Implementation of First Class Functions and Type Checking for a Multiparadigm Language

by

Vinoo Cherian

A research paper submitted in partial fulfilment of the degree of Master of Science

Major Professor: Dr. Timothy A Budd

Department of Computer Science
Oregon State University, Corvallis, Oregon

May 23, 1991

Abstract

A multiparadigm language is one which combines features of different language paradigms. Leda is a strongly typed, compiled, multiparadigm language with facilities for imperative, functional, object oriented and relational programming. This report describes the type checker of the Leda compiler and the implementation of first class functions required for functional programming. Closure analysis is done to determine if a function can outlive its defining context. If the defining context is not on the activation stack at the time of the function invocation, the context is allocated on the heap. Type checking rules for Leda are presented. These rules illustrate the interaction between the different paradigms. The report also describes the development of the Leda compiler using an object oriented paradigm.

Acknowledgements

I would like to thank my advisor Tim Budd and the other members of the Leda compiler development project , Wolfgang Pesch and Jim Shur, for all their help and encouragement.

Table of Contents

Chapter 1

Introduction

Leda is a strongly typed, compiled, multiparadigm language. This work involves the implementation of first class functions for programming in the functional paradigm and the implementation of the type checker of the compiler.

1.1 Language Paradigms and Multiparadigm Languages

Programming languages can be classified into different groups on the basis of the underlying programming model. Each programming model presents a different way to view programming. Each of these models is called a paradigm.

Languages like C, Pascal and Fortran use the imperative paradigm. Programming in this paradigm emphasizes assignments to variables. An assignment changes the state of the machine by changing the value of a location in memory. Imperative programming languages therefore provide a convenient programming model of the underlying machine. Such languages usually have structured control flow statements like loops and conditionals. Fig 1.1 shows a fragment of C code which computes the factorial of 9.

```
void main(void)     /* find the factorial of 9 */
{
   int x, fact;

  x = 9;
  fact = 1;
  while (x > 1) {
     fact = fact * x;
     x = x -1;
  }
}
```

Fig 1.1 The Imperative Paradigm (C)

Languages using the functional paradigm include Lisp, Scheme and ML. They have functions as values and use functional composition to develop higher order functions. A higher order function is one whose arguments or results are themselves functions. In pure functional programming the value of an expression depends only on the values of its subexpressions, if any.

There are no side effects in pure functional programming and therefore no assignments. Many of the functional languages however do allow assignments but programming in them largely follows the pure functional style [Set89]. Storage management is usually taken care of by the system in these languages. Fig 1.2 shows an example of a filter function in ML which copies a binary tree (btree) of integers, applying the filter function to each of the elements as they are copied. The btree is defined in ML as

datatype btree = leaf | node **of** int * btree *btree.

If **x** is a btree, the result of **map(x, sq)** is a new tree in which each value is squared.

```
fun     sq(value) = value*value;

fun     map(leaf, filterFun) =
                leaf
|       map(node(value, left, right), filterFun) =
              node(filterFun(value), map(left, filterFun), map(right, filterFun));
```

Fig 1.2 The Functional Paradigm (ML)

In the object oriented paradigm, the language views data items as representations of real world objects, which interact with each other by sending and receiving messages. These data items, called objects, are instances of classes. The messages which an object can respond to is determined by its class. Objects are seen as representing independent communicating agents in the object-oriented style of programming. Inheritance and data encapsulation are also major concepts. Inheritance allows new classes to be defined as extensions of previously defined ones. Data encapsulation is provided by classes. Methods are provided as the means to manipulate data in a class. This hides the representation of data from the user. Since the representation of data is often unnatural, this data encapsulation enables a programmer to think more in the abstract rather than in terms of a concrete data representation [Kam90]. Smalltalk, Object Pascal and C++ are examples of object-oriented languages. Fig 1.3 shows a C++ example of a class **point** and its subclass **circle**.

The logical paradigm emphasizes relationships between objects. An example of a logical language is Prolog. In logic programming relations are used instead of functions. Relations are used to represent facts and rules provided by the programmer. The languages use deduction to answer queries. A relation treats arguments and results the same. In other words flow of data in a relation is bi-directional. Backtracking is used to find solutions and unification is used to allow the use of variables as place holders for data to be filled in later [Set89]. Fig 1.4 shows Prolog code to represent information about family relationships and queries about the realationships.

A particular paradigm may be chosen to solve a problem because it results in a more

```
class point {                                   // declaration of class point
protected:
  int x;             // fields x and y are protected and can be accessed only through methods
  int y;
public:
  point(int v1, int v2) { x = v1; y = v2;}      // constructor for class point
  void print() { cout << x << y;}               // print is a method
};

class circle : public point {                   // circle is a subclass of point
protected:
  int radius;
public:
  circle(int v1, int v2, int v3) : (v1, v2) {radius = v3;} // constructor for class circle
  void print() {cout << x <<  y << radius;}
                          // this print overriddes in the declaration in the class point
};

...

point p1(5, 6);                                 // create a point
circle  c1(0,1,5);                              // create a circle

p1.print();                                     // print the point
c1.print();                                     // print the circle
```

Fig 1.3 The Object-Oriented Paradigm (C++)

```
mother(MOM, KID) :- child(KID, MOM, DAD).
child(helen, leda, zeus).
child(hermione, helen, menelaus).
child(castor, leda, tyndareus).
child(pollux, leda, zeus).
child(aeneas,aphrodite,anchises).
```

Rules and facts

```
?- mother(leda, helen).
yes
?-mother(leda, X).
X = helen;
X = castor;
X = pollux;
no
```

Queries

Fig 1.4 The Relational Paradigm (Prolog)

succinct, efficient and easy implementation compared to another paradigm. For large problems it is conceivable that different parts of a problem may be best programmed in different paradigms. This requires a language which has features that allow different paradigms to be used in solving a programming problem. Such a language is called a multiparadigm language [Hai86][Kor86].

1.2 Leda

Leda is a multiparadigm language that attempts to combine features of four common language paradigms - imperative, functional, object-oriented and logical. Each paradigm represents a particular outlook of the world as described in the previous section. Leda was designed to enable programming in any of the above four paradigms or in a combination of paradigms [Bud89c] [Bud91a].

Leda is a strongly typed, compiled language. Functions can be passed to other functions and returned from functions and so are first class values. All values are objects. This allows different data types to be treated uniformly. A class hierarchy is provided to enable the user to define new types. Relations are provided to support the relational paradigm. These features enable programming in different paradigms. The different paradigms are illustrated by examples below.

```
var                // find the factorial of 9
   fact, x : integer;
begin
  x : = 9;
  fact : = 1;
  while (x > 1)
    begin
       fact := fact * x;
       x := x -1;
    end;
end;
```

Fig 1.5 Imperative Programming in Leda

Fig 1.5 shows imperative programming in Leda. **while, for** and **repeat-until** loops and **if-then-else** statements are the structured control flow structures available.

Fig. 1.6 shows an example of object oriented programming in Leda. Each class has a shared part which is common to all instances of the class and an unshared part which is unique to each instance of a class. **point** is a class with one shared method, **print**. **circle** is a subclass of **point** and it overrides the definition of **print**. The methods themselves can be assigned to in the body of the program (as **circle.print** is) or declared in a procedural style (as **point.print** is). **p1.new**() creates a new point and values can be assigned to the unshared portion.

```
type
  point := class
     x : integer;
     y : integer;
  shared
     print : method();
  end;

  circle := class of point                      // circle is a subclass of point
    radius : integer;
   shared
    print :  method();
  end;
var
   p1 : point;                                  // define a point and a circle
   c1 : circle;

method point.print();
begin
       x.print();
       y.print();
end;

begin

  circle.print := function(c : circle);         // conversion of function to method
                 begin
                      c.x.print();
                      c.y.print();
                      c.radius.print();
                 end;
  p1 := point.new();                              // create a new point
  p1.x := 5;
  p1.y := 6;
  p1.print();                                   // print the point

  c1 := circle.new();                             // create a new circle
  c1.x := 0;
  c1.y := 1;
  c1.radius := p1.x;
  c1.print();                                   // print the circle
end;
```

Fig 1.6 Object Oriented Programming in Leda

Functional programming in Leda is illustrated in Fig 1.7. A binary tree (**btree**) in Leda can be defined by two classes **leaf** and **node** which are subclasses of a general class **btree.** In a ML like language, btree would be defined as

datatype btree = leaf I node **of** int * btree * btree.

map is defined on the btree data structure. If x is the binary tree, the result of x.map(sq) is a new tree in which each value is squared. **map** can be passed a function of type function(integer)->integer [Set89][Bud89c]

```
function sq ( value : integer)->integer;
begin
  return value*value;
end;

method leaf.map(filterFunc : function(integer)->integer)->btree;
begin
  return self;                          // return the receiver of the message
end;

method node.map(filterFunc : function(integer)->integer)->btree;
begin
  return node.new(filterFunc(value), left.map(filterFunc), right.map(filterFunc));
end;
```

Fig 1.7 Functional programming in Leda

```
relation child(var name, mother, father : names);
begin
  child(helen, leda, zeus).
  child(hermione, helen, menelaus).
  child(castor, leda, tyndareus).
  child(pollux, leda, zeus).
  child(aeneas,aphrodite,anchises).
end;
```

A Relation consisting of only facts

```
relation mother(var mom, kid : names);
var dad : names;
begin
  mother(mom, kid) :- child(kid, mom, dad).
end;
```

Fig 1.8 Relational programming in Leda

Relational programming in Leda is provided by a procedural abstraction called the relation. The body of the relation consists of rules similar to Prolog. The rules can be either facts or rules of inference as illustrated in Fig 1.8. **var** parameters allow the flow of information in either direction. Implicit backtracking is invoked when more than one answer is possible [Bud91a].

One of the difficult parts of multiparadigm language design is to provide a language in which the different paradigms easily blend together. This is more than interfacing features of different paradigms. There must be some internal consistency so that the different paradigms work well together and independently. Leda is a step in this direction of providing a combination of paradigms which is more than the sum of its parts.

Chapter 2

First Class Functions

2.1 The Upward and Downward Funarg Problem

In block structured languages, the environment in which a function is defined can be the main program or any other function. This causes complications when functions are passed as arguments or when functions are returned as values of other functions. This is called the funarg (functional argument) problem.

Consider the Leda program shown in Fig 2.1. The environment of function D is function C. The variable x that is incremented in function D is defined in function C. This variable must be

```
var
  A : function();
begin
  A := function()
       var
          B : function(f : function());
          C : function();
      begin
          B := function(f : function());
               begin
                    f();
               end;

          C := function();
               var
                    x : integer;
                    D : function();
               begin
                    D := function();
                         begin
                            x := x +1;
                         end;

                    B(D);
               end;
          C();
       end;
  ....
```

Fig 2.1 The Downward Funarg Problem

accessible to function D when D is passed as a parameter to B. Since B is invoked from C, the activation record for C is on the stack at the time of the invocation of B. The activation record of C contains the variable x which must be accessible to function D when it is invoked from within B. This can be done by passing along with the functional argument D, a pointer to its environment i.e. a pointer to the activation record of C. The downward funarg problem can therefore be solved by passing a closure which contains a pointer to the code of the function and a pointer to its environment. The environment of a function is guaranteed to be on the stack or in the global area when the function is passed as an argument. [Aho86]

```
type
  intfun := function(integer)->integer;
var
  A : function(integer)->intfun;
  C : intfun;
  ans : integer;
begin
  A := function(a : integer)->intfun;
       begin
          return function (b : integer)->integer;
                 begin
                   return a + b;
                 end;
       end;
  C := A(5);
  ans := C(7);
end;
```

Fig 2.2 The Upward Funarg Problem

Solving the problem of allowing functions to be returned from other functions (the upward funarg problem) is more difficult. In this case the activation record which is the environment of the function being returned may no longer be on the stack. Consider the Leda program in Fig 2.2. Function A returns a pointer to a function which accesses **a**, the parameter of A. This function is assigned to C. The storage for the parameter **a** is destroyed once the invocation of A has been completed. The function assigned to C however requires the value of **a** when it is invoked. Solving this problem requires some additional mechanism to keep the activation record of A around even after its invocation.

2.2 Functions and Methods in Leda

Since Leda supports the functional programming style, functions are first class data types. Functions are therefore no different from other data values. Consider the class declaration in Fig

2.3. **line** is a class with one unshared function **lineFunction** which holds the line equation and three shared methods **drawTransform, getNewEndPoints** and **isVisible**. The shared data is common to all instances of a class while the unshared data is different for each instance of the class. **lineFunction** holds the line equation, **drawTransform** draws the transformation of a line, **getNewEndPoints** transforms a line and obtains its intersections with a graphics screen and **isVisible** indicates if a line is visible.

The difference between methods and functions is as follows. The method **drawTransform** has an instance of line as an implicit receiver. The transformation is done on this receiver. A method can access the class fields of its receiver. A function on the other hand has no

```
line := class
   lineFunction : function(real, real)->boolean;
shared
   drawTransform : method(transform);
   getNewEndPoints : method(transform, var point, var point);
   isVisible : function()->boolean;
end;
```

Fig 2.3 Declaration of class line

```
var
  aLine : line;
  t : transformation;

method line.drawTransform ( t : transform);
  { transforms a line and draws the part which falls within the screen }
var start, end : point;
 begin          // get screen Intersections of transformed line
    self.getNewEndPoints(t, start, end);
    MoveTo(start);              // draw transformed line
    DrawTo(end);
end;

function line.isVisible() -> boolean;
  { indicates if a line is visible}
begin
  return true;
end;

begin
  aLine := line.new();                                    // create an instance of Line
  ...
end;
```

Fig 2.4 Definition of Methods and Functions

```
var
    aLine : line;
    t  :  transformation;

method line.drawTransform  ( t : transform);
  { transforms a line and draws the part which falls within the screen }
var start, end : point;
 begin          // get screen Intersections of transformed line
    self.getNewEndPoints(t, start, end);
    MoveTo(start);                  // draw transformed line
    DrawTo(end);
end;

begin
  ...
  line.isVisible := function () -> boolean;
                         begin
                           return true;
                         end;
  aLine := line.new();                                          // create an instance of Line
  aLine.lineFunction := function(xCord , yCord : real)->boolean;
                         { checks if point (xCord, yCord) is on the line }
                         begin
                                   if (4*xCord + 3*y = 8) then
                                      return true;
                                   else return false;
                         end;
  ...
end;
```

Fig 2.5 Alternate Initialization of Methods and Functions

```
var
  aLine : line;
  t : transformation;
begin
  ...                              // assign a function to a method. Allowed since the first
                                   //  parameter of the function matches the receiver of the
                                   // method  drawTransform
  line.drawTransform  :=function(thisLine : line;  t : transform);
                           { transforms a line and draws the part which falls within the screen }
                                    var start, end : point;
                                   begin
                                      thisLine.getNewEndPoints(t, start, end);
                                      MoveTo(start);
                                     DrawTo(end);
                                   end;
...
end;
```

Fig 2.6 Assigning a function to a method

implicit receiver.

Fig 2.4 shows one form of definition of shared functions and methods. This is merely syntactic sugar for assigning a value to the field. In the case of shared class fields, the function or method name must be prefixed by the name of the class. For example, since **drawTransform** is a field of the class **line**, it is qualified by the class name. Functions or methods which are fields of a class must be declared in the class declaration before being defined (as in Fig 2.3). Functions which are not class fields can also be defined using this syntax and need not be declared earlier. Functions and methods defined using this form can be reassigned other values later in the program using assignment.

Fig 2.5 shows alternate syntax for defining methods and functions by assignment. Shared data fields must be qualified by the class name and unshared data fields must be qualified by the name of the instance of the class when they are being assigned to. Thus **line.isVisible** is legal but **line.lineFunction** is not as targets for assignment. **line.lineFunction** can however be used as a value of an expression. Functions (but not methods) can also be created as nameless values. These values can be passed as arguments, returned from other functions as values and assigned to variables.

Functions can be converted to methods. This is illustrated in Fig 2.6. The first parameter of the function must correspond to the receiver of the method. Methods can also be converted to functions. The receiver of the method must correspond to the first parameter of the function [Bud91b].

Functions or methods can be invoked in the postfix style object oriented fashion or in a prefix style functional fashion. Thus **aLine.drawTransform(t)** can also be invoked as **drawTransform(aLine, t)**.

2.3 Representation of Functions and Methods

Functions and methods in Leda are represented as objects. Each object is an instance of a funtion class. The layout in memory of a method or function is shown in Fig 2.7. The reference count indicates the number of variables that refer to the object. This is required since Leda uses pointer semantics. The object can be garbage collected when the reference count falls to zero. Garbage collection is not done in the current implementation. The shared field points to a shared table of fields which all instances of a class share. Functions and methods do not have any shared table. The next field points to the code of the function and the last field points to the context in which the function was defined. This can be a pointer to an activation record on the stack or a context allocated on the heap, as will be discussed in the next section.

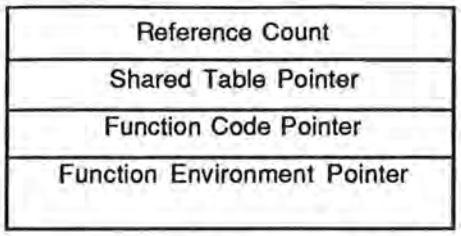

Fig 2.7 Representation of Methods and Functions in Leda

```
class programASTnode : public scope {              // holds mainprogram, functions & methods
 protected:
  DTnode      *args;                // formal parameters
  DTnode      *constantdefs;        // constant definitions
  DTnode      *typedefs;            // type definitions
  DTnode      *variabledefs;        // local variable declarations
  DTnode      *reciever;            // receiver in the case of a method
  ASTnode     *subprogramType;      // a type constructor including the receiver,
                                                  // parameter and return types
  ASTnode     *statements;          // list of statements that make up the subprogram
  escInd      closureInd;           // indicates if there is an escaping closure
  ASTnode     *parentLevel;         // pointer to the enclosing subprogram
 public:
  ...
};
```

Fig 2.8 C++ Class Representation of a Subprogram in the Leda Compiler

```
class funcType : public classType {                // type of a function
 private:
  typeArgsNode        *params;        // types of parameters (including passing mode)
  ASTnode             *returnType;    // return Type. A function need not return a value.
 public:
  ...
};

class methType : public classType {
 private:
  ASTnode             *receiverType;  // type of receiver
   typeArgsNode       *params;        // types of parameters (including passing mode)
  ASTnode             *returnType;    // return Type. A method need not return a value.
 public:
  ...
};
```

Fig 2.9 C++ Class Representation of Function and Method Types in the Leda Compiler

The compiler represents functions and methods as C++ classes. The class holds the formal parameters, local variable declarations, local constant and type definitions, statements in the subprogram, type of the receiver (in the case of a method), the subprogram type and a closure indicator to indicate if there is an escaping closure (discussed in the next section). Fig 2.8 shows the class which represents a function or method.

The function or method type is also represented in the compiler as a C++ class. These classes are shown in Fig 2.9. Type checking involves checking each parameter and the the return type. The reciever type is also type checked in the case of a method.

2.4 Implementation of First Class Functions

Every function in Leda is a closure, that is, an object with a pointer to the code and a pointer to the function environment. Before a function is invoked, its environment pointer is pushed on the stack. This forms the static link. Solving the downward funarg problem is trivial since every function carries with it its environment pointer. Traversing the static link gains access to the environment.

Solving the upward funarg problem is more difficult. Closure analysis is first done to determine whether any nested function can outlive the context in which it is defined. This is called an escaping closure. Closure analysis is done as follows. Every assignment statement is analysed to see if a function is assigned to a variable outside the current block. Every return statement is analysed to determine if a function is being returned. In either case the current function is flagged as having an escaping closure. If a nested function has an escaping closure, the current function is also flagged as having an escaping closure. This is because a block forms part of the environment of any nested block. Closure analysis is done during the first pass [Kra88].

If a function is flagged as having an escaping closure, space is allocated on the heap for its parameters, local variables, return address, static link and dynamic link. The parameters which were pushed on the stack by the calling subprogram are then copied to the heap. No space is allocated on the stack for the local variables since the space on the heap is used. The environment on the heap has the same layout as an activation record as shown in Fig 2.10. The return address and dynamic link fields of the environment on the heap are however unused.This allows uniform treatment of activation records and heap environments when the offsets of the parameters and local variables are calculated. In the current implementation which generates 68000 code, register a4 holds the pointer to the current function environment. This could be either a pointer to the activation record (if there is no escaping closure) or it could be a pointer to environment on the heap (if there is an escaping closure) .

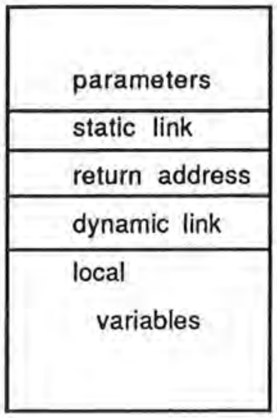

Fig 2.10 Layout of environment on heap and stack

Consider the Leda program in Fig 2.11 which defines a curry. A curry of a binary function permits the arguments to the function to be bound one at a time. In this example **x** is a curry of the binary function **plus.** When x is invoked with a single argument 5, it returns a function of one argument with **f** bound to **plus** and **a** bound to 5. When this function is invoked with argument 7, it returns the sum of 5 and 7.

Closure analysis of the curry example results in curry and the function it returns to be flagged as having escaping closures since both of them return functions. When curry is invoked, its context (environment) is allocated on the heap. The parameter **f** which was pushed on the stack, is copied to the heap. The function object returned by curry and assigned to **x** is shown in Fig 2.12. On completion of the invocation, the dynamic link and the return address on the stack are used to return to the caller. When **x** is invoked, it creates a second context on the heap. This context holds parameter **a** and its static link points to the earlier context . The function object returned by **x** and assigned to **y** is shown in Fig 2.13. Fig 2.14 shows the program state when **y** is invoked. The context for **y** is on the stack since it has no escaping closure. The static link points to the parent context of **y** which is on the heap. This context holds the parameter **b** and inturn points to its calling context which is also on the heap. This context holds the function **f** which is bound to **plus** [Bud89a]. When **f(a,b)** is invoked from within **y**, the list of contexts is searched to find the values that **f, a** and **b** are bound to. Since **f** is bound to **plus, a** to 5 and **b** to 7, 12 is assigned to **i** the main program.

```
type
     binaryFunction := function(integer, integer)->integer;
     unaryFunction := function(integer)->intFunction;
     intFunction := function(integer)->integer;
var
     x : unaryFunction;
     y : intFunction;
     plus : binaryFunction;
     curry : function(binaryFunction)->unaryFunction;
     i : integer;

begin

     plus := function(a, b : integer)->integer;
        begin
              return a+b;
        end;

     curry := function(f : binaryFunction) -> unaryFunction;
           begin
                return function(a : integer)->intFunction;
                  begin
                           return function (b:integer)->integer;
                             begin
                                   return f(a,b);
                             end;
                  end;
        end;

     x := curry(plus);
     y := x(5);
     i := y(7);
     i.print();

end;
```

Fig 2.11 Curry Implementation

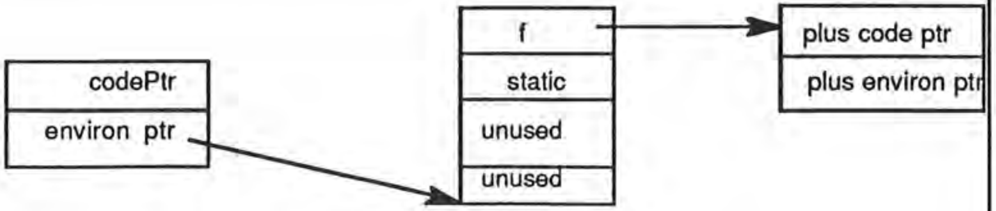

Fig 2.12 Function object assigned to x

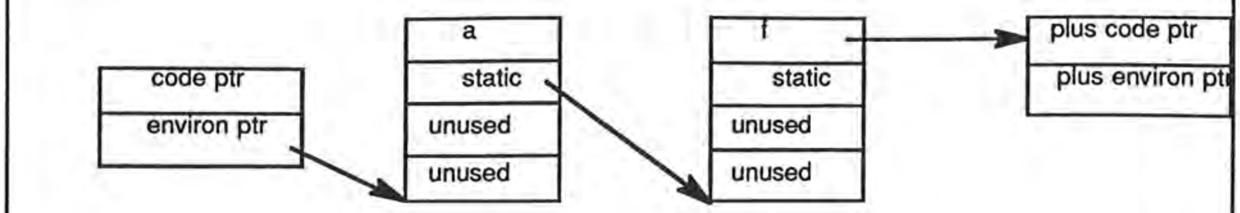

Fig 2.13 Function object assigned to y

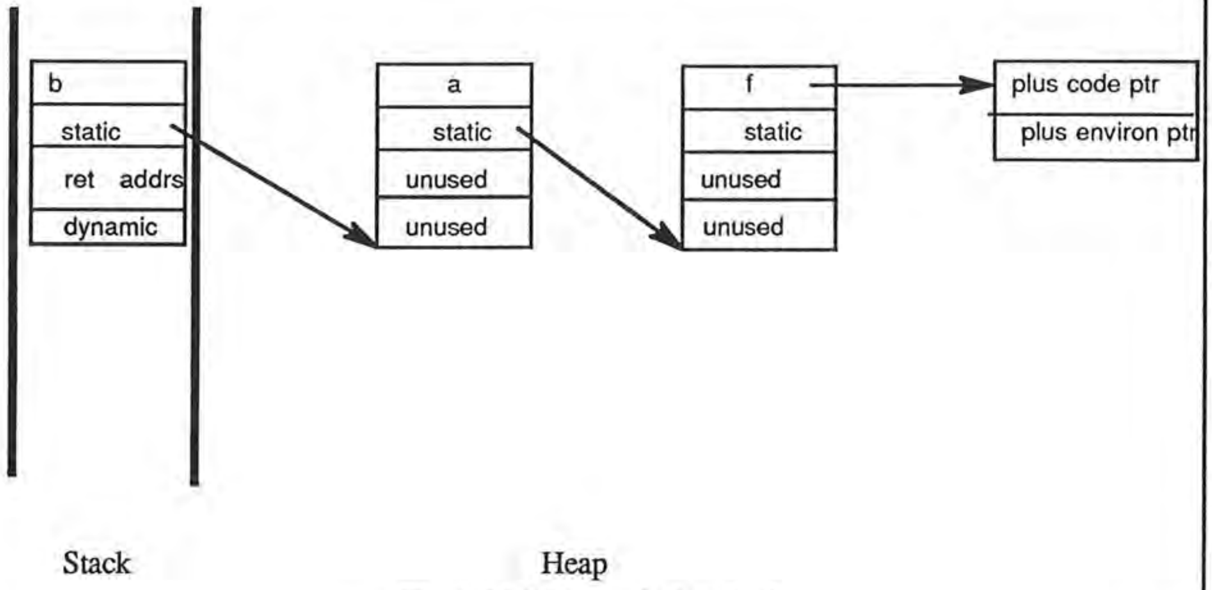

Fig 2.14 When y is invoked

Chapter 3

Type Checking

3.1 Data Types in Leda

Data Types in Leda are divided into 4 categories - standard system types, enumerated types, subprogram types and class types. All types can have aliases.

Standard system types are integer, real and boolean.

Enumerated types can be defined by explicitly listing the values that the type defined. The syntax is as follows.

```
type
        spectrum := (violet, indigo, blue, green, yellow, orange, red);
var
        color : spectrum;
```

Subprogram types can be functions, methods or relations as noted in section 1.2. Subprograms are first class values and hence variables can be declared of this type, passed as arguments and returned from subprograms as values. Parameters can be passed by name or reference (using var). In type declarations only the types of the arguments, the passing mode and the return type is specified.

```
type
        binaryFun := function(integer, var integer)->integer;
```

Class types are the major data structuring mechanism in Leda. A class encapsulates representation dependent code. A class can have many instances. All instances of a class can share a common data area. The syntax for a class definition is shown in Fig 3.1.

Data fields which follow the **shared** key word, are shared by all instances of the class. The variable **self** can be used to refer to the object itself. The **new** message can be sent to a class to create an instance of the class. Subclasses can be created by using the **of** keyword as shown in Fig 3.1.

A Subclass inherits the data fields of its superclass. The subclass can override any of the data fields. In the above example circle overrides the print method in point.

```
type
      point := class
         x : integer;
         y : integer;
      shared
         print : method();
      end;

      circle := class of point                    // circle is a subclass of point
        radius : integer;
       shared
        print :  method();
      end;
var
       p1 : point;                                // define a point and a circle
       c1 : circle;


    begin
     ...

      p1 := point.new();                           // create a new point
```

Fig 3.1 Class Definition Syntax

Parameterized Classes are supported in Leda. The parameters can be instantiated with type values to create a new class. This is illustrated below.

```
type
        list := class(T)
                item : T;
                next : list(T);
           end;
```

A list of integer can be declared as follows:-

```
var
        integerList :  list(integer);
```

All variables are undefined before the first assignment. This can be checked using a system defined boolean function **defined**(x) [Bud89b].

3.2 Type Checking Rules

The following type checking rules are used in Leda.

Integers, Reals and Booleans

Integers match integers, reals match reals and booleans match booleans. No other combinations involving these types match. Integers are automatically coerced to reals on assignments, expressions and function/method calls and returns. There are no other automatic coercions.

Enumerated Types

Instances of the same enumerated type match.

Functions and Methods

Functions match functions if the types, the passing modes and number of the parameters match . The types of the corresponding parameters must match exactly or must be aliases.

Methods match methods if the types, the passing modes and number of the parameters match and if the type of the receivers match. The types must match exactly or be aliases.

Functions match methods if the first parameter of the function matches the implicit receiver of the method and if all the remaining parameters of the function match the parameters of the method. The types must match exactly or be aliases. The passing mode of the first parameter of the function must be call by value for it to match the method receiver.

All methods/functions in Leda are virtual in the C++ sense, that is, the method/function invoked when a message is sent to a variable will depend on the type of object held by the variable at that point (dynamic type) and not on the static type of the variable.

Because of pointer semantics, even if an object is passed by value, its individual fields are passed by reference. This means that changing the field of an object passed by value will change the corresponding field of the actual parameter. Of course, since the object itself is passed by value, assigning to the formal parameter will not change the actual parameter.

Classes

Class A matches class B if either of the following is true.

1. The class numbers of class A and class B are the same.
2. Class A is a subclass of class B. Note that in this case class B does not match class A.

Consider the functions in Fig 3.2 which use the classes in Fig 3.1. Circle is a subclass of point. If a circle is passed as the parameter to printXCoord instead of a point, no problems are caused since a circle also has a field **x** which can be sent the print message. Subclasses can therefore be assigned to variables, passed as parameters and returned from functions or methods.

However if a point is passed to the function printRadius which expects a circle as the parameter, the function will attempt to print the radius field of the parameter and fail. This is because a point does not have a radius field.

```
var
  p1, p2, p3 : point;
  c1, c2, c3 : circle;

function printXCord(p : point);
begin
  p.x.print();
end;

function printRadius(c : circle);
begin
   c.radius.print();
end;

function addPoints(pntA, pntB : point)->point;
begin
  ..
end;

function addCircles(crclA, crclB : circle)->circle;
begin
  ...
end;
...
p1 := addPoints(p2, p3);
p1.x.print();
c1 := addCricles(c2,c3);
c1.radius.print();
```

Fig 3.2 Subclasses can be passed, returned or assigned instead of their superclasses

Binary Expressions

For binary expressions involving the built in types, the types of the left and right subexpressions must be identical after allowing for coercion of integers to reals.

User defined types can use the arithmetic operators +, -, * and /. These operators correspond to the methods plus, minus, times and slash in the user defined class. In this case, the corresponding method must exist, the type of the left subexpresssion must match the type of the receiver of the method and the type of the right subexpression must match the type of the first parameter of the method.

The type of the binary expression is the return type of the method corresponding to the binary operator.

Unary Expressions

The type of the unary expression is the type of the subexpression.

Assignments

The type of the rvalue must match the type of the lvalue. Type aliases match. Assignment of a class to its superclass is allowed. Functions and methods can be assigned to each other if they match. A warning is generated if the lvalue and the rvalue have types which are aliases.

Assignment to a shared field of the class requires that the field be prefixed by the name of the class. Assignment to an unshared field of a class requires that the field be prefixed by the name of an instance of the class.

Function/Method Invocation

If a receiver exists, the type of the receiver must match the type of the class in which the function/method is defined. Functions and methods can be invoked in a postfix style object oriented fashion or in a prefix style functional fashion as discussed in section 2.2. Thus **aLine.drawTransform(t)** can also be invoked as **drawTransform(aLine, t)**. If the latter style is used, the first parameter corresponds to the receiver of the class. When **drawTransform(aLine, t)** is seen by the compiler, a function called drawTransform which is not a class field and with the correct number and type of parameters is searched for. If it is not found, a class field called drawTransform is searched for in the class of the first parameter.

The types of the actual parameters must match the types of the formal parameters. The types of the corresponding parameters can be aliases and subclasses of the formal parameter can be passed. Functions and methods can be passed to each other if they match.

The number of actual parameters must match the number of formal parameters except when the alternate invocation style is used. In this case the first actual parameter corresponds to the receiver and the number of remaining actual parameters must match the number of formal parameters.

Return Expressions

The type of a return expression must match the type of the return type of the function/method in which it appears. Type aliases match. A subclass of the return type can be returned. Functions and methods match if the criteria discussed earlier holds.

Consider again the functions in Fig 3.2 which use the classes in Fig 3.1. If a circle is returned from addPoints instead of a point, no problems are caused since a circle also has a field **x** which can be sent the print message after the invocation. However if a point is returned from addCircles instead of a circle, printing the **radius** field of the returned value will fail. This is because a point does not have a **radius** field. Subclasses of the return type can therefore be returned but not superclasses of the return type.

Class Fields

The identifier after a period must be a field of the object or class which precedes the period. The exception to this rule is the new message to a class which is used to create an instance of a class.

A shared field can be prefixed by the name of the instance if the value of the field is only going to be used in an expression. However it cannot be assigned to. Assigning to a shared field requires that the field be prefixed by the class name. Similarly assigning to an unshared field requires that the field name be prefixed by the name of the instance.

Use of an unshared field without a prefixed instance name is allowed within a method declaration. In this case, the instance to which the field belongs is implicitly the receiver of the method. For example in Fig 3.3, the field **x** of class point is not prefixed by a name. Since **x** is neither a local variable nor a parameter, it is searched for among the class fields of point and its superclasses. If it is not found there it is searched for in the enclosing block.

Classes can override shared fields defined in the superclass. Fields which are not shared cannot be overridden. The subclass cannot move a overridden field from the shared to the unshared portion or vice-versa. With the exception of argument lists in overridden fields in methods and functions, the types of the overridden fields must be identical to the types in the superclass. Issues of covariance and contravariance allow the argument of an overridden method to be enlarged to a more general type [Coo89][Bud91b]. Similarly the return type of an overridden method can be declared as less generalized than in the parent method. This is illustrated in Fig 3.4. Assume that

```
point.plus := method(p : point)->point;
            var
              q : point;
            begin
              q := point.new();
               q.x := x + p.x;           // self is optional
               q.y := self.y + p.y;
              return q;
            end;
```

Fig 3.3 Class fields which are not prefixed

the method **getArea** in class circle is defined to take a parameter of type circle rather than shape. Suppose **s1**, a variable declared of type shape, is assigned a circle. It is legal for **s1** to accept a square as the parameter to the method **getArea**. This leads to a type incorrect situation since the method **getArea** in class circle will be passed a square instead of a circle. This does not arise if the parameters in the overridden method are generalized. Similarly a return type can be declared in an overridden method as less generalized but not as more generalized.

```
type
  shape := class
             xCord, yCord : integer;
          shared
            getArea : method(s : shape);
          end;
  circle := class of shape
             radius : integer;
          shared
            getArea : method(s : shape);
          end;
  square := class of shape
             sideLength : integer;
         end;
```

Fig 3.4 Enlarging parameter types and restricting return types in overridden methods

3.3 Type Checking Implementation

Leda allows a programmer to construct types from the basic types. The basic types are real, integer and boolean. Examples of the constructed types are class, function, method and relation.

Representation of Types

Type checking for Leda is done in an object oriented fashion. All types are subclasses of an abstract type class. Real, integer and boolean are internally represented as classes. As a result they can be treated very similar to user defined classes for the purpose of type checking. Constructed types are also subclasses of this abstract class. Each type has a unique class number which is used to compare types.

In the implementation, a special C++ class called classID (class identifier) is used to hold the information required to type check. This information consists of the name of the type, the type itself and a list of class parameters to store information about instantiated parameterized types (if they exist). Each of the parameters holds the same information.

The structure of the C++ class classID is shown in Fig 3.3. classIDlist is a list of classID.

```
class classID {            // unique identifier of class used for typechecking
  protected:
    char        *className;          // name of the class
    classType   *classStructure;     //  type
    classIDlist *classArgs;          // types of class parameters
  public:
    ...
}
```

Fig 3.3 Class used to Type Check

Type checking two types is done by first checking if the types held by the classStructure fields match. If they do, the classNames are checked for equality. If they do not match the types held in classStructure are aliases. The class parameters held in classArgs are then checked for matches. If they match, the two types match.

Constructed Types

Constructed types are described below along with the type information they hold.

User Defined Class Types

This type holds the types of the data fields of the user defined class, the name of the superclass and the types of the parameters in the case of a parameterized class. The data fields are divided into shared variables and unshared variables.

Typechecking classes is done by first comparing the unique class numbers. If the class numbers are different, the class number of the superclass is fetched and compared. This continues until a match is found or until the top of the class heirarchy is reached without a match.

Parameterized Class Types

Typechecking two parameterized class types is done by first type checking the template of the class using the unique class numbers. Once the template is found to match, each of the parameters is type checked.

Functions

The function type holds type information about the parameters and the return type.

Methods

The method type holds information about the parameters, return type and the type of the receiver.

Constructors and Type Checking

Each statement and expression in the intermediate representation of the compiler is an instance of a C++ class. Every expression has a type which can be obtained by sending a message to the class representing the expression. The type of the expression is set by the constructor of the class representing the expression according to the type checking rules.

Assignment statements, subprogram invocations and return statements are also represented by instances of C++ classes, the constructors of which do the appropriate type checking.

Chapter 4

Writing a compiler in an object oriented style

The implementation of Leda was done in C++ using an object oriented style of programming. The intermediate representation is an abstract syntax tree made up of nodes. Each node is a subclass of an abstract node. Every expression and statement is represented by such a subclass. A part of the class heirarchy is shown in Fig 4.1.

- Abstract Node
  - Assignment Statement
  - Subprogram Invocation Statement
  - Conditional Statement
  - While Loop Statement
  - For Loop Statement
  - Repeat Until Loop Statement
  - Identifier Expression
  - Integer Constant Expression
  - Real Constant Expression
  - Binary Expression
  - Unary Expression
  - Subprogram Expression
    - Method
    - Function
    - Relation
  - Types
    - Integer Type
    - Real Type
    - Enumerated Type
      - Boolean Type
    - Function Type
    - Method Type
    - Relation Type
    - User Defined Class Type

- Abstract Declaration Node
  - Integer Constant Definition
  - Real Constant Definition
  - Type Definition
    - Parameterized Type
  - Variable Declaration
  - Formal Parameter Declaration

Fig 4.1 Class Heirarchy used in the Leda Compiler

Each subclass representing an expression or a statement has methods to type check and generate code. In addition all expression classes have methods to coerce types. All types are also subclasses of an abstract class. The generic messages to generate code, check types and to coerce can therefore be sent to a node without knowledge of the expression or statement the node represents.

Types are checked by sending a message to one of the types with the other type as a parameter. Every type class has code to check if the type which is passed to it is the same as or compatible with itself. Most type checking is done by the constructors, so that most errors are flagged as soon as the line with the error is encountered in the program. As a result the line number of the error need not be stored in most cases to be output as part of the error message.

Code is generated by sending a message to an expression or statement. The expression or statement in turn will send code generation messages to the components of the class or/and will generate code.

Coercion is done by sending a message to an expression with the type to be coerced to as the parameter.

These messages mean very different things to different classes. The classes thus form an excellent abstraction mechanism which is valuable for the compiler writer.

Fig 4.2 shows part of the class which represents the if-then-else statement and pseudocode

```
class ifthenelseStatem : public ASTnode {            // if-then-else  statement
 private:
  ASTnode     *condition;
  ASTnode     *thenStatements;
  ASTnode     *elseStatements;
public:
  ASTnode     *genCode();                       // generate code for the if-then-else statement
  ...
};

 ASTnode *ifthenelseStatem::genCode()
{
   condition->genCode();
 <generate branch labels>

  thenStatements->genCode();
 <generate branch labels>

  elseStatements->genCode();
 <generate branch labels>
}
```

Fig 4.2 Code Generation for the if-then-else Statement

for generating code for the statement. Code generation involves sending the generic **genCode()** message to the condition . The appropriate labels are then generated, the **genCode()** message is sent to the statements which form the then part, labels are generated again and the **genCode()** message is sent to the statements which form the else part. Different statements and condition expressions will interpret the **genCode()** message in different ways. The code generation routine for the the if-then-else statement does not need to know the kind of statements and expressions which are its components.

```
class binaryExpNode : public ASTnode {                // if-then-else  statement
 protected:
  char        *operator;              // operator of binary expression
 ASTnode      *leftChild;             // left sub expression
 ASTnode      *rightChild;            // right sub expression
 ASTnode      *nodeType;              // type of this binary expression (set after typechecking)
 public:
  ...
};
                        // constructor for the binary expression class does the type checking
 binaryExpresssion::binaryExpression(ASTnode *expLeft, ASTnode *expRight, char *op)
{
  <set the operator, left and right sub expressions to the parameters of this constructor>
   leftType = leftChild->getNodeType();                    // get subexpression types
   rightType = rightChild->getNodeType();
   typeCheckResult = leftType->checkType(rightType);       // checksubexpression types
 <set the type of this binary expression according to precedence rules>
}
```

Fig 4.3 Type Checking a Binary Expression

Fig 4.3 shows the class representation of a binary expression and simplified pseudocode for typechecking the expression. The typechecking is done by the constructor of the binary expression class. Typechecking involves sending the generic message **getNodeType()** to the left and right sub expressions. The left type is then sent the generic **checkType(rightType)** message with the right type as the parameter. **checkType()** checks if the two types match and returns a value which indicates the result of the check. The type of the binary expression is then set according to precedence rules. This type will be returned when this expression receives the **getNodeType()** message. As in the case of the conditional statement, the binary expression does not need to know the kind of sub expressions which are its components.

Chapter 5

Conclusions

A simple approach to implementing first class functions has been demonstrated. Type checking for a strongly typed multiparadigm language has also been demonstrated. The type checking rules are quite different from those for conventional languages because of the interaction between paradigms. Efficiency of the target code was not a primary motive for this compiler. Nevertheless the fact the system is compiled rather than interpreted should suggest fast execution. The system requires a garbage collector to collect objects whose reference count has fallen to zero and to collect function environments on the heap. The implementation does not create debugging or linkage information for a debugger or linker. A programming environment with features like browsers and debuggers would also be required for serious programming.

References

[Aho86] Aho, A.V., Sethi, R., Ullman, J.D., Compilers: Principles, Techniques and Tools, Addison Wesley, 1986

[Bud89a] Budd, T.A., Leda: Low Cost First Class Functions, Technical Report 89-60-12, Department of Computer Science, Oregon State University, June 1989

[Bud89b] Budd, T.A., Data Structures in Leda, Oregon State University, Technical Report 89-60-17, Department of Computer Science, Oregon State University, August 1989

[Bud89c] Budd, T.A., Functional Programming in a Object Oriented Language, Technical Report 89-60-16, Department of Computer Science, Oregon State University, August 1989

[Bud91a] Budd, T.A., Blending Imperative and Relational Programming, IEEE Software, Vol 8(1):58-65, January 1991

[Bud91b] Budd, T.A., Sharing and First Class Functions in Object Oriented Languages, February 1991

[Coo89] Cook, W.R., A Proposal for Making Eiffel Type Safe, ECOOP'89, Cambridge University Press 1989

[Hai86] Hailpern, B., Multiparadigm Languages and Environments. IEEE Software, Vol 3(1):6-9 , January 1986

[Kam90] Kamin, S.N., Programming Languages: An Interpreter-Based Approach, Addison Wesley, 1990

[Kor86] Korth, H.F., Extending the Scope of Relational Languages, IEEE Software, Vol 3(1):19-28, January 1986

[Kra88] Kranz, D.A., ORBIT: An Optimizing Compiler for Scheme, PhD Thesis, Yale University, 1988

[Set89] Sethi, R., Programming Languages: Concepts and Constructs, Addison Wesley, 1989