Turning Dynamic Typing into Static Typing by Program Specialization

Karina Olmos
Eelco Visser

Technical Report UU-CS-2003-049
Institute of Information and Computing Sciences
Utrecht University

July 2003

Preprint of:
K. Olmos and E. Visser. Turning Dynamic Typing into Static Typing by Program Specialization. In D. Binkley and P. Tonella, editors, Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03), pages 141–150, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.

ISSN 0924-3275

Address:
Karina Olmos
karina@cs.uu.nl
http://www.cs.uu.nl/~karina

Eelco Visser
visser@acm.org
http://www.cs.uu.nl/~visser

Institute of Information and Computing Sciences
Utrecht University
P.O.Box 80089
3508 TB Utrecht

Turning Dynamic Typing into Static Typing by Program Specialization

Karina Olmos and Eelco Visser

Institute of Information and Computing Sciences, Utrecht University

P.O. Box 80089, 3508 TB Utrecht, The Netherlands.

karina@cs.uu.nl, visser@acm.org

Abstract

Array processing languages such as APL, Matlab and Octave rely on dynamic typechecking by the interpreter rather than static typechecking and are designed for user convenience with a syntax close to mathematical notation. Functions and operators are highly overloaded. The price to be paid for this flexibility is computational performance, since the run-time system is responsible for type checking, array shape determination, function call dispatching, and handling possible run-time errors. In order to produce effecient code, an Octave compiler should address those issues at compile-time as much as possible. In particular, static type and shape inferencing can improve the quality of the generated code. In this paper we discuss how overloading in dynamically typed Octave programs can be resolved by program specialization. We discuss the typing issues in compilation of Octave programs and give an overview of the implementation of the specializer in the transformation language Stratego.

1. Introduction

Array processing languages such as APL, Matlab[1], and Octave rely on dynamic typechecking by the interpreter rather than static typechecking by the compiler. Types of variables and functions are not explicitly declared, but inferred at run-time depending on run-time values. Type errors are detected at run-time and caught by the interpreter. These programming languages are designed to be user-friendly. The syntax is close to the mathematical notation and a rich set of mathematical functions is available. In keeping with mathematical notation, functions and operators can be highly overloaded. For example, a function can operate on scalars as well as matrices, and can have a variable number of arguments. These features make array processing languages easy to use as prototyping languages. However, the price to be paid for this flexibility is computational performance when production quality code is needed.

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

Deferring typechecking to run-time, and in particular, run-time resolution of overloading, prevents high quality compilation. First of all, extra instructions need to be spent in performing dynamic checks. Secondly, overloading requires a more generic data representation, which can be avoided when more precise type information is available. Thus, type information is essential for high quality compilation. Compilation for array processing languages is a topic of ongoing research [8, 3, 4, 1]. Issues that distinguish their compilation from other languages are type and shape inferencing, which become a problem when a language has overloaded operators and 'ad-hoc' overloaded functions as is the case in Octave [6].

In this paper we give an overview of a source-to-source transformation system for Octave which turns dynamically typed programs into statically typed ones by specializing overloaded functions and operators to the types they are actually used with. This transformation system is a front-end for a vectorizing compiler for Octave that we are developing in collaboration with University Dresden.

The transformation system is implemented in Stratego [13], a rewriting system extended with programmable rewriting strategies [15] for the control over the application of rules and dynamic rewrite rules [12] for the propagation of information. The specializer extends the constant propagation techniques developed earlier for use in optimizers [11]. An important aspect of our specializer is that it does not require the annotation of the source program with declarations as other systems do.

The rest of the paper is organized as follows. The next section discusses overloading in Octave and shows how specialization resolves overloading. Section 3 presents the architecture of the transformation system and explains the individual components by means of examples. The remaining sections discuss the implementation of selected components. Section 4 describes the abstract interpretation style of type checking. Section 5 explains the approach used to type user-defined and intrinsic functions. Section 6 discusses shape inference for the detection of the shape of matrix values.

2. Overloading in Octave

Octave, an open source clone of Matlab, is a high-level programming language and development environment which is widely used for rapid prototyping and simulation [6]. Distinctive features of the language are its high-level built-in data types, overloaded operators and a rich set of mathematical functions. Another advantage of this language is that it is more declarative than procedural, providing freedom to the compiler on how to structure the computation of operations.

Although these features make the language easy to use, they are also a source of ambiguities. For instance, the statement `M = X * Y` performs matrix multiplication if `X` and `Y` are matrices. The same statement performs multiplication of scalars such as integers, reals, complex numbers or any combination of them depending on the type of `X` and `Y`. The lack of variable, type, and shape declarations in Octave makes identification of the proper invocation of a subroutine call problematic. It is the task of the interpreter to perform type checks and resolve overloading in order to invoke the proper operation.

In addition, certain functions are interpreted according to their context. The number and type of the actual arguments in a function call determine the behaviour of a function. Furthermore, a function can yield multiple return values determined by the return context. As an example consider the statement `p = find([ 1 0; 5 0])`. The result of the call to `find` is a vector of the non-zero positions in the argument matrix considered as a one-dimensional vector organized by columns. Thus, the outcome is `p = [1 ; 2]`. The same function called in a context where two results are required, as in the statement `[r, c] = find([ 1 0; 5 0])`, gives the non-zero positions *in the matrix* as a pair of coordinate vectors for the row and column positions. Thus the result of the statement is `r = [1 ; 2]` and `c = [1 ; 1]`. If the context requires three results from `find` as in the statement `[r, c, v] = find( [ 1 0; 5 0])`, the last requested result will contain the elements from the matrix which are non zero `r = [1 ; 2]`, `c = [1 ; 1]` and `v = [1 ; 5]`. This example shows how contextual information is required in order to correctly type these statements.

User-defined functions inherit overloaded features from operators. The lack of declarations represents a problem for type inferencing. When defining the signature of a function, the type of the arguments cannot be restricted, which can lead to run-time errors. To prevent this, the function definition could include restrictive type checks explicitly to enforce certain argument types. Furthermore, functions can have different behaviour depending on the type and number of the arguments.

```
function retval = reshape (a, m, n)
  if (nargin == 2 && prod (size (m)) == 2)
    n = m(2);
    m = m(1);
    nargin = 3;
  endif

  if (nargin == 3)
    [nr, nc] = size (a);
    if (nr * nc == m * n)
      retval = zeros (m, n);
      if (isstr (a))
        retval = setstr (retval);
      endif
      retval(:) = a;
    else
      error('reshape: sizes must match');
    endif
  else
    usage('reshape(a, m, n) or ...
           reshape (a, size(b))');
  endif
endfunction
```

```
  c = [ 1 2 3 ];
  s = 'hello';
  c1 = reshape(c, 3, 1);
  s1 = reshape(s, 5, 1);
```

⇓

```
function retval =
  reshape_string_int_int (a , m , n)
   ()
   [nr, nc] = size_lib (a)
   if (nr * nc) == (m * n)
     retval = zeros_lib (m , n);
     retval = setstr_lib (retval);
     retval(:) = a;
   else
     error_lib('reshape: sizes must match')
   endif
end
function retval =
   reshape_matrix(_int)_int_int (a , m , n)
   ()
   [nr, nc] = size_lib(a)
   if (nr * nc) == (m * n)
     retval = zeros_lib(m , n)
     ()
     retval(:) = a
   else
    error_lib('reshape: sizes must match');
   endif
end
```

```
  c = [ 1 2 3 ];
  s = 'hello';
  c1 = reshape_matrix(_int)_int_int(c, 3, 1);
  s1 = reshape_string_int_int(s, 5, 1);
```

Figure 1. Typing by means of program specialization

An example of code which includes different functionality for different argument types is the function `reshape` shown in Figure 1, which is part of the Octave distribution.

In this function there are some notions of contextual information. From the signature of the function we can see that the function is specified to be called with three arguments. In fact, the reshape function can also be called with two arguments. That is the reason for performing the verification of `nargin` which represents the number of incoming arguments. This variable is part of the interpreter and it can be manipulated in any function definition. Observe that the code performs many type checks in order to execute the right computation. Most of these checks can be performed statically based on the arity and types of a function call. For instance, Figure 1 shows how a specialized program is derived by propagating known information to the function definition in order to infer the resulting type. The `reshape` function has been specialized for two different instances of a reshape call. We have introduced `()` where code has been removed from the program. Intrinsic functions have been identified and renamed to indicate it, by means of the suffix `_lib`.

This example shows how context dependent information is needed for static type inferencing. By function specialization the result of type inference is expressed in context independent functions such that subsequent compiler phases can deal with unambiguous code.

3. Architecture

The architecture of `octave-front` is depicted in Figure 2. The transformation system performs a source-to-source transformation. Octave source code is parsed using the octave parser which was updated to generate abstract syntax trees using the ATerm format [2]. The other components of the front-end are implemented in Stratego. Desugaring, variable and function name disambiguation are performed before type and shape inferencing. Type-based specialization is implemented to obtain code suited for clean compilation.

Parsing To parse Octave programs we use Octave itself. The output is an abstract syntax tree which contains user-defined functions as well as Octave functions that are part of the Octave distribution.

Desugaring The abstract syntax obtained after parsing is converted into a simpler representation of the program that reduces the number of constructs to ease subsequent phases. Complex statements, such as `switch` are desugared into semantically simpler ones as is illustrated in the following example:

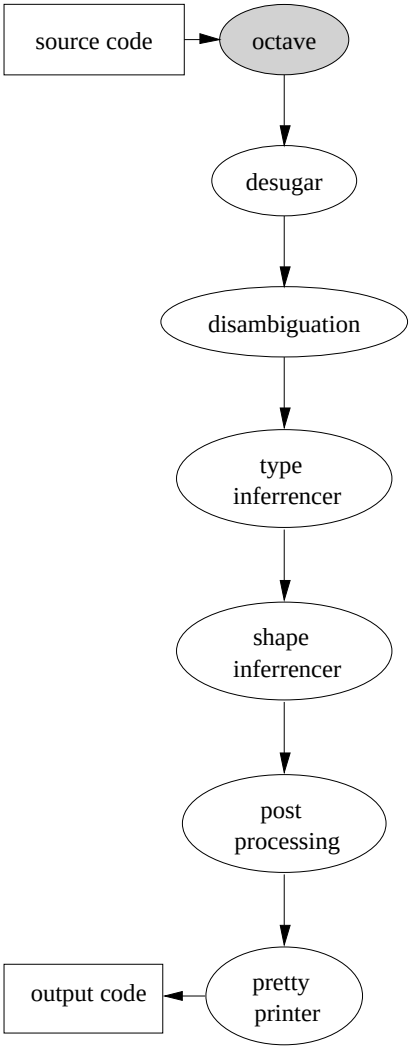

Figure 2. Overview of the system architecture

```
switch a
  case (1)
    x = 1;
  case (2)
    x = 2;
  case (3)
    x = 4;
  otherwise
    x = 5;
end
```

⇒

```
if a == 1
 then x = 1
 else
  if a == 2
    then x = 2
    else
     if a == 3
      then x = 4
      else x = 5
     endif
  endif
endif
```

Side effects in expressions are removed by translating them into statements which have a simpler and cleaner representation, as in the following example:

```
a = [10,11];
++a;
b = ++a ;
```

⇒

```
a = [ 10 11 ];
a = a + 1;
a = a + 1;
b = a;
```

Disambiguation An identifier in Octave can represent a variable or a function call. A mechanism to determine to which name space an identifier belongs is needed. In Octave, assignments are closest to variable declarations. Thus, an occurrence of an identifier in the left hand side of an assignment defines it as a variable. The occurrence of an identifier in the right hand side of an assignment or elsewhere may represent a function call. Without disambiguation, an erroneous interpretation can be given to a program. For instance, consider the following code:

```
function x = test(y)
   a = y + rand;
   rand(3) = a;
   x = rand(2) + 1;
```

⇒

```
function x = test(y)
  a = y + rand();
  rand[3] = a;
  x = rand[2] + 1;
end
```

This example shows three occurrences of the identifier `rand`. The first occurrence is a function invocation without arguments. Note that a function invocation does not have to use parentheses. The second occurrence of this identifier denotes an assignment to the array `rand` with three elements. Although only the third element is being assigned explicitly, this statement will create an array with three elements. The first and the second elements of the array will contain zero. The third occurrence of `rand(2)` denotes a subscript for the array `rand` accessing the second element. The same expression in the absence of the second statement will have a completely different interpretation, i.e., `rand(2)` will denote a function invocation which will produce a two by two matrix of randomly generated values.

Type Inference Octave does not have an explicit notion of types. Type casting is done transparently when the result of a computation requires it. We could assign a *Complex* data type which will represent all types in a proper way, but this leads to a waste of memory allocation for computations that not require this data type. The goal of a type inferencer is to make types explicit and determine the smallest data type for each variable, and characterize proper casting actions when are required.

The type inferencer is implemented by traversing the abstract syntax tree of a program twice. The first traversal is responsible for determining data types and characterizing operations. The following traversal is responsible for determining shapes. The type and shape information is added to every node of the abstract syntax tree and decorates it with the information of shape. The reason for performing type and shape inferencing in two separate traversals is because the first phase performs specialization based on type information. The shape inferencer adds shape information extending the information which is contained in the term.

The following example shows how specialization can eliminate "ad-hoc" function overloading. Different specialized functions are created for different type invocations.

```
function x =
 f(a, b)
  x = a + b;
```

⇒

```
function x =
 f_matrix(_int)_int(a,b)
   x = b + a;
end
function x =
 f_int_int(a,b)
   x = a + b;
end
```

```
c = 4;
r = f(c, c);
y = f([3, 4], r);
```

⇒

```
c = 4;
r = f_int_int(c, c);
y = f_matrix(_int)_int...
  ([3 4], r);
```

User functions inherit overloading features from operators. Different data types can be supplied to the arguments of the function `f` in the example. The type inferencer determines the type of the arguments and propagates this information to specialize functions. Functions are renamed using the name of the type of arguments of the call.

Shape Inference Shape inference consists of determining the structure and size of a matrix. Inferring exact shape information can avoid run-time memory allocation. Furthermore, shape information is vital for matrix calculations. Matrix operators impose shape restrictions for matrix calculations. For instance, to multiply two matrices, the number of columns of the first matrix has to be equal to the number of rows of the second matrix. In matrix addition the argument matrices should have the same shape. In addition, single elements are also considered matrices. These matrices are referred to by the community as *scalars*.

```
a = [1,2;5,3];
b = ones(2);
......
m = a * b;
```

⇒

```
a = [1,2;5,3];
b = ones(2);
......
m = a *m b;
```

Operations have been typed by the previous phase. This reduces the number of constructors that are required to be inspected. When not enough information has been inferred, our system inserts a guard to guarantee that shape operation conditions are satisfied to execute matrix operations. This is illustrated with the following example:

```
a = zeros(n);
b = zeros(n1);
......

m = a + b;
```

⇒

```
a = zeros(n);
b = zeros(n1)
  .....
m =
 if (n == n1)
   then a +m b
   else error(...)
 endif
```

The shape inferencer is able to identify `a` and `b` as square matrices. How to infer shape information from built-in functions will be discussed in Section 6. From the program we cannot statically determine if `a` and `b` are matrices that satisfy the shape restriction for multiplication. In this situation, the program is instrumented to apply this operation

only when the restriction is satisfied. The operation is being guarded by an `if` construction. Other components of the compiler may eliminate this guard when the restriction can be satisfied.

Post processing After type and shape inferencing, the code contains empty statements and the outcome for instrumenting the program with guards. The code needs some restructuring. We have also applied other optimizations after type and shape inferencing, but these optimizations are not the focus of this paper. For example, consider the hoisting of an `if` statement from an assignment in the following transformation:

```
a = zeros(n);
b = zeros(n1)
  .....
m =
 if (n == n1)
   then a +m b
   else error(...)
 endif
```

⇒

```
a = zeros(n);
b = zeros(n1)
  .....

if (n == n1)
  then m = a + b
  else error(...)
endif
```

Pretty-Printing The pretty-printer turns the abstract syntax tree back into readable source code. The output of the transformation is designed for consumption by a compiler back-end. Therefore, the output of the pretty-printer is not completely valid Octave code, since complex identifiers are used for specialized functions. If necessary these identifiers can be mangled into valid Octave identifiers.

4. Type Inference

Octave data types are organized in two sorts. Scalar types such as *Boolean*, *Integer*, *Float*, *Complex*, *Char* and *Matrices* of scalar values. Boolean types are represented by integer numbers as in the C language. Matrices are two dimensional arrays of scalar types. The type inferencer of our transformation system annotates expressions in an Octave program with their static types and shapes. In this section we will discuss the specification of type inference for expressions.

4.1. Expressions

The annotation of expressions with their type is performed with rewrite rules which are applied in a bottom-up traversal over the abstract syntax tree. Each rule infers the type of the expression from the operator and the types of its arguments. Type information is encoded using Stratego term annotations. A term annotation `t{t'}` embeds a term `t'` into the term `t` without altering the signature or the structure of the term `t`. Some typical typing rules for expressions are shown in Figure 3. The first three rewrite

```
TcExp:
  |[ ~int: i ]| -> |[ (~int: i){INT} ]|
TcExp:
  |[~str: s ]| -> |[ (~str: s){STRING}]|
TcExp:
  |[ e1{t} - e2{t} i ]| ->
  |[ (e1{t} - e2{t} i){COMPLEX} ]|
TcExp:
  |[ e1{t1} + e2{t2} i ]| ->
  |[ (e1{t1} + e2{t2} i){FLOAT} ]|
  where <INT + FLOAT> t1
      ; <INT + FLOAT> t2
TcExp:
  |[ /( e1{INT}, e2{INT} ) ]| ->
  |[ /(e1{INT}, e2{INT}){FLOAT} ]|
TcExp:
  |[ bo( e1{t}, e2{t} ) ]| ->
  |[ bo(e1{t}, e2{t}){t} ]|
  where <is-scalar> t
```

Figure 3. Type inferencing rules for simple Octave expressions

```
TcExp:
 |[ bo(e1{t1}, e2{t2}) ]| ->
 MBinOp(bo, e1', e2'){t}
 where <is-Matrix> t1
     ; <is-Matrix> t2
     ; <casting-matrices>
         (e1,t1,e2,t2) => (e1',e2',t)
```

Figure 4. Typechecking rules for matrix operations

rules type leaf nodes for integers, strings and complex numbers. The last three rules type several binary operators. For brevity, we only show a small subset of these rules, the rest of the rules are specified in similar fashion.

Observe that the type inference rules are specified using the concrete syntax of the object language [14]. For example, binary operations are specified using infix notation. Thus, rewrite rules are expressed in a natural way to the reader and are more concise than specifation using only abstract syntax tree notation. The concrete syntax used in the specification of the type inferencer is slightly different from the original Octave language. Since the Octave syntax is ambiguous, we use a concrete syntax that is like Octave, but not ambiguous, which should look transparent for the reader.

4.2. Matrix Expressions

Matrix expressions are typed using rules similarly to normal operators. In order to distinguish matrix operators from scalar operators and to ease the verification of matrix restrictions, they are characterized using special constructors. Figure 4 defines a rule to typecheck matrix operations, it

```
TcAssg:
 |[ v = (e{t}) ]| ->  |[ v{t} = (e{t}) ]|
  where rules(TcVar: |[ v ]| -> |[ v{t}]|)
```

Figure 5. Typing assignments

characterizes them with `MBinOp` to denote matrix binary operations and it insert casting operations if required. In order to apply certain transformations we must have exact information about the shape of the matrix. This shape information is added during shape analysis.

4.3. Matrix Expansion

In Octave a scalar function or operation can be lifted to apply to matrices. For example, consider the statement `a = abs([-2,5;3,-1]);`. While the function `abs` is defined for scalar types, it can be applied to a matrix, which entails the application of the function to each element of the matrix. This feature is called *matrix expansion*. The function that can be applied can be an intrinsic function or a user-defined function. Element-wise operators are instances of matrix expansion operations. When the type system encounters matrix expansion operations, it captures and characterizes them by representing these operations with a special term constructor, which reduces the number of cases to deal with in subsequent phases of the system.

4.4. Assignments

Octave has no explicit variable declarations. Assignments denote a kind of declarations, where the type of a variable is determined by the type of the assigned expression. Inferred types from variables are propagated toward their use by means of dynamic rules [12]. Dynamic rules are used to propagate information following the data flow path imposed by the programs [11, 5]. Figure 5 shows how dynamic rules are created at the point of assignments. The strategy `TcAssg` matches an assignment `v = (e{t})`, that is an assignment of expression `e` with type `t` to variable `v`. For this specific variable `v`, a dynamic rule `TcVar` is generated, which annotates an occurrence of that variable with the type `t`. The dynamic rule `TcVar` thus propagates the type of a variable to all its uses. Multiple assignment statements will create multiple rules to propagate corresponding types of each assigned variable.

4.5. Constant Folding

As part of the type annotation transformation, run-time type checks expressed in the program using intrinsic functions such as `isstr` and `isempty` can be eliminated when the type of the argument expression is known. Instances for these rules can be seen in Figure 6.

```
EvalFunc:
  |[ isstr( e{STRING} ) ]| ->
  |[ (~int: 1) {INT} ]|

EvalFunc:
  |[ isstr( e{t} ) ]| ->
  |[ (~int: 0) {INT} ]|
  where <not(?STRING)> t

EvalFunc:
  |[ isempty([]) ]| ->
  |[ (~int: 1) {INT} ]|
```

Figure 6. Rules for eliminating type checks

5. Typing Functions

The power of problem solving environments in general and Octave in particular relies on the rich set of functions. This type of system contains a specialized library for different fields, which constitutes the main reason to use these systems for rapid prototyping. Octave functions come in two kinds. On the one hand, intrinsic functions are functions that are part of the language and are available in all Octave programs. On the other hand, scripts and user-defined functions are specified by means of the language itself and are stored in ".m" files. Functions have their own scope, where arguments are passed by value. In each function scope there is a copy that holds the argument of a function call. There is no way to have side effects on the arguments of the function call. This does not mean that functions are side-effect free. Functions can only have side-effects modifying `global` variables.

5.1. User-Defined Functions

Due to operator overloading and the lack of declarations, a user-defined function can be called with different types of arguments. In this sense functions admit "ad-hoc" overloading. A function cannot restrict the type of its arguments by means of the signature of its definition. Functions can also be very flexible with respect to the number of arguments which can be present at function calls. Another characteristic of Octave is that functions can yield multiple return values depending on the context of the call. Furthermore, not all return arguments of a function may be defined. All these features make it very hard to correctly type functions. We use program specialization as a technique to find the precise type of a function.

Our strategy is to infer the resulting type of a function call by providing the types of the arguments of a function call and propagating this information to determine the type of the result. Providing context information we can eliminate certain branches within the function body which do not contribute to the result. Context information such as the

```
tc-asg-multi(s) =
{| NumArgsOut:
   ?AssignMulti(vs, _)
   ; where(
       <length> vs => val
       ; rules(
           NumArgsOut: Var("nargout") -> val
         )
     )
   ; AssignMulti(id, s)
   ; try(TcAssg <+ TcAssg(s))
|}

tc-asg(s) =
{| NumArgsOut:
   Assign(id, id)
  ; rules(
      NumArgsOut: Var("nargout") -> 1
    )
  ; Assign(id, s)
  ; try(TcAssg <+ TcAssg(s))
|}
```

Figure 7. Rules to provide context information

```
types-func(tc) = ?(f-name, call-types)
 ; where(
    <GetFunc> f-name =>
     FuncDec(ret-vars, name, args, body)
   ;<zip'(\(Var(x), t)-> Var(x){t}\)>
       (args, call-types) => var-args
   ;<length> call-types <+ !0) => nargin
   ;(<NumArgsOut> Var("nargout") <+ !0)
        => nargout
   ;{| TcVar:
       <typecheck-body(tc)>
        (var-args, nargout, nargin, body)=> body'
     ; <take(?nargout)> ret-vars => num-ret-vars
     ; <map(try(TcVar))> num-ret-vars
         => r-vars-types
    |}
   ;<rename-func> call-types => type-names
   ;<concat-fnames> (f-name, type-names)
         => new-fname
   ;<map(get-type)> r-vars-types => ret-types
   ;<record-funcs>
    FuncDec(r-vars-types, name, var-args, body')
   )
 ;!(name, ret-types)
```

Figure 8. Typing user-defined functions

number of expected results of a function call are injected in the inference process by means of dynamic rules. Values for *built-in variables* such as `nargin` and `nargout` are added to eliminate branches of the program that are not used and/or do not contribute to the required computation.

Figure 7 shows how context information is discovered and injected into the system. At the point of assignments the system discovers the number of the expected values, which is propagated by the local dynamic rule `NumArgsOut`.

5.2. Intrinsic or Library Functions

To type intrinsic functions, context information is also required, recall the example on the function `find` discussed in the introduction. The question arises how to incorporate functions that can yield a set of different values. For intrinsic functions we do not have a program to infer the resulting type from or to extract further information. Our solution is to encode information about intrinsic function in a data base. Although this idea is not new [4], our approach considers functions with multiple results. From this repository dynamic rules are created that will provide the information about the set of possible outcomes. The decision to extract the proper one is delayed until more information is discovered. This delay leads to select the proper resulting type from the set.

Table 1 shows a classification of functions into 7 groups. In the first group, the functions are defined for scalar types, the number of in and out arguments is fixed, and the outcome value is also a scalar type. For example the function `abs(4.3 + 2i)` will result into `4.7424`. These functions can be typed statically provided some information about the argument types.

Functions in the second group can be typed by having information about the value of the argument of the call. The function `sqrt(n)` will give a *Float* type as a result for `n` bigger or equal to zero. For negative instances of `n` the result will be of type *Complex*.

Functions in group three can have a variable number of arguments, the number of results is fixed. To type functions in this group it is required to know the type of the arguments. For example, the function `cumsum` will preserve the type and shape of the first argument of the call.

Functions in group four accept different numbers of arguments and the number of results is static. As an example consider `zeros(4,3)`, which will yield a matrix containing integer values, namely the number zero. Functions in this group such as `eye`, `rand`, `ones` can be typed statically.

The fifth group is characterized by having variable number of in arguments and also a variable number of return values. Functions in this group can be typed with additional context dependent information. An example for this group is the function `size([1 3 6])`, which can yield one or two return values. To infer the type for functions in this group, we encode all the possible outcome types and the selection of the proper type is delayed until the point where further information is encountered.

Groups VI and VII will be discussed in section 6.

```
[// fname,[arg-types], [ret-type]
 ("abs",  [INT],    S([INT]) ),
 ("abs",  [FLOAT],  S([FLOAT]) ),
 ("abs",  [COMPLEX],S([FLOAT]) ),
 ("find", [INT],    M([INT, [INT, INT],
                      [INT, INT, INT]])),
 ("find", [FLOAT],  M([INT, [INT, INT],
                      [INT, INT, FLOAT]]))
]
```

Figure 9. Intrinsic library typing rules

5.3. Function Specialization

Program specialization is a compiler technique to achieve a residual program, given part of the input. In the functional programming community type-based specialization is applied to obtain a residual expression and a residual type [7]. For typing Octave, the same criteria have been used. Data type information inferred by the system is used to trigger specialization. Furthermore, some of the information available to the interpreter, is applied during transformation as well. This includes the arity of function calls, the expected number of results of a function, and the types and shapes of *intrinsic functions.* Residual programs restrict the types that can be assigned to function results.

Function specialization is part of partial evaluation. Basically, there are two sorts of partial evaluators: online and offline. Our function specializer is an online partial evaluator. It determines 'static' type information during transformation and propagates this information as it is encountered and uses it to obtain residual programs. Since the transformation only propagates type information, the normal risks of online partial evaluation do not apply.

The disadvantage of this approach is that it increases code size. However, as a side effect of function specialization, branches that are not reachable are eliminated from residual programs. Furthermore, the code obtained is much more suitable for compilation and optimization.

6. Shape Inference

Shape inference is done using the same traversal as the data type inferencer. The same sort of rules are defined for most of the language constructs such as expressions, statements, and intrinsic and user defined functions. The main difference with type inference is that this phase does not performe function specializations. Shape inference extends the information discovered by adding information about the number of rows or columns. Octave, in contrast to Matlab, only has arrays of two dimensions.

Expressions Inferring shape for scalar expressions is easy, the shape information is added to the annotation term

```
SaExp:
  |[ e{INT}]| -> |[ e{INT, S(1,1)} ]|
SaExp:
  |[ bo( e1{s}, e2{s}){t} ]|  ->
  |[ bo(e1{s}, e2{s}){s} ]|
SaExp:
  MBinOp(op, x{sx}, y{sy}){t} ->
  MBinOp(op, x{sx}, y{sy}){sx}
  where <PLUS + MINUS> op
      ; <eq> (sx, sy)
```

Figure 10. Shape inferencing rules scalar Octave expressions

as is depicted in Figure 10. The construction `S(1,1)` denotes a shape with the number of rows and columns of an expression. The last rule in the picture infers the shape for the matrix operations addition and subtraction. The rule verifies that the shapes of the matrices are equal. Otherwise the expression is transformed into an `if` statement which uses symbolic computation to reproduce run-time behaviour. Similar rules to infer shapes of matrix expressions are defined in the system. Matrix expansion operations are also characterized and these operations preserve the shape of the matrix.

Intrinsic Functions Intrinsic functions can provide means of extracting type and shape information, especially from functions such as `rand`, `zeros`, `ones`, `eye`, which define the shape of arrays.

Joisha et al. classify intrinsic functions in three types [10]. Shape of Type I functions can be determined statically. Symbolic determination is required to find out the shape of functions in Type II. Functions that are not in Type I or II are functions in Type III. In addition, we have also considered the flexibility for different arguments of the call and different return values. We have extended the classification taking these aspects into account. Table 1 contains information on shapes.

Shape can be inferred statically for functions in group I and II. Type and shape information for functions in group III are argument dependent, maybe it is more accurate to say it can preserve type and shape of one argument, in the example it preserves the first argument of the call. To extract shape information for functions in group IV, it is required to know the value of the arguments of the call.

The `size` function is in group V, because it can be typed and shaped according to the context.

Functions in Type VI can be typed statically as for example in case case of the function `find`, the shape information depends on the values that are contained in the argument call. Accurate information is not possible to extract, and this is the reason to open an extra group for *dynamic* shape determination.

Group Func	arguments #In	arguments # Out	Type	Shape
I	Fixed	Fixed	Argument dependent	Static Static
II	Fixed	Fixed	Value dependent	Static Static
III	Variable	Fixed	First Arg. dependent	First Arg. dependent
IV IV	Variable	Fixed	Static	Argument dependent
V	Variable	Variable	Context dependent	Context dependent
VI	Variable	Variable	Static	Dynamic
VII	Variable	Variable	Dynamic	Dynamic

Table 1. Shapes of intrinsic functions

Group VII describes functions that are dynamically typed and shaped, for instance consider the function `eval`. This function can evaluate any function call and the result can be of any type and shape.

7. Discussion

7.1. Related Work

In the literature, the problem of typing Matlab is solved by a combined approach of a static and dynamic type inferencing system [4, 8]. Other approaches include user annotations [3] and type speculation as mechanism for just in time compilation [1].

Rose [4] uses data-flow analysis to propagate type information and they resort to inserting many possible branches when type information is not available. To extract shape information from intrinsic functions, a data base is mentioned in his work, but no further details are provided.

In [8], shape inferencing is modeled by an algebra which infers shapes to resolve operator overloading and to avoid dynamic shape inferencing. Shape information is also discovered by considering the algebraic properties such as identity, associativity, commutativity, and idempotency, preserved by operators. The authors classify intrinsic functions in the three categories that have been discussed above. Because their system can infer types using symbolic computation and using algebraic properties satisfied by an operation, more accurate shape information can be extracted. As a restriction they mention that function calls have to be of the same type, i.e., the same context. This model has been implemented in the MAGICA system [9], which requires the user to provide code in single static assignment form and single operator form. Usings this type system inside a compiler implies a dependence on MATHEMATICA and MAGICA.

The work from [3] is a bit more ambitious than ours, in that they want to infer all possible types which can be valid for a Matlab program. With the information of the inferred types for argument expressions, specialized functions can be invoked using techniques known as telescoping compilers. This system requires some user provided annotations in order to infer proper types. Although they mention that their system can deal with a limited range of variability which respect to number of arguments they do not give further details.

Our system does not require any annotations from the user. It makes overloading explicit by function specialization, which restricts the input and output types of functions to the ones that are used in actual function calls, and produces code that is suitable for compilation. Code that is unreachable based on type checks is eliminated. This approach handles context typing information, and variability in input arguments and output results.

7.2 Conclusion

This is work has been done in order to type Octave and to create a user friendly compiler which does not require further effort from the user. In order to resolve as much type information as possible statically, we have used several compiler techniques, which are normally used for optimization. With a combination of partial evaluation, context information propagation and function specialization, the current framework can eliminate most dynamic type checks from library functions.

Acknowledgments We would like to thank Gordon Cichon for his help during this project.

References

[1] G. Almási and D. Padua. MaJIC: compiling MATLAB for speed and responsiveness. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language desing and implementation*, pages 294–303, Berlin, Germany, July 2002. ACM Press.

[2] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Software—Practice & Experience*, 30:259–291, 2000.

[3] A. Chauhan, C. McCosh, and K. Kennedy. Type-based speculative specialization in a telescoping compiler for matlab, January 2003. Techical Report TR03-411, Rice University.

[4] L. A. de Rose. Compiler techniques for matlab programs, 1996. PhD Thesis, University of Illinois at Urbana-Champaign.

[5] E. Dolstra and E. Visser. Building interpreters with rewriting strategies. In M. G. J. van den Brand and R. Lämmel, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'02)*, volume 65/3 of *Electronic Notes in Theoretical Computer Science*, Grenoble, France, April 2002. Elsevier Science Publishers.

[6] J. Eaton. Octave. `http://www.octave.org/`.

[7] J. Hughes. An Introduction to Program Specialisation by Type Inference. In *Functional Programming*. Glasgow University, July 1996. published electronically.

[8] P. G. Joisha and P. Banerjee. Computing array shapes in MATLAB. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Lecture Notes in Computer Science Series, Cumberland Falls, USA, August 2001. Springer-Verlag.

[9] P. G. Joisha and P. Banerjee. MAGICA a software tool for inferring types in MATLAB, October 2002. Technical report No. CPDC-TR-2002-10-04.

[10] P. G. Joisha, U. N. Shenoy, and P. Banerjee. Technical report no. CPD-TR-2000-10-010, October 2000. Center for Parallel and Distributed Computing, Departament of Electrical and Computer Engineering, Northwestern University.

[11] K. Olmos and E. Visser. Strategies for source-to-source constant propagation. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies (WRS'02)*, volume 70 of *Electronic Notes in Theoretical Computer Science*, page 20, Copenhagen, Denmark, July 2002. Elsevier Science Publishers.

[12] E. Visser. Scoped dynamic rewrite rules. In M. van den Brand and R. Verma, editors, *Rule Based Programming (RULE'01)*, volume 59/4 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, September 2001.

[13] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[14] E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[15] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.