EFFICIENT JIT COMPILATION OF MATLAB LOOPS

by

Matthieu Dubet

School of Computer Science

McGill University, Montréal

Sunday, April 13th 2014

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

MATLAB® is a dynamic numerical scripting language widely used by scientists, engineers and students. It is praised because it allows fast prototyping, especially for numerical programs which manipulate matrices. However, numerical software can be computationally heavy, and MATLAB, as most interpreted languages, suffers from slow performance as compared to traditionally compiled languages such as FORTRAN or *C++*. One way to provide better performance for interpreted languages is through just-in-time compilation, where the program (or part of the program) is compiled at run-time.

In this thesis, we introduce SJIT, a just-in-time compiler for MATLAB which focuses on providing good performance while keeping the compilation time extremely small. It is designed to integrate easily and transparently into an existing interpreter for MATLAB named *Mc*VM, and generates highly efficient assembly code intensive parts of the program, namely loops. In addition to its use for accelerating whole MATLAB programs, it is also suitable for accelerating the execution of fragments of MATLAB code inside an interactive environment such as a read-eval-print loop.

In addition to the SJIT compiler, this thesis also contributes an efficient framework to develop static dataflow analyses, and a type inference analysis implemented within this framework.

The SJIT compiler has been evaluated, both in terms of compilation time and execution time, on a collection of MATLAB benchmarks using traditional features such as matrices and structures. The results show that: (1) the achieved performance is several times faster than the original MATLAB implementation, and (2) that the compilation time is very reasonable, taking only a small fraction of the overall time.

Résumé

MATLAB® est un langage de calcul numérique utilisé par des ingénieurs, scientifiques, et étudiants à travers le monde. Il est apprécié pour faire du prototypage rapide, particulièrement pour les programmes de calcul numérique manipulant des matrices. Cependant, les programmes de calcul numérique peuvent être exigeant en termes de puissance de calcul, et MATLAB, comme la plupart des langages interprétés, souffrent de mauvaises performances comparativement au langages compilés traditionels comme FORTRAN ou *C++*.

Dans cette thèse, nous présentons SJIT, un compilateur just-in-time pour MATLAB qui porte son attention à procurer de bonne performance tout en gardant un temps de compilation extrémement réduit. Il est concu pour s'intégrer facilement et de manière transparente dans un interpréteur déjà éxistant pour MATLAB appelé *Mc*VM, et génère du code assembleur très performant pour les parties du programme lourde en quantité de calcul : les boucles. En plus de son usage pour accélérer des programmes MATLAB complet, il fonctionne aussi pour accélérer l'éxécution de fragments de code MATLAB à l'intérieur d'environments interactifs, comme une boucle d'évaluation read-eval-print.

En plus du compilateur SJIT, cette thèse fournit aussi un outil performant pour le développement d'analyses statiques de dataflow, ainsi qu'une analyse d'inférence de type implémentée avec cet outil.

Le compilateur SJIT a été évalué, à la fois en temps de compilation et en temps d'éxécution, sur une collection de programmes MATLAB qui utilisent des fonctionnalités classiques, telles que les matrices et les structures. Les résultats montrent que : (1) la performance atteinte est bien supérieure à celle de l'implémentation originelle de MATLAB, et que (2) le temps de compilation est très raisonnable, puisqu'il correspond seulement à une faible fraction du temps globale.

Acknowledgements

I would like to thank all the persons who made these two years a nice experience, and thus helped me to make this thesis:

- My family in Canada for their dedication in making my life in Montreal nicer.
- My family in France for their long-distance support, from video calls to handwritten letters.
- The members of the Sable lab, for the lunchs in Chinatown.
- My supervisor, Laurie, for dealing nicely with me.

Table of Contents

List of Figures

List of Tables

List of listings

Chapter 1
Introduction

MATLAB is a popular numeric programming language, used by millions of scientists, engineers and students worldwide[1]. MATLAB programmers appreciate the high-level matrix operators, dynamic typing, the large number of library functions, and the interactive style of program development available with interpreter-style read-eval-print loop. However, even though MATLAB programmers appreciate all of the features that enable rapid prototyping, they often have other ultimate goals. Frequently their programs are quite computationally intensive and they really want an efficient implementation.

Providing efficient implementation for a dynamic language is still an ongoing effort in the compiler world. For MATLAB specifically, the main implementations are:

- the MATHWORKS reference implementation, which is efficient thanks to its just-in-time compiler, but is neither free nor open source.[2]
- the GNU OCTAVE implementation[3], which is an almost complete implementation in terms of semantics, but performs poorly because its default implementation is an interpreter.
- *Mc*VM, implemented by the Sable research group at McGill university, which provides an interpreter for a subset of MATLAB and a just-in-time compiler to improve the performance by some orders of magnitude, but not enough to compete with the Mathworks implementation.

This thesis aims to provide a highly efficient just-in-time compiler for MATLAB, by taking into account some common properties of numerical software and taking advantage of those. The results are very encouraging, being on some benchmarks within a factor of two in terms of execution time compared to a highly-efficient FORTRAN implementation[4], while preserving the dynamic behaviour of the program if necessary.

1.1 Contributions

The main contributions of this thesis are as follows.

- We provide a very efficient just-in-time compiler for *Mc*VM. We discuss different fundamental architectural choices, and also provide overall performance results, as well as more detailed results on the cost of bound-checking.
- We describe a precise type inference analysis for a significant subset of the MATLAB language. In particular, we can handle different data types, such as numerical matrices and array of structures.
- We present a dataflow analysis framework that allows faster and safer development for compiler analysis developers.

1.2 Thesis Outline

This thesis is divided into 7 chapters, including this one, which are structured as follows. Chapter 2 introduces key MATLAB features, showing some of the challenges of compilation. It also discusses the current ongoing work to efficiently execute dynamic languages. Chapter 3 presents our dataflow analysis framework, and describes the different components contributed by this thesis. Chapter 4 explains our type inference analysis. Chapter 5 presents our just-in-time compiler, its implementation and its specificities. Chapter 6 discusses the performance results of our compiler and Chapter 7 concludes.

Chapter 2
Background and Related Work

In this chapter, we present background information helpful to understand this thesis. We begin with a brief presentation of the MATLAB language, and its properties which are of interest for this thesis.

Then, we describe the research which has already be done in the fields of optimization of dynamic languages and numerical languages.

2.1 The MATLAB language

MATLAB is a closed sourced, proprietary programming environment. It was originally designed in the 1970s as an easier approach to numerical computation than FORTRAN[5]. To achieve this goal of friendliness for the programmer, it has been designed as a procedural, dynamically-typed, weakly-typed array-based programming language.

2.1.1 Execution model

The MATLAB execution model is quite different from other programming languages, due to the existence of two units of reusable code: functions and scripts. Variables exist in an environment: a map from variable name to value. There is a global (shared by any code in a given MATLAB session) environment called the *workspace*. Functions can have multiple input and output parameters, and have a local environment where variables are stored and read. Scripts on the other hand only have access to the global *workspace*. Also, the interactive mode (the prompt >>) is implicitly accessing the global *workspace*. If a symbol is evaluated and is not bound to a local variable, the interpreter tries to resolve it with a function or script lookup. MATLAB function or script lookup resolving is based on an interpreter setting which indicates the current directory on the filesystem in which functions or scripts will be located. This setting can be change dynamically with the `cd` function which takes a string representing the target directory path as an input. The `cd` function thus allows the programmer to change the lookup behavior at runtime.

2.1.2 Dynamic type system

As we said, MATLAB is dynamically-typed. It supports several primitives types, such as integers of different precisions (`int8`, `int32`, `int64`), floating-point numbers (`single`, `double`), boolean (`logical`) and character (`character`). It also contains an aggregate type inspirated by C structures called `struct`. It allows the creation of values with named fields.

All those types are implicity treated as if they were matrices:

- A scalar value is actually a 1×1 matrix.
- A structure with a field *f* is actually a structarray of size 1×1 with a field *f*.
- A string like ‘hi’ is actually a 1×2 character matrix.

Also, MATLAB provides the `function handle` type which combined with anonymous function expression provides some support for functional programming. More recently, support for object-oriented programming with classes has been available.

2.1.3 Typical MATLAB programs

MATLAB programs are quite different from the expected programs in general-purpose programming language such as Java or Python. Its numerical computing roots, the fact that most of its users are not programmers but scientists, and some specifc properties of the language itself are the main reasons for this uniqueness.

Thanks to the McBench tool[6], which allows running queries over the structure of a huge number of Matlab benchmarks at once, and to our manual observations of programs from various sources, we got some insights about general properties of Matlab programs.

Functions in Matlab are not as ubiquitous as in other languages: the combination of interactive development within the Matlab IDE and the existence of scripts make functions less used. Related, the usage of recursion in Matlab is low (around 5% of the benchmarks contains one or more recursive functions).

Being an array-based language, the vast majority of Matlab programs are mainly iterations and computations over matrices: while the official documentation advises to use vector operation and ranges, inexperienced programmers find loops more intuitive.

The usage of structures in Matlab is significant, about 30% of the benchmarks take advatange of them, but the usage of structarrays is much less frequent, at only 7%. Arrays of structures are however useful in many programs[1], and we suspect their lack of usage in Matlab to the well-known performance trouble of the MathWorks implementation when it comes to structarrays[7]. Enabling the usage of structarrays without a performance penalty would be useful in our opinion.

Finally, we found that the various dynamic features provided by Matlab are being used quite heavily in real-world programs. For example, a simple query about the usage of the functions *cd*, *eval*, *feval*, *load*, *save*, *exist*, *assignin*, *clear* returns more than 25% of McBench benchmark set.

2.2 Related work

Matlab, as most dynamic language implementations, is run in an execution environment called a Virtual Machine (VM). A VM for a language *X* is a program that is designed specifically to execute programs written in the language *X*. The fact that the execution of a program is done by another program, instead of directly by the processor, introduces a performance penalty compared to the same program executed directly by the processor (so-called compiled program). Unfortunately, implementing a compiler for a dynamic language

[1]Imagine a language where only primitive types could compose a vector

is in general not possible, because of the dynamism of those programs such as non constant variable type, ability to load code at run-time with construct like *eval*, dynamic lookup semantics with *cd* and complex reflection features. As a result, most dynamic languages have an interpreter as their reference implementation such as Python (CPython[8]) and Ruby (YARV[9]).

This section explores differents approaches which have been tried to provide efficient implementation of MATLAB or other dynamic languages.

2.2.1 Static approach to MATLAB execution

*Mc***FOR**[4] and Matlab Coder[10] are compilers for MATLAB to respectively FORTRAN and *C++*. Both offer really high performance for the generated code, but don't support MATLAB dynamic features, thus are not valid for a general usage. However, they could be useful for projects with specific requirement which are developed in the supported static subset of MATLAB. The cumulative work of the *Mc***SAF** to *Mc2***FOR** project is able to compile enough of MATLAB to support real-world programs, but the amount of work necessary to achieve that has been tremendous.

2.2.2 *Mc*JIT

The only existing open source implementation of a just-in-time compiler for MATLAB is *Mc***JIT**[11], the MathWorks implementation being closed-source. *Mc***JIT** is a function-based type specializing JIT compiler: it compiles functions at run-time (on top of the interpreter) based on the type of the function arguments. This has proven to be a performance improvement over the interpreter, however the execution model and the compilation phase is fairly complex, because of the constant interaction between a compiled function and the interpreter. The just-in-time compiler implemented as a contribution of this thesis takes a different approach: lower-level, smaller and more focused, with the hope of providing better performance.

2.2.3 Old research projects about MATLAB

Although we were not able to find publicly available versions, which is not surprising considering that those projects have been done more than 10 years ago, there have been several previous research projects on MATLAB:

- MAGICA[12] is a type inference engine written in Mathematica, which is very precise (but costly). It has been used for the ahead-of-time compiler MAT2C[13].
- FALCON[14] is a static MATLAB to FORTRAN translator with a sophisticated type inference algorithms.
- MaJIC[15] is a MATLAB just-in-time compiler patterned after FALCON with simplified analysis to fit the just-in-time context. While interesting at the time, the performances are not on par with what we would expect nowdays.

2.2.4 TraceMonkey

Recently, thanks to the emergence of the web and JavaScript, a lot of efforts both in the research and the industry communities have been spent to improve the speed of dynamic languages by the usage of just-in-time compilers.

The TraceMonkey VM[16] for the JavaScript language is based on a fast bytecode interpreter that can identify frequently executed bytecode sequences (traces), records them, and compiles them to fast native code. The TraceMonkey JIT compiler is loop-based and assume hot loops to be mostly *type-stable*, meaning that the types of variables are invariants through the loop. Their results show that this assumption is reasonable. This idea is a direct inspiration for the SJIT compiler contributed by this thesis, and we actually think that *type-stabilty* is even more common in a numerical language like MATLAB than in a prototype-based language such as JavaScript, so this approach might make even more sense in the context of MATLAB.

Chapter 3
Dataflow Analysis Framework

An effective JIT compiler is based on the knowledge acquired by analyses such as reaching definitions, live variables and type inference. These analyses can have different properties and definitions, but they often require the same functionalities. For example, all analyses require a way to traverse the program representation. This kind of common functionality should not be rewritten for each analysis.

One contribution of this thesis is the design and the implementation of a framework to make the development of static analysis easier. It does this by providing an implementation for well-known kind of analysis, specifically flow-sensitive and flow-insensitive analyses. The primary goals of the analysis framework are:

Generic

The framework shouldn't make any assumptions about the underlying analysis and the computed abstract values. For example, it should support set-based abstract values such as set of definition points in a reaching definition analysis, or integer ranges in an shape analysis.

Effective

The framework should make the development of analyses easier, faster and safer, by providing the foundations to write an analysis on *Mc*IR.

Flexible

One should be able to use small parts of the framework, if necessary (thus enabling its uses in non standard analysis for example). It should provide different common behaviours, which can be used interchangeably.

Efficient

The framework should introduce no overhead compared to the same hand-written analysis.

3.1 Description of the design

In this section, we describe the design of the framework, first by a brief overview of the program representation on which it operates, *Mc*IR, then with the description of an analysis in the context of the framework, and finally by a technical chapter about the *C++* features used to implement it.

3.1.1 *Mc*IR

The framework has been developed to operate on the *Mc*IR program representation. *Mc*IR is an AST-based representation of a MATLAB program[17]. It is a high-level representation, and the control flow is implicit: it is contained inside some nodes such as `IfElseStatement` and `LoopStatement`, contrary the a control-flow graph (CFG) based IR, where as the name implies, the control-flow is explicit. It is built on top of two base classes: *node* and *statement*, with their subclasses respectively listed in Figure 3.1 and Figure 3.2.

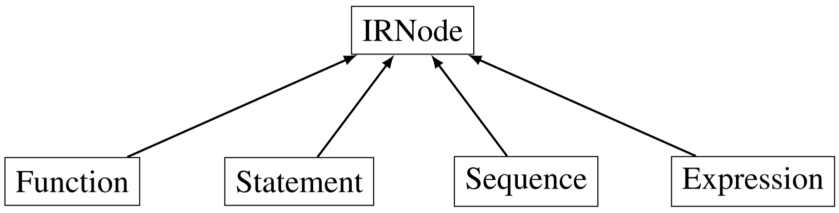

Figure 3.1 *Mc*IR Nodes

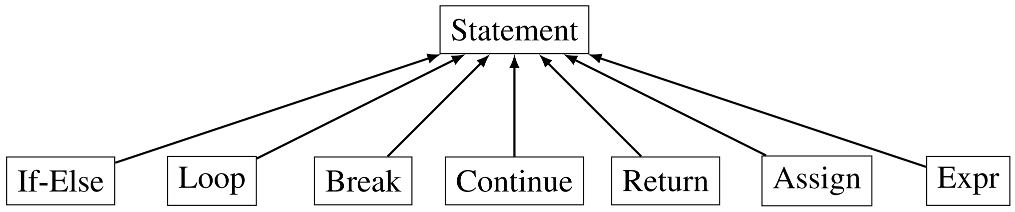

Figure 3.2 *Mc*IR Statements

3.1.2 Definition of an analysis

An analysis is, in essence, the association of a domain of abstract values, and rules which define how those abstract values are computed over the program representation.

Analysis Domain

The framework is completely domain-agnostic. While some frameworks try to enforce domain properties (such as being set-based, or map-based), the one developed for this thesis does not: it allows the domain of integers, for example. This gives complete freedom to the analysis writer about the underlying representation of the domain. However, a valid domain needs to have at least two operations defined on it (`T` being an element of the domain):

- A merge operation, with a signature `T merge(T,T)`.
- An equality operation, with a signature `bool ==(T,T)`.

Analysis Rules

The second part of the definition of an analysis is the definition of the analysis rules, one for each program construct in program representation (*Mc*IR). To be valid, an analysis needs to be defined over the entire program representation. We enforce during the compilation of the analysis that all runtimes behavior has been defined. Thus, an analysis is implemented as a *C++* class to ensure that it contains a definition for at least the functions described in Table 3.1. Those functions correspond exactly to the different subclasses of the class *Statement* in *Mc*IR.

The design of the framework is *component* based. A *component* (also known as a *mixin* in the programming community)[18] represents a fragment of code which implement a defined functionality while respecting a common interface. By definition, a *component* should be interchangeable with another one implementing the same interface. As a contribution of this thesis, we provide different components. Each one implements one or more of the functions required for an analysis. They are described individually in Section 3.2.

3.2 Provided components

In this section, we describe the different components which have been implemented as a contribution of this thesis. We will use the following definitions to describe them:

Flow insensitive

A flow insensitive analysis computes facts that holds for all statements in the program, and thus doesn't take into account the order of execution of the statements[19].

Flow sensitive

A flow sensitive analysis is sensitive to the flow of the data: it takes into account the order of statements in a program. Thus, it records facts on a per-statement basis, and consider all paths in the program as executable, whatever the predicates at the conditional for this path might be[20].

Path sensitive

A path-sensitive analysis computes facts which are dependent on the predicates at conditional branch instructions[21].

3.2.1 Flow sensitive base

This component is the basis for a flow sensitive analysis.

As a flow-sensitive analysis, results are computed on a per-statement basis: it is represented in this component by the `data` member variable in Listing 1, which associates at each node in the program the resulting abstract value. Note that the implementation uses a function call `statement_dispatch` described in Subsection 3.2.7.

Name	Signature	Description
`analyze_statement`	$(T, Statement) \longrightarrow T$	Defines the analysis rule of a *Statement* node, in most analysis, this should forward the computation to the actual *Statement* subclass, (with the `statement_dispatch` function described in Subection 3.2.7).
`analyze_assign`	$(T, AssignStmt) \longrightarrow T$	Defines the analysis rule of an assignment statement. This behavior is entirely analysis-specific, thus we don't provide any component which implements this function.
`analyze_exprstmt`	$(T, ExprStmt) \longrightarrow T$	Defines the analysis rule of an expression statement such as `disp()`. Again this behavior is entirely analysis-specific and thus we don't provide any component which implements this function.
`analyze_sequence`	$(T, Sequence) \longrightarrow T$	Defines the analysis rule of a sequence of statements. We provide three components which implements this function which should correspond to the behavior of most analysis.
`analyze_ifelse`	$(T, IfElseStmt) \longrightarrow T$	Defines the analysis rule of an *IfElseStmt* node. We provide one component which implements this function on top of the `merge` function.
`analyze_loop`	$(T, LoopStmt) \longrightarrow T$	Defines the analysis rule of a *Loop* node. We provide one component which implements this function.
`analyze_continue`	$T \longrightarrow T$	Defines the analysis rule of a continue statement. It is closely related to the behavior of a loop and we provide an implementation for this function in the `loop_fixpoint` component.
`analyze_break`	$T \longrightarrow T$	Same as the analyze_continue but for a break statement.

Table 3.1 Mandatory functions to compose an intraprocedural analysis

```
#define BASE static_cast<derived*>(this)
template <typename derived, typename T>
struct flow_sensitive_base :
public framework::statement_dispatcher<derived,T>
{
        // at each statement, we have a dataflow result
        std::unordered_map<Node*,T> data ;

        T analyze_statement(Statement st, T in) {
            // use helper component
            auto dataflow = BASE->statement_dispatch(st,in);
            // save the analysis result at each statement
            data[st] = dataflow;
            // return the analysis result for this statement
            return dataflow;
        }
}
```

Listing 1 Implementation of component flow_sensitive_base

3.2.2 Sequence of statements

A very common requirement of an analysis is the ability to traverse a sequence of statements in the program representation. This component implements this behaviour in two different flavours: in the forward direction (top to bottom), or in the backward direction (from bottom to top). To avoid code duplication, the difference between those two really close traversal methods is handled with a compile-time boolean value *backward* in Listing 2, thus introducing no runtime overhead compared to an hand-written statement sequence component.

For convenience, we provide two components in Listing 3 with straightforward name for analysis writer, respectively named *forward* and *backward*.

3.2.3 Loop

A loop in *Mc*IR is a statement which is executed multiple times, while a specified condition stays true. It is actually the underlying representation in *Mc*IR for both the `while` and `for` construct in the MATLAB language.

A loop is difficult to abstract in a static fashion, considering that the condition is in the general case not evaluable statically, thus the number of iterations is unknown. A flow-sensitive analysis, whose result depends on the flow of the program by definition, needs to

```
#define BASE static_cast<derived*>(this)
template <typename derived, typename T, bool backward>
struct sequence
{
            T analyze_sequence(StmtSequence seq, T in)
            {
                auto current = in ;
                auto stmtsequence = seq->getStatements() ;

                // Handle the backward case with a
                // compile time boolean parameter
                if (backward)
                    std::reverse (
                        std::begin(stmtsequence),
                        std::end(stmtsequence));

                // Iterate through all statements
                for (auto st: stmtsequence) {
                    current = BASE->analyze(
                            st,
                            current);
                }
                return current;
            }
};
```

Listing 2 Implementation of component sequence

```
template <typename derived, typename T>
        using forward = sequence<derived,T,false>;

template <typename derived, typename T>
        using backward = sequence<derived,T,true>;
```

Listing 3 Implementation of component forward and backward

approximate the behaviour of the loop statically.

We provide a component which implements a standard method of doing an iterative fixpoint over the loop.

Fixed-point iterative solver

The flow of data is described in Figure 3.3, and the *C++* implementation is in Listing 4. The described loop contains severals statements, and also a `continue` and a `break` statement.

The fixed-point algorithm is: the computation operates by storing the previous result of analyzing the body, and comparing it with the new result. If they are equal in terms of

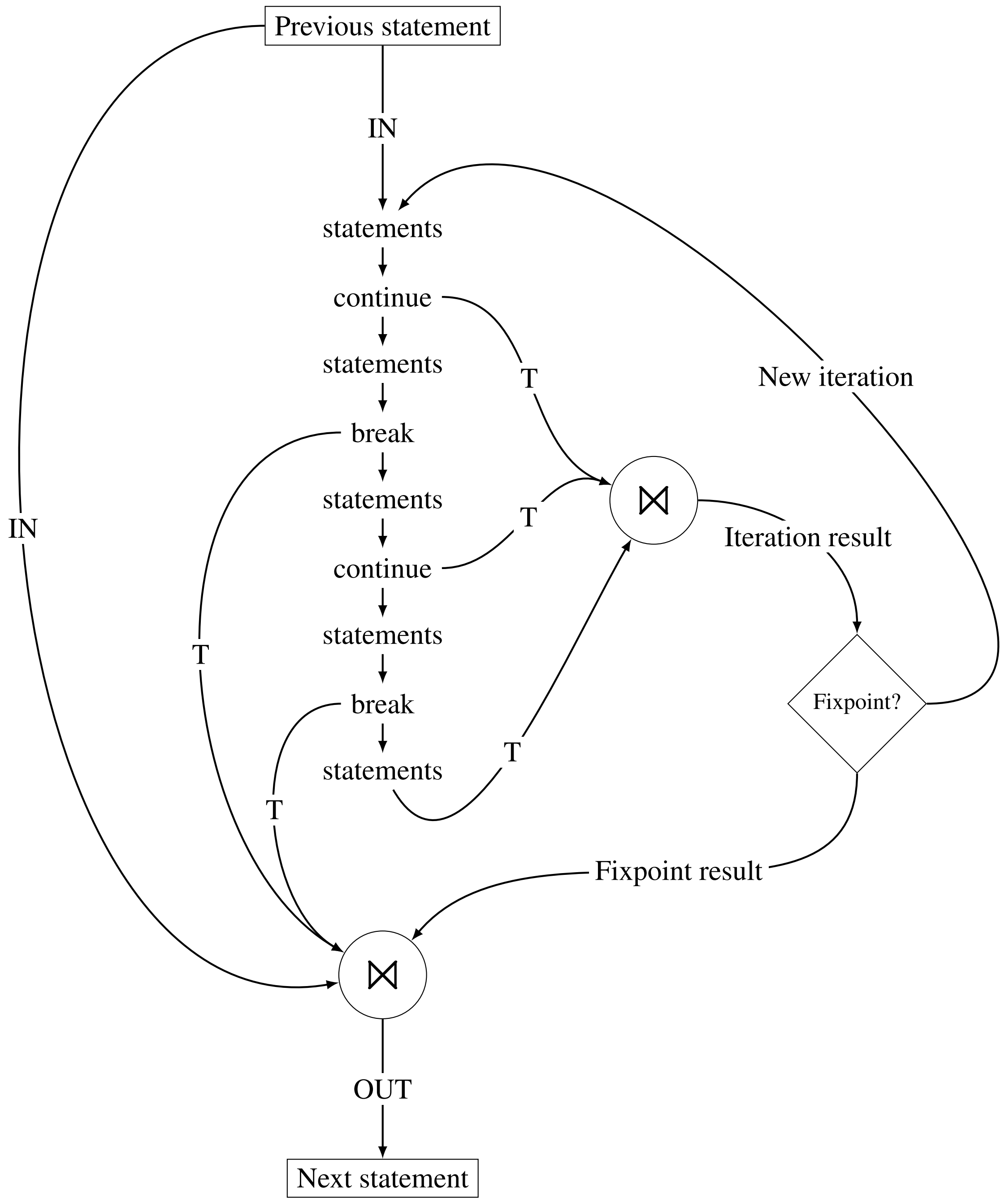

Figure 3.3 Iterative fixpoint dataflow behavior on a loop

abstracted values then a fixed-point is reached and the computation is done for this loop, else another iteration of the fixed-point computation is executed.

Also, special attention is given to `break` and `continue` statements:

- whenever a `break` statement is encountered, we merge the current dataflow with the *breaks* dataflow, which will eventually be merged with the end result at the exit of the loop.
- whenerver a `continue` statement is encountered, we merge the current dataflow with a *continues* dataflow, which will eventually be merged with the result of the analysis at the current iteration.

This component is path insensitive, as it doesn't take into account the conditional of the loop. It would be possible to implement another component which is path sensitive.

3.2.4 Conditional

A conditional statement is a statement which, depending on a conditional, will execute one path or another. It is common in imperative languages and is represented as *if(cond) then S1 else S2*, where `cond` is a boolean expression and `S1` and `S2` are statements. Depending on the properties on the analysis (flow sensitivity, path sensitivity,...), the analysis rule for such a statement will vary.

We provide a flow-sensitive, path-insensitive component for the *IfElseStatement* construct in *Mc*IR.

The dataflow for this component is described in Figure 3.4, and its code is in Listing 5. Its behavior is straighforward: it analyzes the two different paths independently (given them an input dataflow from the previous statement), and merges the two results into a single dataflow result.

We acknowledge that path-sensitivity could be useful for several analysis, such as range analysis. Future work would be to implement such a component.

```
#define BASE static_cast<derived*>(this)
template <typename derived, typename T>
struct loop_fixpoint
{
    T analyze_loop(LoopStmt loop, T in)
    {
        // for the first iteration, we just define
        // the previous dataflow result to be the input dataflow
        T old = in;
        // loop until a fixpoint is reached
        while (true)
        {
            // analyze the body of the loop with
            // the result of the previous iteration
            // as the input dataflow
            T end_body= analyze_sequence(loop.body, old);
            // use the merger helper to merge all the dataflow results
            // at each continue statement into one
            T continues_merged = merger(continues) ;
            // merge all the continue statements result
            // with the end of the body result at this iteration
            T end_iteration = merge(end_body,continues_merged);
            // We have reached a fixpoint
            if (old == end_iteration)
            {
                // use the merger helper to merge all the dataflow results
                // at each break statement into one
                T breaks_merged = merger(breaks);
                // merge all the break statements results
                // with the result at the end of the fixpoint and return.
                return merge(end_iteration,breaks_merged);
            } else {
                // the continue statements results are cleaned at
                // each iteration
                continues.clear();
                // store the result for this iteration
                old = end_iteration;
            }
        }
    }

    T analyze_continue(T in)
    {
        continues.push_back(in);
        return in;
    }

    T analyze_break(T in)
    {
        breaks.push_back(in);
        return in;
    }

    // store the dataflow result at each continue statement
    std::vector<T> continues;
    // store the dataflow result at each break statement
    std::vector<T> breaks;
};
```

Listing 4 Implementation of component loop_fixpoint

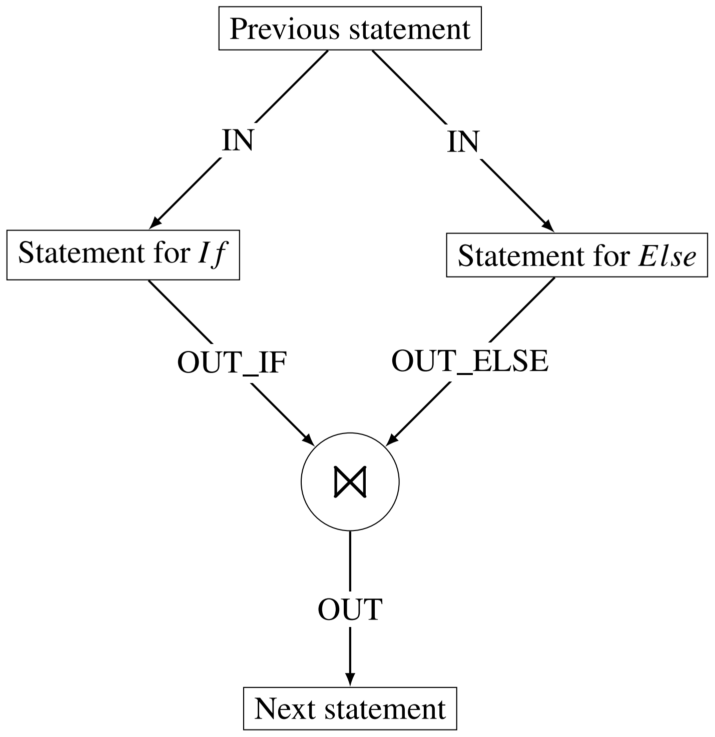

Figure 3.4 Flow sensitive, path insensitive conditional component dataflow

```
#define BASE static_cast<derived*>(this)
template <typename derived, typename T>
struct ifelse_sensitive
{
            T analyze_ifelse(IfElseStatement ifelse, T in)
            {
                T end_if = BASE->analyze_sequence(
                                    ifelse->getIfBranch(),
                                    in);
                T end_else = BASE->analyze_sequence(
                                    ifelse->getElseBranch(),
                                    in);
                return BASE->merge(end_if,end_else);
            }
};
```

Listing 5 Implementation of component ifelse_sensitive

3.2.5 Function call expression

A function call is nothing but an expression in MATLAB: it can happen either inside an assignment statement, like `myfunc(param)` in Listing 6, or in an expression statement, like `disp(a)` in the same listing.

```
    param = 1;
    a = myfunc(param);
    disp(a);
```

Listing 6 MATLABcode showing function calls

We haven't discussed how to analyze expressions in the framework, because there is no general semantic, it's entirely analysis-specific.

However, we found that for function calls, we could provide a component with a behavior which would help analysis writer to handle something hard: recursion. In MATLAB, functions can be self recursive (call itself), or mutually recursive (call others functions which again call it), such as the examples in Listing 7.

```
function [res] = fib(n)
if fib < 2
    res = 1;
else
    res = fib(n-1) + fib(n-2) ;


function [res] = odd(n)
if n == 1
    res = true;
elseif n == 0
    res = false;
else
    res = even(n-1);


function [res] = even(n)
if n == 0
    res = true;
elseif n == 1
    res = false;
else
    res = odd(n-1);
```

Listing 7 Example of MATLAB recursive functions

Analyzing a function call expression thus requires to analyze itself recursively. As we saw for loops, an interesting behavior is to compute this result with an iterative fixpoint solver.

We provide two components to handle functions in *Mc*IR, one supporting function recursion and the other not. Both of them implement the methods described in Table 3.2.

Name	Signature	Description
`analyze_function`	$(T, Function) \longrightarrow T$	Defines the analysis rule for a function call expression.
`analyze_return`	$T \longrightarrow T$	Defines the analysis rule for a return statement.

Table 3.2 Mandatory functions to analyze a function call expression

The first one, without support for recursion, will do a simple call to the `analyze_sequence` function of the analysis, and returns the result after having dealt with the possible multiple `return` statements inside the body. The code is showed in Listing 8.

The more interesting component is the one with included support for recursive function. In terms of dataflow, a function being recursive means that the dataflow result for a function call actually depends on itself, in the same way that with a loop (see Subsection 3.2.3).

We dealt with this problem by using partial result of function analysis: every time a recursion is identified (because the function currently analyzed is already in the call stack), we store partial result for each function currently being analyzed in the call stack. Then, we do a fixpoint iteration until the result for all the functions in the call stack doesn't change.

3.2.6 Flow insensitive

A flow insensitive analysis is an analysis which doesn't take the control flow into account to compute its result. Such an analysis just does some traversal of the program statements and computes a single result for the whole program. We provide a simple component which handles that by doing a top-down traversal of the program.

```
#define BASE static_cast<derived*>(this)
template <typename derived, typename T>
struct function_simple :
public framework::merger<derived,T>
{
    // analyze a function call expression
    T analyze_function(Function f, T in)
    {
        // analyze the body of the function
        T result = BASE->analyze_sequence(f->getBody(),in);

        // use the merger helper to merge all the dataflow result
        // at each return statement together
        T returns_merged = BASE->merger(returns);

        // merge the result at the end of the body with all the results
        // at each return statement
        return merge(result,returns_merged);
    }

    T analyze_return(T in)
    {
        returns.push_back(in);
        return in;
    }

    // set of dataflow result
    std::vector<T> returns ;
};
```

Listing 8 Implementation of component function_simple

3.2.7 Helper components

Those components implement functions which are provided to help with common operations an analysis writer might find useful. We provide three helper components:

- a statement dispatcher function (called in Listing 1 and Listing 10) which takes an object of superclass *Statement* and dispatches a call to its runtime subclass, such as *AssignStmt* or *LoopStmt*. This is part of the necessary double dispatching used by the Visitor pattern[22], which the framework derived from. The complete code of this component is listed in appendix A.
- a merger function, which takes a list of dataflow results, and merge them together one-by-one. Of course, it uses the `T merge(T,T)` function which is analysis specific, and thus should be implemented inside the analyzer class. Figure 3.5 explains how it works graphically. The complete code of this component is listed in appendix B.

```
#define BASE static_cast<derived*>(this)
template <typename derived, typename T>
struct function_fixpoint
{
         T analyze_function(Function f, T in)
         {
              partial_result[f] = {} ;

              while (true)
              {
              T result = BASE->analyze_sequence(f->getBody(),in);
              T returns_merged = BASE->merger(returns);
              T final = BASE->merge(result,returns_merged);

              // we have reached a fixpoint
              if (final == partial_result[f])
                   return final;

             // we continue the analysis
             partial_result[f] = final ;
              }

         }

         T analyze_return(T in)
         {
             returns.push_back(in);
             return in;
         }

        // set of dataflow result
        std::vector<T> returns ;

        // map to store partial dataflow results
        std::map<function,T> partial_results ;

};
```

Listing 9 Implementation of component function_fixpoint

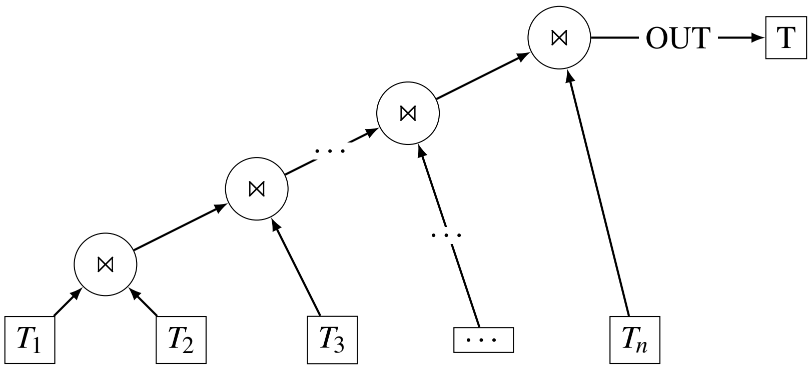

Figure 3.5 Graphical explanation of the `T merger(std::vector<T>)` function

```
#define BASE static_cast<derived*>(this)
template <typename derived, typename T>
struct flow_insensitive_base :
public framework::statement_dispatcher<derived,T>,
public framework::forward<derived,T>
{
    T analyze_statement(
        const Statement* st,
        const T& in)
    {
        T dataflow = BASE->statement_dispatch(st,in);
        return dataflow;
    }

    T analyze_ifelse(
        const IfElseStmt* ifelse,
        const T& in)
    {
        T end_if = BASE->analyze_sequence(
                                ifelse->getIfBranch(),
                                in);
        T end_else = BASE->analyze_sequence(
                                ifelse->getElseBranch(),
                                end_if);
        return end_else;
    }

    T analyze_loop(
        const LoopStmt* loop,
        const T& in)
    {
        return BASE->analyze_sequence(
             loop->getBody(),
             in);
    }

    T analyze_break(const T& in)
    {
        return in;
    }

    T analyze_continue(const T& in)
    {
        return in;
    }

}
```

Listing 10 Implementation of flow_insensitive_base

3.3 Implementation

The implementation of the framework requires a way to select between different algorithm's behavior (also known as the *Strategy pattern*). A classical implementation of this pattern would use the inheritance mechanism and the virtual method call dispatching mechanism available in object-oriented language to resolve that[17]. However, two problems would arise with this approach:

- the *C++* language doesn't allow virtual generic function.
- this would means that every dispatch call in the framework will have the overhead of a virtual method call (which has been evaluate to be non negligable performance-wise[23][24]).

We decided here to take advantage of C++ template programming functionalities to allow a generic and modular implementation, without any runtime overhead. This programming idiom is called *Curiously Recursive Template Pattern*[25]. The definition of this idiom is that a class `Derived` derives from a class template instantiation using `Derived` itself as template argument of the base class.

```
 // The Curiously Recurring Template Pattern (CRTP)
template<class Derived>
class Base
{
// methods within Base can use template
// to access members of Derived
};
class Derived : public Base<Derived>
{
// ...
};
```

Listing 11 Description of CRTP

The trick here is that the code in the class Base can access any member in Derived, by doing a cast to the type parameter `Derived`. Thus, you just have to know which method signature and properties are available in `Derived`, but not the actual implementation.

For a flow-sensitive forward analysis named `analyzer` with fixpoint solver, we could inherit from all the components we need taking care to pass the class `analyzer` itself as the `Derived` template parameter.

A real-world example is Listing 12, which is taken directly from the implementation of the type inference analysis described in the next chapter.

```
struct analyzer:
// analyzer is passed itself as the
// derived template parameter
public framework::flow_sensitive_base <analyzer,I> ,
public framework::forward <analyzer,I> ,
public framework::loop_fixpoint <analyzer,I> ,
public framework::function_fixpoint <analyzer,I>
{
    // ...
}
```

Listing 12 Example from the type inference analysis described in Chapter 4

By analyzing the resulting assembly code in Listing 13, we see that (as expected) it only contains static calls, which are much faster than dynamic calls (because of branch prediction and pipelining issue). This is fairly impressive considering that the call to `analyze_sequence` might be a user defined function, or one of the functions provided by the framework. This idiom is thus often called *static polymorphism* or *static mixin* because it provides an efficient way to *plug* a functionality inside a class statically and without any overhead.

```
  fw_ifelse.h * auto info_else =
  fw_ifelse.h * static_cast<derived*>(this)->analyze_sequence(
  fw_ifelse.h *           static_cast<StmtSequence*>(stmt_else),
  fw_ifelse.h *           in);
  leaq        368(%rsp), %rdi
  movq        %rbx, %rcx
  movq        %r14, %rdx
  movq        %r12, %rsi
  call        _ZN4mcvm8analysis9framework8sequenceINS0_13...
    ...typeinference8analyzerENS3_1IELb0EE16analyze_seque
```

Listing 13 Resulting assembly code compiled with GCC 4.8.2 -O3

3.4 Example of a reaching definition analysis implementation

To demonstrate the process of creating an analysis, we present an example analysis and step through the process. The example being implemented is the well-known reaching definitions analysis.

A *definition* is an assignment of a value to a variable. A definition d : `x = E` *reaches* a program point p (equivalently, d is a *reaching definition* at p) if there is a control flow path from d to p such that `x` is not redefined anywhere along that path.

In order to implement this analysis, we will complete the following tasks:

- Define the type which represents the domain `T`.
- Define a merging function over the domain: `T merge(T,T)`.
- Define a equality comparison function over the domain: `bool ==(T,T)`.
- Define the analyzer class by either including components or writing the required functions manually.

Based on the description of the analysis, we define our dataflow domain as a mapping from variable names to sets of assignments statements.

```
using ReachDefDomain =
    std::map<std::string,std::set<AssignStmt>>;
```

Listing 14 Domain for the reaching definitions analysis

The next step is to implement the merging function over the domain defined in Listing 14. The semantics of the merging operation is entirely analysis-specific, thus the framework doesn't try to provide any component for this. For the reaching definition analysis, the merge function is implemented in Listing 15. We define the merge of a and b as follows:

- if neither a nor b have a definition for variable v, then the resulting abstract value doesn't have a reaching definitions set for variable v.
- if only one of a and b has a reaching definitions set for variable v, then the resulting abstract value has the same set of reaching definitions for variable v as this one.

- if both input abstract values *a* and *b* have a reaching definitions set for variable *v*, then the resulting abstract value has the reaching definitions set resulting from the union between the set of reaching definitions for variable *v* in *a* and the set of reaching definitions for variable *v* in *b* for variable *v*.

```
ReachDefDomain merge (ReachingDefDomain a, ReachDefDomain b)
{
    ReachDefDomain out = a;
    for (auto pair: b)
    {
        auto itr = out.find (pair.first);
        if (itr != std::end(out))
        {
            // union the sets
            out[pair.first].insert (
                pair.second.begin(),
                pair.second.end());
        } else
        {
            // simply add the reaching definition
            // in b for this symbol
            out[pair.first] = pair.second;
        }
    }
}
```

Listing 15 Semantic of the merging function for the reaching definitions analysis

Then, we need to implement the equality comparison operator over the domain which is defined as: two reaching definitions abstract values are equal if they contains the same set of reaching definitions for every variables. This is actually the exact behavior of the standard *C++*equality operator, thus we don't have to define it by ourself.

At that point, we have defined a dataflow domain and the necessary functions to operate over it.

A reaching definition analysis is a flow-sensitive, forward, may analysis. We will take advantage of the framework to traverse the program representation, while respecting those properties.

We start by defining the analysis class in Listing 16, and we inherit from the components we need: `forward`, `flow_sensitive_base`, `loop_fixpoint` and `ifelse_sensitive`. To be complete, an analysis requires a definition for at least each of the functions described in Table 3.1. With the already inherited components, only the `analyzer_assign()` function isn't defined yet.

```
class ReachDefAnalyzer:
    public framework::forward
        <ReachDefAnalyzer,ReachDefDomain>,
    public framework::flow_sensitive_base
        <ReachDefAnayzer,ReachDefDomain>,
    public framework::loop_fixpoint
        <ReachDefAnalyzer,ReachDefDomain>,
    public framework::ifelse_sensitive
        <ReachDefAnalyzer,ReachDefDomain>
    {
    };
```

Listing 16 Declaration of the reaching definition analyzer

```
ReachDefDomain ReachDefAnalyzer::analyze_assign (
    AssignStmt* stmt,
    ReachDefDomain in)
{
    std::string x = stmt->getLeftHandSide() ;

    if (in.contains(x))
        in.remove(x) ;

    in.add(x,stmt) ;
    return in ;
}
```

Listing 17 Implementation of the analyze_assign() function for the reaching definition analysis

The implementation of the `analyze_assign()` function is straightforward as shown in Listing 17 (however it doesn't respect the complete semantic of MATLABwith constructs such as *eval*; supporting that is beyond the scope of this chapter). For an assignment `x = E`, the function verifies if `x` already has one or more reaching definitions in the entry set *in*. If there is, it removes them and add the current assignment statement to the set of reaching definition, else, it simply adds it.

Our reaching definition analysis is now complete. The full code of the analysis is given in appendix C.

3.5 Summary

In this section, we have presented our dataflow analysis framework. By using advanced programming techniques such as *CRTP* and by providing useful components such as the iterative fixpoint solver for the loop statement, the framework should allow future analysis writers to develop analysis more easily and faster, without sacrifing the performance over the convenience.

Chapter 4
Type Inference Analysis

In this chapter, we explain in detail the type inference analysis used by the just-in-time compiler described in Chapter 5. We begin with a simple textual description of the analysis, its roots in abstract interpretation and its domain. This is followed by a complete explanation of the inference rules used by the analysis. Finally, we describe the value analysis which has been developed to improve the type inference in specific cases.

4.1 Description of the algorithm

The type inference analysis described in this thesis is a forward, flow-sensitive analysis. It determines at each program point, a type (in the domain described in Section 4.2) to each variable in the program. Because it is used in the context of a just-in-time compiler, it aims at being simple and thus fast to execute. It is designed with the precise requirements of our compiler in mind, and doesn't try to be as complete and extensive as it could be.

By being based on the analysis framework described in Chapter 3, it supports any entry point in the program: it takes an *entry* abstract value and infers the types for every variable at every program point reachable from entry.

This analysis is based on abstract interpretation[26]. It propagates a type for each variable through all possible branches of a given program fragment. At each statement, it simulates the effect that these statements would have on the type of each variables. It does that in a conservative way, thus the result is always valid (but sometimes not precise enough to be of any subsequent interest). The inference follows a set of rules described in Section 4.3.

This dataflow analysis approach to type inference is well suited for dynamic languages, while statically typed languages such as OCaml generally use a type inference derived from the Hindley-Milner algorithm[27, 28]. Our forward iterative approach allows us to work on small code fragment. Also, it works even if a portion of the code fragment is invalid code: the computed abstract value will still be correct (but conservative).

4.2 Domain

Abstract interpretation is a way to simulate the execution of programs over an abstract domain. In MATLAB, being a dynamically typed language, variables are not associated with types but with values. Our abstract domain is a mapping of variables to types, the set of possible types being described in Figure 4.1. This set has important properties:

- it is partially ordered.
- every two elements have a *supremum* (also called a least upper bound or join), and an *infimum* (also called a greatest lower bound or meet).
- it has a finite height.

Those properties ensure that the number of iterations needed to find a fixpoint during the computation of the abstract result is finite, thus the algorithm will end eventually. By being a *must* analysis, we assure certainty of the result. At each program point, each variable is mapped to one precise type of the domain. In the case where multiple types could be possible at run-time, we just assume pessimistic knowledge ($\top$).

The domain illustrated in Figure 4.1 represents the subset of MATLAB we aim to compile: it focus on data structures which are frequently used in loops (iteration over matrices of double, the default type for numbers in MATLAB, and over structarrays), and logical for loop condition.

It contains tree branches, each of them having an associated base type:

- on the left, the *double* base type.

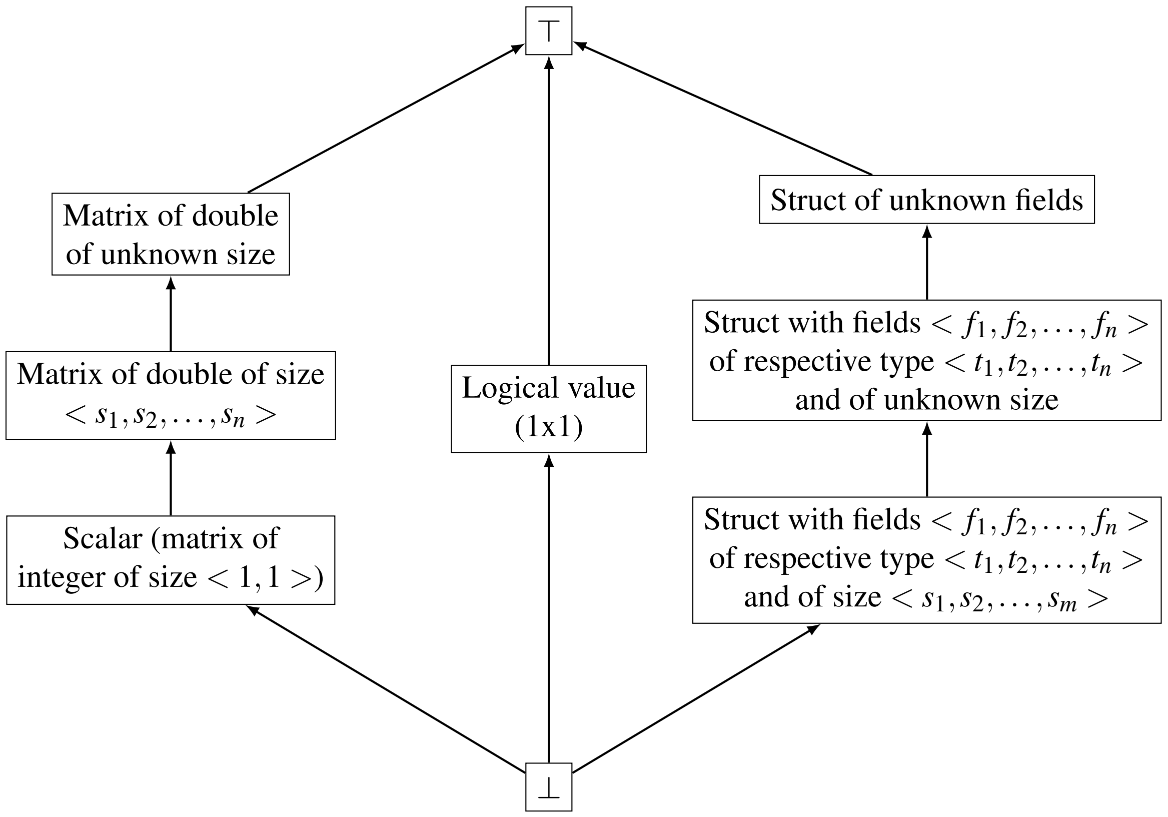
$\top$
Matrix of double of unknown size
Logical value (1x1)
Struct of unknown fields
Matrix of double of size $< s_1, s_2, \ldots, s_n >$
Struct with fields $< f_1, f_2, \ldots, f_n >$ of respective type $< t_1, t_2, \ldots, t_n >$ and of unknown size
Scalar (matrix of integer of size $< 1, 1 >$)
Struct with fields $< f_1, f_2, \ldots, f_n >$ of respective type $< t_1, t_2, \ldots, t_n >$ and of size $< s_1, s_2, \ldots, s_m >$
$\perp$

Figure 4.1 Domain of the type inference analysis

- on the middle, the *logical* base type.
- on the right, the *struct* base type.

Knowing only the basic type (such as *double* or *struct*) is not enough to build a truly efficient just-in-time compiler. The type inference analysis described in this thesis approximates, not only the base type but also other interesting properties about the variable. Thus, the domain is actually more complicated than only those three base type

The *double* branch: This branch of the domain contains 3 differents elements:

- the most precise result we can get is when we are able to determinate that a double matrix is actually a scalar number (remember that in MATLAB, a scalar number such as *1* is actually considered as a double matrix of size 1x1). In loops, the increment is almost always done on an integer number (such as *for i=1:10*), and generating code which works with integer numbers instead of floating-point numbers allows better performance.

- another possible result is when the analysis determines the size of the matrix but don't infer that this is a scalar. The result type is a *double* matrix of size $< s_1, s_2, \ldots, s_m >$. Knowing the size of the matrix will help to generate faster code by removing some boundchecking which would have happen at run-time without this information.
- when the analysis can't infer the size, it just infers that the result type is a *double* matrix of unknown size. This is the least precise result for the *double* branch.

The *struct* branch: A structure in MATLAB is a multi-dimensional value which contains several named fields, each one holding a value. Thus, the domain contains 3 differents members for the *struct* branch, depending on the knowledge of:

- the name of each field composing that structure.
- the type of each field composing that structure.
- the size of the structure.

To efficiently compile structure-based MATLAB code, we need to know its memory representation (the same kind of information that a C compiler knows about `struct`): this means that we have to know the number of fields and recursively the in-memory representation of each field. Thus, our type inference analysis can output:

- a structure with all fields and their respective types known, and the size of the struct known (remember that any struct is actually a structarray). This is the most precise result, and the size information might help to avoid runtime array-bound check (see Subsection 5.2.6).
- the same but without the size information.
- a structure with unknown fields. We could be more fine grained for this case by keeping the fields that certainly exist, but not knowing a single field is enough to prevent compilation anyway, so we decided to keep the domain simple by just considering that all the fields should be known.

The *logical* branch: This branch contains a single element: our domain only considers logical as a single value, while MATLAB consider all logicals as a matrix of logical actually. This is because the usage of logical inside loops is the vast majority of the time a single value and not a matrix. Matrices of logical are not considered by our type inference analysis.

4.3 Rules

To simulate the effect of a program over our domain, we need to define a type inference rule for all possible nodes in the program representation.

Besides the statements which are dedicated to controlling the flow of the code, for which rules are described in Subsection 4.3.3, and expression statements which have the identity rule (except for edge cases, such as *eval*[1]), the core of the analysis is the assignment statement, described in Subsection 4.3.2.

4.3.1 Expression

The first step is to define the rules for expressions. The rules are straighforward. For each kind of expression in *McIR*, we give them a specific type, for example:

- if *expr* is a symbol expression `e`, its type is the type T_e in the input type inference map.
- if *expr* is a dot expression *x.y*: if the type of *x* is *struct*, and *x* contains a field *y* of type *t*, then the infered type for *expr* is *t*, else it is $\top$
- if *expr* is a matrix expression `[ 1 2 ; 3 4 ]`, its type is a *double* of size *<2,2>*.
- if *expr* is an integer constant, for example `2`, its type is the *scalar* type (in the *double* branch).
- if *expr* is a floating point constant, for example `2.34`, its type is a *double* of size *<1,1>*.
- if *expr* is a binary operation expression `x OP y`, the type rule is specific to each *OP* in MATLAB: for example, the result of a binary addition (+) between *x* and *y* both

[1]Which are easy to handle correctly with a $\top$ result, but not of any interest for this thesis

being the *scalar* type is the *scalar* type, but it's a *double* if the binary operation is a division between those two.

- if *expr* is a call to a library function provided by MATLAB, the result is again specific to each specific function. For example, the *floor* builtin function returns a type *double* with the scalar property, but the *abs* doesn't.
- if *expr* is a call to a user-provided function, its type is the type of the returned value by analyzing this function. The special case of recursion is described in Subsection 4.3.5.

MATLAB is well known for its extensive set of operators, each one applicable to several different types and resulting in different results, and its huge number of builtin functions. Providing a type rule for each of them would be a tremendous engineering task, thus we focus on supporting the most frequently used operators and builtin functions.

4.3.2 Assignment statement

Assignment statements are the essential part of our analysis: without considering the corner cases introduced by MATLAB dynamic features (such as *eval*), those are the only kind of statement which can define a variable, and thus a type in our analysis.

We need to introduce several terms which are part of the rule 2 before giving the actual rule.

First, the *root* of an expression is (recursively) defined as the left-most symbol:

- for a symbol expression such as *a*, the root is *a* itself.
- for a parameterized expression such as *a(b)*, the root is the root of *a*.
- for a dot expression such as *a.b*, the root is the root of *a*.

The set of variables in a program is the set of *roots* of all the assigned expressions in the program. For example, in the program described in Listing 18, the set of variables is the singleton $\{a\}$.

Another term introduced in the rule 2 is the *inclusion* of one type into another. Let's see an example to understand why it is necessary to introduce this operation: At the end of Listing 19, the inferred types are:

```
a = 1 ;
a.b = 1 ;
a(x).c.d = 2 ;
a(a(1).c.d) = 1 ;
```

Listing 18 MATLAB code fragment example to illustrate the *root* of an expression

```
1 a = 1.1;
2 a = true;
3
4 b = 1.1;
5 b.c = 1.1;
6
7 d.e = 1.1;
8 d.f = 1.1;
```

Listing 19 MATLAB code fragment example to illustrate *inclusion* between types

- *a* of base type *logical*.
- *b* of base type *struct*, with field *c* of type *double*.
- *d* of base type *struct*, with field *e* of type *double* and field *f* of type *double*.

We see that the assignment in line 1 has been overriden by the assignment in line 2 because they were both assigning to the same variable *a*, that the assignment in line 4 has been overriden by the assignment in line 5 because they were both assigning to the same variable *b*, but that the assignment in line 7 has not been overriden by the assignment in line 8, even if they were both assigning to the same variable *d*. Thus, we define the notion of *inclusion*:

Rule 1. *The inclusion of a type* a *into a type* b*, written* $include(a,b)$*, is a type defined as:*

- *if* a *and* b *don't have the same base type, then the result is* b.
- *if their base type is* struct*, then the result contains all the fields in* a *which don't exist in* b *with their respective type in* a*, all the fields in* b *which don't exist in* a *with their respective type in* b*, and for each field* f *which exists in both* a *with type* T_a *and in* b *with type* T_b*, the result has a field* f *with its type being defined recursively has* $include(T_a,T_b)$

With those definitions, the rule for the assignment statement is defined as follows:

Rule 2. *Consider an assignment statement* s *of the form* lhs = rhs*, where* lhs *and* rhs *are expressions, and a type inference mapping before this statement T.*

Then the type inference map after this statement, T', is the same as T except for the variable root(lhs)*, which have type:* $T'_{root(lhs)}$ = include($T_{root(lhs)}$,assign(lhs,type_expr(rhs)))

The function assign(lhs,type) *returns a type, and is defined as:*

- *if* lhs *is a symbol expression, then it returns* type.
- *if* lhs *is a dot expression* X.Y*, then it returns* assign(X,temp) *where* temp *is of base type* struct *with a field* y *of type* type.
- *if* lhs *is a parameterized expression* X(index)*, then it returns* assign(X,temp) *where* temp *is the same as* type *except for the size*[2].

We illustrate the assignment rule by inferring the type of a program using structures step-by-step, from Listing 20 to Listing 21. We assume the entry inference map is the empty set ∅.

```
1  a.a = 1.1;
2  a.b = true;
3  a.c.a = 1.1;
```

Listing 20 Example part 1

The type is straighforward in line 1,2 and 3. The inferred type T_a for variable a after line 3 is a *struct* with fields:

- *a* of type *double*
- *b* of type *logical*
- *c* of type *struct* with a field *a* of type *double*.

```
4  a.c.d = a.c.a;
```

Listing 21 Example part 2

[2]The exact algorithm to calculate the new size is not complex but too long to explain here because of the weird MATLAB indexing possibilities.

Let's explain the assignment rule at line 4: we start by figuring the type of the right hand side expression, here `a.c.a`. Thanks to the entry type inference map at this line, we know that this expression is of type *double*. Then we apply the recursive assignment rule: `a.c.d` is a dot expression, thus we start by constructing *temp* which is a *struct* with a field *d* of type *double*. We apply the recursion, and `a.c` is a dot expression, thus we again have to construct a *temp* with a field *c* of type the previous *temp*: a *struct* with a field *d* of type *double*. We apply the recursion again, and `a` is a symbol expression, so we have to do the inclusion of T_a into the most recent *temp* type. T_a is of type *struct*, and *temp* is too, thus we include the field *a* and *b* from T_a into our result type. For the field *c*, we apply the inclusion rule recursively thus the field *c* is of type *struct* with fields *a* and *c*.

4.3.3 Control-Flow statements

Our type inference analysis is a forward analysis: we iterate through every statement in a sequence in execution order (first to last). It is implemented by inheriting the `forward` component from the analysis framework described in Subsection 3.2.2. Our type inference analysis is flow-sensitive, which means it takes into account the different possible runtime paths followed by the program execution. However, the only nodes which has an effect in the program are the assignment statements, as described before. Expressions which are not part of an assignment, such as the condition for an if-else statement or a loop statement, are meaningless in the context of the type inference analysis. Thus, the control flow nodes have been implemented as:

- the if-else statement doesn't consider the condition, and simply runs the analysis in each branch, then merges the two results (see Subsection 4.3.4 for the merge definition).
- the loop statement doesn't consider the condition either, so it could be seen as simply running the analysis through the body of the loop until a fixed point is reached. Note that the real-world implementation is more complex because of the `continue` and `break` statements. We implemented this behaviour with the `loop_fixpoint` component available in the analysis framework (see Subsection 3.2.3 for the exact description of its behaviour).

4.3.4 Merge operation

Our analysis being flow-sensitive, it requires a merge operator to implement some control flow statements. The merge operation used in this type inference analysis can again be seen as sound, as it ensures correctness of the result, by being conservative if needed.

The merge operation for the base type is fairly straighforward, and we will explain it in prose:

- merging any abstract base type with $\top$ outputs $\top$.
- merging any abstract base type v with $\perp$ outputs v.
- merging any abstract base type v with another abstract base type w and v being not equal to w, outputs $\top$.
- merging any abstract base type v with another abstract base type w and v being equal to w, outputs the abstract type v.

In terms of the domain described in Figure 4.1, it means that merging elements from different branches outputs $\top$, which is obvious graphically.

In the case where the base type are the same (which means we are merging types of the same branch in the domain), we need to merge between the different possibles types in this specific branch.

For the *double* branch:

- if both are *scalar* types, the result is *scalar* type.
- if one is *scalar* type, and the other is *double* type with size 1×1, the result is *double* type with size 1×1.
- if both are *double* with the same size S, the result is *double* with size S.
- in any other cases, the result is *double* of unknown size.

For the *struct* branch:

- merging a *struct* of unknown fields with any other *struct* outputs a *struct* of unknown fields.
- merging a *struct a* with fields $< f_1, f_2, \ldots, f_n >$ of respective type $< t_1, t_2, \ldots, t_n >$, and a *struct b* with fields $< f'_1, f'_2, \ldots, f'_{n'} >$ of respective type $< t'_1, t'_2, \ldots, t_{n'} >$, outputs:

– if $n == n'$ and $\forall x \in [1:n], f_x == f'_x$, a *struct* with fields $< f_1, f_2, \ldots, f_n >$ of respective type $< merge(t_1,t'_1), merge(t_2,t'_2), \ldots, merge(t_n,t'_n) >$. If both a and b have size S, the size of the outputed *struct* is S, else its size is unknown.
– else a *struct* of unknown fields.

For example, in Listing 22, we are merging two types for variable a. Assuming the entry map is $\perp$ before line 1, a being a *struct* on both branches, the result is a *struct*. However, because the fields are not same, the variable a after line 7 is a *struct* with unknown fields.

```
1 if true
2     a.c = 1.1;
3     % end of if
4 else
5     a.a = 1.1;
6     % end of else
7 end
```

Listing 22 Merging two structures with different fields

4.3.5 Recursion

Recursion has been showed to be fairly rare in MATLAB program, and actually doesn't happen in our set of benchmarks. However, for the purpose of correctness of the analysis, we developed a simple strategy to support it. Thanks to the analysis framework, most of the support is already done by just using the adequate component, the function fixpoint, described in Subsection 3.2.5. The problem lies is the function call expression typing (when its calling the current function): because the analysis is not over yet, we only have a partial (uncomplete) result for the type returned by this function. Thus, a function call expression is typed as, assuming there is a map from the functions currently in the call stack and the type of their input parameters to the (partial) returned type:

- if the callee function is not in the map, then the callee is simply analyzed.
- if the callee function is in the map (and the input parameters are of the same type as the current ones), we are in a recursion, thus we return the last result for this function if it exists.

- if there is no last result for this entry in the map, it returns ⊥.

We illustrate this behaviour with the factorial function described in Listing 23. The function

```
function res = factorial(n)
if n < 2
    res = 1;
else
    res = n * factorial(n-1);
end
```

Listing 23 A user-defined factorial function in MATLAB

is called with as a parameter an integer number (`factorial(5)` for example). On the if branch, the variable *res* is of type double<1,1>(scalar). On the else branch, however, its type depends on the type of the recursive call. Because we don't have a partial result yet, the inferred type is ⊥, and the multiplication operator with operands of type double and ⊥ returns ⊥. We merge the double<1,1>(scalar) and ⊥ at the end of the if-else statement, which gives the partial result that the variable *res* is of type double<1,1>(double).

On the second iteration, the result for the if branch is the same. For the else branch however, we use the partial result of last iteration for the returned type of *factorial(double<1,1>(scalar))*, which was double<1,1> (scalar). The type of variable *res* is the same on both branches, the merge thus gives a result of double<1,1> (scalar).

The result being the same on two consecutive iteration, the result will not evolve again, thus the analysis is done and the final inferred type for res is double<1,1> (scalar) when the function factorial is called with an integer parameter.

We don't show a trace but the inference also works for mutually recursive functions.

4.4 Value analysis

We found that the type inference analysis was sometimes unable to determine the size of *double* matrices and structarrays, because real-world MATLAB program uses the array-growth feature of MATLAB. For example, the code pattern in Listing 24 is frequent, and because of the array-growth feature, the *size* of the variable *a*, which depends on the runtime value of variable *i*, is unknown during and after the loop, whatever it was before. This lack

```
for i=1:100
    a(i) = i ;
end
```

Listing 24 MATLABcode fragment example which might use array-growth

of information might induce some performance cost for the JIT compiler as explained in Chapter 5. We implemented a small value analysis to statically determine the *size* of some variables in simple cases, such as the one is Listing 24.

We designed a small *maximum value* analysis. It approximates the maximum integer value of each variable at each program point, to be able to determine more often the size of matrices and structures in the program.

It is a must, flow-sensitive and forward dataflow analysis over the domain is the integer numbers ($\mathbb{N}$).

We only consider integer values (because array indexing variable have to be integer), and we don't compute any binary operation on it (to keep the analysis simple and fast). Remember that the use case is really specific, it doesn't try to be a partial evaluation analysis.

Assignment statements are straighforward, control-flow is using the framework with the appropriate component (forward, flow sensitive). One difficult part of this analysis is the treatment of `for` loops. In Listing 25:

```
previous
for i=start:stop
    body
end
next
```

Listing 25 Definition of a `for` loop

- the body of the loop is analyzed with its entry information being the output of the `previous` statement, except for variable *i* which has integer value *stop*.
- then, `next` statement is analyzed with its entry information being the output of the `body` statement.

This particular rule is because of the specific semantic of a `for` loop in MATLAB.

We illustrate the analysis and the behaviour of MATLAB in Listing 26:

- after line 1, the variable *i* has maximum value *100*.
- after line 2, the variable *i* has maximum value *100* and the variable *j* has maximum value *30*.
- after line 3, the variable *i* has maximum value *100* and the variable *j* has maximum value *2*.
- next lines (4,5 and 6) don't change the maximum value of variables *i* and *j*.

Also, to ensure correctness of our analysis, we do a simple structural analysis on every `for` loop: a depth first search traversal of the body of the for loop to verify that there is no break or continue statements inside: this asserts that the `for` loop will be executed exactly for the specified range. Combining the maximum value analysis and the type inference analysis

```
1 for i=1:100
2     for j=1:30
3             j = 2;
4             b(i,j) = 1.1;
5     end
6 end
```

Listing 26 Example of the analysis running on two nested `for` loops

allows us to statically determine that after line 6, the variable b is of type *double* and has size *<100,2>*.

This is a simple range analysis, tailored to our specific use case. It would be possible to use other more sophisticated range analysis[29, 30].

Chapter 5
Simple Just-In-Time Compiler

This chapter describes our Simple just-in-time (SJIT) compiler, written as a contribution of this thesis. The SJIT is completely independent and fundamentally different from the already existing just-in-time compiler for *Mc*VM named *Mc*JIT[11], which has been described in Chapter 2. We will compare their performances in Chapter 6.

We begin this chapter by analyzing how the *Mc*VM interpreter works, and how it is extended by the SJIT compiler. Then, we describe how the SJIT works internally: the compilation strategy, the architectural choices we made, and the reasons for those choices. Finally, we explain how each supported language construct in *Mc*IR is implemented in our target language, LLVM IR.

5.1 Design of SJIT compiler

5.1.1 The *Mc*VM interpreter

*Mc*VM is an interpreter for the MATLAB language: it interprets the program at run-time. It is also highly interactive, which is praised by MATLAB users: it provides an interface where commands can be typed at the prompt, commonly called an Read-Eval-Print-Loop (REPL).

To understand how our SJIT interact with the interpreter, we first describe how the *Mc*VM interpreter is designed. Whenever a code fragment has to be interpreted, a frontend called *Mc*LAB parses it and outputs an untyped high-level abstract syntax tree, *Mc*IR. Then, the actual interpretation of the program happens: each statement is decoded, executed, until the program terminates. During the interpretation process, the interpreter maintains an environment: a map from variable names to pointers in the heap representing the data processed by the program. As described in Figure 5.1, data is handled with boxed objects which wraps the actual values:

- A matrix of double is represented by an object which has several attributes, such as the current size of the matrix and a pointer to the actual content of the matrix, which is stored contiguously in a row-major order in the heap. The same representation is used for logicals.
- Structarrays in MATLAB are highly dynamic, fields can be added on the fly and the fields in a structarray can have different types at different positions in the array.
  This is illustrated in Listing 27: we start by assigning a double value in the field *id_like_a_double_field* on the 4th element of the structarray *a*, then we assign a string value in the same field but on the 3rd element of the structarray. We print the same field on line 3, and we see that it contains value of different types, and even empty matrices.
  To represent this behavior, *Mc*VM represents a structarray as a matrix of map from field name to boxed object pointer.

```
>> a(4).id_like_a_double_field = 1.0 ;
>> a(3).id_like_a_double_field = 'hi' ;
>> a.id_like_a_double_field
ans =
     []
ans =
     []
ans =
hi
ans =
     1
```

Listing 27 Example of a MATLAB session which demonstrates that structarrays are weakly typed: the field `id_like_a_double_field` contains an empty double matrix, a chararray and a double matrix

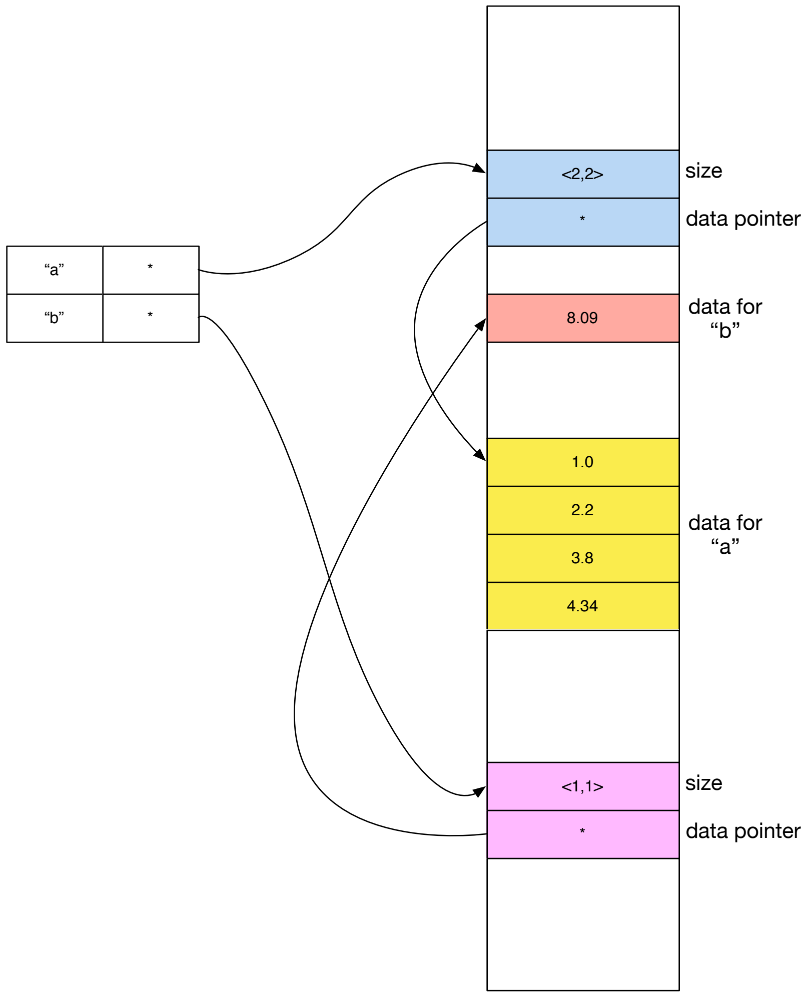

Figure 5.1 An environment containing two variables: *a* which is a matrix [1.0 2.2 ; 3.8 4.34] and *b* which is a matrix [8.09]

5.1.2 Performance troubles

It is well-known that interpreting programs is extremely costly in terms of performance, compared to running an already compiled program[31, 11].

For example, Listing 28 shows a MATLAB code fragment which computes the sum of all the elements in a vector of size 100. For each iteration of the loop, the interpreter has to:

- decode the statement: it is an assignment statement.

```
for i=1:100
    sum = sum + mymatrix(i);
end
```

Listing 28 Loop to compute the sum of all the matrix elements in MATLAB

- decode the rhs expression `sum + mymatrix(i)`: this is a binary operation.
- decode which binary operation expression: this is an addition operation.
- decode the lhs `sum` of the binary expression this is a symbol expression.
- get the pointer from the environment to the *sum* object.
- get the pointer to the raw data for the *sum* matrix (which is a 1x1 here).
- load the actual value from that pointer.
- decode the rhs `mymatrix(i)` of the binary expression: this is a matrix indexing expression.
- decode the indexing expression `i`: this is a symbol expression.
- get the pointer from the environment to the *i* object.
- get the pointer to the raw data for the *i* matrix (which is a 1x1 here).
- load the actual value from that pointer.
- decode the lhs `mymatrix` of the matrix indexing expression: this is a symbol expression.
- get the pointer from the environment to the *mymatrix* object.
- get the pointer to the raw data for the *mymatrix* matrix
- load the value at the correct offset (the indexing value computed before).
- compute the addition.
- decode the lhs `sum` expression of the statement: this is a symbol expression.
- get the pointer from the environment to the *sum* object.
- get the pointer to the raw data for the *sum* matrix (which is a 1x1 here).
- store the computed result at the pointed address.

While the cost those operations is acceptable with modern processors, running all those steps inside a potentially long loop isn't.

Out of those operations, several of them are interpreter-overhead, and some of them are boxed-value overhead. The advantage of compiled program is that they remove both, thus performing only the *mandatory* operations, by moving some of those operations from run-time to compile time, for example, Listing 29 shows the generated code by the SJIT for the `sum = sum + mymatrix(i)` statement, which has many fewer operations to perform than the interpreter.

```
"mysum = (mysum + mymatrix(i))":
; preds = %"i = temp_0"
  %60 = load double* %mysum
  %62 = load i32* %i
  %63 = getelementptr double* %mymatrix, i32 %62
  %64 = load double* %63
  %65 = fadd double %60, %64
  store double %65, double* %mysum
```

Listing 29 Generated code by the SJIT for the statement in Listing 28

Compiling MATLAB has already been tried (as explained in Chapter 2), and some projects have been really efficient in terms of performance[4]. However, those approachs don't support some real-world MATLAB programs which use dynamic features. By looking at real-world MATLAB programs, we realize that most of them use the same pattern:

- load and initialize the data (with functions such as *load*, *ones*, and *zeros*), ask user for input values.
- compute the result, mostly with long running numerical loops accessing (read and write) data.
- do something with the result (*disp*, *plot*, *save*).

Supporting this with a static approach requires complex and heavy analysis (only determining the call-graph statically is already extremely complex[32]), which are not suitable for a JIT compiler. Also, we are not aware of any static approach which supports common MATLAB features such as *save*, *load*,*cd* and *eval* and we suspect that there is not easy solution to this problem.

Thus, we decided to take a completely different approach in the design of the SJIT compiler. It should:

Be fast and simple

MATLAB is a complicated language, with a lot of idiosyncrasies and complex dynamic features. Supporting all of them in a compiler would range from hard to impossible. In the case where supporting them is possible, it would make the compiler require expensive analysis, which is not affordable in a JIT compiler. The SJIT should focus on compiling code which is responsible of the core of the computation, which is generally quite simple.

Produce very efficient code

However, we want to produce really fast code when the SJIT is triggered. To achieve that goal, we ensure that the SJIT produces *idiomatic* LLVM code, and then rely on the powerful LLVM JIT compiler to apply state-of-the-art optimizations.

Support useful data structures

The C language, for example, demonstrates that you can have very efficient and useful C-struct and C-array, which would translate in MATLAB with matrices and structarrays. The SJIT should support those.

To achieve all those goals at the same time, we decided to make the SJIT loop-based. This choice has a lot of advantages:

- loops are small compared to a whole program or a whole function, which makes analysis and compilation time small.
- in MATLAB, loops are the main reason for the huge interpreter overhead, as we saw with Listing 28 and Chapter 2. By compiling loops, which are computational intensive block of code, we get great performance improvement.
- loops rarely contains complex dynamic code such as *load* or *save*, which makes static analysis possible and fairly easy.
- loops have the same environment has their surrounding function in MATLAB, which makes interacting with the environment straighforward.

5.1.3 Execution model

Simply, our SJIT compiler is designed to replace the interpreter at executing loops, while working on the same data. Its execution mode is simple: everytime a loop has to be executed by the interpreter, the interpreter asks if the current loop is compilable within the current environment. If it is, it tells the SJIT to execute the loop within the current environment, if it's not, the interpreter executes the loop itself. This behavior is illustrated in Figure 5.2.

Thus, the SJIT compiler consists of only two public methods:

- `bool sjit::is_compilable (LoopStmt, Environment);`
  This function takes as an input the loop statement to be executed and the current environment, and returns a boolean value to tell wether or not this loop can be handled by the SJIT compiler.
- `void sjit::run (LoopStmt, Environment);`
  This function takes an an input the loop statement to be executed and the current environment, and executes the loop.

Both of those functions pass as arguments the loop to be executed, and the environment. The environment is a mapping from all the currently existing program variables to a pointer for their boxed value. Whether the loop is executed by the interpreter or by the SJIT compiler has the same effect on the data (those in the given environment) at the end of the loop, this is illustrated in Figure 5.2.

The loop compilation produces a completely standalone loop, without any call back to the interpreter until the end of the loop. This means that if a loop is compilable, every code underneath (function call included) are compilable. In Figure 5.3, we represent loops in the program in Listing 30: the circles represent the ones which are compiled, and the squares are interpreted. The important point of the picture is to realize that whenever a loop is compiled, every nested loop is compiled too.

```
for
    for
        for
        for
    for
        for
    for
        for
        for
```

Listing 30 Structure of a MATLAB program with nested loops

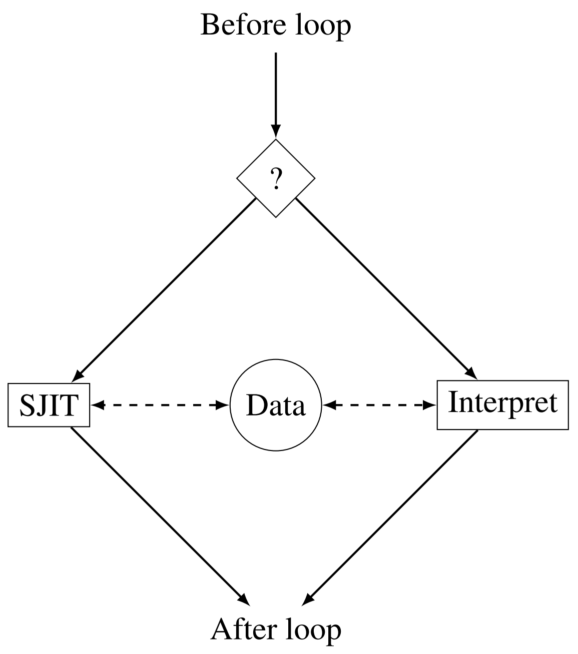

Figure 5.2 Execution of a loop in *Mc*VM enhanced with the SJIT

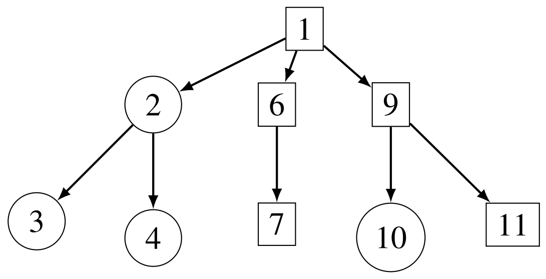

Figure 5.3 The names correspond to the line number of the loop in Listing 30

Determining if a loop is compilable

The first part of the SJIT compiler is to determine whether or not a loop is compilable to assembly code. As explained before, there are many choices of what is and is not compiled, which is a tradeoff between the complexity involved, both in terms of engineering but also in terms of speed of the SJIT compiler, and the number of real-world features which are highly optimizable.

To determine if this loop is compilable by our system, we run three analyses.

Structural analysis: This verifies that the loop uses only the feature-set which is the most optimizable and the most frequent in numerical programs: vectors, matrices, structures manipulation (read/write). This analysis only depends on the structure (the AST) of the loop statement.

Type inference analysis: This is the type inference analysis as described in Chapter 4. We give as an entry map the current type environment, by first converting each variable value in the given environment to a variable type.

This conversion is straighforward for matrices because all the elements inside a matrix have the same type. However, it is most difficult for structarrays because they are weakly type as we explained in Listing 27. Because fields with the same name can have different types inside a structarray, and those are unsupported by the SJIT compiler (because we could not use LLVM builtin instructions for structures), we need to verify that a field is always used with the same type through the structarray. It means that, for example with a structarray of size 10$\times$10 with fields *a* and *b*, we need to iterate 100 times to verify that all the fields *a* store the same variable type, and again 100 times for field *b*. This can be quite costly if the structarray is big or has a lot of fields and it might have some negative impact in terms of time spent to compute the initial entry map for the type inference analysis.

Type inference weeder: To simplify the code generation, we also run another analysis on top of the type inference result to exclude loops where variables have different types at different program points inside the loop. In practice, it rarely happens because using the same variable with different types is generally consider a bad programming practice[33]. We name this analysis the *type inference weeder*, because it weeds out from compilation programs which use the same variable with different types.

The *type inference weeder* analysis is flow-insensitive: it computes a single result (a mapping from variable name to type) which is valid at each statement in the program fragment it analyzes. The goal of this algorithm is twofold:

- to identify usage of the same variable name with different base types inside the loop. In Listing 31, the type inference analysis result after line 1 says that the variable *a* has base type *double*, but that it has base type *logical* after line 2. The type inference weeder analysis rejects those cases, thus determine that the variable *a* has unknown type ($\top$).

- for variables which are used with a single base type inside the loop, determine the other attributes (size, fields,...). This simplifies the code generation and also improves the performance, by avoiding multiple memory reallocations. In Listing 32, we see that the type inference weeder determines that for this whole code fragment, we should consider the variable *a* as a *struct* with fields *a* and *b*, even if the field *b* doesn't actually exist after line 1. Intuitively, it always tries to construct the biggest possible structure. For size, we use the same "take the biggest possible" size. The idea is described more precisely in the Subsection 5.2.6, which discusses the code generation for the array bound checks and the potential difficulties introduced by this simplication.

```
a = 1.1;
a = true;
```

Listing 31 The type inference weeder determines that the variable *a* is of type $\top$ in this program fragment

```
a.a = 1.1;
a.b = 1.2;
```

Listing 32 The type inference weeder determines that the variable *a* is of type *struct* with fields *a* of base type *double* and *b* of type *double* in this program fragment

Caching results

By design, the interpreter always asks the SJIT to execute a loop, and if not possible executes it itself. In the case where the loop is compilable, this is a gain. However, one of the analysis may determine that a loop is not compilable. In this case, there is an overhead for the interpreter, compared to directly interpreting the loop: we run some analyses but can't take advantage of them. The time of analysis is negligible compared to the gain when the loop is compiled, but the overhead might be costly when analyses are run too often.

An example of a problematic case is given in Listing 33: because the outer loop (the one from 1 to X) is not compilable, the inner loop (the one from 1 to Y) would be analyzed X times, which correspond to the number of time this loop is encountered by the interpreter.

```
for i=1:X
    for j=1:Y
        body;
    end
uncompilable_statement;
end
```

Listing 33 MATLAB code with a loop inside an uncompilable loop

A solution to this potential problem is to cache the results of the three analyses. Our SJIT compiler maintains two different maps:

- one which is a map from loop to boolean (the result of the structural analysis).
- another which is a map from loop to type inference weeder result.

Everytime the interpreter asks if a loop is compilable, we first loop in the caching maps and only if the result doesn't exist yet, we actually perform the analyses. It is described in pseudocode in Listing 34.

```
boolean function is_compiable (loop, environment)

if !structural_cache.contains(loop)
    structural_cache[loop] = structural_analyze(loop)

if !structural_cache[loop]
    return false;

env_type = construct_type_from_variables(environment)
key = make_pair(loop,env_type)

if !typeinferenceweeder_cache.contains(key)
    typeinference_cache[key] =
        typeinference_weeder_analyze(loop,env_type)

return typeinference_cache[key]
```

Listing 34 Pseudocode for the is_compilable function with cache

5.2 Code Generation

This section explains how some interesting constructs in *Mc*IR, such as loop statement or array access, are expressed in LLVM IR. We don't describe obvious translations such as sequence of statements and binary expressions.

5.2.1 LLVM IR

The LLVM project is a collection of modular and reusable compiler and toolchain technologies. It is widely used in both research and industy[34]. The LLVM IR is a powerful intermediate repsentation for efficient compiler transformations and analysis. For this thesis, the important characteristics of the LLVM IR are:

Low-level instructions

It is designed at a level close to real assembly languages like X86 or ARM. It doesn't have high-level language constructs such as loops. Control-flow has to be expressed with explicit constructs: branching, conditional branching or function call/return.

Higher-level typed data structures

It provides builtin support for C-style arrays and structures, and a powerful way to access elements in them with the *getelemptr* instruction[35]. All the data structures (scalar, matrices, structures) are typed to provide security and better opportunities for optimization.

Efficient

It can be used as an in-memory representation, and the LLVM JIT compiler can apply advanced optimizations on the IR such as constant propagation, vectorization and scalar replacement of aggregates.

5.2.2 Type-specialized block

As we explained before, a compilation is loop-based, within a certain environment. It means that the same loop might be compiled several times, because the compilation also depends on the surrounding environment (because the type inference analysis depends itself on the surrounding environment).

The code generation process, after having determined that the loop is compilable and having gotten the results of the type inference weeding analysis, is:

- Create an LLVM function which takes as arguments pointers to each variable raw data in the environment.
- Cast each variable to its inferred type.
- If the variable is a scalar (logical, integer or double), allocate a local copy of the variable and store in the local copy the current value from the environment.
- Compile the actual loop statement.
- If the variable is a copy, store the result back into the environment.

Creating local variables instead of using the one from the environment directly, which means loading their value at the beginning of the loop and storing them back at the end of the loop, is necessary because several LLVM optimizations only works on local variables (variables allocated inside the LLVM function with the `alloca` instructions). This is the case of the crucial *mem2reg* pass which move local variables into registers and and is the recommended way to generate LLVM IR[36].

```
1 a(10,10) = 1.1;
2 b = false;
3 c = 3;
4
5 % The compilation starts here
6 for cond
7     body
8 end
```

Listing 35 Example of a loop compilation

For example in Listing 35, the first three statements, from line 1 to 3, are interpreted and at line 5, the loop is compiled within an environment which contains:

- *a* with a base type *double*.
- *b* with a base type *logical*.
- *c* with a base type *scalar integer*.

```
define void @loop_ptr_35935600([6 x i64*]* %params_vec) {
entry:

; cast and load variable c
%0 = getelementptr [6 x i64*]* %params_vec, i64 0, i64 0
%1 = bitcast i64** %0 to double**
%c = load double** %1

; cast and load variable b
%2 = getelementptr [6 x i64*]* %params_vec, i64 0, i64 1
%3 = bitcast i64** %2 to i1**
%b = load i1** %3

; cast and load variable a
%4 = getelementptr [6 x i64*]* %params_vec, i64 0, i64 2
%5 = bitcast i64** %4 to double**
%a = load double** %5

; copy variable c
%c_copy = alloca i64
%12 = getelementptr double* %c, i64 0
%13 = load double* %12
%14 = fptosi double %13 to i64
%15 = getelementptr i64* %c_copy, i64 0
store i64 %14, i64* %15

; copy variable b
%b_copy = alloca i1
%16 = getelementptr i1* %b, i64 0
%17 = load i1* %16
%18 = getelementptr i1* %b_copy, i64 0
store i1 %17, i1* %18

; branch to the loop statement
br label %loop_init
...
```

Listing 36 The environment variables are passed as arguments to the LLVM function, cast to their inferred type (`double` for *a*, `i1` for *b*, `i64` for *c*, and copied (except *a* which is too big).

We compile the base loop as an LLVM function, taking as arguments pointers to each data in the environment. Part of the resulting assembly code is in Listing 36.

Sometimes, data is created inside the loop, and thus isn't contained in the initial environment. We need to take care of initializing an empty object of the correct type (the inferred one) before compiling the loop. In Listing 37, this means that we initialize a double matrix object (without any data inside) for variable *a* and for variable *mymatrix* and we put those variables in the environment. Thus, they will be accessible by the interpreter when it takes the control back (after the loop). If the size of the matrices *a* and *mymatrix* at the end of the loop is inferred by the type inference analysis, we store it directly in the object (for example, *val* is inferred to be of size 1x1), else it will be handle at run-time by the SJIT (with a performance cost, see Subsection 5.2.6).

```
a = 1.1;
for i=1:100
    val = a * i;
    mymatrix(i) = 1;
end
disp(a);
disp(val);
disp(mymatrix);
```

Listing 37 Loop compilation within an environment which contains variables only *a* before, but also *val* and *mymatrix* after.

5.2.3 If-Else statement

Like most assembly languages, the LLVM IR doesn't provide a direct language construct to express an If-Else statement from *McIR*. We implement this with named basic-blocks (*labels*) and branching instruction (*gotos*).

The compilation is straightforward. We generate 4 basic blocks: the one where the condition is evaluated (named *cond_ifelse* in Listing 39), one for the if body (*sequence1*), one for the else body (*sequence2*) and one for the end of the statement (*end_ifelse*).

```
if a
    b = 1;
else
    b = 2;
end
```

Listing 38 Simple example of an If-Else statement in MATLAB

5.2.4 Loop, Continue and Break statements

Again, the LLVM IR doesn't have a direct way to represent loop statements. The code generation strategy is based on 3 blocks:

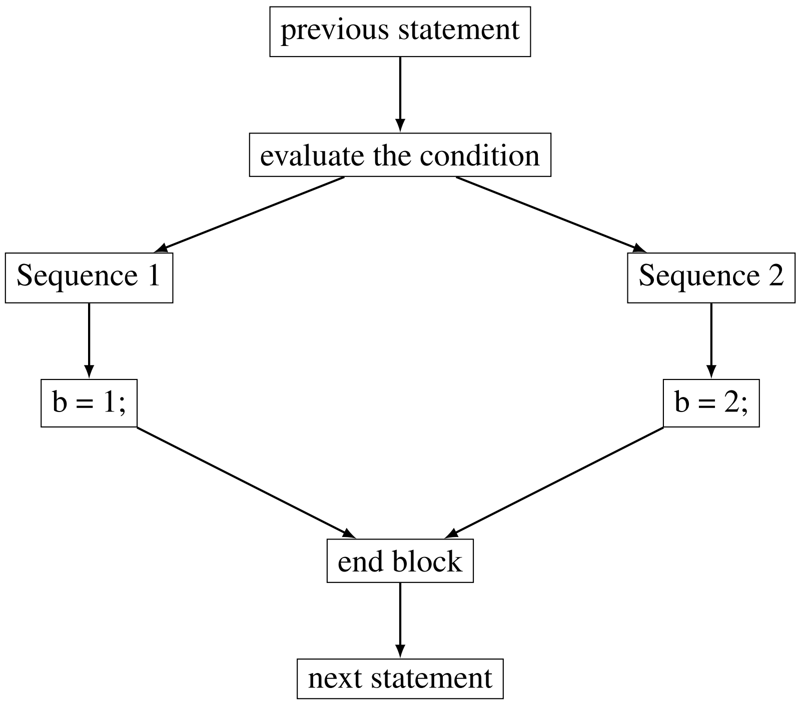

Figure 5.4 Control Flow Graph representing the LLVM code (basicblock and branching) for an If-Else statement

- an end block, which branchs to the instruction after the loop statement.
- a body block, which branchs to the first instruction in the body of the loop.
- a condition block, which evaluates the condition of the loop and branchs to either the end block or the body block.

Generating a `continue` statement is as simple as generating an LLVM branch instruction to the condition block. Similarly, a `break` statement is translated to a branch instruction to the end block.

5.2.5 Matrix and structure access

LLVM provides direct language support for C-style arrays and structures, and we take advantage of that in the SJIT.

```
cond_ifelse:
; preds = %"i = temp_0"
  %64 = load i1* %a_copy
  br i1 %64, label %sequence1, label %sequence2

end_ifelse:
; preds = %"b = 2", %"b = 1"
  br label %loop_incr

sequence1:
; preds = %cond_ifelse
  br label %"b = 1"

"b = 1":
; preds = %sequence1
  store i32 1, i32* %b_copy
  br label %end_ifelse

sequence2:
; preds = %cond_ifelse
  br label %"b = 2"

"b = 2":
; preds = %sequence2
  store i32 2, i32* %b_copy
  br label %end_ifelse
```

Listing 39 Assembly code for an If-Else statement in LLVM

Matrix

In LLVM IR, `getelemptr` is an instruction which takes a pointer to a data structure and an integer which represents the offset, and returns a pointer to the element of that data structure at the corresponding offset.

For example, in Listing 40, we store into the variable *b* the 33rd (MATLAB uses one based indexing but LLVM uses zero based indexing) element of the *mat* matrix.

```
"b = mat(33)":
; preds = %"i = temp_0"
%60 = getelementptr double* %mat, i32 32
; %60 is a pointer to the element
%61 = load double* %60
; %61 is the element value
```

Listing 40 Assembly code for a matrix access in LLVM IR

Structarray

For structarrays, the dynamic representation inside the interpreter is a map from field name (a string) to pointer of data object. This representation isn't suitable for LLVM structure access instructions: they operate on a C-like representation (a contiguous region of memory which contained each data field aligned). Our solution is to create an LLVM friendly representation before executing the loop: we allocate a region of memory (with standard `malloc`) of the current size of structarray (it can be extended after) by the size of each field, and we put in each field a pointer to the actual variable in the dynamic representation. This is illustrated in Figure 5.5 (note that we haven't represented all the pointers to keep the figure readable). With this representation, the LLVM *getelemptr* instruction works correctly, and we get a pointer to a field of the structarray in the environment.

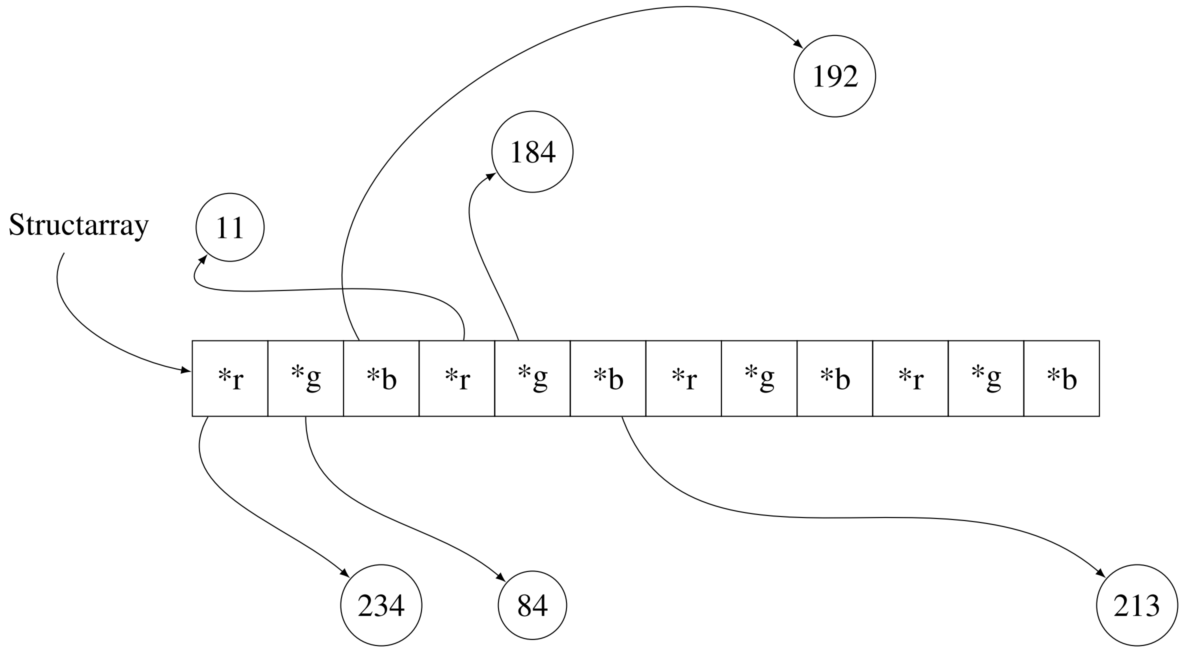

Figure 5.5 Memory representation of a 2×2 structarray with fields `r`, `g` and `b` suitable for LLVM

In Listing 41, we store into a structarray *st* at index <1,1> in the field *a*:

- At line 6, we get a pointer to the beginning of the structure at index <1,1> in the

```
1 "st(1, 1).a = 4.4":
2   %nbrows = load i64* bitcast (i64 40957360 to i64*)
3   %nbrcolumns = load i64* bitcast (i64 40957368 to i64*)
4   %55 = mul i64 0, %nbrows
5   %56 = add i64 0, %55
6   %57 = getelementptr %0* %st, i64 %56
7   %58 = getelementptr %0* %57, i32 0, i32 0
8   %ptr_to_elem = load double** %58
9   store double 0x40119999A0000000, double* %ptr_to_elem
```

Listing 41 Assembly code for a structarray access in LLVM IR

structarray.

- At line 7, we get a pointer to the element inside this structure. The offset is 0 because the field *a* is the first one in this structure (and LLVM uses zero based indexing).
- As a result, at line 8, the variable *%58* is a pointer to pointer of double. Thus, we dereference it to get the actual pointer to the element (named *%ptr_to_elem*).
- Finally, at line 9, we can store the number 4.4 at the pointed address.

5.2.6 Array Bound Checking

As we explained in Chapter 2, MATLAB provides two fundamental features which are appreciated by users:

- every read access to a matrix or a structure is bound checked: if the indices is bigger than the actual size of the accessed object, MATLAB emits a warning and gratefully prevents the operation (contrary to for example the C language which would crashes the program with a segmentation fault).
- every write access to a matrix or a structure is also bound checked. However, MATLAB has the rare feature of providing array-growth: the accessed object will be grown transparently to the necessary size to make the write possible.

The SJIT compiler, being a proper MATLAB subset implementation, follows this behavior. However, thanks to the type inference analysis, the code generation strategy emits the least possible runtime checks, to provide better performance (the performance cost of bound checking is described in detail in Chapter 6).

There are three different code generation strategies:

- if we know the size of the accessed object at this statement thanks to the original (not the weeded one) type inference analysis, and we also know the value of the index, we can verify the validity of the read or write at compile time, and directly generate either the error message or a call to the array growth function.
- if we know the size but not the index, like in Listing 42, we generate a single comparison instruction (on line 4) between the statically known size (10 in the example) and the indexing value.

```
1  "x = a(i)":
2  ; preds = %"i = temp_0"
3    %60 = load i32* %i_copy
4    %abc_check = icmp slt i32 10, %60
5    br i1 %abc_check, label %throw_abc_exception,
6                      label %"valid_abc_for_a(i)"
7
8  "valid_abc_for_a(i)":
9  ; preds = %throw_abc_exception, %"x = a(i)"
10   %61 = sub i32 %60, 1
11   %62 = getelementptr double* %a, i32 %61
12   %63 = load double* %62
13   store double %63, double* %x_copy
14   br label %loop_incr
15
16 throw_abc_exception:
17 ; preds = %"x = a(i)"
18   call void @static_throw_abc(i32 10, i32 %60)
19   br label %"valid_abc_for_a(i)"
```

Listing 42 Bound check generated when the size of the accessed object is known at compile-time (10) but the index isn't.

- if we don't know the size of the accessed object at this statement, we generate a runtime call to the get the size of the accessed object, and a runtime comparison to the indexing value. This is the most frequent case in practice, and it's the case with the biggest runtime overhead: at each access, we need to load a value from memory (the current size of the accessed object), and compare it to the index.

With the example given in Listing 43, the generated bound checking code at each line would be, assuming *a* is of type *double* and of size <1x10> in the entry type inference map:

1. *a* has size 1x10, the assignment to *a(10)* is valid thus no array growth check is needed here.

2. *a* has size 1x10, the matrix read *a(8)* is valid thus no boundcheck is generated here.
3. *a* has size 1x10, the matrix read *a(x)* with the value of *x* unknown requires a runtime boundcheck which check if *x* is less or equal to *10*.
4. *a* has size 1x10 before this statement, the matrix write *a(20* is valid and is handled by the type inference analysis, thus require no array growth check (it will be directly initialized to the statically determined maximum size at the beginning of the compilation).
5. *a* has size 1x20, the matrix read *a(15)* is valid thus no boundcheck is generated here.
6. the matrix write *a(x)* with the value of *x* unknown need a runtime array growth check.
7. *a* has unknown size now, thus any code after that accessing (read/write) the matrix *a* will need boundcheking code.

```
a(10) = 1.1;
b = a(8);
c = a(x);
a(20) = 2.1;
d = a(15);
a(x) = 4.23;
e = a(200) ;
```

Listing 43 MATLAB code to show when a runtime boundcheck is generated

5.3 Summary

In this chapter, we described our SJIT compiler. Its core idea is to interact with the data objects already existing in the environment, by first getting the data and adapting them in a suitable way (by copying, casting and reorganizing the data if necessary). It then produces straighforward LLVM IR code and thus benefits from robust LLVM optimizations. We saw that there is a direct translation for regular data where MATLAB has some type constraints (such as double matrices, where all the elements has to be doubles), but for weakly typed data such as structarrays in MATLAB, where each element can have the type of its fields different from another element, the translation requires more complex analysis and data transformation.

Chapter 6
Performance Evaluation

In this chapter, we present the performance results of our SJIT compiler, compared to previous dynamic implementations of the MATLAB language: *Mc*JIT, *Mc*VM, the reference MATHWORKS implementation, and OCTAVE. We start by describing the different benchmarks which are used to validate the performance of our SJIT, we continue with the actual results and some interpretation of them, and we finish with a discussion about the compilation time introduced by the just-in-time compilers.

6.1 Benchmarks

In this section, we describe our benchmarks, which come from various sources such as the FALCON project[14] and McGill numerical computation classes. They focus on the numerical aspect of MATLAB: matrices and structures. They don't use uncommon features such as lambda functions or GUI librairies.

bubl is an implementation of the bubble sort algorithm in MATLAB. It uses a vector of size 1×20000 filled with random numbers between 0 and 100. Its implementation is a single function with takes the unsorted array an input and outputs a sorted one: this function uses doubly nested loops.

euler31 is an implementation of a solution for problem 31 of the Euler project[37]. It uses a dynamic programming algorithm and store the results in a two-dimensional matrix of size 9×100000.

fiff is part of the benchmark set used by the FALCON project. It is an implementation of finite-difference solution to the wave equation. It performs scalar operations on a matrix of size 10000×10000.

dich is an implementation of the Dirichlet solution to Laplace's equation. It is a loop-based program that performs scalar operations on a two-dimensional matrix of size 667×667. It is from the FALCON project.

crni is an implementation of the Crank-Nicholson solution to the heat equation. It is a loop-based program that performs scalar operations on a two-dimensional matrix of size 600×600. It is from the FALCON project.

capr is an implementation of the computation of the capacitance of a transmission line using a finite difference and Gauss-Seidel method. It is a loop-based program that involves two small matrices and performs scalar operations on them. It is from Chalmers University of Technology[4].

clos calculates the transitive closure of a directed graph. Its execution time is for the vast majority spent in a matrix multiplication between two matrices of size 2000×2000. It is a benchmark from the OTTER project[38].

rgbneg performs a calculation of the negative of a small image represented in the RGB color space[39], which is stored using a structarray of size 1024×1024 containing fields *r* (red), *g* (green) and *b* (blue). We found this representation more intuitive than the three-dimensional matrix used by MATLAB librairies to represent images.

rgbtohsv performs a transformation of an RGB image (using a structarray of size 1024×1024 and fields *r*, *g* and *b*) to the same one using an HSV representation[40] (using a structarray of size 1024×1024 and fields *h*, *s* and *v*). The MATLAB code for this algorithm is given in appendix D. Interestingly, MATLAB provides a builtin function to do this conversion (named `rgb2hsv`). It takes a three dimensional matrix has input and outputs a three dimensional matrix: by using the exact same representation for two actually different image representations, we believe that this is confusing for the user and error-prone. Using structarrays would help make the code self-documenting and thus more explicit.

6.2 Performance results

We compare:

- our SJIT implementation (on top of the *Mc*VM interpreter).
- the SJIT with array-bound checking enabled (both for reads and writes).
- the previous *Mc*JIT implementation (on top of the *Mc*VM interpeter) without array-bound checking.
- the reference MATHWORKS MATLAB implementation R2013a 64-bits.
- the *Mc*VM interpreter without any just-in-time compiler.
- OCTAVE 3.8 with the default compilation configuration.

The results are shown in Table 6.1. They have been run on an Intel Core i7-3517U with 10GB of memory, with the operating system being Arch Linux (kernel version 3.12.9).

Benchmarks `bubl`, `euler31`, `fiff` and `dich` have similar results: they generally show that the SJIT is the fastest implementation, followed closely by *Mc*JIT. They are between 5 and 10 times faster than the MATHWORKS implementation. The interpreters are performing poorly, but OCTAVE is particularly slow.

The `clos` benchmark has unique results because it is almost exclusively performing a matrix multiplication, thus actually depends on the performance of the underlying matrix multiplication implementation. The MATHWORKS implementation and the SJIT are the fastest because they both use Intel Math Kernel Library[41] as their matrix multiplication

	SJIT	SJIT (with ABC)	*Mc*JIT	MATLAB	*Mc*VM	OCTAVE
bubl	0.88	1.23	0.96	11.45	287	3484
euler31	0.74	0.73	0.71	2.45	167	1519
fiff	0.46	0.6	0.99	4.24	212	2187
dich	0.81	0.95	1.38	5.47	421	3373
clos	4.98	4.97	7.16	4.36	11.87	20.22
crni	0.41	0.41	2.59	0.09	4.25	20.24
capr	0.66	0.64	0.45	2.03	273	632
rgbneg	0.149	0.157	N/A	2.24	251	53
rgbtohsv	0.185	0.197	N/A	13.53	428	398

Table 6.1 Performance results (in seconds)

implementation, which is known to be very efficient. The other implementations are in the same order of magnitude in terms of performance.

The `crni` benchmark is largely dominated by the MATHWORKS implementation, which is about 4 times faster than SJIT (with or without array-bound checking), and more than 25 times faster than *Mc*JIT. This benchmark contains a long function (more than 20 instructions), which contains a small loop inside it of only 3 instructions: 2 functions calls and 1 slice assignment. The SJIT doesn't have support for slice operation yet, and thus the loop is executed by the interpreter. *Mc*JIT is unable to fully compile this loop neither and produces code which uses some level of interpretation. However, the MATHWORKS implementation seems to compile it just fine. Note that the SJIT is still 5 times faster than *Mc*JIT because the function calls contains inner loops, which are compiled by the SJIT because it takes advantage of the state of the environment just before those loops.

The `capr` benchmark yields interesting results: it is the single benchmark where *Mc*JIT performs better than the SJIT (but both run about 3 times faster than the MATHWORKS implementation). This benchmark is made of a loop (running 100 times) containing 3 instructions, 2 of them being function calls and containing 5 inner loops inside them (which are responsible for the main computation), and the last one being a library call (Listing 44 represents it in pseudocode). *Mc*JIT compiles the whole function and everything inside it fine, however, the SJIT doesn't compile the outer loop because we didn't implement this library call, thus this loop is given back to the interpreter. However, the interpreter tries to

```
1  function capr
2  ...
3  outer_loop
4      inner_loop1
5          ...
6      end
7      inner_loop2
8          ...
9      end
10     inner_loop3
11         ...
12     end
13     inner_loop4
14         ...
15     end
16     inner_loop5
17         ...
18     end
19     library_function_call();
20 end
21 ...
```

Listing 44 Pseudocode for the `capr` benchmark. Three dots are representing instructions.

defer the execution of each inner loop to the SJIT (as explained in Subsection 5.1.3), which works because those five inner loops are simple matrix iterating loops. Thus, the SJIT still achieves good (but not optimal) performance because the instructions which would be at lines 5, 8, 11, 14 and 17 are compiled. We believe that this is an example of the fact that targeting loops allows the SJIT to be simple: even if it doesn't support some library calls yet, it always has good performance because *at some point* inside a program, the loops are the vast majority of time really simple and thus handled by the SJIT.

For the `rgbneg` and the `rgb2hsv` benchmarks, the MATHWORKS implementation performance is abysmal, which is expected considering that those benchmarks are extensively using structarrays. The SJIT doesn't suffer from the usage of structarrays, and by looking at the just-in-time compiled assembly code, we see that using structures in LLVM only add one instruction per access (the multiplication by the size of the structure to get the correct address) compared to accessing a double matrice, which is almost invisible in terms of running time.

6.2.1 Array-bound checking performance

The first observation made analysing the generated code is that eliminating array-bound checking in MATLAB is difficult and would require a full-featured range analysis to be of any use. This type inference doesn't provide it, and thus the SJIT with array-bound check enabled generates checks for all array reads and writes.

In terms of performances, we discover that the cost of array-bound checking in the SJIT varies: it is negligible in most benchmarks (`euler31`, `dich`, `crni`...) but sometimes yield some performance penalty (to a maximum of +40% in `bubl`). We suspect that this behavior is related to branch prediction and caching, which sometimes makes the cost of array-bound checking invisible.

In general, we see that the performance cost of array-bound checking in MATLAB is fairly constrained (it stays in the same order of magnitude), compared to the cost of interpreting for example.

6.3 Overhead of just-in-time compilation

6.3.1 Compilation and analysis time

During this thesis, we have stated that the SJIT compiler, as the name implies, should stay simple, and thus fast. The compilation time results show that our insights were valid, and the SJIT is really fast in terms of compilation (analyses included) time: around 10 milliseconds on all benchmarks, depending on the size of the loops in terms of number of statements.

In comparison, *Mc***JIT** has much higer compilation time, because it always compiles a lot more statements (complete function vs loop, which in the `dich` benchmark for example means compiling 48 instructions but only 13 for the SJIT) and thus the analyses are much more complex and time consuming.

One particular case where the compilation time is quite heavy in the SJIT is the presence of structarrays (benchmarks `rgbneg` and `rgbtohsv`). As we explained in Section 5.2.5, before the analysis starts, we need to verify that structarrays in the environment follows certains properties (all structures in the structarray have the same fields with the same types) which are not enforced by the MATLAB interpreter. For example, if we consider an environment containing a structarray of size 1024×1024 with 3 fields, it means verifying 3145728 (1024×1024×3) the dynamic type of an element.

	SJIT	*Mc*VM
bubl	0.002	0.103
euler31	0.003	0.025
fiff	0.005	0.242
dich	0.010	0.337
crni	0.004	0.341
clos	0.008	0.133
capr	0.013	0.416
rgbneg	0.097	N/A
rgbtohsv	0.109	N/A

Table 6.2 Compilation time (in seconds)

6.3.2 Cached analysis time

By the design of our SJIT compiler, every time a loop is executed, the interpreter asks if this loop is runnable by the SJIT. As already described, the `crni` benchmark contains an outter loop which is not compiled by the SJIT, but executed 600 times. This loop containing 3 nested loops, everytime this loop is executed by the interpreter, it asks the SJIT compiler 1800 times if the inner loops are compilable (3 × 600). Thanks to the caching mechanism, the cost of asking if some loops are compilable 1800 times results in an overhead of around 10ms, which is negligible compared to executing even a few instructions in the interpreter anyway. Thus, the pure overhead introduced by the SJIT when it doesn't compile (which means running the analyses for no benefits) is extremely constrained.

Chapter 7
Conclusions and Future Work

This thesis has introduced the SJIT compiler, an efficient just-in-time compiler for MATLAB. It takes advantage of existing ideas (such as dataflow-based type inference, type specialization, and loop-based compilation) and applies them on a widely used numerical language, MATLAB. The performance results are impressive, being much faster than the reference MATHWORKS implementation in most benchmarks.

The important points of this thesis are:

- Loop-based just-in-time compilation fits nicely with numerical programs, which are often based around small computation kernels iterating over matrices.
- Targeting a well-known assembly language like LLVM allows the SJIT to benefit from state-of-the-art optimizations.
- The essential analysis to be able to generate valid and efficient code is the type inference analysis. Determining ranges for bound-checking optimization is minor in comparison (while being actually harder).
- Just-in-time compilation allows us to keep the compiler simple compared to ahead-of-time compilation for a dynamic language such as MATLAB, while still achieving excellent performance. It also allows us the integrate it in interactive development environment thanks to the extremly small compilation time.

7.1 Future Work

The current SJIT focuses on the general usage of MATLAB: double matrices, logical and structarrays. Future work should go toward supporting more MATLAB features such as:

- Complex numbers: some work has already been done in other projects[4].
- Cell arrays would be a good opportunity for optimization: its usage is quite important in MATLAB and we suspect that even if MATLAB doesn't put any type constraint on it, most users use cell arrays as a structured data, sometimes in place of more adequate data structures (such as structarrays or plain matrices).
- Function handles (which represent lambda functions in MATLAB) could benefit from type inference to generate specialized (and thus optimized) functions on the type of its arguments.

Aside from supporting more MATLAB features, we think that several ideas could be explored to further improve performance:

- value-based specialization: our approach compiles a loop specialized on the current type of the variables in the input environment (before the loop). One could imagine a compilation strategy which compiles a loop specialized on the current value of the variables in the input environment. This could yield great performance results in some cases (thanks for example to constant propagation) but could also introduce too many compilations.
- better memory usage: MATLAB, by being a copy-based language, creates many memory allocations which could sometimes be avoid with dataflow analyses.

Appendices

Chapter A
Statement dispatcher code

```
#pragma once

#include "statements.h"
#include "assignstmt.h"
#include "ifelsestmt.h"
#include "loopstmts.h"
#include "exprstmt.h"
#include "stmtsequence.h"
#include "returnstmt.h"

#define BASE static_cast<derived*>(this)

namespace mcvm {
namespace analysis {
namespace framework {

    template <typename derived, typename T>
        struct statement_dispatcher {

            T statement_dispatch (
                    const Statement* st,
                    const T& in) {
                switch (st->getStmtType()) {
                    case Statement::ASSIGN:
                        return BASE->analyze_assign(
                                static_cast<const AssignStmt*>(st),
                                in) ;

                    case Statement::IF_ELSE:
                        return BASE->analyze_ifelse(
                                static_cast<const IfElseStmt*>(st),
                                in) ;

                    case Statement::LOOP:
                        return BASE->analyze_loop(
                                static_cast<const LoopStmt*>(st),
                                in);
```

Listing 45 Implementation of the statement_dispatcher component (1/2)

```
                    case Statement::BREAK:
                        return BASE->analyze_break(
                                static_cast<const BreakStmt*>(st),
                                in);

                    case Statement::CONTINUE:
                        return BASE->analyze_continue(
                                static_cast<const ContinueStmt*>(st),
                                in);

                    case Statement::RETURN:
                        return BASE->analyze_return(
                                static_cast<const ReturnStmt*>(st),
                                in);

                    case Statement::EXPR:
                        return BASE->analyze_exprstmt(
                                static_cast<const ExprStmt*>(st),
                                in) ;
                }
            }
        };

}}}
```

Listing 46 Implementation of the statement_dispatcher component (2/2)

Chapter B
Merger code

```
#pragma once

#include <vector>

namespace mcvm {
namespace analysis {
namespace framework {

template <typename derived, typename T>
struct merger {
    T merge_list(const std::vector<T>& infos) {
        if (infos.empty())
            return T{} ;

        auto merged = infos.front() ;
        for (auto& info : infos) {
            merged = static_cast<derived&>(*this).merge
                (info,merged) ;
        }
        return merged ;
    }
};

}}}
```

Listing 47 Implementation of the merger component

Chapter C
Reaching definition analysis code

```
#pragma once

#include <unordered_map>
#include <set>

#include <analysis/framework/helper.h>
#include <analysis/framework/fw_sequence.h>
#include <analysis/framework/fw_ifelse.h>
#include <analysis/framework/fw_merger.h>
#include <analysis/framework/fw_flow_sensitive_base.h>
#include <analysis/framework/fw_loop_fixpoint.h>


namespace mcvm {
namespace analysis {
namespace reachdef {


    using I = std::unordered_map<std::string,std::set<const AssignStmt*>> ;

struct analyzer :
    public framework::flow_sensitive_base <analyzer,I>,
    public framework::forward <analyzer,I>,
    public framework::ifelse <analyzer,I>,
    public framework::loop_fixpoint <analyzer,I> ,
    public framework::merger <analyzer,I>
    {

        I merge(const I&a, const I&b) const ;
        I analyze_assign(const AssignStmt* stmt, const I& in) ;
        I analyze_exprstmt(const ExprStmt* stmt, const I& in) ;

    };

}}}
```

Listing 48 reachdef.h

```
#include "reachdef.h"

namespace mcvm {
namespace analysis {
namespace reachdef {

        I analyzer::merge(const I&a, const I&b) const {
            I out = a ;
            for (auto& pair:b)
                out[pair.first].insert
                    (pair.second.begin(),pair.second.end());
            return out;
        }

        I analyzer::analyze_assign(
            const AssignStmt* stmt,
            const I& in) {

            auto left = stmt->getLeftExprs();
            auto out = in ;
            for (auto& id:left)
                out[id->toString()] = {stmt};
            return out;

        }

        I analyzer::analyze_exprstmt(
            const ExprStmt* stmt,
            const I& in) {

            return in;
        }
```

Listing 49 reachdef.cpp

Chapter D
RGB2HSV benchmark

```
function hsv = rgbtohsv(rgb,x,y)

% Preallocation
hsv(x,y).hue = 0;
hsv(x,y).sat = 0;
hsv(x,y).val = 0;

for i=1:x
    for j=1:y

        % Calculate the max rgb
        if rgb(i,j).r < rgb(i,j).g
            if rgb(i,j).g < rgb(i,j).b
                rgbmax = rgb(i,j).b;
            else
                rgbmax = rgb(i,j).g;
            end
        else
            if rgb(i,j).r < rgb(i,j).b
                rgbmax = rgb(i,j).b;
            else
                rgbmax = rgb(i,j).r;
            end
        end

        % Calculate the min rgb
        if rgb(i,j).r < rgb(i,j).b
            if rgb(i,j).r < rgb(i,j).g
                rgbmin = rgb(i,j).r;
            else
                rgbmin = rgb(i,j).g;
            end
        else
            if rgb(i,j).b < rgb(i,j).g
                rgbmin = rgb(i,j).b;
            else
                rgbmin = rgb(i,j).g;
            end
        end

        % Calculate the value (V) is the maximum
        % of the r, g and b
        hsv(i,j).val = rgbmax;

        % Normalize value to one
        rgb(i,j).r = rgb(i,j).r / hsv(i,j).val;
        rgb(i,j).g = rgb(i,j).g / hsv(i,j).val;
        rgb(i,j).b = rgb(i,j).b / hsv(i,j).val;
        rgbmin = rgbmin / hsv(i,j).val;
        rgbmax = rgbmax / hsv(i,j).val;

        % Calculate the saturation (S)
        hsv(i,j).sat = rgbmax - rgbmin;
```

Listing 50 MATLAB implementation of a conversion between an RGB and an HSV image representation (1/2)

```
        % Normalize saturation to one
        rgb(i,j).r = (rgb(i,j).r - rgbmin) / (rgbmax - rgbmin);
        rgb(i,j).g = (rgb(i,j).g - rgbmin) / (rgbmax - rgbmin);
        rgb(i,j).b = (rgb(i,j).b - rgbmin) / (rgbmax - rgbmin);

        % Calculate the max rgb
        if rgb(i,j).r < rgb(i,j).g
            if rgb(i,j).g < rgb(i,j).b
                rgbmax = rgb(i,j).b;
            else
                rgbmax = rgb(i,j).g;
            end
        else
            if rgb(i,j).r < rgb(i,j).b
                rgbmax = rgb(i,j).b;
            else
                rgbmax = rgb(i,j).r;
            end
        end

        % Calculate the min rgb
        if rgb(i,j).r < rgb(i,j).b
            if rgb(i,j).r < rgb(i,j).g
                rgbmin = rgb(i,j).r;
            else
                rgbmin = rgb(i,j).g;
            end
        else
            if rgb(i,j).b < rgb(i,j).g
                rgbmin = rgb(i,j).b;
            else
                rgbmin = rgb(i,j).g;
            end
        end

        % Calculate the hue (H)
        if rgbmax == rgb(i,j).r
            hsv(i,j).hue = 0.0 + 60.0 * (rgb(i,j).g - rgb(i,j).b);
            if hsv(i,j).hue < 0.0
                hsv(i,j).hue = hsv(i,j).hue + 360.0;
            end
        elseif rgbmax == rgb(i,j).g
            hsv(i,j).hue = 120.0 + 60.0 * (rgb(i,j).b - rgb(i,j).r);
        else
            hsv(i,j).hue = 240.0 + 60.0 * (rgb(i,j).r - rgb(i,j).g);
        end

        hsv(i,j).hue = hsv(i,j).hue / 360 ;

    end
end
```

Listing 51 MATLAB implementation of a conversion between an RGB and an HSV image representation (2/2)

Chapter E
Example of LLVM IR generated by the SJIT

```
cond_ifelse:
  %111 = load i64* %i_copy
  %112 = sub i64 %111, 1
  %113 = getelementptr double* %A, i64 %112
  %114 = load double* %113
  %115 = load i64* %i_copy
  %116 = add nuw nsw i64 %115, 1
  %117 = sub i64 %116, 1
  %118 = getelementptr double* %A, i64 %117
  %119 = load double* %118
  %120 = fcmp ogt double %114, %119
  br i1 %120, label %sequence6, label %sequence7
sequence6:
  br label %"swap_val = A(i)"
"swap_val = A(i)":
  %121 = load i64* %i_copy
  %122 = sub i64 %121, 1
  %123 = getelementptr double* %A, i64 %122
  %124 = load double* %123
  store double %124, double* %swap_val_copy
  br label %"A(i) = A((i + 1))"
"A(i) = A((i + 1))":
  %125 = load i64* %i_copy
  %126 = add nuw nsw i64 %125, 1
  %127 = sub i64 %126, 1
  %128 = getelementptr double* %A, i64 %127
  %129 = load double* %128
  %130 = load i64* %i_copy
  %131 = sub i64 %130, 1
  %132 = getelementptr double* %A, i64 %131
  store double %129, double* %132
  br label %"A((i + 1)) = swap_val"
"A((i + 1)) = swap_val":
  %133 = load double* %swap_val_copy
  %134 = load i64* %i_copy
  %135 = add nuw nsw i64 %134, 1
  %136 = sub i64 %135, 1
  %137 = getelementptr double* %A, i64 %136
  store double %133, double* %137
  br label %end_ifelse
sequence7:
  br label %end_ifelse
```

Listing 52 LLVM IR generated by the SJIT compiler for part of the `bubl` benchmark

Bibliography

[1] Cleve Moler. *The Growth of MATLAB and The MathWorks over Two Decades*. http://www.mathworks.com/company/newsletters/news_notes/clevescorner/jan06.pdf.

[2] MATLAB. *version 7.10.0 (R2010a)*. Natick, Massachusetts: The MathWorks Inc., 2010.

[3] Octave community. *GNU Octave 3.8*. 2014. URL: www.gnu.org/software/octave/.

[4] Jun Li. "McFOR: A MATLAB to FORTRAN 95 Compiler". MA thesis. Aug. 2009.

[5] Cleve Moler. *The Origins of MATLAB*. http://www.mathworks.com/company/newsletters/news_notes/clevescorner/dec04.html.

[6] Soroush Radpour. "UNDERSTANDING AND REFACTORING THE MATLAB LANGUAGE". MA thesis. Aug. 2012.

[7] *MATLAB 55 times slower compared to C#*. [Online; accessed 24-January-2014]. Jan. 2014. URL: http://www.mathworks.com/matlabcentral/newsreader/view_thread/259806.

[8] Wikipedia. *CPython — Wikipedia, The Free Encyclopedia*. [Online; accessed 5-February-2014]. 2014. URL: %5Curl%7Bhttp://en.wikipedia.org/w/index.php?title=CPython&oldid=588973499%7D.

[9] Koichi Sasada. “YARV: Yet Another RubyVM: Innovating the Ruby Interpreter”. In: *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA ’05. San Diego, CA, USA: ACM, 2005, pp. 158–159. ISBN: 1-59593-193-7. DOI: `10.1145/1094855.1094912`. URL: `http://doi.acm.org/10.1145/1094855.1094912`.

[10] MathWorks. *MATLAB Coder*. `http://www.mathworks.com/products/matlab-coder/`.

[11] Maxime Chevalier-Boisvert. “McVM: an Optimizing Virtual Machine for the MATLAB Programming Language”. MA thesis. Aug. 2009.

[12] Pramod G. Joisha and Prithviraj Banerjee. “Correctly Detecting Intrinsic Type Errors in Typeless Languages Such As MATLAB”. In: *Proceedings of the 2001 Conference on APL: An Arrays Odyssey*. APL ’01. New Haven, Connecticut: ACM, 2001, pp. 7–21. ISBN: 1-58113-419-3. DOI: `10.1145/570407.570408`. URL: `http://doi.acm.org/10.1145/570407.570408`.

[13] Pramod G. Joisha and Prithviraj Banerjee. “A Translator System for the MATLAB Language: Research Articles”. In: *Softw. Pract. Exper.* 37.5 (Apr. 2007), pp. 535–578. ISSN: 0038-0644. DOI: `10.1002/spe.v37:5`. URL: `http://dx.doi.org/10.1002/spe.v37:5`.

[14] Luiz De Rose and David Padua. “A MATLAB to Fortran 90 Translator and Its Effectiveness”. In: *Proceedings of the 10th International Conference on Supercomputing*. ICS ’96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 309–316. ISBN: 0-89791-803-7. DOI: `10.1145/237578.237627`. URL: `http://doi.acm.org/10.1145/237578.237627`.

[15] George Almási and David Padua. “MaJIC: Compiling MATLAB for Speed and Responsiveness”. In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. PLDI ’02. Berlin, Germany: ACM, 2002, pp. 294–303. ISBN: 1-58113-463-0. DOI: `10.1145/512529.512564`. URL: `http://doi.acm.org/10.1145/512529.512564`.

[16] Andreas Gal et al. “Trace-based Just-in-time Type Specialization for Dynamic Languages”. In: *SIGPLAN Not.* 44.6 (June 2009), pp. 465–478. ISSN: 0362-1340. DOI: 10.1145/1543135.1542528. URL: http://doi.acm.org/10.1145/1543135.1542528.

[17] Jesse Doherty. “McSAF: An Extensible Static Analysis Framework for the MATLAB Language”. MA thesis. McGill University, Dec. 2011.

[18] Wikipedia. *Component-based software engineering — Wikipedia, The Free Encyclopedia*. [Online; accessed 20-January-2014]. 2014. URL: %5Curl%7Bhttp://en.wikipedia.org/w/index.php?title=Component-based_software_engineering&oldid=590512677%7D.

[19] Manuvir Das. “Unification-based pointer analysis with directional assignments”. In: *PLDI*. 2000, pp. 35–46.

[20] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0-201-10088-6.

[21] Glenn Ammons and James R. Larus. “Improving Data-flow Analysis with Path Profiles”. In: *PLDI*. 1998, pp. 72–84.

[22] Wikipedia. *Visitor pattern — Wikipedia, The Free Encyclopedia*. [Online; accessed 24-January-2014]. 2014. URL: %5Curl%7Bhttp://en.wikipedia.org/w/index.php?title=Visitor_pattern&oldid=589484391%7D.

[23] Karel Driesen and Urs Hölzle. “The direct cost of virtual function calls in C++”. In: *ACM Sigplan Notices*. Vol. 31. 10. ACM. 1996, pp. 306–323.

[24] *The cost of dynamic (virtual calls) vs. static (CRTP) dispatch in C++*. 2013. URL: http://eli.thegreenplace.net/2013/12/05/the-cost-of-dynamic-virtual-calls-vs-static-crtp-dispatch-in-c/ (visited on 01/15/2014).

[25] James O. Coplien. “Curiously Recurring Template Patterns”. In: *C++ Rep.* 7.2 (Feb. 1995), pp. 24–27. ISSN: 1040-6042. URL: http://dl.acm.org/citation.cfm?id=229227.229229.

[26] P. Cousot and R. Cousot. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”. In: *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York, NY, 1977, pp. 238–252.

[27] Luis Damas and Robin Milner. “Principal Type-schemes for Functional Programs”. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’82. Albuquerque, New Mexico: ACM, 1982, pp. 207–212. ISBN: 0-89791-065-6. DOI: `10.1145/582153.582176`. URL: `http://doi.acm.org/10.1145/582153.582176`.

[28] StackOverflow. *StackOverflow - Implementing type inference*. [Online; accessed 24-January-2014]. 2014. URL: `%5Curl%7Bhttp://stackoverflow.com/questions/415532/implementing-type-inference%7D`.

[29] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. “ABCD: Eliminating Array Bounds Checks on Demand”. In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. PLDI ’00. Vancouver, British Columbia, Canada: ACM, 2000, pp. 321–333. ISBN: 1-58113-199-2. DOI: `10.1145/349299.349342`. URL: `http://doi.acm.org/10.1145/349299.349342`.

[30] Clark Verbrugge, Phong Co, and Laurie Hendren. “Generalized Constant Propagation A Study in C”. In: *In 6th Int. Conf. on Compiler Construction, volume 1060 of Lec. Notes in Comp. Sci*. Springer, 1996, pp. 74–90.

[31] Wikipedia. *Interpreter (computing) — Wikipedia, The Free Encyclopedia*. [Online; accessed 29-January-2014]. 2014. URL: `%5Curl%7Bhttp://en.wikipedia.org/w/index.php?title=Interpreter_(computing)&oldid=592908221%7D`.

[32] Anton Dubrau. “TAMING MATLAB”. MA thesis. Apr. 2012.

[33] *Is changing the type of a variable partway through a procedure in a dynamically typed language bad style?* 2013. URL: `http://programmers.stackexchange.com/questions/187332/is-changing-the-type-of-a-variable-partway-through-a-procedure-in-a-dynamically` (visited on 01/15/2014).

[34] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California, Mar. 2004.

[35] LLVM. *The Often Misunderstood GEP Instruction*. [Online; accessed 24-January-2014]. Jan. 2014. URL: `http://llvm.org/docs/GetElementPtr.html`.

[36] LLVM. *LLVM's Analysis and Transform Passes*. [Online; accessed 24-January-2014]. Jan. 2014. URL: `http://llvm.org/docs/Passes.html`.

[37] Project Euler. *Project Euler: Coin Sums (Problem 31)*. [Online; accessed 24-January-2014]. Jan. 2014. URL: `http://projecteuler.net/index.php?section=problems&id=31`.

[38] "Results from a Parallel MATLAB Compiler". In: *Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*. IPPS '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 81–. URL: `http://dl.acm.org/citation.cfm?id=876880.879565`.

[39] Wikipedia. *RGB color model — Wikipedia, The Free Encyclopedia*. [Online; accessed 10-February-2014]. 2014. URL: `%5Curl%7Bhttp://en.wikipedia.org/w/index.php?title=RGB_color_model&oldid=593781990%7D`.

[40] Wikipedia. *HSL and HSV — Wikipedia, The Free Encyclopedia*. [Online; accessed 10-February-2014]. 2014. URL: `%5Curl%7Bhttp://en.wikipedia.org/w/index.php?title=HSL_and_HSV&oldid=592589294%7D`.

[41] *Using Intel MKL with MATLAB*. [Online; accessed 24-January-2014]. Jan. 2014. URL: `http://software.intel.com/en-us/articles/using-intel-mkl-with-matlab`.