$.*

Proof of Theorem A.1 : We prove the first statement in the proof by structural induction over the normalization rules.

Case [Value]: trivial, since the result of the normalization is the same as in the original statement.

Case [Var]: trivial, since the result of the normalization is the same as the original statement.

Case [App]: Let Γ' be such that $\Gamma' \supseteq \Gamma$, and $\Gamma' \vdash g_i : t_i$ for all $i \in \{1, \ldots, n\}$. Then, $\Gamma' \vdash g_1 := e_1; \ldots; g_n := e_n; x := f(g_1, \ldots, g_n)$. Therefore, by the induction hypothesis there exists a Γ'' such that $\Gamma'' \supseteq \Gamma'$ and $\Gamma'' \vdash nrm[g_i := e_i]$ for all $i \in \{1, \ldots, n\}$. Therefore, $\Gamma'' \vdash nrm[x := f(e_1, \ldots, e_n)]$.

Case [Elem]: Let Γ' be such that $\Gamma' \supseteq \Gamma$, and $\Gamma' \vdash g_1 : t_1$ and $\Gamma' \vdash g_2 : t_2$. Then, $\Gamma' \vdash g_1 := e_1; g_2 := e_2; x := x'(g_1, g_2)$. Therefore, by the induction hypothesis $\exists \Gamma''$ such that $\Gamma'' \supseteq \Gamma'$ and $\Gamma'' \vdash nrm[g_1 := e_1], \Gamma'' \vdash nrm[; g_2 := e_2]$. Therefore, $\Gamma'' \vdash nrm[x := x'(e_1, e_2)]$.

Case [Proc]: By induction hypothesis, there exists a $\Gamma' \supseteq \Gamma$ and a $\Gamma'' \supseteq \Gamma$ such that $\Gamma' \vdash nrm[c]$ and $\Gamma'' \vdash nrm[P]$. Therefore $\Gamma''' = \Gamma' \cup \Gamma''$ gives us $\Gamma''' \vdash nrm[c; P]$.

The second statement of the theorem is trivial since $\Gamma' \supseteq \Gamma$ □

Proof of Theorem 4.2 (By Induction Over Procedure Bodies): Base Case: Let P be *stop*. Then the derivation with respect to any Γ is:

$$\frac{}{\Gamma \vdash stop}.$$

Induction step: Assume P is a normalized procedure, Γ is well-formed with respect to P, and Γ allows for a valid type derivation of P. Let Γ' be another type environment. We can treat type environments as though they are existentially quantified, so that we can meaningfully combine type environments by conjoining them. Note that this corresponds to performing a remapping of the sizes before conjoining the type environments so that no \$-variables are shared across Γ and Γ'. Let $\Gamma'' = \Gamma \wedge \Gamma'$ be well-formed with respect to $c; P$. Then, since Γ is a subset of Γ'', there must be a valid derivation of P with respect to Γ''.

$$\frac{\cdots}{\Gamma'' \vdash P}$$

We must show that this derivation can be extended to include c. Assume c be $x := f(e_1, \cdots, e_n)$. Since Γ'' is well-formed, there must be a mapping, M from \$-variables to linear expressions over the \$-variables such that for one of the clauses in the statement constraint corresponding to c, M does not take matrices to scalars. Let m be the mapping from \$-vars to \$-vars that was used in generating the statement constraints. Let $\mu = M \circ m$. Then there exists a mapping, μ, that

that satisfies the restrictions for μ from the type system for the types in Γ to be in $Types(f)$. Therefore, we can use μ to prove that there is a type-derivation over c with respect to Γ, and we can append this derivation to that of P to get the type derivation for $c; P$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma''(x) = t}}{\Gamma'' \vdash x : t} \quad \dfrac{}{t_1 \cdots t_n \to t_0 \in Types(f)}}{\Gamma'' \vdash e_1 : t_1 \cdots \Gamma'' \vdash e_n : t_n}}{\Gamma'' \vdash f(e_1, \ldots, e_n) : t_0}}{\Gamma'' \vdash x := f(e_1, \ldots, e_n)} \quad \dfrac{\cdots}{\Gamma'' \vdash P}$$
$$\Gamma'' \vdash x := f(e_1, \ldots, e_n); P$$

Statements of the form $x := v$, $x := x'$, and $x := x'(e_1, e_2)$ are trivial, since they are the same constraints used in Γ for these statements.

When the expressions passed to f involve subscripts, extra constraints stating that these expressions have to have a size of $< 1, 1 >$ are added. These constraints match those given in the type system.

By theorem A.1, we have that $\$Gamma \vdash P'$, where P' is the pre-normalization form of P if $\Gamma \vdash P$.

□

Proof of Theorem 4.3 (By Contradiction): Assume there exists a valid type derivation for the normalized procedure P with respect to Γ, where Γ is not well-formed with respect to P. Then for some statement, c, the statement constraint, for every clause, for some atom, there does not exist a mapping, M, that can take the atom to Γ. Therefore, there does not exist a type assignment in $Types(f)$ for f called at statement c, since there can be no valid mapping, μ. Therefore by theorem A.1, we have that $\$Gamma \nvdash P$, means that if $\Gamma \nvdash P'$ where P' is the pre-normalization form of P.

□