MIX10: A MATLAB TO X10 COMPILER

by

Vineet Kumar

School of Computer Science
McGill University, Montréal

Thursday, October 31st 2013

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

TBD

Résumé

TBD

Acknowledgements

TBD

Table of Contents

Abstract i

Résumé iii

Acknowledgements v

Table of Contents vii

List of Figures ix

List of Tables xi

1 Introduction 1

1.1 Contributions 2

1.2 Thesis Outline 4

2 Introduction to X10 programming language 7

2.1 Overview of X10's key sequential features 8

2.1.1 Object-oriented features 9

2.1.2 Statements 10

2.1.3 Arrays 12

2.1.4 Types 13

2.2 Overview of X10's concurrency features 14

2.2.1 Async 15

2.2.2 Finish 15

2.2.3 Atomic . . . 16
2.2.4 At . . . 17
2.3 Overview of X10's implementation and runtime . . . 17
2.3.1 X10 implementation . . . 17
2.3.2 X10 runtime . . . 18

3 Background and High level design **21**

4 Static analyses for performance and extended feature support **23**

5 Code generation **25**

6 Techniques for efficient compilation of MATLAB arrays **27**
6.1 The Function Collection Object . . . 28
6.2 The Interprocedural Analysis Framework . . . 29
6.2.1 Contexts . . . 30
6.2.2 Call Strings . . . 31
6.2.3 Callsite . . . 32
6.2.4 Recursion . . . 33
6.3 Summary . . . 35

7 Evaluation **37**

8 Related work **39**

9 Conclusions and Future Work **41**

Appendices

A XML structure for builtin framework **43**

B isComplex analysis Propagation Language **45**

C MIX10 IR Grammar **47**

Bibliography **49**

List of Figures

2.1 Architecture of the X10 compiler 19

2.2 Architecture of the X10 runtime 20

6.1 A small program where *main* calls f calls g. 31

6.2 A small program showing two calls to a function 32

6.3 Multiple possible callsites from one statement 33

6.4 A recursive example 34

6.5 Example program showing an infinite chain of calls. 35

List of Tables

6.1 The different kinds of Function Collection objects. 29

Chapter 1
Introduction

MATLAB is a popular numeric programming language, used by millions of scientists, engineers as well as students worldwide[Mol]. MATLAB programmers appreciate the high-level matrix operators, the fact that variables and types do not need to be declared, the large number of library and builtin functions available, and the interactive style of program development available through the IDE and the interpreter-style read-eval-print loop. However, even though MATLAB programmers appreciate all of the features that enable rapid prototyping, their applications are often quite compute intensive and time consuming. These applications could perform much more efficiently if they could be easily ported to a high performance computing system.

X10 [IBM12], on the other hand, is an object-oriented and statically-typed language which uses cilk-style arrays indexed by *Point* objects and rail-backed multidimensional arrays, and has been designed with well-defined semantics and high performance computing in mind. The X10 compiler can generate C++ or Java code and supports various communication interfaces including sockets and MPI for communication between nodes on a parallel computing system.

In this thesis we present MIX10, a source-to-source compiler that helps to bridge the gap between MATLAB, a language familiar to scientists, and X10, a language designed for high performance computing systems. MIX10 statically compiles MATLAB programs to X10 and thus allows scientists and engineers to write programs in MATLAB (or use old programs already written in MATLAB) and still get the benefits of high performance computing without having to learn a new language.Also, systems that use MATLAB for prototyping and C++ or Java for production, can benefit from MIX10 by quickly converting MATLAB prototypes to C++ or Java programs via X10

On one hand, all the aforementioned characteristics of MATLAB make it a very user-friendly and thus popular application to develop software among a non-programmer community. On the other hand, these same characteristics make MATLAB a difficult language to compile statically. Even the de facto standard, Mathworks' implementation of MATLAB is essentially an interpreter with a *JIT accelarator*[The02] which is generally slower than statically compiled languages. GNU Octave, which is a popular open source alternative to MATLAB and is mostly compatible with MATLAB, is also implemented as an interpreter[Oct]. Lack of formal language specification, unconventional semantics and closed source make it even harder to write a compiler for MATLAB. Furthermore, the use of arrays as default data type and the dynamicity of the base types and shapes of arrays also make it harder to add support for concurrency in a static MATLAB compiler. Mathworks' proprietary solution for concurrency is the *Parallel Computing Toolbox*[Mat13], which allows users to use multicore processors, GPUs and clusters. However, this toolbox uses heavyweight worker threads and has limited scalability.

Built on top of *Mc***LAB** static analysis framework[Doh11, DH12], MIX10, together with its set of reusable static analyses for performance optimization and extended support for MATLAB features, ultimately aims to provide MATLAB's ease of use, to benefit from the advantages of static compilation, and to expose scalable concurrency.

1.1 Contributions

The major contributions of this thesis are as follows:

Identifying key challenges: We have identified the key challenges in performing a semantics-preserving efficient translation of MATLAB to X10.

Overall design of MIX10: Building upon the *Mc***LAB** frontend and analysis framework, we provide the design of the MIX10 source-to-source translator that includes a low-level X10 IR and a template-based specialization framework for handling builtin operations.

Static analyses: We provide a set of reusable static analyses for performance optimization and extended support for MATLAB features. These analyses include: (1) *IntegerOkay analysis* - We provide an analysis to automatically identify variables that can be safely declared to be of type `Int` (or `Long`) without affecting the correctness of the generated X10 code. This helps to eliminate most of, otherwise necessary, typecast operations which our experiments showed to be a major performance bottleneck in the generated code; (2) *Variable renaming for type collision* - MATLAB allows a variable to hold values of different types at different points in a program. However, in statically typed languages like X10 this behaviour cannot be supported since a variable's type needs to be declared statically by the programmer and cannot be changed at any point in the program. We provide an analysis to identify and rename such variables if their different types belong to mutually exclusive UD-DU webs; and (3) *isComplex value analysis* - We designed an analysis for identification of complex numerical values in a MATLAB program. This helped us to extend MIX10 compiler to also generate X10 code for MATLAB programs that involve use of complex numerical values.

Code generation strategies for key language constructs: There are some very significant differences between the semantics of MATLAB and X10. A key difference is that MATLAB is dynamically-typed, whereas X10 is statically-typed. Furthermore, the type rules are quite different, which means that the generated X10 code must include the appropriate explicit type conversion rules, so as to match the MATLAB semantics. Other MATLAB features, such as multiple returns from functions, a non-standard semantics for `for` loops, and a very general range operator, must also be handled correctly. MIX10 not only supports all the key sequential constructs but also supports concurrency constructs like `parfor` and can handle vectorized instructions in a concurrent fashion. We have also designed and implemented a template-based system that allows us to generate specialized X10 code for a collection of important MATLAB builtin operations.

Techniques for efficient compilation of MATLAB arrays: Arrays are the core of MATLAB. All data, including scalar values are represented as arrays in MATLAB. Efficient compilation of arrays is the key for good performance. X10 provides two types of array representations for multidimensional arrays: (1) Cilk-styled, region-based arrays and (2) rail-backed *simple* arrays. We compare and contrast these two array forms for a high performance computing language in context of being used as a target language and provide techniques to compile MATLAB arrays to two different representations of arrays provided by X10.

Working implementation and performance results: We have implemented the MIX10 compiler over various MATLAB compiler tools provided by the *McLAB* toolkit. In the process we also implemented some enhancements to these existing tools. We provide performance results for different X10 backends over a set of benchmarks and compare them with results from other MATLAB compilers including Mathworks' MATLAB implementation and Octave.

1.2 Thesis Outline

This thesis is divided into 9 chapters, including this one and is structured as follows.

Chapter 2 provides an introduction to the X10 language and describes how it compares to MATLAB from the point of view of language design. *Chapter 3* gives a description of various existing MATLAB compiler tools upon which MIX10 is implemented, presents a high-level design of MIX10, and explains the design and need of MIX10 IR. In *Chapter 4* we provide a description of the *IntegerOkay* analysis to identify variables that are safe to be declared as `Long` type, *variable renaming for type conflict* to rename variables with conflicting types in isolated UD-DU webs and *isComplex* analysis to identify complex numerical values. *Chapter 5* gives details of code generation strategies for important MATLAB constructs. In *Chapter 6* we introduce different types of arrays provided by X10, we identify pros and cons of both kinds of arrays in the context of X10 as a target language and describe code generation strategies for them. *Chapter 7* provides performance results for code generated using MIX10 for a suite of benchmarks. *Chapter 8* provides an overview of related work and *Chapter 9* concludes and outlines possible future work.

Chapter 2
Introduction to X10 programming language

In this chapter, we describe key X10 semantics and features and contrast them with MATLAB to help readers unfamiliar with X10 and MATLAB to have a better understanding of the MIX10 compiler.

X10 is an award winning open-source programming language being developed by IBM Research. The goal of the X10 project is to provide a productive and scalable programming model for the new-age high performance computing architectures ranging from multi-core processors to clusters and supercomputers [IBM12].

X10, like Java, is a class-based, strongly-typed, garbage-collected and object-oriented language. It uses Asynchronous Partitioned Global Address Space (APGAS) model to support concurrency and distribution [SBP+13]. The X10 compiler has a *native backend* that compiles X10 programs to C++ and a *managed backend* that compiles X10 programs to Java.

In contrast to X10, MATLAB is a commercially-successful, proprietary programming language that focuses on simplicity of implementing numerical computation application [Mol]. MATLAB is a weakly-typed, dynamic language with unconventional semantics and uses a JIT compiler backend. It provides restricted support for high performance computing via Mathworks' parallel computing toolbox [Mat13].

2.1 Overview of X10's key sequential features

X10's sequential core is a container-based object-oriented language that is very similar to that of Java or C++ [SBP+13]. A X10 program consists of a collection of classes, structs or interfaces, which are the top-level compilation units. X10's sequential constructs like `if-else` statements, `for` loops, `while` loops, `switch` statements, and exception handling constructs `throw` and `try...catch` are also same as those in Java. X10 provides both, implicit coercions and explicit conversions on types, and both can be defined on user-defined types. The `as` operator is used to perform explicit type conversions; for example, `x as Long{self != 0}` converts `x` to type `Long` and throws a runtime exception if its value is zero. Multi-dimensional arrays in X10 are provided as user-defined abstractions on top of `x10.lang.Rail`, an intrinsic one-dimensional array analogous to one-dimensional arrays in languages like C or Java. Two families of multi-dimensional array abstractions are provided: *simple arrays*, which provide a restricted but efficient implementation, and *region arrays* which provide a flexible and dynamic implementation but are not as efficient as *simple arrays*. Listing 2.1 shows a sequential X10 program that calculates the value of π using the Monte Carlo method. It highlights important sequential and object-oriented features of X10 detailed in the following subsections.

```
package examples;
import x10.util.Random;
public class SeqPi {
  public static def main(args:Rail[String]) {
    val N = Int.parse(args(0));
    var result:Double = 0;
    val rand = new Random();
    for(1..N) {
      val x = rand.nextDouble();
      val y = rand.nextDouble();
      if(x*x + y*y <= 1) result++;
    }
    val pi = 4*result/N;
    Console.OUT.println("The value of pi is " + pi);
  }
}
```

Listing 2.1 Sequential X10 program to calculate value of π using Monte Carlo method

2.1.1 Object-oriented features

A program consists of a collection of *top-level units*, where a unit is either a *class*, a *struct* or an *interface*. A program can contain multiple units, however only one unit can be made `public` and its name must be same as that of the program file. Similar to Java, access to these *top-level units* is controlled by *packages*. Below is a description of the core object-oriented constructs in X10:

Class A class is a basic bundle of data and code. It consists of zero or more *members* namely *fields*, *methods*, *constructors*, and member classes and interfaces [IBM13b]. It also specifies the name of its *superclass*, if any and of the interfaces it *implements*.

Fields A field is a data item that belongs to a class. It can be mutable (specified by the keyword `var`) or immutable (specified by the keyword `val`). The type of a mutable field must be always be specified, however the type of an immutable field may be omitted if it's declaration specifies an *initializer*. Fields are by default instance fields unless marked with the `static` keyword. Instance fields are inherited by subclasses, however subclasses can shadow inherited fields, in which case the value of the shadowed field can be accessed by using the qualifier `super`.

Methods A method is a named piece of code that takes zero or more *parameters* and returns zero or one value. The type of a method is the type of the return value or `void` if it does not return a value. If the return type of a method is not provided by the programmer, X10 infers it as the least upper bound of the types of all expressions `e` in the method where the body of the method contains the statement `return e`. A method may have a type parameter that makes it *type generic*. An optional *method guard* can be used to specify constraints. All methods in a class must have a unique signature which consists of its name and types of its arguments.

Methods may be inherited. Methods defined in the superclass are available in the subclasses, unless overridden by another method with same signature. Method overloading allows programmer to define multiple methods with same name as long as they have different signatures. Methods can be access-controlled to be `private`, `protected` or `public`. `private` methods can only be accessed by other methods in the same class. `protected` methods can be accessed in the same class or its subclasses. `public` methods can be accessed from any code. By default, all methods are *package protected* which means they can be accessed from any code in the same package.

Methods with the same name as that of the containing class are called constructors. They are used to instantiate a class.

Structs A struct is just like a class, except that it does not support inheritance and may not be recursive. This allows structs to be implemented as *header-less* objects, which means that unlike a class, a struct can be represented by only as much memory as is necessary to represent its fields and with its methods compiled to static methods. It does not contain a *header* that contains data to represent meta-information about the object. Current version of X10 (version 2.4) does not support mutability and references to structs, which means that there is no syntax to update the fields of a struct and structs are always passed by value.

Function literals X10 allows definition of functions via literals. A function consists of a parameter list, followed optionally by a return type, followed by =>, followed by the body (an expression). For example, `(i:Int, j:Int) => (i<j ? foo(i) : foo(j))`, is a function that takes parameters `i` and `j` and returns `foo(i)` if `i<j` and `foo(j)` otherwise. A function can access immutable variables defined outside the body.

2.1.2 Statements

X10 provides all the standard statements similar to Java. Assignment, `if - else` and `while` loop statements are identical to those in Java.

for loops in X10 are more advanced and apart from the standard C-like for loop, X10 provides three different kinds of for loops:

***enhanced* for loops** take an index specifier of the form `i in r`, where `r` is any value that implements `x10.lang.Iterable[T]` for some type `T`. Code listing 2.2 below shows an example of this kind of `for` loops:

```
def sum(a:Rail[Long]):Long{
    var result:Long = 0;
    for (i in a){
        result += i;
    }
    return result;
}
```

Listing 2.2 Example of enhanced for loop

for loops over LongRange iterate over all the values enumerated by a `LongRange`. A `LongRange` is instantiated by an expression like `e1..e2` and enumerates all the integer values from `a` to `b` (inclusive) where `e1` evaluates to `a` and `e2` evaluates to `b`. Listing 2.3 below shows an example of a `for` loop that uses `LongRange`:

```
def sum(N:Long):Long{
    var result:Long = 0;
    for (i in 0..N){
        result += i;
    }
    return result;
}
```

Listing 2.3 Example of for loop over `LongRange`

for loops over Region allow to iterate over multiple dimensions simultaneously. A `Region` is a data structure that represents a set of *points* in multiple dimensions. For instance, a `Region` instantiated by the expression `Region.make(0..5,1..6)` creates a 2-dimensional region of *points* `(x,y)` where `x` ranges over `0..5` and `y` over `1..6`. The natural order of iteration is lexicographic. Listing 2.4 below shows an example that calculates the sum of coordinates of all points in a given rectangle:

```
def sum(M:Long, N:Long):Long{
    var result:Long = 0;
    val R:Region = x10.regionarray.Region.make(0..M,0..N);
    for ([x,y] in R){
        result += x+y;
    }
    return result;
}
```

Listing 2.4 Example of for loop over a 2-D `Region`

2.1.3 Arrays

In order to understand the challenges of translating MATLAB to X10, one must understand the different flavours and functionality of X10 arrays.

At the lowest level of abstraction, X10 provides an intrinsic one-dimensional fixed-size array called `Rail` which is indexed by a `Long` type value starting at `0`. This is the X10 counterpart of built-in arrays in languages like C or Java. In addition, X10 provides two types of more sophisticated array abstractions in packages, `x10.array` and `x10.regionarray`.

Rail-backed Simple arrays are a high-performance abstraction for multidimensional arrays in X10 that support only rectangular dense arrays with zero-based indexing. Also, they support only up to three dimensions (specified statically) and row-major ordering. These restrictions allow effective optimizations on indexing operations on the underlying `Rail`. Essentially, these multidimensional arrays map to a `Rail` of size equal to number of elements in the array, in a row-major order.

Region arrays are much more flexible. A *region* is a set of points of the same rank, where `Points` are the indexing units for arrays. Points are represented as n-dimensional tuples of integer values. The `rank` of a point defines the dimensionality of the array it indexes. The rank of a region is the rank of its underlying points. Regions provide flexibility of shape and indexing. *Region arrays* are just a set of elements with each element mapped to a unique point in the underlying region. The dynamicity of these arrays come at the cost of performance.

Both types of arrays also support distribution across places. A *place* is one of the central innovations in X10, which permits the programmer to deal with notions of locality.

2.1.4 Types

X10 is a statically type-checked language: Every variable and expression has a type that is known at compile-time and the compiler checks that the operations performed on an expression are permitted by the type of that expression. The name `c` of a class or an interface is the most basic form of type in X10. There are no primitive types.

X10 also allows *type definitions*, that allow a simple name to be supplied for a complicated type, and for type aliases to be defined. For example, a type definition like `public static type bool(b:Boolean) = Boolean{self=b}` allows the use of expression `bool(true)` as a shorthand for type `Boolean{self=true}`.

Generic types X10's generic types allow classes and interfaces to be declared parameterized by types. They allow the code for a class to be reused unbounded number of times, for different concrete types, in a type-safe fashion. For instance, the listing 2.5 below shows a class `List[T]`, parameterized by type `T`, that can be replaced by a concrete type like `Int` at the time of instantiation (`var l:List[Int] = new List[Int](item)`).

```
class List[T]{
    var item:T;
    var tail:List[T]=null;
    def this(t:T){
        item=t;
    }
}
```

X10 types are available at runtime, unlike Java(which erases them).

Constrained types X10 allows the programmer to define `Boolean` expressions (restricted) constraints on a type [T]. For example, a variable of constrained type `Long{self != 0}` is of type `Long` and has a constraint that it can hold a value only if it is not equal to `0` and throws a runtime error if the constraint is not satisfied. The permitted constraints include the predicates == and `!=`. These predicates may be applied to constraint terms. A constraint term is either a final variable visible at the point of definition of the constraint, or the special variable `self` or of the form `t.f` where `f` names a field, and `t` is (recursively) a constraint term.

2.2 Overview of X10's concurrency features

X10 is a high performance language that aims at providing productivity to the programmer. To achieve that goal, it provides a simple yet powerful concurrency model that provides four concurrency constructs that abstract away the low-level details of parallel programming from the programmer, without compromising on performance. X10's concurrency model is based on the Asynchronous Partitioned Global Address Space (APGAS) model [IBM13a]. APGAS model has a concept of global address space that allows a task in X10 to refer to any object (local or remote). However, a task may operate only on an object that resides in its partition of the address space (local memory). Each task, called an *activity*, runs asynchronously parallel to each other. A logical processing unit in X10 is called a *place*. Each *place* can run multiple *activities*. Following four types of concurrency constructs are provided by X10 [IBM13b]:

2.2.1 Async

The fundamental concurrency construct in X10 is `async`. The statement `async S` creates a new *activity* to execute `S` and returns immediately. The current activity and the "forked" activity execute asynchronously parallel to each other and have access to the same heap of objects as the current activity. They communicate with each other by reading and writing shared variables. There is no restriction on statement `S` and can contain any other constructs (including `async`). `S` is also permitted to refer to any immutable variable defined in lexically enclosing scope.

An activty is the fundamental unit of execution in X10. It may be thought of as a very light-weight thread of execution. Each activity has its own control stack and may invoke recursive method calls. Unlike Java threads, activities in X10 are unnamed. Activities cannot be aborted or interrupted once they are in flight. They must proceed to completion, either finishing correctly or by throwing an exception. An activity created by `async S` is said to be *locally terminated* if `S` has terminated. It is said to be *globally terminated* if it has terminated locally and all activities spawned by it recursively have themselves globally terminated.

2.2.2 Finish

Global termination of an activity can be converted to local termination by using the `finish` construct. This is necessary when the programmer needs to be sure that a statement `S` and all the activities spawned transitively by `S` have terminated before execution of the next statement begins. For instance in the listing 2.5 below, use of `finish` ensures that the `Console.OUT.println("a(1) = " + a(1));` statement is executed only after all the asynchronously executing operations (`async a(i) *= 2;` have completed.

```
//...
//Create a Rail of size 10, with i'th element initialized to i
val a:Rail[Long] = new Rail[Long](10,(i:Long)=>i);
finish for (i in 0..9){
//asynchronously double every value in the Rail
```

```
    async a(i) *= 2;
}
Console.OUT.println("a(1) = " + a(1));
//...
```

Listing 2.5 Example use of `finish` construct

2.2.3 Atomic

`atomic S` ensures that the statement (or set of statements) `S` is executed in a single step with respect to all other activities in the system. When `S` is being executed in one activity all other activities containing `s` are suspended. However, the `atomic` statement `S` must be *sequential*, *non-blocking* and *local*. Consider the code fragment in listing 2.6. It asynchronously adds `Long` values to a linked-list `list` and simultaneously holds the size of the list in a variable `size`. The use of `atomic` guarantees that no other operation, in any activity, is executed in between (or simultaneously with) these two operations, which is necessary to ensure correctness of the program.

```
//...
    finish for (i in 0..10){
        async add(i);
    }
//...
    def add(x:Long){
        atomic {
            this.list.add(x);
            this.size = this.size + 1;
        }
    }
//...
```

Listing 2.6 Example use of `atomic` construct

Note that, `atomic` is a syntactic sugar for the construct `when (c)` . `when (c)` is the conditional atomic statement based on binary condition `(c)`. Statement `when (c) S` executes statement `S` atomically only when `c` evaluates to true; if it is false, the execution blocks waiting for `c` to be true. Condition `c` must be *sequential*, *non-blocking* and *local*.

2.2.4 At

A *place* in X10 is the fundamental processing unit. It is a collection of data and activities that operate on that data. A program is run on a fixed number of places. The binding of places to hardware resources (e.g. nodes in a cluster, accelerators) is provided externally by a configuration file, independent of the program.

`at` construct provides a place-shifting operation, that is used to force execution of a statement or an expression at a particular place. An activity executing `at (p) S` suspends execution at the current place; The object graph G at the current place whose roots are all the variables V used in `S` is serialized, and transmitted to place `p`, deserialized (creating a graph G' isomorphic to G), an environment is created with the variables V bound to the corresponding roots in G', and `S` executed at `p` in this environment. On local termination of `S`, computation resumes after `at (p) S` in the original location. The object graph is not automatically transferred back to the originating place when `S` terminates: any updates made to objects copied by an `at` will not be reflected in the original object graph.

2.3 Overview of X10's implementation and runtime

In order to understand the compilation flow of the MIX10 compiler and enhancements made to the X10 compiler for efficient use of X10 as a target language for MATLAB, it is important to understand the design of the X10 compiler and its runtime environment.

2.3.1 X10 implementation

X10 is implemented as a source-to-source compiler that translates X10 programs to either C++ or Java. This allows X10 to achieve critical portability, performance and interoperability objectives. The generated C++ or Java program is, in turn, compiled by the platform C++ compiler to an executable or to class files by a Java compiler. The C++ backend is referred to as *Native* X10 and the Java backend is called *Managed* X10.

The source-to-source compilation approach provides three main advantages: (1) It makes X10 available for a wide range of platforms; (2) It takes advantage of the underlying classical and platform-specific optimizations in C++ or Java compilers, while the X10 implementation includes only X10 specific optimizations; and (3) It allows programmers to take advantage of the existing C++ and Java libraries.

Figure 2.1 shows the overall architecture of the X10 compiler [].

2.3.2 X10 runtime

Figure 2.2 shows the major components of the X10 runtime and their relative hierarchy [].

The runtime bridges the gap between application program and the low-level network transport system and the operating system. X10RT, which is the lowest layer of the X10 runtime, provides abstraction and unification of the functionalities provided by various network layers.

The X10 Language Native Runtime provides implementation of the sequential core of the language. It is implemented in C++ for native X10 and Java for Managed X10.

XRX Runtime, the X10 runtime in X10 is the core of the X10 runtime system. It provides implementation for the primitive X10 constructs for concurrency and distribution (`async`, `finish`, `atomic` and `at`). It is primarily written in X10 over a set of low-level APIs that provide a platform-independent view of processes, threads, synchronization mechanisms and inter-process communication.

At the top of the X10 runtime system, is a set of core class libraries that provide fundamental data types, basic collections, and key APIs for concurrency and distribution.

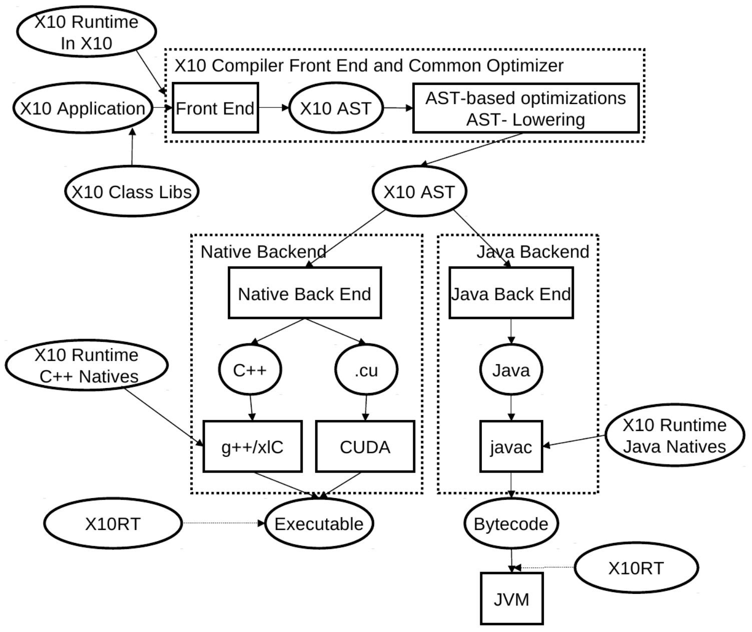

Figure 2.1 Architecture of the X10 compiler

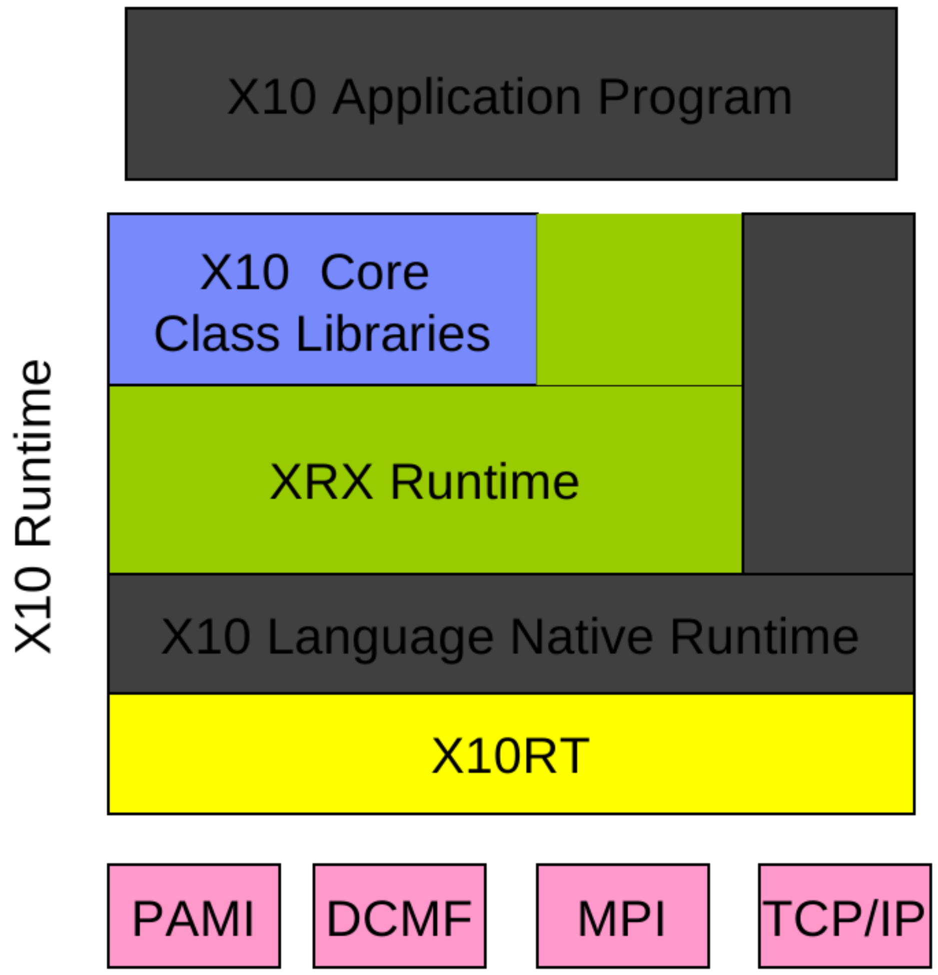

Figure 2.2 Architecture of the X10 runtime

Chapter 3
Background and High level design

TBD

Chapter 4
Static analyses for performance and extended feature support

Chapter 5
Code generation

TBD

Chapter 6
Techniques for efficient compilation of MATLAB arrays

This chapter introduces the interprocedural analysis framework. We have previously introduced the builtin framework and the Tame IR. In the next chapter we will introduce the value analysis, an interprocedural analysis that uses all these tools to build a callgraph with annotated type information. In order to implement this interprocedural analysis, we have developed the interprocedural analysis framework.

The interprocedural analysis framework builds on top of the Tame IR and the MCSAF intraprocedural analysis framework. It allows the construction of interprocedural analyses by extending an intraprocedural analysis built using the MCSAF framework. This framework works together with a callgraph object implementing the correct MATLAB look up semantics. An analysis can be run on an existing callgraph object, or it can be used to build new callgraph objects, discovering new functions as the analysis runs.

In the following sections we will introduce the interprocedural analysis framework as an extension of the intraprocedural analysis framework, and how it works in tandem with callgraph objects and the lookup objects, as well as how the framework deals with recursion. To help potential analysis writers, we have indicated the names of Java classes that correspond to the contexts in bold.

6.1 The Function Collection Object

In order to represent callgraphs we use an object which we call **FunctionCollection**. It is, as the name suggests, a collection of nodes representing functions, indexed by so function reference objects. Objects of type **FunctionReference** act as unique identifiers for functions. They store a function's name, in which file it is contained if applicable, and what kind of function it is (primary function, subfunction, nested function, builtin function, constructor). For nested functions, it stores in which function it is contained. Function reference objects give enough information to load a function from a file.

Nodes in the function collection not only store the code of the function and a function reference; they also provide information about its environment. The node provides a MATLAB function lookup object which is able to completely resolve any function call coming from the function. It includes information about the MATLAB path environment and other functions contained in the same file. The lookup information is provided given a function name, and optionally an mclass name (to find overloaded versions); and will return a function reference allowing the loading of functions.

The lookup information allows us to build a callgraph knowing only an entry point and a path environment, and using semantics for finding functions that correspond to the way MATLAB finds functions at runtime. This is bridging the gap between a dynamic language and static compilers, which usually require specifying what source code files are required for compilation.

The simple function collection uses only the lookup information contained in its nodes to built an approximation of a callgraph, which is naturally incomplete. We have used it for the development of the Tamer framework, as it provides a simple way to generate a callgraph which excludes discovering overloaded calls and propagation of function handles. We have implemented slightly different versions of the function collection, which are described in the table below.

SimpleFunctionCollection	A simple callgraph object built using MATLAB lookup semantics excluding overloading. Function Handles are loaded only in the functions where the handle is created. Obviously this an incomplete callgraph, but may be used by software tools that do not need a complete callgraph, and where the simplicity can be useful.
IncrementalFunctionCollection	callgraph that does the same lookup as the FunctionCollection, but does not actually load functions until they are requested. This is used to build the callgraph
CompleteFunctionCollection	callgraph that includes call sites for every function node and correctly represents overloading can calling function handles. This is produced by the Tamer using interprocedural analyses. This callgraph can be used to build further interprocedural analysis that are not extensions of the value analysis. It can also be used as a starting point for static backends.

Table 6.1 The different kinds of Function Collection objects.

6.2 The Interprocedural Analysis Framework

The interprocedural analysis framework is an extension of the intraprocedural flow analyses provided by the McSAF framework. It is context-sensitive to aid code generation targeting static languages like FORTRAN. FORTRAN's polymorphism features are quite limited; every generated variable needs to have one specific type. The backend may thus require that every MATLAB variable has a specific known mclass at every program point. Functions may need to be specialized for different kinds of arguments, which a context-sensitive analysis provides at the analysis level.

An interprocedural analysis is a collection of interprocedural analysis nodes, called **InterproceduralAnalysisNode**, which represent a specific intraprocedural analysis for some function and some context. The context is usually a flow representation of the passed arguments. Every such interprocedural analysis node produces a result set using the contained intraprocedural analysis. An InterproceduralAnalysisNode is generic in the intraprocedural analysis, the context and the result - these have to be defined by an actual implementation of an interprocedural analysis.

Every interprocedural analysis has an associated FunctionCollection object, which may initially contain only one function acting as the entry point for the program (i.e. when building a callgraph using an IncrementalFunctionCollection). The interprocedural analysis requires a context (argument flow set) for the entry point to the program.

Algorithm

The analysis starts by creating an interprocedural analysis node for the entry point function and the associated context, which triggers the associated intraprocedural flow analysis. As the intraprocedural flow analysis encounters calls to other functions, it has to create context objects for those calls, and ask the interprocedural analysis to analyze the called functions using the given context. The call also gets added to the set of call edges associated with the interprocedural analysis node.

As the interprocedural has to analyze newly encountered calls, the associated functions are resolved, and loaded into the callgraph if necessary. The result is a complete callgraph, and an interprocedural analysis.

6.2.1 Contexts

In order to implement an interprocedural analysis, one has to define a context object. These may be the flow information of the arguments of a call; but it could be any information. The analysis itself is context-sensitive, meaning that if there are multiple calls to one function with different contexts, they are all represented by different interprocedural analysis nodes. The interprocedural analysis framework never merges contexts, which would have to be done by the specific analysis if desired.

Interprocedural analysis nodes are cached. Thus if a function/context pair is called a second time, the information will be readily available.

Note that in order to completely resolve calls, the flow information and the contexts have to include mclass information for variables and arguments. In order to resolve calls to function handles, the contexts have to store which arguments may refer to function handles (and which functions they refer to).

Once the complete callgraph is built, further analyses don't need to flow mclass information, because all possible calls are resolved. But this information may still be useful to obtain more accurate analysis result, by knowing which information to flow into which calls for ambiguous call sites (see Sec. 6.2.3) - that is why the value analysis presented in *Chapter* **??** allows extending the flow-sets, to allow flowing information for different analyses together in one analysis, and get a more precise overall result.

6.2.2 Call Strings

When analyzing a function f for a given context c_f, and encountering a call to some other function g, the interprocedural analysis framework suspends the analysis of f in order to analyze the encountered call. The flow analysis has to provide a context c_g for the call to g, and an intraprocedural analysis will be created that will analyze g with c_g.

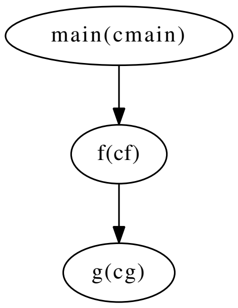

Figure 6.1 A small program where *main* calls f calls g. The call string for $g(c_g)$ in this example may be $main(c_{main}) : f(c_f) : g(c_g)$.

The set of currently suspended functions (in *Figure 6.3* *main* and f), which are awaiting results of encountered calls that need to be analyzed correspond to the callstack of these functions at runtime, at least for non-recursive programs. We call the chain of these functions, together with their contexts a **CallString**. Every function/context pair, i.e. the associated interprocedural analysis node, has an associated call string, which corresponds to one possible stack trace during runtime. Note that interprocedural analysis nodes are cached, and may be reused. Thus in the above example, if the main function also calls g

with context c_g, the results of the interprocedural analysis node created for the call encountered in function f will be reused.

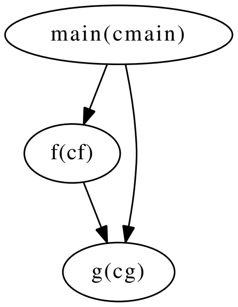

Figure 6.2 Here, *main* also calls g, also with context c_g. Since the interprocedural analysis node for $g(c_g)$ is reused, the call string will be reused as well.

Since the interprocedural analysis node is reused, it will have the same call string. So the call string is not an exact representation of a call stack for every call, it is merely the exact representation of one possible call stack that will reach a given function/context pair. Note that for purposes of error reporting, the call string can be presented to the user as a stack trace.

6.2.3 Callsite

Any statement representing a call may actually represent multiple possible calls. For example a call to a function *g* may be overloaded, so if arguments may have different possible mclasses, different functions named *g* may be called. Also, because it is up to an actual analysis to define its notion of what a context is, it is possible that an analysis may decide to produce multiple contexts for one call to a function f. This would create specialized versions of a function from a single call (this is actually possible in the value analysis presented in *Chapter* **??**). A third way in which a statement may represent multiple possible calls is via function handles. An TIRArrayGet statement may trigger a call if the represented array is actually found to be a function handle (we call the variable accessed in a TIRArrayGet statement an 'array' simply because it is used in an array-indexing operation, but it could be any variable). If that function handle may refer to multiple possible functions at runtime, then the function handle access may refer to multiple possible calls.

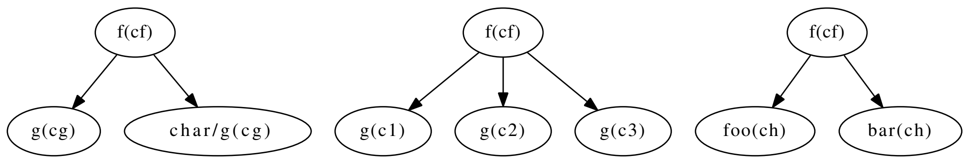

Figure 6.3 This figure shows examples how it is possible for one single call site to refer to multiple possible calls. This may be due to overloading, creation of multiple contexts for a single call, or function handles.

In order to be able to represent multiple possible call edges coming out of a statement, we associate any statement that includes any calls with a **Callsite** object. This callsite can store multiple possible call edges as function/context pairs, which we call a "call" in the interprocedural analysis framework. An intraprocedural analysis, in order to request the result of a call, has to request a callsite object for a calling statement. It may then request arbitrary calls from that callsite object, which will all get associated with the calling statement.

6.2.4 Recursion

The interprocedural analysis framework supports simple and mutual recursion by performing a fixed point iteration within the first recursive interprocedural analysis node. In order to identify recursive and mutually recursive calls we use the call strings introduced in Sec. 6.2.2. While we established that there is no guarantee which stack trace the call string represents, we know that it will always represent one possible stack trace. Since the call stacks of all recursive and mutual recursive calls must include the function, we merely need to check, for any call, whether it already exists in its call string.

If it does, we have identified a recursive call, and must perform a fixed point iteration. To do so, we label the intraprocedural analysis node associated with the recursive call (i.e. the call to $f(c_f)$ in *Figure 6.4*) as recursive. This will trigger the fixed point iteration. Because we need a result for the recursive call to continue analyzing, an actual analysis implementation has to provide a default value as a first approximation, which may be just

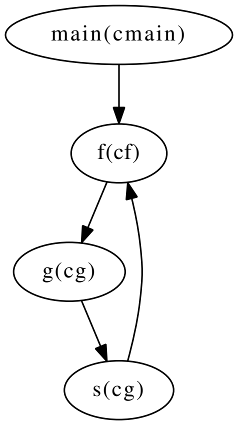

Figure 6.4 Example of a recursive program. The call in $s(c_s)$ to $f(c_f)$ triggers the fixed point iteration of $f(c_f)$. $f(c_f)$ is the first recursive interprocedural analysis node.

bottom. Once the intraprocedural contained in the interprocedural analysis node associated with the recursive call is completed, the result is stored as a new partial result. The analysis is then recomputed, using this new partial result for the recursive call. When a new partial result is the same as a previous partial result, we have completed the fixed point iteration. Note that the computation resulting in the new partial result uses the previous partial result for its recursive call - but since they are the same, we have made a complete analysis using the final result for the recursive call.

Note that the while the fixed point iteration is being computed, all calls below the recursive call (i.e. the calls $g(c_g)$ and $s(c_s)$ in *Figure 6.4*) always return partial results. Thus we cannot cache the nodes and their results, and have to continuously invalidate all the corresponding interprocedural analysis nodes.

Note that the analysis treats calls to the same function with different contexts as different functions. No fixed point iteration is performed to resolve recursive calls with different contexts, because they represent different underlying intraprocedural analyses. Thus it is possible to create infinite call strings, as shown in *Figure 6.5*. It is up to the actual analysis implementation to ensure this does not happen. A simple strategy would be to ensure that there are only a finite number of possible contexts for every function. Another strategy is for the intraprocedural analysis to check the current call string before requesting a call, to

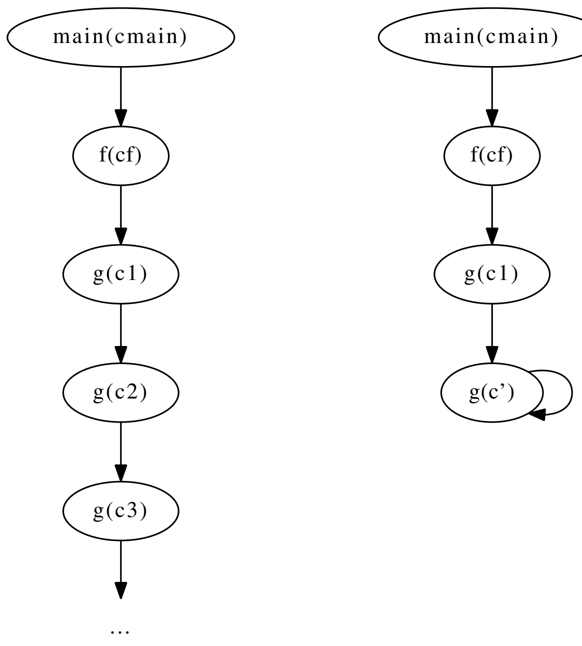

Figure 6.5 Example of a recursive program, showing how recursive calls with different contexts can create infinite chains of calls on the left. An interprocedural analysis implementation has to catch such cases and create a finite number of contexts, as shown on the right, where the contexts c_2 and onward are replaced with c'. In this case the interprocedural analysis framework will perform a fixed point iteration on $f(c')$.

ensure that the function to be called does not already exist in the call string. If it does, the intraprocedural analysis should push up the context to a finite representation (shown in *Figure* 6.5).

6.3 Summary

We have presented an interprocedural analysis framework that we hope is flexible enough to allow different kinds of full-program analyses, while powerful enough to deal with issues such as recursion and ambiguous call sites. This analysis framework is a key component of our value analysis (presented in the next chapter), and the overall callgraph construction of the Tamer.

Chapter 7
Evaluation

TBD

Chapter 8
Related work

Chapter 9
Conclusions and Future Work

TBD

Appendix A
XML structure for builtin framework

TBD

Appendix B
isComplex analysis Propagation Language

TBD

Appendix C
MiX10 IR Grammar

TBD

Bibliography

[DH12] Anton Dubrau and Laurie Hendren. Taming MATLAB. In *Proceedings of OOPSLA 2012*, 2012, pages 503–522.

[Doh11] Jesse Doherty. McSAF: An Extensible Static Analysis Framework for the MATLAB Language. Master's thesis, McGill University, December 2011.

[IBM12] IBM. X10 programming language. http://x10-lang.org, February 2012.

[IBM13a] IBM. Apgas programming in x10 2.4, 2013. http://x10-lang.org/documentation/tutorials/apgas-programming-i

[IBM13b] IBM. An introduction to x10, 2013. http://x10.sourceforge.net/documentation/intro/latest/html/intr

[Mat13] MathWorks. Parallel computing toolbox, 2013. http://www.mathworks.com/products/parallel-computing/.

[Mol] Cleve Moler. The Growth of MATLAB and The MathWorks over Two Decades. http://www.mathworks.com/company/newsletters/news_notes/clevesc

[Oct] GNU Octave. http://www.gnu.org/software/octave/index.html.

[SBP$^+$13] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. *X10 Language Specification, Version 2.4*. December 2013.

[The02] The Mathworks. Technology Backgrounder: Accelerating MATLAB, September 2002. http://www.mathworks.com/company/newsletters/digest/sept02/accel_mat