Static Conformance Checking for Matrices

by

Roy Seto

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Engineering in Electrical Engineering and Computer Science

and

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

Author .....
Department of Electrical Engineering and Computer Science
May 16, 1994

Certified by .....
Raymie Stata
Department of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .....
John Guttag
Associate Head, Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Students

Static Conformance Checking for Matrices

by

Roy Seto

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 1994, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Electrical Engineering and Computer Science
and
Bachelor of Science in Computer Science and Engineering

Abstract

Abstractions such as matrices or state-space models can be viewed as families with common structure, modulo dimensional parameters such as numbers of rows and columns (for matrices) or numbers of inputs, outputs, and states (for systems). Such constructs occur frequently in engineering software. This thesis explores ways to automatically check "conformance" properties of these dimensional parameters such as the requirement that the number of columns of a matrix product's first factor equal the number of rows of the second factor. Our approach subsumes these parameters as dependent type parameters and includes conformance checking in type checking. A stylized language ("DP") is defined, along with typing rules and a checking algorithm. It includes a mechanism (similar to ML's type inference mechanism [12]) which infers implicitly instantiated parameters; this improves programmer convenience while preserving safety. A script from the MATLAB [17] control toolbox is hand-translated to DP and passed through a DP checker implementation to evaluate the type system's usefulness. Possible extensions are described.

Thesis Supervisor: Raymie Stata
Title: Graduate Student, Department of Electrical Engineering and Computer Science

Thesis Supervisor: John Guttag
Title: Associate Head, Computer Science and Engineering

Acknowledgments

My primary supervisor, Raymie Stata, proposed this project and provided essential guidance from beginning to end. He supplied numerous hints, steered me away from quite a few dead ends, and suggested invaluable improvements to this document. This thesis would not have been possible without his help.

My other supervisor, Professor John Guttag, also kept me on track with his practical and cheerful advice.

I would like to express my thanks to my parents for their support during my years at MIT.

Contents

List of Figures

Chapter 1

Introduction

Good type systems enhance software quality by helping programmers avoid making mistakes. They achieve this goal both by documenting the program's meaning and structure so that it is more comprehensible to human readers and by providing information that enables compilers to check automatically for errors.

This thesis aims to improve safety for software relying heavily on a certain class of data structures. Specifically, the data structures considered here can be characterized as families of abstractions that have the same behavior modulo dimensional parameters—matrices and systems of linear differential equations are two examples that arise frequently in engineering and scientific applications. We enhance safety by incorporating what we call "conformance" checks of these dimensional parameters into the type system so that they can be expressed naturally in the programming language and checked automatically by the language implemention. An example of a conformance check is the well-known requirement for matrix multiplication that the number of columns in the first operand be equal to the number of rows of the second.

Unfortunately, while type systems help to deal with the problem of erroneous programs, they also introduce problems of their own:

1. Meaningful programs may not be legal because the type system is too strict.

2. Information in the source language demanded by the type system may make programs more awkward for programmers to write.

This chapter introduces a combination of type-system features which provide safety while minimizing these problems. Specific features introduced include dependent types, parametric polymorphism, and type inference. The discussion will motivate each feature as it is introduced, explaining how the feature helps meet a need in the language.

1.1 Naive Approach

The simplest approach to the dimension conformance problem is to avoid dealing with it altogether in the type system. Matrix conformance is not supported by the compiler; instead, the programmer must manually write such checks on an individual basis. Conformance failures in such systems are runtime errors.

Though this approach (or lack of approach) seems simpleminded, it is taken by most systems in current use. For example, programmers representing matrices in C must explicitly keep track of matrix bounds by defining their own variables. The language provides no built-in support for checking conformance. Programming errors related to dimensional conformance tend to manifest themselves at runtime as memory protection violations.

Other languages provide somewhat more forgiving responses to errors. CLU's [10] abstract typing and exception facilities allow the programmer to build representations for types having dimensional parameters that provide somewhat more graceful runtime errors. For example, an abstract type defined to represent matrices could include a definition of a matrix multiplication operation. This definition could first test the dimensional conformance of its arguments. An exception would be signalled at runtime in the event of a conformance failure, which would allow a more graceful recovery than an arbitrary memory-access error would. As another example, the numerical computation tool MATLAB [17] provides dynamically resizable matrices. However, in these languages, conformance failures still are not detected until runtime.

1.2 Dependent Types

Especially for applications relying heavily on matrices, the ability to detect conformance errors at compile time would aid program safety. One method for doing this is to include the shape of the matrix in its type. In such a system, the matrix type is "dependent" because the type depends on value-parameters—namely, the numbers of rows and columns. A dependent type is a type that depends on a value [7, 11]. Subsuming dimensions in dependent matrix types enables the typechecker to verify conformance statically. Early versions of Pascal took this approach for arrays.

In our language, we declare the family of dependent matrix types with the "type generator" `mat`, which has two integer-valued parameters, as follows:

```
type mat[int,int];
```

`mat` can be specialized by instantiating it with type parameters to generate a ground type.

Unfortunately, this approach causes the first problem listed above, the rejection of meaningful programs, to arise because it makes the granularity of types too fine. If matrices of different shapes have distinct types, then separate versions of each procedure must be defined to handle each shape of matrix. A matrix-multiplication procedure that works for any two matrices $A_{m \times n}$ and $B_{n \times p}$ should be expressible as a meaningful computational notion, but such an approach makes it illegal.

Later versions of Pascal as well as Modula-3 (a Pascal descendant) avoid this problem by using "open types" [14]. They continue to include the bounds of array variables as part of the type. However, they allow array bounds to be unspecified ("open") in the signatures of procedures. The actual bounds can be queried at runtime. This approach makes the matrix-multiplication example above legal.

Open types, though, negate much of the advantage of putting array bounds in the type system since they actually indicate parts of types which are *not* specified or checked statically. Detection of conformance errors is deferred until runtime, just as if matrix and array bounds were not part of their types. This hole in the Pascal type system is similar to the other type-specification loopholes examined in [18].

1.3 Parametric Polymorphism for Procedures

We can reconcile static typechecking with dependent types by using parametric polymorphism, which was explored as a central aspect of the ML language [13, 12]. In ML, types, including types of functions, can be parameterized by type variables which range over types. We adapt the notion of polymorphism by parameterizing procedures by parameters which range over the integers. This enables the definition of procedure generators accepting arguments whose types have parameters. (We will always use the term "parameter" to denote a value or name which generalizes types, and use the term "argument" to denote a value or name which generalizes "procedure generators." The term "procedure generator" denotes a parameterized procedurel abstraction.)

In such a scheme, a procedure-generator for matrix multiplication could be declared as follows:

```
proc mmul[r,c,i](A: mat[r,i], B: mat[i,c]) returns mat[r,c];
```

In this declaration, the parameters of the `mmul` procedure generator are `r`, `c`, and `i`. The arguments are the parameterized types `A` and `B`. We don't need to indicate that the parameters `r`, `c`, and `i` are integer-valued because this thesis considers only integer-valued parameters. Now the expression

```
mmul[4,3,5](X,Y)
```

denotes the application of the specialized procedure `mmul[4,3,5]` to `X` and `Y`, which must have type `mat[4,5]` and type `mat[5,3]` to match the argument types in `mmul`'s signature. Exposing this conformance requirement in the signature facilitates its verification at compile time. Assuming that `mmul`'s definition is consistent with its signature in this type system, then the type of `mmul[4,3,5](X,Y)` could then be statically determined to be `mat[4,5]`.

This provides the safety benefit gained by expressing dimensional conformance requirements in the type system without incurring the unacceptable limits on expressiveness suffered by the simple-minded dependent-type scheme of Section 1.2.

1.4 Parameter Inference

While the preceding section's scheme enables the type system to include conformance checks while avoiding type-system problem (1), it exacerbates type-system problem (2) by making the procedure-call syntax unwieldy and inconvenient. Consider a naming environment in which matrices `A`, `B`, `C`, and `D` have types `mat[4,3]`, `mat[3,6]`, `mat[4,6]`, and `mat[4,6]`. If we want to assign the expression $AB + C$ to `D`, we are forced to write

```
D := madd[4,6](mmul[4,3,6](A,B), C);
```

Even an infix operator syntax would need to include tedious bookkeeping details:

```
D := A *[4,3,6] B +[4,6] C;
```

The need to specify these parameter values seems especially galling since we have already declared elsewhere the parameter values of `A`, `B`, `C`, and `D`'s types. It seems reasonable to expect the compiler to infer what the proper procedure-generator parameter values should be to make the procedure call well-typed. This would allow the far more intuitive syntax

```
D := madd(mmul(A,B), C);
```

or even

```
D := A * B + C;
```

The compiler would verify correct typing with respect to the type parameters—that is, conformance—by deriving constraints on those parameters at each call site and then attempting to find a consistent solution to the constraint equations. An inconsistent set of constraints would indicate a static type error.

Figure 1-1 lists an example program informally illustrating the parameter-inference process. The typechecking algorithm reconstructs the types of expressions bottom-up. Therefore, it begins by performing the inference process on the expression `double(C)` to find its type if it is well-defined. The formal argument `X` of `double` corresponds to

the actual argument C. The checker equates corresponding type parameters, deriving the constraint equations $r = i$ and $c = j$. Since for all i and j, there exist r and c which make these equations true, `double(C)` is provably well-typed regardless of what values `i` and `j` are instantiated with. Its type in the context of `test`'s body is `mat[2*i,2*j]`, which was obtained by applying the substitution denoted by the constraint equations to `double`'s result type, `mat[2*r,2*c]`.

```
type mat[int,int];

% Declare matrix-addition procedure
proc m_add[r,c](A,B: mat[r,c]) returns mat[r,c];

% Declare a procedure which takes a matrix,
% returning one twice as big
proc double[r,c](X: mat[r,c]) returns mat[2*r,2*c];

% Define a test procedure
proc test[i,j](C:mat[i,j],D:mat[2*i,2*j])
    returns mat[2*i,2*j]
{
    return m_add(double(C), D);
}
```

Figure 1-1: Example Program Requiring Parameter Inference

The next typechecking step is to perform the same process on the expression `m_add(double(C), D)`. Instantiating the types of `m_add`'s formal arguments `A` and `B` with the actual-argument types `mat[2*i,2*j]` and `mat[2*i,2*j]` gives us the constraint equations $r = 2i$, $c = 2j$, $r = 2i$, $c = 2j$. Like the system for `double(C)` above, this system is consistent for all `i` and `j`, so conformance is proven correct at compile time for all values of the parameters. The type of the expression `m_add(double(C), D)` which is returned by `test` is `mat[2*i,2*j]` as advertised by `test`'s signature, so `test`'s definition type-checks statically.

1.5 Related Work

Perhaps due to the specialized nature of the application, there has been little prior research on the specific problem of static conformance checking. Raymie Stata developed a type system that subsumes units-checking in typechecking [16]. His system bears a close resemblance to this one, incorporating units-polymorphism and units-inference. It differs significantly in the fact that the algebra of units only has the operations of multiplication, division, and equality, while the algebra of integers includes addition, subtraction, and the less-than and greater-than operators too. This difference makes the equation-solving problem for the inference algorithm more difficult.

Cardelli and Wegner [5] give a good general overview of type-system theory, including parametric polymorphism and type inference. Polymorphism and type inference were introduced to the language community by ML [13]. Milner [12] wrote the seminal paper defining ML's type system. Cardelli [4] explains this material at a more introductory level. Note that ML's style of polymorphism and type inference differs from the kind used in this thesis, as discussed below.

Gupta [8] discusses techniques for optimizing array-bound checks by taking advantage of dataflow information. Some of the possible extensions speculated on by Chapter 4 could profit from similar techniques.

Dependent types were introduced (for another purpose) by Martin-Löf's theory of intuitionistic types [11]. Cardelli [3] explains (in another context) the tradeoff between expressiveness and static typing that is involved in dependent typing.

1.5.1 Differences from ML Type Inference

The differences between this flavor of parameter inference and ML's have important consequences for the typechecking algorithm. Most importantly, our language requires that procedure parameters be declared explicitly and that the parameters be used in the types of the arguments of the procedure. ML doesn't require this—it infers formal *parameters* themselves, as well as their values for the actual parameters. This makes

the inference problem for ML much more ambitious in that respect. There are also some other differences:

- ML type variables are themselves types, so that ML parameterized types may be arbitrarily deep type graphs. Our type parameters are always integer-valued, so the structure of types is flat, not recursive. This allows the pattern-matching process we use to be much simpler than ML's, which must remember matched terms from other parts of the type graph.

- The algebra of ML types, the matched entities in ML inference, has only one operation, equality. However, the algebra of integers, used in our style of type inference, is much more complex, including operations for comparison, addition, multiplication, division, and conceivably others, in addition to equality. As the example program in Chapter 3 shows, the equality, addition, and multiplication operations are definitely useful in realistic programs; and subtraction, division, and inequality operators would also have a lesser usefulness. This implies that a sophisticated checker for our language would need to embody much more semantic knowledge than the ML typechecker, which only needs to decide whether types are equal.

1.6 Overview of the Thesis

Chapter 2 gives a formal definition of the language introduced by this chapter. It provides specifications for the type system and a typechecking algorithm carrying out those specifications. Chapter 3 evaluates this language ("DP") by assessing how well it satisfies its design goals in a case study of a practical program. The case-study example has been verified by a working typechecker that implements the specifications of Chapter 2. Chapter 4 proposes extensions to the DP type system. Chapter 5 describes some of the lessons that we have learned from the project.

Chapter 2

A Type System with Implicit Dependent Parameters

In its discussion of checking matrix programs, Chapter 1 referred to several well-known type-system features. By making matrix dimensions part of the type to improve safety, the discussion introduced a form of dependent typing. The chapter suggested using parametric polymorphism for procedures to increase the generality of the fine-grained types created, and discussed using type inference to make the resulting system more convenient without compromising its safety. A language incorporating these features was informally introduced in the course of the discussion.

This chapter will formally define the syntax and type system of the language introduced in Chapter 1. For convenience, we will refer to the language as "DP" ("dependent parameters"). The discussion begins by specifying the abstract syntax and some context-sensitive syntactic restrictions. It then gives a set of type rules and an algorithm for checking those rules at compile time.

Note that DP was designed for exploration of the typing issues at hand, not practical use. It therefore omits niceties important in real languages.

2.1 Abstract Syntax

2.1.1 Terminology

In this thesis, the term *procedure generator* refers to parameterized procedure declarations and definitions, while *procedure* denotes a ground procedure value derived from a procedure generator by instantiating the parameters.

Likewise, a *type generator* represents the family of all ground *types* with the same name but different instantiations for the parameters.

A *parameter* is a name that denotes integer values specializing procedure and type generators. The parameters of a procedure generator's signature represent integers (bound at compile time by the parameter-inference process) within that procedure generator's argument types. *Arguments*, on the other hand, are names which denote values instantiated within ground procedures at evaluation time.

2.1.2 Emphasized Language Features

Since DP was designed to explore dependent parameterization, that is the most completely elaborated feature in the language. In particular, a procedure declaration or definition includes a list of formal type parameters which represent integer values within the procedure's formal arguments and body. Procedure applications, on the other hand, do not explicitly specify any parameters in the syntax; they are not necessary because the compiler infers actual parameters at each call site. (As the case study in Chapter 3 will show, it is worthwhile to extend the language to *allow* explicit parameter instantiations, but the checker prototype built as part of the thesis project did not allow this.) Type generators which take 0 or more integer parameters are declared by the user at the beginning of each program. DP also includes the primitive types **bool** and **int**.

DP skimps on features not directly related to checking dependent types. To enable realistic evaluation of programs, it would be necessary to elaborate the type generator declarations mentioned above into an abstract type facility. This abstract type

facility would allow the definition of constructor procedures creating objects of abstract (and parameterized) type from arguments of primitive type. To avoid defining such a facility, DP permits procedures to be declared without giving definitions; this provision allows the modeling of type constructors for the purpose of typechecking but not evaluation. Procedure declarations also allow us to model primitive operators and library procedures. Another realistic feature omitted from DP is **while** loops, which could be added as syntactic sugar for lambda-lifted [9] tail-recursive procedures. The syntax does not distinguish a main procedure, which would be necessary to provide a code entry point for evaluation. We consider traditional (non-dependent) type parameters orthogonal to our dependent parameters and omit them as well.

2.1.3 Syntax Description

Figure 2-1 gives the abstract syntax of DP, which is imperative and block-structured.[1] A program consists of a sequence of type declarations, a sequence of procedure declarations, and a sequence of procedure definitions. Type declarations introduce new type generators into the program. Procedure declarations provide signatures but not bodies for procedure generators.

The signature for each procedure declaration and definition lists a number of P ("parameter") identifiers, which are formal names for the type parameters in the signature. These formal parameter names are bound in the argument types (*i.e.*, T's associated with the I's) and body statements of the procedures.

For example, in the procedure declaration for matrix multiplication given in figure 2-2, the formal parameters `r`, `c`, and `i` are bound in the argument types `matrix[r, i]` and `matrix[i, c]`.

A statement can be a block, an **if** statement, an assignment to a variable which has previously been declared as a formal argument or by a **var** construct, or a **return**

[1]The literature on type systems seems to favor example languages derived from the lambda-calculus rather than from Algol. We felt that, although following this tradition would make the syntax and type rules more elegant and concise, the block-structured form was closer to a language suited to practical work with matrices, systems, and the like. There would probably not be any fundamental differences in the type-system features under discussion if we translated this work to a functional language.

$$
\begin{array}{lcl}
\textit{program} & ::= & \textit{typedecl}^* \ \textit{procdecl}^* \ \textit{procdef}^* \\
\\
\textit{typedecl} & ::= & \textbf{type}\ I\,[\textbf{int}^*] \\
\\
\textit{procdecl} & ::= & \textbf{proc}\ I\,[P^*]\,(\ \{I : T\ \},^*)\ \textbf{returns}\ T; \\
\\
\textit{procdef} & ::= & \textbf{proc}\ I\,[P^*]\,(\ \{I : T\ \},^*)\ \textbf{returns}\ T\ \textit{stmt} \\
\\
\textit{stmt} & ::= & \{\ \{\textbf{var}\ I := \textit{exp};\ \}^*\ \textit{stmt}^*\} \\
& | & \textbf{if}\,(\textit{exp})\ \textit{stmt}_1\ \textbf{else}\ \textit{stmt}_2 \\
& | & I\ :=\ \textit{exp}; \\
& | & \textbf{return}\ \textit{exp}; \\
\\
\textit{exp} & ::= & I \\
& | & P \\
& | & I\,(\textit{exp}^*) \\
& | & \textit{intlit} \\
& | & \textit{boollit} \\
\\
T' & ::= & T \\
& | & \textbf{proctype}\,[P^*]\,(T^*)\ \rightarrow T \\
\\
T & ::= & \textbf{int} \\
& | & \textbf{bool} \\
& | & I\,[\textit{term}^*] \\
\\
\textit{term} & ::= & \textit{intlit} \\
& | & P \\
& | & \textit{term op term} \\
& | & (\ \textit{term}\ ) \\
\\
\textit{op} & ::= & * \ |\ /\ |\ +\ |\ - \\
\\
P & ::= & I \\
\\
I & ::= & \textit{identifier}
\end{array}
$$

Figure 2-1: Abstract Syntax

```
proc mmult[r,c,i: int](A: matrix[r,i], B: matrix[i,c])
    returns matrix[r,c];
```

Figure 2-2: Syntax Example: Procedure-Generator Parameters

statement. A block is a sequence of bindings of block-local variables to initial expressions, followed by a sequence of statements. The types of variable names declared by **var** constructs are reconstructed from the initial expressions. Assignments can be made only to variables, not parameters, since parameters are considered constant within each application.

Expressions can be variable or parameter references, procedure calls, or integer or boolean literals. Note again that calls explicitly instantiate arguments, but not parameters, which are inferred. Procedures are not first-class and hence cannot be expressions.

There are two primitive types, **int** and **bool**. All other types are denoted by type generators instantiated with 0 or more value terms, where a value term is a restricted integer expression whose identifiers are type parameters. Type generators must be declared by *typedecl*'s at the beginning of the program; a *typedecl* gives the name and number of parameters of a type generator. The abstract syntax includes a **proctype** type-expression which is used internally in the typing rules but is not allowed as the type of arguments or variables (*i.e.*, procedures are not first-class).

2.1.4 Scope and Naming Issues

Procedure definitions have global, mutually recursive (*i.e.* "letrec") scope. Type generator names have global scope. Local variables introduced by blocks have "let" scope. All identifiers must be declared before use. In particular, this includes the requirement that parameter names in a procedure generator's arguments and body be bound by the procedure generator's signature, which will have important implications for the parameter inference process. Multiple declarations of the same name in the same scope are illegal. Declarations of variables in inner scopes shadow declarations in

outer ones. There are separate namespaces for type generators, procedure generators, parameters, and variables. Names used as expression rvalues may be either variables or parameters; variables take precedence over parameters in name conflicts.

2.1.5 A Syntactic Restriction on Signatures

We require that each formal parameter of a procedure generator appear by itself (*i.e.*, as an atomic term) in the type of at least one formal argument. This means, for example, that we would disallow the signature

```
proc foo [i, j] (A: matrix[i+2*j, i+j]) returns matrix[i, j];
```

because neither `i` nor `j` appear by themselves in the types of `foo`'s arguments. Note that the same relation between input and output types can be expressed by the change of variables $y = i + 2j$ and $z = i + j$, producing

```
proc foo [y, z] (A: matrix[y, z]) returns matrix[2*z-y,y-z];
```

As subsection 2.3.5 explains, this restriction greatly simplifies the typechecking algorithm. We feel that it should not inconvenience programmers because we canot find realistic signatures in which the generalized form seems more natural.

2.2 Type-Correctness Rules

This section specifies the set of rules for proving that programs are type-correct. Conformance of dimensional parameters is included in our notion of type correctness.

In the type rules, "type assumptions" (also known as "type environments") are partial functions, represented by sets of ordered pairs of identifiers and types, mapping identifiers to types. Formally, type assumptions are subsets of the set

$$\{(I,T) \mid I \in \mathit{Identifiers} \wedge T \in \mathit{Types}\}.$$

The operator $+$ denotes environment extension:

$$A + A' = \{(I,T) \mid ((I,T) \in A \wedge (I,T) \notin \wedge A') \vee (I,T) \in A'\}$$

We also define

$$\sum_{i=1}^{n} A_i = A_1 + \cdots + A_n$$

The notations $\vec{x}$ and $\vec{y}$ denote vectors of parameter terms; the notation $\vec{P}$ denotes a vector of parameter names.

Since DP includes statements as well as expressions, the typechecking rules include both rules asserting that a statement is type-correct and rules asserting that an expression has a certain type. For statements (and the procedures and programs built up from them), the predicates *StmtOK*, *ProcdefOK*, and *OK* denote the assertions that statements, procedure generator definitions, and programs are type-correct. For an expression E and a type T, the notation $E{:}T$ denotes the assertion that E has type T. The rules for proving type-correctness of statements are derived from [1].

2.2.1 Type-Correctness of Programs

Figure 2-3 gives the rule for proving that programs are type-correct. We can show a program is type-correct by proving the type-correctness of each procedure definition in a type environment extended with type bindings for all the procedures in the program. This simultaneous extension of the environment reflects the mutually recursive scope of procedure definitions. *SigToName* and *SigToType* are simple syntactic macros which express the extraction of procedure names and types from their declarations and definitions.

2.2.2 Type-Correctness of Procedure-Generator Definitions

Figure 2-4 gives the rule for typechecking definitions of procedure generators. This rule is complicated by the fact that each formal parameter of a procedure generator may be instantiated with any value by a caller. We express this requirement in terms of the substitution of a universally-quantified vector of parameter values $\vec{x}$ for the vector of formal parameters $\vec{P}$. Thus, the rule is that a procedure-generator definition type-checks if its body type-checks for all possible values of the procedure generator's parameters.

$$\frac{\begin{array}{c}\mathrm{A}' = \mathrm{A} + \sum_{i=1}^{n} \{(\mathrm{SigToName}(\mathit{procdecl}_i), \mathrm{SigToType}(\mathit{procdecl}_i))\} \\ \mathrm{A}'' = \mathrm{A}' + \sum_{i=1}^{p} \{(\mathrm{SigToName}(\mathit{procdef}_i), \mathrm{SigToType}(\mathit{procdef}_i))\} \\ \bigwedge_{i=1}^{p} \mathrm{A}'' \vdash \mathrm{ProcdefOK}(\mathit{procdef}_i)\end{array}}{\mathrm{A} \vdash \mathrm{OK}(\mathit{typedecl}_1 \ldots \mathit{typedecl}_m\ \mathit{procdecl}_1 \ldots \mathit{procdecl}_n\ \mathit{procdef}_1 \ldots \mathit{procdef}_p)}$$

where

$$\mathrm{SigToName}\,(\mathbf{proc}\, I\,[P_1, \ldots, P_m : \mathbf{int}](I_1 : T_1, \ldots, I_n : T_n)\,\mathbf{returns}\, T_{body}) = I$$

$$\mathrm{SigToType}\,(\mathbf{proc}\, I\,[P_1, \ldots, P_m : \mathbf{int}](I_1 : T_1, \ldots, I_n : T_n)\,\mathbf{returns}\, T_{body}) = \mathbf{proctype}\,[P_1, \ldots, P_m](T_1, \ldots, T_n) \rightarrow T_{body}$$

Figure 2-3: Typing Rule for Programs

$$\frac{\forall \vec{\mathbf{x}} \in \mathbf{int}^m \cdot \mathrm{A} + \sum_{i=1}^{n} \{(I_i, [\vec{\mathbf{x}}/\vec{\mathbf{P}}]\, T_i)\} \vdash \mathrm{StmtOK}\,([\vec{\mathbf{x}}/\vec{\mathbf{P}}]\mathit{stmt}, [\vec{\mathbf{x}}/\vec{\mathbf{P}}]\, T_{body})}{\mathrm{A} \vdash \mathrm{ProcdefOK}\,(\mathbf{proc}\, I\,[P_1, \ldots, P_m](I_1 : T_1, \ldots, I_n : T_n)\,\mathbf{returns}\, T_{body}\ \mathit{stmt})}$$

where

$$\vec{\mathbf{P}} = P_1, \ldots, P_m$$

Figure 2-4: Typing Rule for Procedure Definitions

Imposing this universal quantification in the rule for procedure definitions enables separate compilation of procedures. We check procedure bodies once for all possible instantiations rather than rechecking them relative to each specific instantiation in the program. Procedure calls can then be checked with knowledge of the callee's signature but not its body.

2.2.3 Type-Correctness of Statements

Figure 2-5 gives the rules for typing statements. The *StmtOK* predicate includes a second argument which specifies the return type expected if the statement is or includes a **return**. A block is type-correct if its statements type check in an environment extended with bindings for each local variable declared in a **var** construct. (The types of the local variables are reconstructed by type checking their initializer expressions). An **if** statement is type correct if its predicate has type **bool** and its consequent statement is type-correct. An assignment is type-correct if the type of the assigned expression is identical to the type of the assigned variable. A **return** statement is type-correct if the type of the returned expression is equal to the type declared by the signature of the enclosing procedure definition.

2.2.4 Provable Types of Expressions

Figure 2-6 gives the rules for typing expressions. The rules for literals are axioms. Variable references have the types bound to the variables in the type environment, and parameter references always have integer type.

The interesting rule is the one for procedure applications. The complication in this rule arises because the actual-argument types are parameterized by the caller's parameters and the formal-argument types are parameterized by the callee's formal parameters. We substitute the vector of actual-parameter terms $\vec{\mathbf{y}}$ for the vector of formal-parameter names $\vec{\mathbf{P}}$ in the types of the callee's formal arguments T_i. Actual parameters are inferred rather than explicitly instantiated; this leads to the existential quantification of $\vec{\mathbf{y}}$. That is, procedure calls have a provable type if there exists a set

$$\frac{\bigwedge_{i=1}^{m} \mathrm{A} \vdash exp_i : T_i \quad \mathrm{A} + \sum_{i=1}^{m} \{(I_i, T_i)\} \vdash \mathrm{StmtOK}(stmt_i, T_{return})}{\mathrm{A} \vdash \mathrm{StmtOK}(\{\mathbf{var} I_1 := exp_1; \ldots; \mathbf{var} I_m := exp_m;\ stmt_1; \ldots; stmt_n;\}, T_{return})}$$

$$\frac{\mathrm{A} \vdash exp : \mathbf{bool} \quad \mathrm{A} \vdash \mathrm{StmtOK}(stmt, T_{return})}{\mathrm{A} \vdash \mathrm{StmtOK}(\mathbf{if}(exp)\ stmt, T_{return})}$$

$$\frac{\mathrm{A} \vdash exp : \mathbf{bool} \quad \mathrm{A} \vdash \mathrm{StmtOK}(stmt_1, T_{return}) \quad \mathrm{A} \vdash \mathrm{StmtOK}(stmt_2, T_{return})}{\mathrm{A} \vdash \mathrm{StmtOK}(\mathbf{if}(exp)\ stmt_1\ \mathbf{else}\ stmt_2, T_{return})}$$

$$\frac{\mathrm{A} \vdash I : T \quad \mathrm{A} \vdash exp : T}{\mathrm{A} \vdash \mathrm{StmtOK}(I := exp;, T_{return})}$$

$$\frac{\mathrm{A} \vdash exp : T_{return}}{\mathrm{A} \vdash \mathrm{StmtOK}(\mathbf{return}\ exp;, T_{return})}$$

Figure 2-5: Typing Rules for Statements

$$\vdash \mathit{intlit} : \mathbf{int}$$

$$\vdash \mathit{boollit} : \mathbf{bool}$$

$$\{..., (I, T), ...\} \vdash I : T$$

$$\vdash P : \mathbf{int}$$

$$\frac{\exists \vec{\mathbf{y}} \in \mathbf{int}^m \cdot \bigwedge_{i=1}^{n} \mathrm{A} \vdash \mathit{exp}_i : [\vec{\mathbf{y}}/\vec{\mathbf{P}}]\, T_i \qquad \mathrm{A} \vdash I : \mathbf{proctype}[P_1, \ldots, P_m](T_1, \ldots, T_n) \rightarrow T_{body}}{\mathrm{A} \vdash I(\mathit{exp}_1, \ldots, \mathit{exp}_n) : [\vec{\mathbf{y}}/\vec{\mathbf{P}}]\, T_{body}}$$

where

$$\vec{\mathbf{P}} = P_1, \ldots, P_m$$

Figure 2-6: Typing Rules for Expressions

of actual parameters which allow the formal and actual argument types to match. If it exists by that criterion, the type of a call expression is the result type of the callee with actual parameters substituted for formal ones.

Note that the precondition side of the procedure-call rule does not constrain the expected type T_{body} of the call expression. This makes DP's flavor of type inference different from ML type inference [12], which propagates constraints on type variables more globally through programs. We feel that this choice enhances modularity when reasoning about DP programs and simplifies the checking algorithm by making the propagation of type constraints travel exclusively up the syntax tree.

2.3 Typechecking Algorithm

To implement a compile-time type checker, we require a algorithm which determines whether or not the rules in Section 2.2 can be used to prove that syntactically correct programs (where syntactic correctness includes the restrictions in subsections 2.1.4 and 2.1.5) are type-correct. Fortunately, the rules in Section 2.2 are syntax-directed; thus, this algorithm can be structured as a type evaluator that propagates expression types and results of OK-predicates up the syntax tree. The evaluator performs a depth-first traversal of the parse tree, constructing type environments on its way down and examining the types that came up from the lower-level nodes on its way up.

The only potentially problematic parts of the rules are the existential and universal quantification of parameters. As subsection 2.3.1 will show, our syntactic restriction on signatures takes care of the existential quantifiers in the application rule. However, we can't always get rid of the universal quantifiers in the abstraction rule at compile time; some checking must be deferred until runtime, when there is enough information to deal with this problem.

2.3.1 Algorithm for Expressions

Since the checker collects typing information about the the syntax tree bottom-up, we begin this discussion by specifying its behavior for expressions. Because the rules for typing literal expressions and variable-reference expressions are straightforward, it is also straightforward to write an algorithm which determines what type, if any, the rules can prove for such expressions. Figure 2-7 gives the definition, for these simple cases, of the expression-checking function $\mathcal{C}_e$. $\mathcal{C}_e$ takes a type environment and an expression and normally returns a type and set of *runtime checks*. (The set of runtime checks is always empty for these simple cases, so we delay the detailed explanation of runtime checks until it becomes relevant below.) $\mathcal{C}_e$ can also return **error**, which indicates that a type error has been detected statically. The figure assumes that the types **bool** and **int** are sugar for the types **bool**[] and **int**[]. The runtime checks

$$\mathcal{C}_e(A, \mathit{intlit}) = \langle \mathbf{int}[], \{\} \rangle$$

$$\mathcal{C}_e(A, \mathit{boollit}) = \langle \mathbf{bool}[], \{\} \rangle$$

$$\mathcal{C}_e(A, P) = \langle \mathbf{int}[], \{\} \rangle$$

$$\mathcal{C}_e(A, I) = \begin{cases} \mathbf{error} & \text{if } A \text{ undefined at } I \\ \langle T[\alpha_1, \ldots, \alpha_n], \{\} \rangle & \text{if } A(I) = T[\alpha_1, \ldots, \alpha_n] \end{cases}$$

Figure 2-7: Definition of $\mathcal{C}_e$ for Literals and Variable References

encode typechecking verifications that must be deferred until evaluation.

The restriction from subsection 2.1.5 that each formal parameter occur by itself as the term of a formal-argument type makes finding a parameter of the caller to substitute for each parameter of the callee trivial. This can be done by performing syntactic pattern-matching between the callee's formal-argument types and the reconstructed types of the actual-argument expressions. Each atomic term defines the substitution for one of the callee's formal parameters. Since an atomic term must exist for each formal parameter, the substitution is always fully defined.

However, there may be additional matches to make between caller's and callee's parameters because the callee can have more terms in its argument types than parameters. These additional matches become additional constraints that must be met in order for the typing decision for the call expression to be sound. We can transform these additional constraints into equations in terms of the caller's parameters by applying the substitution derived above to the callee's formal parameters.

Unfortunately, these additional constraint equations cannot be verified until runtime because there is no way to prove in general that they hold for all possible instantiations of the caller's formal parameters. However, once the actual parameters are known for any particular call, it's easy to determine whether the constraints are met. Hence, the additional constraints become runtime checks. Failure to satisfy a runtime check is a type error because the static typing-determination is unsound under those circumstances.

$$\frac{\begin{array}{c} A(I) = \textbf{proctype}[p_1, \ldots, p_m](T_1', T_2') \rightarrow R[\phi_1, \ldots, \phi_k] \\ T_1' = T_1[p_1, \ldots, p_m, \beta_1, \ldots, \beta_q] \\ T_2' = T_2[\beta_{q+1}, \ldots, \beta_n] \\ \langle T_1[\alpha_1, \ldots, \alpha_{m+q}], C_1 \rangle = \mathcal{C}_e(A, e_1) \\ \langle T_2[\alpha_{m+q+1}, \ldots, \alpha_{m+n}], C_2 \rangle = \mathcal{C}_e(A, e_2) \\ C = \{\alpha_{m+i} = [\alpha_j / p_j]\beta_i\} \cup C_1 \cup C_2 \\ \theta_i = [\alpha_j / p_j]\phi_i \end{array}}{\mathcal{C}_e(A, I(e_1, e_2)) = \langle R[\theta_1, \ldots, \theta_k], C \rangle}$$

Figure 2-8: Definition of $\mathcal{C}_e$ for Call Expressions

Figure 2-8 formalizes the above discussion in terms of $\mathcal{C}_e$. To simplify notation, this figure does not consider the fully general invocation. It makes the following assumptions:

- The callee takes two arguments.
- The parameters can be found in the leftmost positions of the first argument type in sorted order.

It should be obvious how this rule generalizes. In the figure, p_1 through p_m are the formal-parameter names of the callee. The ϕ's and β's are integer expressions in terms of the p's. The α's are integer expressions in terms of the caller's formal parameters.

The algorithm is expressed in terms of pattern-matching rules. The rules attempt to destructure the call expression passed to $\mathcal{C}_e$ into a form allowing the pattern-matching process described above to take place on the call expression's constituents. If the call expression does not destructure into the specified form the result of $\mathcal{C}_e$ is **error**.

2.3.2 Algorithm for Statements

The typechecking algorithm for statements, like the algorithm for literal expressions and variable-reference expressions, is fairly straightforward since the statement-typing rules are straightforward. The only thing $\mathcal{C}_s$ has to do is extend the type environment for subnodes, check subnodes, and collect the runtime checks from subnodes.

As in the definition of $\mathcal{C}_e$ for call expressions, $\mathcal{C}_s$ is presented in terms of pattern-matching rules. The rules define $\mathcal{C}_s$ over all inputs if we specify that failure of a pattern-match, which occurs if the result of applying $\mathcal{C}_e$ or $\mathcal{C}_s$ above the horizontal bar is **error**, causes the application of $\mathcal{C}_s$ below the bar to yield **error** as well. Figure 2-9 gives $\mathcal{C}_s$. Note that $\mathcal{C}_s$ takes the expected return type as an additional input.

$$\frac{\begin{array}{c}\langle T_i, C_i\rangle = \mathcal{C}_e(A, T_r, e_i) \\ C_j = \mathcal{C}_s(A + \sum\{(I_i, T_i)\}, T_r, s_j) \\ C = (\bigcup_i C_i) \cup (\bigcup_j C_j)\end{array}}{\mathcal{C}_s(A, T_r, \textbf{var } I_i = e_i; \ldots; s_i; \ldots) = C}$$

$$\frac{\begin{array}{c}\langle \textbf{bool}[], C_t\rangle = \mathcal{C}_e(A, e) \\ C_c = \mathcal{C}_s(A, T_r, s_c) \\ C_a = \mathcal{C}_s(A, T_r, s_a) \\ C = C_t \cup C_c \cup C_a\end{array}}{\mathcal{C}_s(A, T_r, \textbf{if } e\ s_c\ \textbf{else}\ s_a) = C}$$

$$\frac{\begin{array}{c}A \vdash I : T \\ \langle T, C\rangle = \mathcal{C}_e(A, e)\end{array}}{\mathcal{C}_s(A, T_r, I := e) = C}$$

$$\frac{\begin{array}{c}\langle T[\alpha_1, \ldots, \alpha_n], C'\rangle = \mathcal{C}_e(A, e) \\ C = C \cup \alpha_i = \theta_i\end{array}}{\mathcal{C}_s(A, T[\theta_1, \ldots, \theta_n], \textbf{return } e) = C}$$

Figure 2-9: Definition of $\mathcal{C}_s$

2.3.3 Algorithm for Procedure Generators and Programs

The algorithm that determines whether a procedure generator type-checks is to apply $\mathcal{C}_s$ to the procedure generator's body statement and expected return type. If this does not result in **error** then we conclude that the procedure generator's body is faithful to its signature contingent on runtime verification of the runtime checks that $\mathcal{C}_e$ returns.

The algorithm for programs is simply to apply the algorithm for procedure gener-

```
type matrix[int, int];
proc mmul[r,c,i](A:matrix[r,i], B:matrix[i,c])
    returns matrix[r,c];

proc wrapper[x,y,z](D:matrix[x,z], E:matrix[z,y])
    returns matrix[x,y]
{
    var tmp := mmul(D, E);
    if (true) { }
    return tmp;
}
```

Figure 2-10: Trivial Example Program

ators over each procedure-generator definition in the program in a type environment that assumes that each signature in the program is correct. This corresponds precisely to the typing rule for programs.

Since type errors, including failures of parameter inference detectable at compile time, are detected at the lowest levels of the syntax tree, error messages provided by a checker can provide quite useful information: they can cite the line number of the erroneous expression and name the kind of problem (*i.e.*, non-matching type generators for formal and actual arguments, provably inconsistent parameter equations, or wrong numbers or arguments). The checker implementation built for the thesis project does provide such localized error messages.

2.3.4 Typechecking Example

To make the preceding discussion concrete, we trace the typechecking algorithm as it verifies the type-correctness of the example DP program in figure 2-10. To prove the OK predicate for the program, we attempt to prove the ProcdefOK predicate for the procedure definition of **wrapper** in the type environment

mmul: proctype[r,c,i](matrix[r,i],matrix[i,c]) → matrix[r,c]

wrapper: proctype[x,y,z](matrix[x,z],matrix[z,y]) → matrix[x,y]

To prove the ProcdefOK predicate for **wrapper**, we must prove

StmtOK(body of **wrapper**, matrix[x,y])

in the type environment

mmul: proc[r,c,i](matrix[r,i],matrix[i,c]→matrix[r,c]

wrapper: proc[x,y,z](matrix[x,z],matrix[z,y]→matrix[x,y]

D: matrix[x,z]

E: matrix[z,y]

This reduces to proving the type correctness of the **if** statement and the type correctness of `return tmp;` in a type environment extended with a binding of `tmp` to the type of the expression `mmul(D,E)`.

Reconstructing the type of `mmul(D, E)`, within the above type environment, we derive a system of equations between formal and actual parameters by performing pattern matching between the formal-argument and actual-argument types of `mmul` as follows: The instantiation of formal argument A with actual argument D gives us the type equation

matrix[r,i] = matrix[x,z]

and the instantiation of formal argument B with actual argument E gives the type equation

matrix[i,c] = matrix[z,y]

Pattern matching the parameters in these type equations yields the system of equations between the callee's formal parameters and actual parameters

$$\forall x, y, z \cdot \exists r, i, c \cdot \begin{cases} r = x \\ i = z \\ i = z \\ c = y \end{cases}$$

The actual parameters x, y, and z are universally quantified, while the formal parameters r, i, and c are existentially quantified, as a consequence of the corresponding quantifications in the type rules. These quantifiers can be eliminated by thinking of the existentially quantified parameters r,i, and c as "variables" and of the universally quantified parameters x,y, and z as "constants." We choose one of the equations with i on the left-hand-side to construct the substitution $r \rightarrow x$, $i \rightarrow z$, $c \rightarrow y$. Now we apply that substitution to the left-hand-side of the redundant equation $i = z$ to get the additional check-equation $z = z$. We also derive the type of the call expression `mmul(D,E)` by substituting the actual parameters x, z, and y for the formal parameters r, i, and c in `mmul`'s result type matrix[r,c] to yield the type matrix[y,z].

Now we bind the type matrix[y,z] to the identifier `tmp` to check the statement `return tmp;` in the type environment

mmul: proc[r,c,i](matrix[r,i],matrix[i,c]→matrix[r,c]

wrapper: proc[x,y,z](matrix[x,z],matrix[z,y]→matrix[x,y]

D: matrix[x,z]

E: matrix[z,y]

tmp: matrix[y,z]

Since `tmp`'s type matches the return type expected by `wrapper`, and the `if` statement type-checks, the definition of `wrapper` type-checks. Therefore the program type-checks, with the provisos that we have collected the runtime check-equation $z = z$ that resulted from the call `mmul(D,E)` and placed it at the beginning of `wrapper`'s body, and that this equation holds true when execution reaches that point in the code. This is one example of a "runtime" check that is easy for the compiler to eliminate statically with a small amount of extra work.

Note that the checker must maintain a distinction between the formal parameters on the left-hand-side of each equation and the actual parameters on the right-hand-side. The need for result types to be expressed only in terms of actual parameters is one reason for this distinction. The need to avoid name conflicts between formal and actual parameters is another reason.

If we had not been able to optimize away the check equations at compile time, it would be necessary to leave some of them in the compiled code to be confirmed at runtime. For example, if we revised the example program to look like the one in figure 2-11, we would need to insert the run-time check $x = y$ in the code generated for `wrapper2` before the call to `mmul(D,E)`.

```
type matrix[int, int];
proc mmul[r,c,i: int](A:matrix[r,i], B:matrix[i,c])
    returns matrix[r,c];

proc wrapper2[w,x,y,z: int](D:matrix[w,x], E:matrix[y,z])
    returns matrix[w,z]
{
    return mmul(D, E);
}
```

Figure 2-11: Example of Program Requiring Runtime Parameter Checks

2.3.5 Pragmatic Considerations in Typechecking

The checking algorithm reduces the problem of proving the rules of Section 2.2 to the problem of determining whether a system of nonlinear, integer equations is inconsistent. We perform this consistency check at runtime for each instantiation by inserting runtime checks into the compiled code which raise runtime errors if they are not satisfied. This guarantees that all code that is executed is type-correct, because a runtime error would be raised before the execution of any code with unsound typing judgments.

The computational complexity of typechecking is polynomial in the length of the program text. Let M denote the number of syntax-tree leaves (*i.e.*, literals, variable-references, and calls that do not contain other calls) of a program, and let N denote the maximum number of parameters appearing in any procedure-generator's signature. The checking algorithm traverses down the syntax tree, collects reconstructed types at the leaves, and then traverses back up to verify that no error occurred at any leaf. Hence each leaf is visited exactly once. At each leaf (call expression), the checker

extracts at most M equations on parameters, splitting them into the set of "defining occurrences" and "extra equations." This takes $O(M)$ time since each equation is extracted by a constant-time pattern match. Multiplying, the overall time complexity of typechecking is $O(MN)$. Since the length of the program text is proportional to M, the time complexity of typechecking can also be expressed as $O(N\times$ (length of program text)).

It should be clear from the above discussion why the syntactic restriction on procedure signatures in subsection 2.1.5 simplifies the checking algorithm. Since each formal parameter must appear in an atomic term in at least one formal argument type, the parameter-equation set will automatically include an already-reduced substitution for each formal parameter, with that formal parameter alone on the left-hand-side. If a procedure application has more constraint equations than parameters, then additional equations (possibly non-atomic on the left-hand-side) will be inserted into the compiled program as runtime conformance checks.

If DP did not include this restriction, then the algorithm for determining the existence of consistent parameter substitutions (*i.e.*, the algorithm for checking procedure calls) would need to find integer solutions to possibly nonlinear equations. This is the decision problem for integer diophantine equations, also known as Hilbert's Tenth Problem, which is undecidable [6].

One optimization for the typechecker would be to simplify or eliminate the residual checks statically. This optimization opportunity reduces to the problem of reducing nonlinear, integer equations.

Chapter 3

Application Case Study

This chapter evaluates how well, in practice, DP meets the needs for safety, convenience, and flexibility outlined in Chapter 1. We will argue that the DP typing constructs successfully make programs cleaner and safer, though sometimes at a cost in convenience. We will also show that the type system lacks the power to handle certain ideas that could benefit from static verification.

We use the LQR (linear-quadratic regulator design) algorithm from control engineering, described below in Section 3.1, as the example for the study. The original code for LQR is given in the control and linear-systems toolbox of MATLAB [17, 15], a commercial tool for numerical computation. This example is well suited for practical evaluation of DP. It is widely used to do practical work in an engineering discipline, so its length and complexity should be representative of procedures used in real life to compute with matrices.

The chapter begins by providing a little background on the LQR algorithm and briefly describing its original implementation in the MATLAB toolbox. It then gives a translation of the algorithm to DP. This translation was checked in the exact form given in this chapter by a typechecker implementing the specifications of Chapter 2. Strengths and weaknesses of the type system are evaluated in terms of the quality of the fit between the type system and the algorithm's computational notions.

3.1 The LQR Algorithm

The LQR algorithm is a widely-used algorithm for designing control systems [2]. It takes four matrix inputs: A, B, Q, and R. A and B represent the physical system to be controlled in terms of its state space, given by a linear differential equation of the form

$$\frac{d}{dt}\mathbf{x} = A\mathbf{x} + B\mathbf{u}$$

where $\mathbf{x}$ is a vector representing the state of the physical system and $\mathbf{u}$ is a vector representing inputs to the system.

The matrices Q and R are design parameters. It is possible to give them physical interpretations as weight matrices in a cost function. In practice, however, designers usually treat this physical interpretation loosely, trying many different Q and R in the process of finding a design that meets their criteria. Hence, the LQR algorithm is used inside an outer loop, so factors influencing performance, including runtime conformance checks, are worth optimizing. (LQR isn't in the innermost loops, though—the eigenvalue decomposition algorithm is.)

The output of the LQR algorithm is a constant matrix K that is used to scale the state of the system in a full-state feedback loop; thus, the closed-loop differential equation for the system is

$$\frac{d}{dt}\mathbf{x} = (A - BK)\mathbf{x}.$$

3.2 MATLAB Implementation of LQR

A slightly simplified version of the MATLAB LQR script is given in figure 3-1. (Note that the line numbers are not part of the script.) The complete original listing can be found in Appendix A.

The script begins by performing conformance checking on the dimensions of the arguments in an ad-hoc manner in lines 3–17. Although these checks look verbose, they amount to confirming that A has dimensions $n \times n$, B has dimensions $n \times nb$, Q has dimensions $n \times n$, and R has dimensions $nb \times nb$. These dimensions make

```
 1 function [k,s,e]=lqr(a,b,q,r)
 2
 3 error(abcdchk(a,b));
 4 if ~length(a) | ~length(b)
 5         error('A and B matrices cannot be empty.')
 6 end
 7
 8 [m,n] = size(a);
 9 [mb,nb] = size(b);
10 [mq,nq] = size(q);
11 if (m ~= mq) | (n ~= nq)
12 error('A and Q must be the same size');
13 end
14 [mr,nr] = size(r);
15 if (mr ~= nr) | (nb ~= mr)
16 error('B and R must be consistent');
17 end
18
19 % Check if q is positive semi-definite and symmetric
20 nq = norm(q,1);
21 if any(eig(q) < -eps*nq) | (norm(q'-q,1)/nq > eps)
22 disp('Warning: Q is not symmetric and positive semi-definite');
23 end
24 % Check if r is positive definite and symmetric
25 nr = norm(r,1);
26 if any(eig(r) <= -eps*nr) | (norm(r'-r,1)/nr > eps)
27 disp('Warning: R is not symmetric and positive definite');
28 end
29
30
% Start eigenvector decomposition by finding eigenvectors of Hamiltonian:
31 [v,d] = eig([a b/r*b';q, -a']);
32 d = diag(d);
33 [e,index] = sort(real(d));  % sort on real part of eigenvalues
34 if (~( (e(n)<0) & (e(n+1)>0) ))
35 error('Can''t order eigenvalues, (A,B) may be uncontrollable.');
36 else
37   e = d(index(1:n));  % Return closed-loop eigenvalues
38 end
39 chi = v(1:n,index(1:n));  % select vectors with negative eigenvalues
40 lambda = v((n+1):(2*n),index(1:n));
41 s = -real(lambda/chi);
42 k = r\(b'*s);
```

Figure 3-1: MATLAB Implementation of LQR Algorithm (Simplified)

sense physically if we consider that nb is the number of inputs and n is the number of states.

The script continues by checking that Q and R are positive definite (actually, positive semi-definite in Q's case) and symmetric, applying numerical conditions based upon the norms and eigenvalues of Q and R and printing runtime warnings if this condition is not met.

The algorithm proper does not start until line 31, which finds the eigenvector decomposition of the "Hamiltonian matrix"—a $2n \times 2n$ matrix formed by "gluing" together four $n \times n$ submatrices. The script sorts the eigenvectors and corresponding eigenvalues by the real parts of the eigenvalues and selects out the eigenvalue/eigenvector pairs whose eigenvalues have negative real parts (a numerical condition for stability). In the process, it checks if there are enough stable eigenvalues. If all is well, the script computes K as well as the auxiliary matrix S and auxiliary vector of eigenvalues **e**. Computing these final results requires permuting columns of matrices and elements of vectors according to the sorted ordering computed in line 33. This shuffling of matrix components is expressed by the array indexing operators in lines 37, 39, and 40. The notation `d(index(1:n))` in line 37 denotes the permuted vector

$$\left[D_{index_1}, \cdots, D_{index_n}\right],$$

and the notation `v(1:n,`*index*`(1:n))` denotes the permuted matrix

$$\begin{bmatrix} V_{1,index_1}, \cdots, V_{1,index_n} \\ \ddots \\ V_{n,index_1}, \cdots, V_{n,index_n} \end{bmatrix}.$$

Note that this notation also specifies a subsectioning operation that selects $s \times s$ matrices from $2s \times 2s$ ones.

3.3 DP Translation of the MATLAB LQR Script

This section gives a handwritten translation of the above MATLAB script to DP. The resulting script expresses the constraints among dimensional parameters more clearly than the MATLAB script does, providing better documentation to the human reader and greater safety through compile-time checking. The DP program given here has been passed as shown through the checker implementation, which discharged all the conformance checks (though not the bounds checks, as we will see) statically. This provides confirmation of the DP type system's usefulness for practical applications.

The example also points out some awkward aspects of the translation that result from the fact that DP omits all features irrelevant to parameter checking. Subsection 3.3.2 explains why these aspects would not be problematic were the interesting DP features integrated into a production language.

Finally, the example points out some genuine limitations of the type system which cannot be solved by simple fixes; subsection 3.3.3 explains what the fundamental problems are.

3.3.1 Successful Aspects of DP

Figures 3-2 and 3-3 list the main DP source file for the LQR algorithm. The most noteworthy feature is the exposure of conformance requirements in `lqr`'s signature. This has two major benefits. First, the parameters in the signature compactly document the conformance requirements for procedure generators. In the example, the signature highlights the fact that there are two dimensional parameters of interest: `ip`, the number of inputs, and `sp`, the number of states. Moreover, it makes the conformance requirements for the arguments A, B, Q, and R immediately apparent in terms of `ip` and `sp`. Contrast this clarity with the lack of information about conformance requirements given by the MATLAB LQR script. The signature there gives *no* information about conformance requirements. Neither, in fact, does the long

introductory comment in the original script (see Appendix A). It is necessary to read the error-handling code in lines 3–17 of figure 3-1 to discern the conformance requirements. These hand-written checks are verbose and idiosyncratically written; hence, significantly more effort is necessary to deduce the conformance requirements from them. Also, hand-written conformance checks like those are a likely source of bugs since they involve repetitive and bothersome details, especially compared to the more compact form encoded in the DP signature.

The second major benefit of exposing conformance requirements in the signature is that it allows the conformance check to be moved to the call site from the start of the procedure-generator definition. This allows the compiler to perform a deeper analysis of conformance properties; in fact, it opens up the possibility of eliminating the check altogether. The checker implementation succeeded very well with this on the LQR program; it was able to optimize away *all* of the conformance checks by applying the algorithm of Section 2.3 and then eliminating check-equations between syntactically identical parameter expressions.

3.3.2 Insignificant Problems in the Example

There are some awkward constructs in this translation that stem solely from DP's exclusion of features not directly related to parameter checking. Integrating these well-known features would not involve difficult interactions with the dependent parameters:

- Since there are no builtin operators, we simulate them with procedure-generator declarations; however, this leads to a more verbose expression syntax. For example, the check in lines 11–13 that Q and R are positive definite and symmetric is delegated to the helper procedures `is_posdef` and `is_symmetric` merely to cope with this verbosity. Adding operator overloading would make programs more concise but would pose no difficulty with dependent types since overloading would resolved among the type *generators* of arguments, not the type *parameters*. Dependent parameterization would be orthogonal to overloading.

```
 1 % Translation of the MATLAB lqr script to DP.
 2
 3 #include "matlab_builtins.h"
 4
 5 proc lqr [ip,sp](a:mat[sp,sp], b:mat[sp,ip], q:mat[sp,sp], r:mat[ip,ip])
 6      returns mat[ip,sp]
 7 % parameter ip is the number of system inputs
 8 % parameter sp is the number of system states
 9 {
10   % Check if q and r are positive definite and symmetric
11   if (not (and (and (is_posdef(q), is_symmetric(q)),
12         and (is_posdef(r), is_symmetric(r)))))
13     { var dummy := disp(); } % Print warning to display
14
15   {
16     % Eigenvector decomposition of Hamiltonian
17     var H := mglue(A, mat_mul(rightdiv(B,R), trans(B)),
18                    Q, mat_scale(real_neg1(), trans(A)));
19     { var v := eigvec(H);
20       var d := eigval(H);
21
22       % sort on real part of eigenvalues
23       { var e2 := sort1(realvec(d));
24         var index := sort2(realvec(d));
25         if (not (and (real_lt(vec_fetch(e2,sp), real_zero()),
26                       real_gt(vec_fetch(e2,int_plus(sp,1)),
27                               real_zero()))))
28           % Can't order eigenvalues, (A,B) may be uncontrollable.
29           { var dummy := error(); }
30
31         { % Return closed-loop eigenvalues
32           var e := vec_chop(vec_permute(d,index),firstrow(A),1);
33           % select vectors with negative eigenvalues
34           var chi := mat_chop(mat_cpermute(v,index),A,1,1);
35           var lambda := mat_chop(mat_cpermute(v,index),A,
36                                  int_plus(sp,1),1);
37           { var s := mat_scale(real_neg1(), rightdiv(lambda,chi));
38             { var k := leftdiv(r,mat_mul(trans(b),s));
39               return k; }
40           }
41         }
42       }
43     }
44   }
45 }
```

Figure 3-2: Translation of the LQR Script to DP, Part 1

```
47 proc is_posdef[n](X:mat[n,n]) returns bool
48 {
49   var nx := norm(X);
50   if (vec_any_lt(eigval(X), real_mul(real_neg1(),real_mul(eps(),nx))))
51     return false;
52   else
53     return true;
54 }
55
56 proc is_symmetric[n](X:mat[n,n]) returns bool
57 {
58   var nx := norm(X);
59   if (real_gt(real_div(norm(mat_sub(X,trans(X))), nx), eps()))
60     return false;
61   else
62     return true;
63 }
```

Figure 3-3: Translation of the LQR Script to DP, Part 2

- MATLAB supports multiple return values while DP does not. This leads to, for example, the separate eigvec and eigval constructs in lines 19 and 20 which find the eigenvectors and eigenvalues of the same matrix; the original MATLAB script does both with the **eig** function (line 31 of figure 3-1). Also, the MATLAB script returns three values: K, S, and E. The DP program returns only K, the most useful. However, it does compute all three. Multiple return values could be integrated with dependent types with a trivial change to the return-statement typing rule.

- Because DP omits traditional type parameters, the type system does not distinguish between matrices of complex numbers and matrices of reals. Such a distinction would provide greater safety for operations such as the **realvec** procedure called on lines 23 and 24. However, traditional type parameters and our value parameters would be orthogonal constructs.

In light of these explanations, most of the translation from the MATLAB script to the DP program should be fairly clear.

```
%%% Declarations to model the MATLAB builtin functions.

#include "type_builtins.h"
#include "scalar_builtins.h"
#include "vector_builtins.h"
#include "matrix_builtins.h"
#include "misc_builtins.h"
```

Figure 3-4: matlab_builtins.h

```
%%% Declarations to model the MATLAB builtin types

type real;
type complex;
type vec[int];
type mat[int,int];
```

Figure 3-5: type_builtins.h

A few constructs remain somewhat difficult to understand. Chief among these are probably the **mat_chop** and **vec_chop** procedures. The lack of clarity of expression here stems from some fundamental limitations of the DP type system which this case study brings to attention. These limitations will be discussed in the next section.

3.3.3 Limitations of the DP Type System

As mentioned above, the case study does illustrate some fundamental limitations of DP's type system. The most troublesome parts of the LQR script from this point of view are the subvector and submatrix selection operations used in lines 37, 39, and 40 of the original MATLAB script. This operation takes a vector or matrix along with a set of indices for a subvector or submatrix as arguments. The translation, in terms of the **chop** operations in lines 32, 34, and 35 of the DP program, takes the original matrix or vector, a second matrix or vector, and an index of a starting element as arguments, and returns a new matrix of the same dimensions as the second matrix, composed of enough elements from the original matrix to fill the new matrix, starting from the specified starting index element. The following limitations explain the need

```
%%% Declarations to model the MATLAB builtin scalar functions

%%% model boolean-operator builtins
proc or(a,b:bool) returns bool;
proc and(a,b:bool) returns bool;
proc not(a:bool) returns bool;

%%% model integer builtins
proc int_plus(a,b:int) returns int;

%%% model real-number builtins
proc real_zero() returns real; % a literal real-valued 0
proc real_neg1() returns real; % a literal real-valued -1
proc real_mul(a,b:real) returns real;
proc real_div(a,b:real) returns real;
proc real_lt(a,b:real) returns bool;
proc real_gt(a,b:real) returns bool;


%%% model complex-number builtins
proc get_real(x:complex) returns real;
```

Figure 3-6: scalar_builtins.h

```
%%% Declarations to model the MATLAB builtin vector functions

proc sort1[n](X:vec[n]) returns vec[n]; % returns sorted elements
proc sort2[n](X:vec[n]) returns vec[n]; % returns permutation vector
proc realvec[n](X:vec[n]) returns vec[n]; % vector containing real components
proc vec_scale[n](x: real, A: vec[n]) returns vec[n];
proc vec_fetch[n](X: vec[n], i:int) returns real;

% chop X down to the size of Y, starting from index i
proc vec_chop[n1,n2](X:vec[n1],Y:vec[n2],i:int) returns vec[n2];

% permute X by permutation vector pi
proc vec_permute[n](X,pi:vec[n]) returns vec[n];

% returns true iff any element of X is less than y
proc vec_any_lt[n](X:vec[n],y:real) returns bool;
```

Figure 3-7: vector_builtins.h

```
%%% Declarations to model the MATLAB matrix builtins

% matrix "gluing" constructor
proc mglue[m1,n1,m2,n2]
     (A: mat[m1,n1], B: mat[m1,n2],
      C: mat[m2,n1], D: mat[m2,n2])
     returns mat[m1+m2, n1+n2];

% matrix transpose and inverse
proc trans[r,c](X: mat[r,c]) returns mat[c,r];
proc inv[r,c](X: mat[r,c]) returns mat[c,r];

% multiply a matrix by a matrix
proc mat_mul[r,i,c](A: mat[r,i], B: mat[i,c]) returns mat[r,c];

% subtract two matrices
proc mat_sub[r,c](A,B: mat[r,c]) returns mat[r,c];

% scale a matrix by a real
proc mat_scale[r,c](x: real, A: mat[r,c]) returns mat[r,c];

% Left division and right division are formally equivalent
% to the definitions below.  MATLAB actually uses
% a different algorithm for numerical reasons.

proc leftdiv[r,i,c](A: mat[i,r], B: mat[i,c]) returns mat[r,c]
{
  return mat_mul(inv(A), B);
}

proc rightdiv[r,i,c](B: mat[r,i], A: mat[c,i]) returns mat[r,c]
{
  return mat_mul(B, inv(A));
}
```

Figure 3-8: matrix_builtins.h, Part 1

```
% matrix norm
proc norm[m,n](x:mat[m,n]) returns real;

% columns of result are the eigenvectors of x
proc eigvec[n](x:mat[n,n]) returns mat[n,n];

% find the vector of eigenvalues
proc eigval[n](x:mat[n,n]) returns vec[n];

% chop matrix X down to the size of matrix Y, starting from element [i,j]
proc mat_chop[m1,n1,m2,n2](X:mat[m1,n1],
   Y:mat[m2,n2],
   i,j:int) returns mat[m2,n2];

% permute columns of matrix X according to permutation vector pi
proc mat_cpermute[r,c](X:mat[r,c],pi:vec[c]) returns mat[r,c];

% get the first row of X
proc firstrow[m,n](X:mat[m,n]) returns vec[n];
```

Figure 3-9: matrix_builtins.h, Part 2

for this convoluted translation:

- Most seriously, parameter expressions cannot be expressed in terms of arguments. The MATLAB submatrix selection operation allows the indices for the submatrix to be indicated as an argument, providing maximum flexibility for the shape of the submatrix. This flexibility to defer decisions about matrix shape until runtime, however, comes at a price for static typing: namely, a static type system cannot prove anything about those shapes. It would be possible to extend the DP type system to allow parameter expressions to be expressed in terms of arguments. However, the usefulness of this would be quite questionable if determining much of the new type information was impossible at compile time.

  Therefore, the translation of the submatrix operations must avoid using an argument variable to specify the dimensions of the new submatrix. Fortunately, the example in the case does not require that much runtime flexibility because we can determine statically that the submatrix must be of dimensions $sp \times sp$.

This allows the translation to solve this problem by specifying the size of the new submatrix in terms of one of the caller's argument matrices. Then the dimensions once again can be inferred in terms of parameters.

If we allowed parameter expressions to be expressed in terms of arguments, the typechecker would have to perform dataflow analysis to determine whether those argument values could be evaluated at compile time. If so, then checking would proceed using those optimized values; otherwise, typechecking would have to be deferred to runtime.

- In DP's current form, parameter values *must* be left implicit. Situations sometimes arise in which it is much more natural for the programmer to *explicitly* instantiate parameter values. The solution to the **chop** problem above still seems unsatisfactory because the second matrix is provided solely for its shape information, which seems rather inelegant. Allowing the parameters to be specified explicitly instead of forcing all parameters to be inferred would make the translation seem more natural. Unlike the binding-time problem with arguments above, there is no fundamental reason why the DP type system could not be extended to allow for this.

- The existing parameter constraints are all in terms of *equalities* which indicate whether type parameters which must conform actually do so. Some other kinds of checks we would like to make would need to be *inequalities*. The check in lines 4–6 of the original MATLAB script, which makes sure that the row and column sizes of the A and B matrices are greater than zero, is a situation in which this need arises: we would like to be able to encode the requirements that $\mathbf{ip} > 0$ and $\mathbf{sp} > 0$.

  Adding inequalities would introduce some new complications. Signatures would be allowed to include inequality constraints on their parameters which would need to be satisfied for a call to be valid. Now, procedure-generator bodies could be checked using these inequality constraints as additional assumptions. A check for the inequality constraints would have to be included at call-sites

where the callee imposed such an inequality constraint.

One major consequence of those limitations is that, while the type system does an excellent job of reconciling dimensional *conformance*, it does not handle *bounds* checking. *Conformance* refers to correct agreement between type *parameters* and signatures. It refers to *shapes* of data objects in the program. The case study shows that these shapes can often be deduced at compile time, allowing them to be subsumed in the type system. Examples of conformance checking include matrix multiplication, transpose, permutation, and addition.

Bounds checking, on the other hand, relates argument values (which, in the general case, cannot be determined until runtime) to type parameters. Matrix and vector element fetch, as well as the inequality checks that would be desirable in the submatrix operation as described above, are examples of operations that would benefit from bounds checking. However, this kind of analysis requires too much dynamic information at compile time to be as successful as the conformance checks.

These problems illustrate the central tradeoff of type systems. Mark Manasse writes,

> The fundamental problem addressed by a type theory is to insure that programs have meaning. The fundamental problem caused by a type theory is that meaningful programs may not have meanings ascribed to them. The quest for richer type systems results from this tension. [3]

This tradeoff is evident in this case study: Applying the DP type system to the LQR script certainly gives more confidence that the dimensions of the matrices used throughout the program make sense. However, this comes at the price of making the operation of cutting up big matrices into smaller submatrices harder to express legally. Likewise, we feel uneasy about the lack of static confirmation of matrix bounds, but it seems evident that an attempt to force programs to be statically bounds-checkable would limit the expressiveness of the language so much that it would be almost useless. Compromises in safety of the DP type system—such as the choice of deferring parts of typechecking until runtime if necessary—result from attempts to deal with the

fundamental tension in the problem.

Chapter 4

Extensions

This chapter describes possible extensions to the type system defined in Chapter 2. The extensions can be classified into three groups by their motivations: expressiveness, safety, or performance. Each extension is discussed in terms of the benefit it would provide and the impact it would have on the typechecking algorithm. Some of the extensions listed may not be worthwhile because they may be too difficult to implement.

4.1 Expressiveness

4.1.1 Explicit Parameter Instantiation

As defined in Chapter 2, DP does not provide a way to instantiate parameters explicitly. Subsection 3.3.3 gives an example of a situation that would be much less awkward without this limitation.

Integrating parameterized types with an abstract type facility would also require explicit parameter instantiation. To make DP's parameterized types really useful, programs must be able to define constructor procedures that build representations for them given arguments of primitive types. Such a constructor procedure would require explicit parameters to give the dimensions of the constructed objects if the arguments passed to the constructor were not parameterized themselves.

Adding the ability to specify procedure-generator parameters explicitly would pose no fundamental difficulties. However, for maximum expressiveness, explicit instantiations of parameters in terms of arguments as well as parameters and literals would be allowed. The compiler could evaluate any such expressions whose values could be determined at compile time, allowing checking to proceed statically. Remaining checks would have to be deferred until runtime. This compromise would trade safety for expressiveness.

4.1.2 Elimination of the Signature Restriction

It would be nice to eliminate the requirement defined in subsection 2.1.5 that each parameter of a procedure generator appear by itself in one of the formal-argument types. However, this may not really improve the usefulness of the language; we have not been able to think of any practical problems for which the generalized form seems natural.

Also, if we eliminated this restriction, the checker would need to solve systems of arbitrary integer equations in order to reconstruct types of call-expressions. This is undecidable (see subsection ??).

4.1.3 Generalized Parameter Expressions

DP specifies a separate grammar for the expressions that are allowed to instantiate parameterized types. This grammar allows only the operators $+$, $-$, $\times$, and $/$. Allowing general expressions to instantiate parameterized types would increase flexibility. To implement this, the typechecker would have to verify that these expressions had integer type. If such an expression's value could be determined at compile time (for example, because the expression was in terms of manifest constants), then checking could still be done statically. Checking would have to be deferred to runtime if the expression could not be statically evaluated.

4.2 Safety

4.2.1 Provably Unsatisfiable Runtime Checks

The typechecker creates runtime-check equations to encode constraints that it has not verified at compile time. If the typechecker can prove that no assignment of parameters in the runtime checks will allow them to be consistent, then it can detect a larger portion of erroneous programs statically.

Implementing this feature would require a more sophisticated equation-solving algorithm, or, at the least, a set of heuristics that detected some cases of unsatisfiable check equations.

4.2.2 Backward Check Propagation in the Flow Graph

An interprocedural flow analysis could identify code points at which a later runtime check would inevitably be made. This would enable the compiler to move the runtime checks backwards through the code, possibly disqualifying more errors based upon the added information exposed.

For example, runtime checks at the beginnings of procedure-generator bodies could be moved to the call sites. Specific information about each call site could then be applied to disqualify calls that had no possibility of being correct.

Note that this movement may also enable some checks to be discharged (proven true for all parameter values) at compile time.

4.2.3 Integration of Conformance and Bounds Checking

Subsection 3.3.3 explains why a large portion of conformance checking can be performed statically, but bounds checking generally cannot. However, bounds checking and conformance checking can be integrated into the same framework. This would require the system to express parameter constraints in terms of inequalities as well as equalities.

[8] discusses techniques for reducing the runtime overhead of bounds checking.

These techniques include elimination of redundant checks, possibly enabled by moving them through the flow graph. It is possible that the analysis needed to do this kind of bounds-check optimization can be combined with the analysis used in bounds checking.

4.3 Performance

4.3.1 Loops

DP omitted loops for simplicity. This did not reduce its expressive power because, with the help of lambda lifting, loops can be desugared into tail recursion. However, explicit inclusion of loops may allow the optimization of redundant checks by allowing the motion of loop-invariant runtime checks outside the loops.

4.3.2 Elimination of Provably Correct Checks

A more sophisticated equation-solving algorithm would be able to determine whether check equations were true for all parameter values, as well as whether checker equations were unsatisfiable regardless of the parameter values. This would allow the runtime checks to be eliminated, speeding up execution and improving confidence in the program.

The working implementation of the DP checker already does this in the simplest cases. It can recognize check equations in which the left-hand and right-hand sides are syntactically the same, eliminating them. This allowed all the checks for the LQR example to be eliminated statically.

A more sophisticated optimizer would embody more semantic knowledge. It would know and apply simple algebraic properties of the integers such as

$$\forall x \in \mathrm{Int} \cdot x + 0 = x$$
$$\forall x \in \mathrm{Int} \cdot 1x = x$$
$$\forall x \in \mathrm{Int} \cdot x + x = 2x$$

Algebraic simplification would allow more check equations to be proven true statically.

Chapter 5

Conclusions

As subsection 3.3.1 argues, the combination of type-system features we have described significantly improves the safety of matrix programs and the documentation of conformance requirements in the language. This shows that conformance checking can be successfully integrated into type systems.

There are, however, fundamental limits to the power of the system in terms of the tradeoff between safety and expressiveness. Most importantly, as subsection 3.3.3 notes, it is fundamentally impossible to perform static analysis on dynamic information. Thus, if programs enjoy the flexibility gained by making dimensional parameters dependent on runtime values, they must forego the safety afforded by static typechecking for the affected constructs. Many of the limitations of this dependent type system are variants of this problem.

This work also highlights the general fact that dependent type systems must embody significant knowledge about their value domains. For maximum effectiveness, the checker for DP should be able to take advantage of many semantic properties of the algebra of integers in order to discharge checks or prove them unsatisfiable statically. The need for this domain-specific knowledge implies that the transferability of technology among type systems dependent on varying value domains would be limited. For example, a type system applying the features of DP to types dependent on boolean values would probably be significantly different from this one, and it is questionable how much the two systems could be integrated into the same framework.

This fact limits the usefulness of dependent type systems for general-purpose languages. It is in specialized application domains—such as the domain of matrix computation explored here—that they will prove most useful.

Appendix A

Listing of the Original MATLAB LQR Script

This appendix contains a complete listing of the original MATLAB LQR script discussed in Chapter 3. Differences between this script and the simplified one given in figure 3-1 are very minor and should not affect the applicability of the script as a case-study example:

- The case-study version in figure 3-1 omits the long introductory comment.
- The case study omits the optional fifth argument nn, the "cross-term." The only operations done with nn are matrix multiplication, transpose, and addition. Since these operations are used elsewhere in the script, omitting nn does not impact the level of language expressiveness the script requires.

```
function [k,s,e]=lqr(a,b,q,r,nn)
%LQR Linear quadratic regulator design for continuous systems.
% [K,S,E] = LQR(A,B,Q,R)  calculates the optimal feedback gain
% matrix K such that the feedback law  u = -Kx  minimizes the cost
% function:
% J = Integral {x'Qx + u'Ru} dt
%
```

```
% subject to the constraint equation:
% .
% x = Ax + Bu
%
% Also returned is S, the steady-state solution to the associated
% algebraic Riccati equation and the closed loop eigenvalues E:
%   -1
% 0 = SA + A'S - SBR  B'S + Q      E = EIG(A-B*K)
%
% [K,S,E] = LQR(A,B,Q,R,N) includes the cross-term N that relates
% u to x in the cost function.
%
% J = Integral {x'Qx + u'Ru + 2*x'Nu}
%
% The controller can be formed with REG.
%
% See also: LQRY, LQR2, and REG.

% J.N. Little 4-21-85
% Revised 8-27-86 JNL
% Revised 7-18-90 Clay M. Thompson
% Copyright (c) 1986-93 by the MathWorks, Inc.

error(nargchk(4,5,nargin));
error(abcdchk(a,b));
if ~length(a) | ~length(b)
        error('A and B matrices cannot be empty.')
end

[m,n] = size(a);
[mb,nb] = size(b);
[mq,nq] = size(q);
```

```
if (m ~= mq) | (n ~= nq)
error('A and Q must be the same size');
end
[mr,nr] = size(r);
if (mr ~= nr) | (nb ~= mr)
error('B and R must be consistent');
end

if nargin == 5
[mn,nnn] = size(nn);
if (mn ~= m) | (nnn ~= nr)
error('N must be consistent with Q and R');
end
% Add cross term
q = q - nn/r*nn';
a = a - b/r*nn';
else
nn = zeros(m,nb);
end

% Check if q is positive semi-definite and symmetric
nq = norm(q,1);
if any(eig(q) < -eps*nq) | (norm(q'-q,1)/nq > eps)
disp('Warning: Q is not symmetric and positive semi-definite');
end
% Check if r is positive definite and symmetric
nr = norm(r,1);
if any(eig(r) <= -eps*nr) | (norm(r'-r,1)/nr > eps)
disp('Warning: R is not symmetric and positive definite');
end

% Start eigenvector decomposition by finding eigenvectors of Hamiltonian:
```

```
[v,d] = eig([a b/r*b';q, -a']);
d = diag(d);
[e,index] = sort(real(d));  % sort on real part of eigenvalues
if (~( (e(n)<0) & (e(n+1)>0) ))
error('Can''t order eigenvalues, (A,B) may be uncontrollable.');
else
  e = d(index(1:n));  % Return closed-loop eigenvalues
end
chi = v(1:n,index(1:n));  % select vectors with negative eigenvalues
lambda = v((n+1):(2*n),index(1:n));
s = -real(lambda/chi);
k = r\(nn'+b'*s);
```

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] Brian D. O. Anderson and John B. Moore. *Optimal Control: Linear Quadratic Methods.* Prentice Hall, New Jersey, 1990.

[3] Luca Cardelli. A polymorphic λ-calculus with Type:Type. Technical Report DEC/SRC TR-10, Digital Equipment Corporation Systems Research Center, May 1986.

[4] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2), April 1987.

[5] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[6] Michael Garey and David Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[7] David Gifford and Franklyn Turbak. Course notes for 6.821 programming languages. Unpublished lecture notes, 1992.

[8] Rajiv Gupta. A fresh look at optimizing array bound checking. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation.*, pages 272–282. ACM SIGPLAN, 1990.

[9] Thomas Johnsson. Lambda-lifting—transforming programs to recursive equations. Technical report, University of Goteborg Programming Methodology Group, June 1986.

[10] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* MIT Press, Cambridge, MA, 1986.

[11] P. Martin-Löf. Intuitionistic type theory. Notes of Giovanni Sambin on a series of lectures given in Padova, Italy, June 1980.

[12] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[13] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[14] Greg Nelson. *Systems Programming with Modula-3.* Prentice Hall Series in Innovative Technology. Prentice Hall, 1991.

[15] Katsuhiko Ogata. *Solving Control Engineering Problems with MATLAB.* MATLAB Curriculum Series. Prentice Hall, 1993.

[16] Raymie Stata. Dimensional analysis for software. Unpublished article, September 1992.

[17] *The Student Edition of MATLAB.* MATLAB Curriculum Series. Prentice Hall, 1992.

[18] J. Welsh, W. Sneeringer, and C. Hoare. Ambiguities and insecurities in Pascal. *Software Practice and Experience*, November 1977.