JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Faculty of Engineering
and Natural Sciences

TruffleClojure: A self-optimizing AST-Interpreter for Clojure

Master's Thesis

submitted in partial fulfillment of the requirements
for the academic degree

Diplom-Ingenieur

in the Master's Program

Computer Science

Submitted by
Thomas Feichtinger, BSc.

At the
Institut für Systemsoftware

Advisor
o.Univ.-Prof. Dipl.-Ing. Dr.Dr.h.c. Hanspeter Mössenböck

Co-advisor
Dipl.-Ing. Matthias Grimmer
Dipl.-Ing. Dr. Thomas Würthinger

Linz, June 2015

Abstract

Truffle is a framework for the implementation of programming language interpreters. It was successfully used for dynamic languages, but no functional language implementations existed at the time of writing. This thesis presents a Clojure interpreter, called *TruffleClojure*, implemented on top of the *Truffle* language interpreter framework to show that the framework can be used for the implementation of functional programming languages. In contrast to existing byte-code generating Clojure implementations, our interpreter is based on an abstract syntax tree (AST) which optimizes itself during execution by collecting language-specific profiling information and dynamically adapting the AST to efficiently execute the program under the observed conditions. By using the techniques provided by the Truffle framework we were able to implement an interpreter for the Clojure programming language. Since our interpreter does not compile to Java byte-code we were also able to add support for tail call optimization to the language.

Kurzfassung

Truffle ist ein Framework zur Implementierung von Interpretern. Es wurde erfolgreich verwendet um dynamische Sprachen effizient zu Implementieren. Funktionale Sprachen wurden bisher nicht implementiert. In dieser Arbeit präsentieren wir einen Interpreter für die funktionale Programmiersprache Clojure um zu demonstrieren das Truffle für funktionale Sprachen geeignet ist. Im Gegensatz zu existierenden Clojure Implementierungen basiert unser Interpreter auf einem abstrakten Syntaxbaum (AST). Der AST optimiert sich selbständig zur Laufzeit indem er Profiling-Informationen sammelt und den AST spezialisiert. Durch die Verwendung des Truffle Frameworks ist es uns gelungen einen effizienten Interpreter für Clojure zu entwickeln. Da unser Interpreter im Gegensatz zu Clojure nicht zu Java bytecode kompiliert wird, ist es möglich Techniken wie die Entfernung von Endrekursionen der Sprache hinzuzufügen.

Contents

Chapter 1

Introduction

This chapter presents TruffleClojure, an abstract syntax tree interpreter for the Clojure programming language built on top of the Truffle framework. We discuss the goals of the project and point out the challenges that were encountered.

1.1 Motivation

Clojure [13] is a dynamic and functional programming language that runs on a Java Virtual Machine (JVM) [17, 36, 37] . Programs are executed by compiling Clojure source code into Java bytecode which is subsequently executed by the JVM. Compiling guest languages to Java bytecode is a common technique to implement programming languages on top of the JVM. Examples for such language implementations include Python [22, 27], Ruby [5] or Scala [19]. Targeting virtual machines has the advantage of being able to reuse services such as automatic memory management or just-in-time (JIT) compilation. However, generating bytecode also

introduces several problems: Java bytecode was specifically designed for the Java language, hence an argument can be made that it is not well suited for languages that build on different abstractions and paradigms than the imperative and statically typed Java language. For example, Java bytecode is statically typed, which may lead to a considerable amount of type checks when executing programs of a dynamic language [32]. Furthermore, Java bytecode does not offer support for tail call optimization (TCO) which is an essential feature for the functional programming paradigm. There is a thesis proposing a solution for tail call optimization on the JVM [24], however it is not part of the JVM at the time of writing. For non-imperative languages like Java the lack of TCO is not a problem: recursive programs can be implemented by using side-effect based control structures like *for*- or *while*- loops. In most functional programming languages, however, there are no side-effects and recursion is thus the only looping construct. By recursively calling a function the call stack grows with each invocation. Without TCO programs run the risk of causing a stack overflow depending on the depth of recursion. To counter this problem, the Clojure designers introduced a special keyword which transforms tail-recursive (a function that calls itself in tail position) function calls into a loop at the bytecode level. However, this is not possible for general calls in tail position (for example mutually recursive functions) and Clojure explicitly does not offer general TCO because of the JVM's lack thereof.

In this thesis we present an implementation of an abstract syntax tree (AST) interpreter for Clojure which is not restricted by the limitations imposed on the language by compiling to Java bytecode. We base our work on Truffle [32], a framework written in Java for the development of programming language interpreters. It provides several APIs for the implementation of efficient self-optimizing AST interpreters [32]. As a language implementor one simply writes a Java based AST interpreter. Truffle then compiles the AST to native machine code using the Graal compiler [20]. The Truffle framework has been successfully used to implement dynamic languages such as JavaScript [31] and Ruby [25].

Although some of the existing Truffle language implementations contain functional features (such as R or Python), at the time of writing no primarily functional language implementation was implemented with Truffle. By basing our interpreter on Truffle instead of compiling to Java bytecode we were able to bring support for TCO to the Clojure programming language. We also show that the Truffle framework allows the implementation of functional programming languages. The Clojure programming language was identified as a potential fit and language that could profit by being implemented on top of the Truffle framework for the following reasons:

Functional:

Clojure is a functional language and as such has higher order functions. Immutability is at the core of the language: Clojure provides immutable and persistent built-in data structures. The addition of Clojure as a Truffle language should serve as confirmation that Truffle may be used as an interpreter framework for functional languages as proposed in [32].

Dynamic type system:

Clojure features a dynamic type system. The Truffle framework was designed for high performance implementations for languages with a dynamic type system. As a consequence, the Clojure programming language could also benefit from the optimizations performed by Truffle-based interpreters.

JVM language:

The default Clojure implementation is written in Java. This allows us to reuse certain parts for our interpreter, for example the built-in Clojure data structures and parts of the parser. In contrast to the default Clojure implementation, our interpreter does not generate any bytecode. This allows us to introduce concepts not supported by Java bytecode, such as TCO.

1.2 Goals, Scope and Results

The main goal of this work is to implement an AST interpreter for the functional programming language Clojure. AST interpreters are an intuitive way to implement a programming language. In contrast to the original Clojure implementation, which is executed by compiling programs to Java bytecode, we base our implementation on the Truffle-based interpretation [32]. The implementation presented in this work reuses parts from the original Clojure[1] implementation.

A secondary goal of this work is to implement the first primarily functional language on top of the Truffle framework in order to determine the suitability of Truffle for functional languages. At the time of writing the existing language implementations on Truffle were mostly dynamic languages like JavaScript [30] and Ruby [25]. Other languages include R [16] and Python [29]. Some of these languages contain functional features which are already successfully implemented under Truffle, but no primarily functional language exists.

Language completeness is defined as an explicit non-goal and intentionally left for future work. Features of Clojure that were not implemented in this prototype is concurrency support. Concurrency for Truffle languages is an active area of research. For this prototype this feature was declared out of scope even though it is an essential part of Clojure.

Our evaluation shows that the TruffleClojure interpreter can compete with the performance of the official Clojure implementation. More detailed results can be found in Chapter 8.

[1] `https://github.com/clojure/clojure`

1.3 Challenges

The lack of a formal specification of the Clojure language was a major challenge we had to overcome. The only resource is the reference implementation of Clojure. There is no clear distinction between language level feature and implementation detail. For example, one could argue that type hints are an implementation detail and not a language feature. In the official Clojure implementation type hints provide additional information for the compiler to avoid the use of reflection and generate optimized code, whereas in the Truffle implementation these hints are not necessary because the executing program gathers profiling information from which these hints can be inferred.

Adding support for tail call optimization was another challenge. We base our implementation on exceptions, which is the only way to unwind the stack in Java. In contrast to the solution presented in [24], which extended the JVM to support TCO, our implementation exists solely in the guest language interpreter and the Truffle framework together with the Graal compiler is then able to generate efficient code. Instead of executing an actual call, which would allocate a new activation frame on the call stack, we throw a special type of exception. The Truffle framework has special knowledge about this exception type and can generate optimized code. For example, the machine code produced for tail recursive calls resembles the machine code produced by a simple iterative loop. In order to preserve source code compatibility with existing Clojure code the special form *loop/recur* is still supported, although not necessary anymore: any call in tail position will be optimized.

1.4 Structure of the Thesis

In the remainder of this work we will refer to the original Clojure implementation[2] simply as *Clojure* and to our interpreter as *TruffleClojure*.

Chapter 2 explains the context of the TruffleClojure interpreter. The chapter introduces the Graal VM and the Truffle framework. It highlights and describes some of the features provided by Truffle that the TruffleClojure interpreter makes use of.

Chapter 3 briefly introduces the Clojure programming language and discusses how the language can benefit from an implementation on top of the Truffle framework.

Chapter 4 describes the architecture and dependencies of TruffleClojure. It further discusses the implementation of the parser, functions and macros.

Chapter 5 discusses the implementation of the AST nodes and describes the implementation of tail call optimization in our interpreter. Chapter 6 presents our implementation of Tail Call Optimization and in chapter 7 we discuss a small case study of an example Clojure program.

The performance of our interpreter is discussed in chapter 8 and finally the thesis concludes with future work in Chapter 9, related work in Chapter 10 and a conclusion in Chapter 11.

[2] `https://www.clojure.org`

Chapter 2

System Overview

This chapter introduces the Graal VM and the Truffle language implementation framework. It describes the features of Truffle that allow us to build highly efficient AST interpreters.

2.1 The Graal VM

The Graal virtual machine [3, 4, 20] is a modification of the Java HotSpot VM. The Graal VM extends the HotSpot VM by a new compiler (called *Graal* compiler), which aims to produce highly efficient machine code for Java. Graal takes Java bytecode as input and transforms it into the Graal Intermediate Representation (IR) [3]. The Graal IR is based on a directed graph structure and is in static single assignment form (SSA) [3]. Graal performs several optimizations on the IR and subsequently compiles it to native machine code [3]. In contrast to the client and

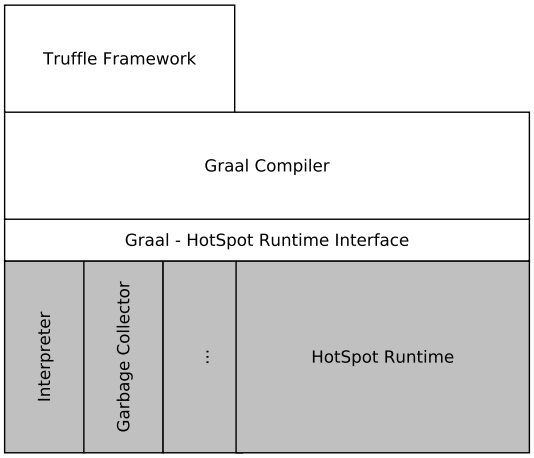


Figure 2.1: System architecture of the Graal VM including the Truffle interpreter framework.

server compilers of the HotSpot VM, the Graal compiler is written in Java. The Graal VM reuses other parts of the HotSpot VM, for example the garbage collector [20, 21].

2.2 The Truffle Framework

Truffle [32] is a multi-language interpreter framework built on top of the Graal VM. Guest languages are implemented in Java as AST interpreters, which are easier and more intuitive to design and implement than compilers [32]. During program interpretation the AST interpreter collects profiling information, e.g. dynamic type information, which can subsequently be used to speculate on certain conditions and optimize the executing program during runtime.

The Truffle framework provides several APIs that can be used to implement abstract syntax tree (AST) interpreters for a wide variety of different programming languages. Services like automatic memory management are provided by the JVM and become reusable to guest languages without any implementation effort. Figure 2.1 shows an overview of the Graal/Truffle architecture.

2.2.1 Self-Optimization

Truffle provides functionality to replace a node with another node during execution of the AST, essentially allowing the AST to optimize itself during execution. The guest language makes use of the framework to optimize the AST based on collected profiling information. Optimization happens through tree rewriting, i.e., nodes are replaced by nodes better suited to handle the observed conditions (referred to as *(node) specialization*) [32]. The basic idea of specialization is to use the most constrained and thus fastest possible implementation of a node for the conditions that hold while executing the program. This means that the actual guest-language code that is executed changes dynamically when executing a guest-language method [32]. Node rewriting is illustrated in Figure 2.2a. The AST starts in an uninitialized state. During execution nodes rewrite themselves with specialized versions based on the types encountered.

Type Specialization:

One common optimization performed by node rewriting is type specialization [32]. In dynamic languages, operators can often be applied to a variety of operand types. Thus, the implementation of an operand must be able to handle all possible cases. Such an implementation must include type checks in order to choose the appropriate action to execute, which introduces an overhead for each invocation of the operand. Listing 2.1 provides an example for

```
class AddNode {
  Node left;
  Node right;

  Object execute()  {
    Object l = left.execute();
    Object r = right.execute();
    if (l instanceof Integer && r instanceof Integer) {
      //return int addition
    } else if (l instanceof Double && r instanceof Double) {
      //return double addition
    } else if (l instanceof String && r instanceof String) {
      //return string concatenation
    }
  }
}
```

Listing 2.1: Example implementation of a + operator in a dynamic language.

such an operand. However, it is often the case that the operand types are stable for a given operand instance in the AST. By speculating on this assumption we can improve performance by only including only the code for one particular case. For example, if the operands to the *add* node depicted in example 2.1 are integers, the node specializes to the int case, shown in Listing 2.2. In case the observed types change at some point during execution of the AST, the specialized node rewrites itself into a more general version [32]. It is in the responsibility of the language implementor to guarantee completeness, i.e., the programmer must provide node specializations for *all* possible cases.

Polymorphic Inline Caches:

Another potential technique for self optimization are polymorphic inline caches [14]. Nodes may be chained to form a cache. For every new entry in the cache a new node is added. When a node is executed the first entry in the cache checks whether the entry matches. In case it does it continues the execution. If the entry does not match the condition, it delegates control to the next node

```
class AddIntIntNode {
  Node left;
  Node right;

  Object execute()  {
    int l = left.executeInt();
    int r = right.executeInt();
    return l+r;
  }
}
```

Listing 2.2: A specialized version of the generic AddNode for the *int* case.

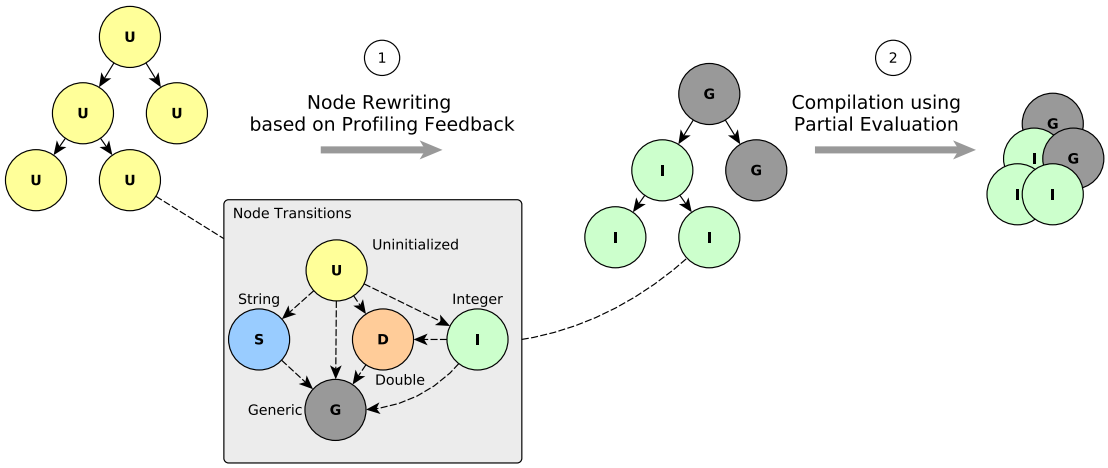
in the inline cache. If the maximum length of the cache is reached the whole chain replaces itself by a generic implementation which can handle the whole megamorphic case [32].

Resolving Operations:

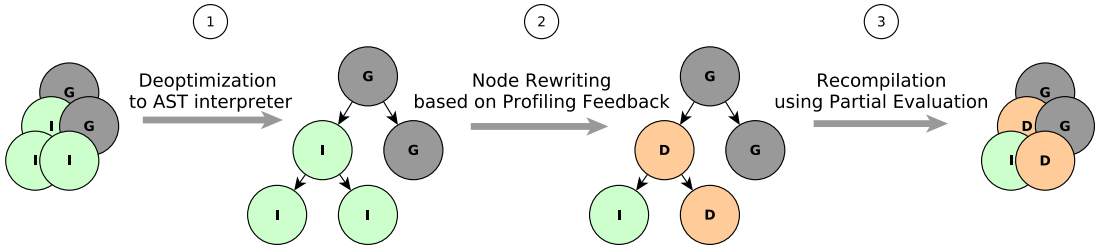
Certain operations may include a lookup step (e.g. method resolving). If the result of the lookup phase is constant, it is beneficial to only perform the lookup once and cache the result. This can be achieved by node rewriting: Executing a node for the first time includes the lookup phase; the node then rewrites itself to a version with the resolved operation. Instead of performing the lookup the next time, the resolved version is directly used [32].

2.2.2 Partial Evaluation

When a tree reaches a stable state it is subject to partial evaluation [7]. We call a tree stable when no specializations occur anymore. During partial evaluation, Graal inlines all execute methods of the AST nodes, forming one combined compilation unit for an entire tree. This eliminates the need for dynamic dispatch between individual nodes and speeds up execution [32]. The partial evaluation step is indicated in Figure 2.2.



(a) Based on profiling feedback nodes may rewrite themselves to use the most efficient implementation for the observed conditions (1). Once the tree reaches a stable state it is compiled to optimized machine code by using partial evaluation (2).



(b) If assumptions are invalidated the tree is deoptimized and control is transferred from machine code back to the interpreter (1). Nodes will again specialize themselves based on the observed conditions (2) and once a stable state is again reached (and the code section is hot) the tree is compiled again (3).

Figure 2.2: Self-optimization by node rewriting and de-optimization [32].

In case the stable tree assumption turns out to be false and a node replacement would be necessary after compilation, the compiled code is deoptimized and execution transfers back to the interpreter. The tree can rewrite itself again to adapt to the encountered conditions and another compilation using partial evaluation may be triggered once it reaches a stable state again.

2.2.3 The Truffle API

The Truffle API provides an abstract base class `Node` which all custom nodes must extend. This class handles various common tasks such as node replacement through the `replace` method. Every non-abstract node must implement an `execute` method. A nodes child nodes must be annotated with the `@Node` annotation, or `@Children` if the field is an array of sub-nodes. These annotation are used by the framework to identify direct children of a node [30].

The `Frame` is the main data structure to store the transient state of a function activation [30]. The state contains local variables, the arguments passed to the function and possibly intermediate results. In Truffle, a `Frame` object is created upon function entry and is passed through the entire tree of the function as an argument. A `FrameDescriptor` describes the contents of a single frame. A frame's data is either stored in an `Object[]` array or, for primitive types, in an `long[]` array. Note that all primitive types are mapped to long values. For example, `Double` values are converted using `Double.doubleToRawLongBits()` and `Double.longToDoubleBits()` methods [30].

2.2.4 Truffle Object Storage Model

The Truffle Object Storage Model (OSM) [31] is a generic, language-agnostic and portable object model provided by the Truffle framework. The OSM is designed to benefit from the optimizing compiler in the Truffle framework [31]. For example, the actual types of properties may specialize themselves during runtime based on the actual content that is assigned to them. This allows dynamic languages to exploit the performance of Java *primitive* types instead of being forced to use the more generic *Object* type.

2.2.5 Truffle DSL

The Truffle DSL [15] is an annotation based domain specific language that aims to reduce the amount of code a language implementer has to write when implementing a Truffle AST interpreter. When writing node specialization without this DSL one needs to implement the rewrite logic by hand. This is not only error prone but also highly repetitive. The Truffle DSL allows the programmer to concentrate on the business logic of the specializations and will generate the rewriting logic.

For example, the Truffle DSL was successfully used in the TruffleJS implementation reducing the code size considerably [15]. Listing 2.3 shows an example of a node implemented with the Truffle DSL. Different versions of a particular node (*AddNode* in this example) are annotated with the `@Specialization` annotation. The DSL will generate all specified nodes and also the rewrite logic which contains the transitions from one specialization to another one. Produced nodes in this particular example include an *AddNode* whose operands are of type *int*, one whose arguments are doubles and a generic version which deals with all kind of types.

```
class AddNode extends Node {

  @Specialization
  int addIntInt(int a, int b) {
    return Math.addExact(a, b);
  }

  @Specialization
  double addDoubleDouble(double a, double b) {
    return a + b;
  }

  @Specialization
  Object addObjectObject(Frame f, Object a, Object b) {
    // Handling of String omitted for simplicity.
    Number aNum = toNumber(f, a);
    Number bNum = toNumber(f, b);
    return Double.valueOf(aNum.doubleValue() + bNum.doubleValue());
  }

  //... remaining code omitted
}
```

Listing 2.3: Example node implementation using the Truffle DSL for specializations.

Chapter 3

Clojure on Truffle

This chapter introduces the Clojure programming language and its features and discusses possible implementation strategies in regard to the Truffle-based interpreter which is described in Chapter 4. Furthermore, benefits and drawbacks of using Truffle as platform for Clojure are discussed.

Clojure is a general-purpose programming language originally developed by Rich Hickey [13, 33]. Among Common Lisp and Scheme, Clojure is nowadays one of the most widely known general-purpose dialects of Lisp [28]. The original Lisp (which stands for **LIS**t Processor) language was designed by John McCarthy and appeared in 1958 [18]. This makes it one the oldest general-purpose languages which are still in use today [1, 18].

No formal specification of the Clojure language itself exists. The official implementation is open source and can be found on GitHub [1]. The official documentation [33] and an overview of all features of Clojure can be found on the project's homepage[2].

Since the original Clojure implementation was released, many derivatives of the language appeared. One popular such derivative is called *ClojureScript* [35], an implementation that targets *JavaScript* as the host platform instead of the *JVM*. *ClojureScript* compiles programs to *JavaScript* which is intended to be executed in a browser. It is not fully compatible to the original implementation and *Clojure* libraries cannot be reused without fulfilling the assumptions by *ClojureScript* [3]. Other implementations of Clojure target other runtimes as host platform, for example the CLR [34]. We base our implementation on the Truffle framework [32].

3.1 Types

Clojure supports the following types:

- `boolean` – Represented by default Java `booleans`. In Clojure both `nil` and `false` are treated as `false` and everything else is treated as `true` [6].
- `String` – Represented by regular Java Strings, i.e., `java.lang.String`.
- Numbers
  - JVM primitives (`int`, `long`, `double`, ...).

[1] `https://github.com/clojure/clojure`
[2] `https://www.clojure.org`
[3] `https://github.com/clojure/clojurescript/wiki/Differences-from-Clojure`

  - `clojure.lang.Ratio`: represents a ratio between integers. Divisions that cannot be reduced to integer values return a ration instead of a floating point value. For the TruffleClojure prototype the `Ratio.java` implementation from the Clojure project is reused.

  - `java.lang.BigInteger` and `java.lang.BigDecimal`

- `Symbol` – Symbols are used to refer to arbitrary values. A Symbol has a name and an optional namespace. If a namespace is supplied the symbol is called *namespace-qualified*. Symbols can refer to local bindings, function parameters or to values in a namespace. Only non-namespace-qualified symbols can be used for the first two cases. Symbols of the form `foo` are not namespace-qualified whereas symbols of the form `a-namespace/foo` are namespace-qualified. Symbols provide fast equality comparisons and can have meta-data attached [33].

- `Keyword` – Keywords evaluate to themselves and provide fast equality comparisons. Furthermore, they can be used as keys into maps. Keywords always start with a colon [33].

3.2 Vars and Namespaces

Namespaces maintain the mappings between simple *Symbols* and *Vars*. A *Var* is basically an object that keeps and manages a reference to some value and are used in Clojure to maintain state. There are other objects in Clojure that can be used to manage state with different semantics (e.g. *agents*), but were not implemented in TruffleClojure. During parsing of an expression all *Vars* are resolved, minimizing

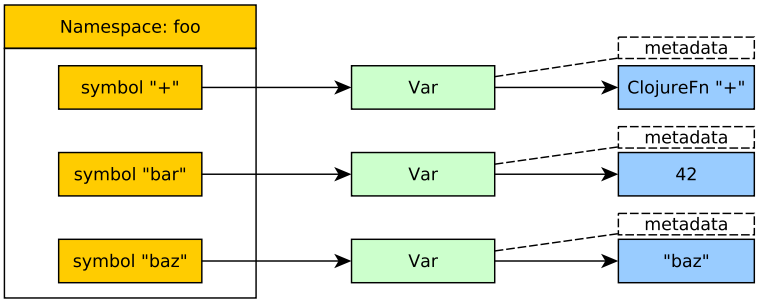


Figure 3.1: The relationship between *Symbols, Vars and Namespaces*: A *Namespace* defines bindings between *Symbols* and *Vars* which may reference any value. *Vars* may have meta-data, i.e., a map with *:keyword*/value entries) attached.

the lookup cost. See Figure 3.1 for a simple illustration of the relationships between *Namespaces*, *Symbols* and *Vars*. Listing 3.1 shows code snippets to explain Namespaces, Vars and Symbols.

3.3 Persistent and Immutable Datastructures

Clojure distinguishes between four built-in data structures: Lists, Vectors, Maps and Sets. All data structures are immutable and persistent. Immutable data structures cannot change their internal state. One advantage of immutable objects is that they are implicitly thread-safe, since no modification can occur. Goetz et al. furthermore argue that immutable objects are easier to understand and reason about [8]. Persistent data structures preserve the previous version when they are modified. Modifications do not alter the internal state but instead yield a new updated structure. In order to efficiently achieve persistency, Clojure's data structures are based on linked data structures.

```
(ns foo) ;set current namespace to 'foo'

foo=> (type 'a) ;' will prevent a form from being evaluated
clojure.lang.Symbol

foo=> (type 'b)
clojure.lang.Symbol

foo=> (def a 1) ;bind value 1 to symbol a
#'foo/a ;returns var in namespace foo named a

foo=> a ;evaluate symbol a
1

foo=> (def b "a string")
#'foo/b

foo=> b
"a string"

foo=> (. #'foo/a ns)
#<Namespace foo>   ;the namespace symbol a is associated with
```

Listing 3.1: Namespaces, Vars and Symbols in Clojure explained.

Lists:

Lists in Clojure are implemented as linked lists. Appending to the front of the list is a constant time operation ($O(1)$). The length of a list is cached in each node ($O(1)$). Indexing requires to iterate the contents of the lis ($O(n)$).

Vectors:

A vector is a collection of values that can be accessed by an index. Vectors are created by either a call to the `vector` function or by using the vector literal form. Vectors are based on Ideal Hash Trees invented by Phil Bagwell [2]. The trees used for implementing have a branching factor of 32, hence insertions and indexed access in vectors have a $O(log_{32}n)$ runtime complexity. The branching factor of 32 guarantees that the trees stay relatively flat.

	map	set	vector	list
conj	$O(log_{32}n)$	$O(log_{32}n)$	$O(1)$ *(tail)*	$O(1)$ *(head)*
assoc	$O(log_{32}n)$	-	$O(log_{32}n)$	-
dissoc	$O(log_{32}n)$	-	-	-
disj	-	$O(log_{32}n)$	-	-
nth	-	-	$O(log_{32}n)$	$O(n)$
get	$O(log_{32}n)$	$O(log_{32}n)$	$O(log_{32}n)$	-
pop	-	-	$O(1)$ *(tail)*	$O(1)$ *(head)*
peek	-	-	$O(1)$ *(tail)*	$O(1)$ *(head)*
count	$O(1)$	$O(1)$	$O(1)$	$O(1)$

Table 3.1: Runtime complexities of common operations on Clojure's data structures [6, 28, 33].

Maps:

Maps represent Clojure's associative data structure. Like lists and vectors they are also immutable and persistent. The implementation is also based on hash array mapped tries (HAMTs). The branching factor of a single node is 32, similar to that of the vector implementation.

Sets:

Clojure's immutable and persistent set implementation based on HAMTs.

Although it would be possible to implement, for example, vectors by using simple arrays it would be inefficient to achieve persistency and immutability. If an element were added to a vector, the backing array's state needs to either be mutated (thus violating the immutability property) or copied before performing the change (high overhead, an otherwise constant time operation would now be linear).

The internal data structure used to implement Clojure's datastructures are *Tries* (the term trie comes from re**trie**val). Tries are tree data structures used to store a dynamic set or associative arrays. Values are kept in leaf nodes and the path from root to leaf is used to identify the entry. Clojure's implementation is based on Phil Bagwell's Ideal Hash Trees [2]. An overview of the runtime complexities of

common operations on Clojure's datastructures is given in Table 3.1. The remainder of this section illustrates the update and insert operations on a Clojure vector as an example of how these datastructures work.

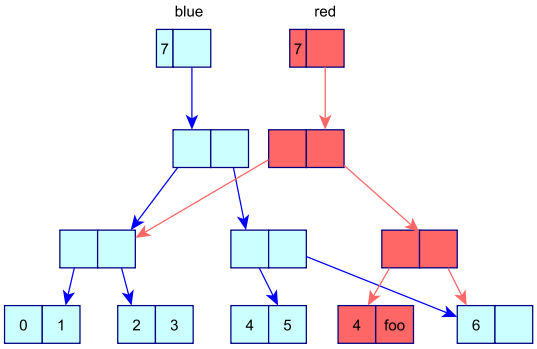
Updates:

Figure 3.2a shows how persistent updates are achieved in Clojure's vectors. The *blue* vector contains elements $0, 1, 2, 3, 4, 5, 6$. By replacing the value 5 with *foo* we get a new *red* vector. The old *blue* version is still intact after the modification. Note how only the path from the root node to the individual elements is copied. Furthermore, some structure of the path is shared between the two versions. If vectors were backed by simple mutable arrays we would need to copy the whole array in order to achieve persistency.

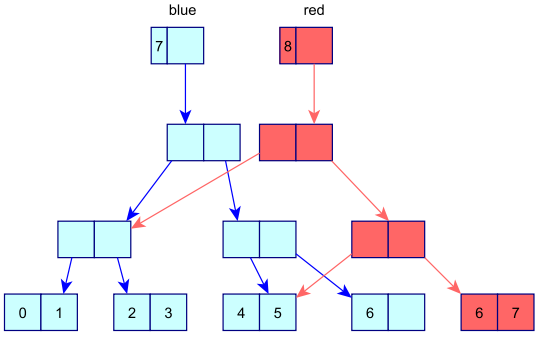
Inserts:

Insertion works similarly as shown in Figure 3.2b. In this example the *blue* vector contains the elements $0, 1, 2, 3, 4, 5, 6$ and we insert value 7. Note that the root node will keep track of the current size, so the `length` operation is constant in time. In the example there is still room left in the node so we simply copy that node and insert the value. Again, we are mostly only copying the path. If there is no room left we generate a new node. In bigger vectors most of the path will be unchanged and share structure with the old version.

In the examples we used a tree with a branching factor of two, i.e., every node in the tree has two child nodes. In reality, Clojure uses trees with a branching factor of 32. As a consequence all trees stay very shallow. For example, a tree with less than one billion elements will be at most 6 levels deep: before performance becomes a problem, the system will run out of memory.



(a) Updating the contents of a vector: the value 5 is replaced by *foo*. The *blue* vector is unchanged after the operation (preserving the *persistency* property). Note that merely the paths and one leaf need to be copied. Both vectors share structure, reducing the memory impact.



(b) Insert operation in a persistent vector. The paths need to be copied. The starting vector is unmodified and structure between the two vectors is shared to reduce memory impact.

Figure 3.2: Update and insert operations on a persistent and immutable vector.

3.4 Truffle as Platform

Clojure is a hosted language, which means that it is designed to run on a platform like the JVM or the CLR. Our interpreter is also hosted on the JVM, but instead of compiling Clojure programs to Java bytecode, we execute the programs as a Truffle-based AST interpreter.

3.4.1 Benefits

The Clojure language benefits from an implementation with Truffle in the following ways:

Type Hints:

Type hints are metadata tags placed on symbols or expressions that are consumed by the Clojure compiler. They are used to avoid the use reflection when performing Java interop calls. Listing 3.2 shows Clojure source code annotated with type hints: there are two version of the same method, with and without type hints. The execution time by using type hints increases by a factor of 10. The `len` method invokes the function *length* on its argument. Without type hints, the Clojure compiler can not know the target ahead of time and thus generates code that resolves the method to be invoked via reflection at runtime. The `len2` method, on the other hand, is annotated with a type hint indicating to the compiler that the argument passed is always of type *String*. In this case, the compiler will emit the bytecode *invokevirtual*, no reflection is used at all. However, the method no longer accepts arguments of a different type. If we were to pass an argument other than of type String to the `len2` method, the program fails at runtime with a *ClassCastException*.

```
1   (defn len [x]
2     (.length x))
3
4   (defn len2 [^String x]
5     (.length x))
6
7   user=> (time (reduce + (map len (repeat 1000000 "asdf"))))
8   "Elapsed time: 3007.198 msecs"
9   4000000
10  user=> (time (reduce + (map len2 (repeat 1000000 "asdf"))))
11  "Elapsed time: 308.045 msecs"
12  4000000
```

Listing 3.2: An example of type hints in Clojure.

Type hints must be added manually by the programmer. With Truffle however, we are able to optimize such calls automatically by caching the result of the method lookup.

Tail Call Optimization:

Clojure cannot perform general TCO because of the lack of support for TCO by the JVM. The TruffleClojure implementation is not bound to these restrictions because we do not compile to Java bytecode. We were able to efficiently implement tail call optimization by throwing exceptions (for a more detailed description of TCO see Chapter 6).

Multi-Language Interoperability:

Clojure has excellent support for Java interoperability due to the fact that it is executeed on top of the JVM. The same is possible with our TruffleClojure implementation. However, thanks to Truffle, we are able to enhance interoperability [11] to other Truffle languages like C [12], Ruby [25], Python [29], JavaScript [30] and R [16].

3.4.2 Drawbacks

Class generation:

Clojure generates Java bytecode for programs before it executes them. These classes can also be packaged into *jars* which can subsequently be used in any other Java project. By targeting Truffle, we are not able to support this feature.

Ahead-Of-Time (AOT) compilation:

By targeting Truffle as compilation backend it is no longer possible to pre-compile Clojure programs. In Clojure, one can set metadata that instructs the compiler to emit the generated class files, which can then be shipped pre-compiled. Again, since we do not generate any byte-code this is not possible anymore.

Core libraries:

The lack of a formal specification of the Clojure language makes it hard to distinguish between implementation detail and language feature. For example, the documentation of *defrecord* states that "[...]. Dynamically generates compiled bytecode for class with the given name [...]"[4]. In our interpreter however, we do not generate bytecode. Instead, we use the object model provided by the Truffle framework to implement Clojure records. It is thus not (fully) possible to reuse the implementation of the *core* libraries.

[4] `https://clojuredocs.org/clojure.core/defrecord`

Chapter 4

Interpreter Architecture

This chapter presents the architecture of TruffleClojure. It describes the different modules of the interpreter and the dependencies to other projects. Furthermore, the reused components of the official Clojure implementation are outlined.

4.1 Overview

Figure 4.1 illustrates the control and data flow during the execution of a Clojure program. There are two possibilities to execute a program: either by putting the source code into a file (of multiple files) and running it as a script or by using a REPL (**r**ead-**e**val-**p**rint-**l**oop) console where the user can enter expressions dynamically. The *Main* class instructs the *ClojureParser* to load an entire source file or a single expression (1). The *ClojureParser* then reads the next s-expression from the input and invokes the *LispReader* (2), which transforms the s-expression into a set of Clojure datastructures (i.e., lists, maps etc). Parsing is then performed on that

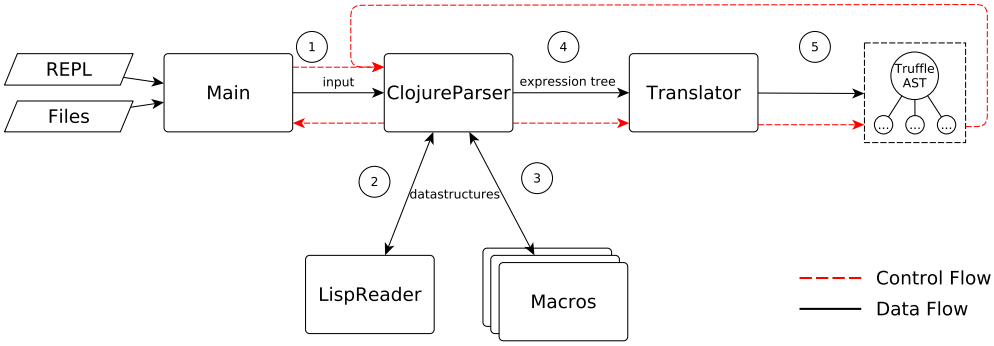


Figure 4.1: Program execution flow.

datastructure. Macro calls are expanded during this stage (3). Macros are Clojure functions that extend the functionality of the parser. The result of the *ClojureParser* is an *expression-tree* (reused from the Clojure implementation), which serves as input to the *Translator* component (4). The *Translator* transform this expression tree into an executable *Truffle AST* (5). After executing the AST, control is transferred back to the parser which continues with parsing the next expression. After handling all expressions control is trasferred back to the Main class which may continue executing the *-main* method, if one was defined.

4.2 Parser

In Clojure, the parser is part of the runtime environment. In contrast to most imperative languages, Clojure parses expressions sequentially and immediately executes them before continuing parsing the next expression. In Java, for example, the whole program is parsed and compiled to Java bytecode before it is executed. The parser has full access to the global namespaces of the running environment. Declared

macros can immediately be used by the parser in the next expression. Clojure gains a lot of its expressiveness from the macro system, which can be used to extend the language (i.e., the parser) with new control structures.

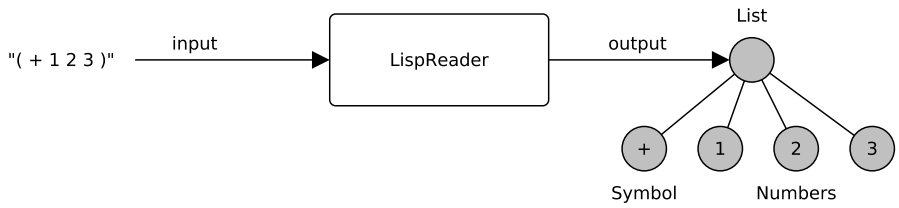
TruffeClojure reuses the Clojure parser with some modifications. We call the AST produced by the original Clojure implemenation the Clojure AST, and the AST produced by our interpreter the Truffle AST. Figure 4.2 illustrates the parsing process. The individual steps of this process are:

1. The *LispReader* accepts a stream of characters as input and transforms it into Clojure datastructures, as shown in Figure 4.2a. The TruffleClojure prototype uses the *LispReader* implementation from the official Clojure project. No modifications were necessary.

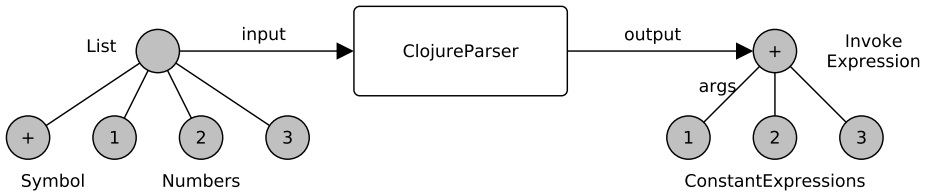
2. The *ClojureParser* component accepts Clojure data structures as input (produced by the *LispReader*) and produces the Clojure AST, as shown in Figure 4.2b. Macros are expanded during this stage. After expanding a macro, parsing continues on the expanded data structure. The end result of parsing is a Clojure AST.

   In the official Clojure project this part is done by the `Compiler` class. We extracted the parsing logic from this class and reuse it for our ClojureParser. The bytecode generation was removed.

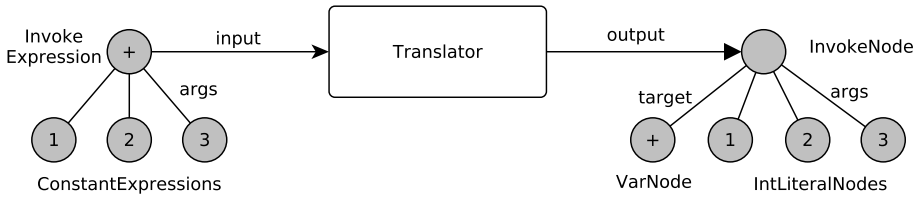
3. As a last step, the *Translator* transforms the Clojure AST into our Truffle AST by visiting each node of the Clojure AST, as shown in Figure 4.2c.



(a) The *LispReader* accepts a stream of characters as input and produces Clojure data structures as output.



(b) The *ClojureParser* accepts Clojure datastructures as input and builds the Clojure AST. Macros are expanded during this stage.



(c) The *Translator* accepts a Clojure AST produced by the *ClojureParser* and transform it into the Truffle AST which is subsequently executed.

Figure 4.2: Parsing Clojure programs.

4.3 Functions

Clojure functions are first-class elements. This means that they can be used like any other value: Functions can be passed as arguments to other functions (higher-order functions), functions can be returned from functions and assigned to global or local state. In Clojure, a single function can have multiple implementations which differ in the amount of arguments it accepts (arity). A single implementation is referred to as function-method. A maximum of 20 function-methods can be defined per function.

4.3.1 Invocation

In Clojure a function call is a list whose first element resolves to a function. The remaining elements of the list are the arguments passed to it. Before the arguments are passed to the function they are eagerly evaluated. Clojure only looks at the number of arguments provided in a function call to find the correct overloaded function to invoke. Functions may also define a *variadic* overload, which accepts an arbitrary amount of arguments.

In TruffleClojure each overloaded method implementation is represented by a Truffle AST and can be invoked by obtaining a call target to a root node of the tree. The corresponding `ClojureFn` object keeps an array of references to the call targets to these methods. Retrieving the correct call target happens when the function is called, which is trivial because at that point the amount of arguments passed to the function are known. In case no target exists for the given amount of parameters the variadic target is used. The excess parameters are wrapped into a sequence object and the variadic overload is then invoked.

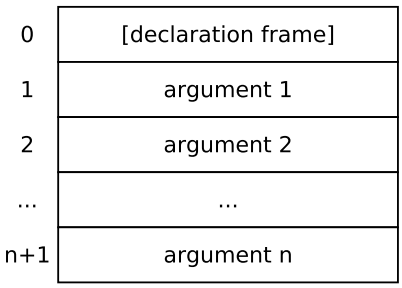


Figure 4.3: The contents of the arguments array passed to a function invocation. At index 0 the materialized outer frame is passed, indexes 1 to $n+1$ contain the actual arguments.

4.3.2 Arguments

Arguments to functions are passed as an `Object[]` array. The array contains the arguments and, optionally, a reference to the outer frame in case the function needs access to its lexical context. The class *ClojureArguments.java* contains several helper methods to access the arguments array. Figure 4.3 shows the contents of the arguments array passed to function invocations.

4.3.3 Closures

Nested functions form a closure around their declaring scope. In TruffleClojure this is achieved by keeping a reference to the outer frame of a function. The example presented in Listing 4.1 shows how closures are formed. Upon activation of the function assigned to *multiplier* a new function objects is created which keeps a reference to the materialized frame of *multiplier*. When *multiplier* returns this frame and its contents will live on in the newly created function object. When the *inner_fn* is invoked later the saved frame is passed as additional argument. At line 7 the value in slot *mult* is 2, while at line 8 the value of *mult* is 3.

```
;;returns a function which accepts one argument and multiplies it
    with 'mult'
(def multiplier
  (fn outer_fn [mult]
    (fn inner_fn [arg1]
      (* mult arg1))))

(println ((multiplier 2) 2)) ; 4
(println ((multiplier 3) 2)) ; 6
```

Listing 4.1: Closures

4.4 Macros

Macros are Clojure functions. The difference between macros and functions is the time when they get evaluated: macros are evaluated during *parse-time* (i.e., while expressions are parsed by the *ClojureParser*) while functions are evaluated during *run-time*. With macros, a programmer can essentially extend the language without needing to modify the language itself.

Macros take as input a Clojure expressions and produce transformed Clojure expressions. Evaluating macros is refered to as *macro expansion*. This is possible due to the fact that Clojure programs are expressed in terms of Clojure data structures. In contrast to functions, macro are applied during the parsing phase and can be used to extend the expresiveness of the language. It is important to note that for function calls arguments are eagerly evaluated, whereas the arguments to macro calls are not evaluated.

Listing 4.2 shows an example of a built-in macro in Clojure. The `when` macro executes multiple expressions if and only if its condition expression evaluates to true. We can expand macros in the REPL and see which result it yields by invoking the `macroexpand` method, as shown in Listing 4.2. Note that it is not possible to

```
; expanding the macro
(macroexpand '(when boolean-expression
               expression-1
               expression-2
               expression-3))

; yields
(if boolean-expression
  (do expression-1
      expression-2
      expression-3))
```

Listing 4.2: A simple macro example.

implement this behaviour as a function. Clojure evaluates expressions eagerly before passing them as arguments which could lead to exceptions when, for example, one of the expressions contains a division by 0.

4.5 Builtins

In TruffleClojure we use so called *builtins* to implement some of the core library methods. A builtin is a special node in the AST. We extended the Parser with a new type of expression, namely *builtin*. The parser will instantiate the corresponding AST node whenever it encounters a builtin special form. We use builtins to implement core methods that benefit from the node specialization feature of Truffle. For example, consider the core method *contains?* in Listing 4.3. Note the call to the `contains` method defined in class `clojure.lang.RT`, the implementation of which is shown in Listing 4.4. As we can see, the method contains a multitude of type checks and selects the correct action based on the types encountered. These checks will be executed on every invocation of the `contains?` function. By creating Truffle nodes with specializations corresponding to the type checks we can pick the correct implementation automatically (through node specialization) and avoid the superfluous code.

```
(defn contains?
  {:added "1.0" :static true}
  [coll key] (. clojure.lang.RT (contains coll key)))
```

Listing 4.3: The core method 'contains?'

```
  static public Object contains(Object coll, Object key) {
        if (coll == null)
            return F;
        else if (coll instanceof Associative)
            return ((Associative) coll).containsKey(key) ? T : F;
        else if (coll instanceof IPersistentSet)
            return ((IPersistentSet) coll).contains(key) ? T : F;
        else if (coll instanceof Map) {
            Map m = (Map) coll;
            return m.containsKey(key) ? T : F;
        } else if (coll instanceof Set) {
            Set s = (Set) coll;
            return s.contains(key) ? T : F;
        } else if (key instanceof Number && (coll instanceof String
            || coll.getClass().isArray())) {
            int n = ((Number) key).intValue();
            return n >= 0 && n < count(coll);
        }
        throw new IllegalArgumentException("contains? not supported
            on type: " + coll.getClass().getName());
    }
```

Listing 4.4: RT.java implementation of 'contains?'

4.6 Core Library

Clojure's standard library provides general-purpose functions which are necessary to build most Clojure programs. The functions are defined in the `clojure.*` namespace. The biggest such library is distributed in the `clojure.core` namespace. It contains methods for basic arithmetic operations, comparisons, bitwise operations, type casts, instance tests and others. For *TruffleClojure*, we reused most of the function implementations. Some selected functions, however, were implemented as builtins in order to benefit from node specialization.

The original implementation of Clojure is done in Java as well as Clojure itself. Some methods in the core libraries will generate bytecode which is then executed. We removed these methods from the core libraries for our interpreter since we do not target Java bytecode. Some notable examples are the instantiation of *Multimethods* or *Protocols* and *Records*. We replace these methods by builtins.

Chapter 5

Interpreter Implementation

This chapter contains a detailed description of the nodes implementing our Clojure interpreter. Note that the list of nodes is not complete.

In the following description words enclosed in angular brackets (< and >) denote arbitrary expressions. Table 5.1 shows the meaning of the operators used to describe the grammar of the core language constructs.

operator	meaning
*	0...n times
+	1...n times
?	0 or 1 times

Table 5.1: Operators used for describing the grammar of the core language constructs.

5.1 Special Forms

Clojure has a small set of core language constructs which are referred to as *special forms*[1] [6, 33]. All Clojure programs are translated (via macro expansion) into expressions of these special forms before being executed. The following section describes the grammar and implementation of most special forms. The description of the special forms is based on Clojure v1.4. The remainder of this section describes the semantics of the special forms and how we implemented them in TruffleClojure.

Def:

```
(def <name> <init>?)
```

Listing 5.1: Grammar of the *def* special form.

The `DefineNode` consists of a final reference to a `Var` and a child expression. When executed the child expression is executed and its returning value is assigned to the var. Defining something is considered a rare operation and in most Clojure programs is mostly done to set up global state and define functions. It is recommended to avoid the use of *def* and thus limit global state and complexity [6].

If:

```
    (if <test> <then> <else>?)
```

Listing 5.2: Grammar of the *if* special form.

[1] `http://clojure.org/special_forms`

```
public class IfNode extends ClojureNode {
    @Child private ClojureNode testNode;
    @Child private ClojureNode thenNode;
    @Child private ClojureNode elseNode;

    private final ConditionProfile condition = ConditionProfile
      .createCountingProfile();

    //remaining code omitted

    @Override
    public Object execute(VirtualFrame frame) {
        if (condition.profile(evaluateCondition(frame))) {
            return thenNode.execute(frame);
        } else {
            if (elseNode != null) {
                return elseNode.execute(frame);
            }
        }
        return CLNil.SINGLETON;
    }
}
```

Listing 5.3: *IfNode* implementation with condition profiling.

The `if` special form is the only conditional expression in Clojure. It can be directly matched to a Java *if* expression. The alternative branch is optional.

The `IfNode` has three subnodes representing a condition expression, a consequent expression and an optional alternative expression. In Clojure everything is *true* except *nil* and *false*. If the condition evaluates to true then the consequent expression is executed and its result will be returned. If the condition evaluates to something that is *false*, then the alternative branch is executed and returned. In case the condition evaluates to *false* and no alternative expression exists the value *nil* will be returned instead. Furthermore, the `IfNode` collects profiling information on the condition. This allows the compiler to generate better branch predictions. Listing 5.3 shows the implementation of the `IfNode`. The *ConditionProfile* collects runtime information about the condition of the if statement. This information is used by the Graal compiler to produce optimized machine code.

Fn:

```
(fn <name>? [<params>* ] <exprs>*)
(fn <name>? ([<params>* ] <exprs>*)+)
```

Listing 5.4: Grammar of the *fn* special form.

The `fn` special form defines a function. In Clojure, functions can be overloaded by arity ranging from 0 to 20. Furthermore, one overload can be variadic, which means that this implementation accepts a variable amount of arguments. To declare a variadic argument it must be preceded by an ampersand. The result of the *fn* expression is a single *ClojureFn* object which contains multiple call targets for all implementations provided.

Do:

```
(do <exprs>*)
```

Listing 5.5: Grammar of the *do* special form.

The `DoNode` can be used to execute multiple expressions in sequential order. The value of the last expression will be returned as a result of this node. Therefore, it only makes sense to put expressions with side effects in front of the last expression. Some other special forms will implicitly include a `DoNode`, for example the `Let` form.

A `DoNode` consists of an array of child nodes. When executing a `DoNode`, the children are evaluated sequentially and only the value of the last expression will be returned. If no expressions are supplied then `nil` will be returned.

```
1    (let [x 1]              ; slot x_1
2        (let [x 2]          ; slot x_2
3            (println x))    ; slot x_2
4        (println x))        ; slot x_1
```

Listing 5.7: Local bindings are hidden by nested declarations (shadowing).

Let:

```
(let [<bindings>* ] <exprs>*)
```

Listing 5.6: Grammar of *let* special form.

The let special form evaluates the expression in a lexical context where the symbols in the binding forms are bound to their respective init expressions. The bindings are sequential, which means that bindings can refer to other bindings that appear before them. Consider the example in Listing 5.7. Two unique frameslots are created for both definitions of x. Internally, these slots are name `x_1` and `x_2`, respectively. The parser will create the correct `ReadLocalSymbolNode` instances to access the correct binding.

In TruffleClojure the values of let expressions are assigned to the Truffle frame in order to allow type specialization. Once initialized the bindings are immutable. The body expression is evaluated after all init expressions were evaluated.

Let expressions can be nested and, as a consequence, bindings can be shadowed. In TruffleClojure, shadowing is achieved by creating a unique frame slot for each binding. New frames are only allocated for functions, not let expressions. The accessing nodes that read the values of bindings from the frame are instantiated during the parsing stage.

Loop and Recur:

```
(loop [<bindings>* ] <exprs>*)
```

Listing 5.8: Grammar for *loop* special form.

```
(recur exprs*)
```

Listing 5.9: Grammar for *recur* special form.

The *loop/recur* construct is the only non stack consuming loop construct in Clojure [33]. A loop defines a recursion point for recur expressions, i.e. after evaluating the recur expression control flow jumps to the first expression of the loop body. In Clojure, the *loop/recur* construct is compiled to a Java loop. We implement this by throwing an exception which can be optimized by the Truffle framework.

The *LoopNode* is an extension of the *LetNode* and as such can be used to establish local bindings. In addition, it also serves as recursion point for a recur expression. The implementation of the *LoopNode* is shown in Listing 5.10. The node catches *RecurException*s in an endless loop and invokes the body expression as long as a *RecurException* is thrown. A corresponding *RecurNode* throws these exception when being evaluated. Before throwing the exception, the *RecurNode* updates the local bindings.

The recur special form must always be in a tail position, otherwise an exception is thrown. Recur evaluates its child expression and rebinds the bindings defined in the loop node. An example of *loop/recur* is shown in Listing 5.11. The loop expression defines two local bindings called *cnt* and *acc*. The values are initialized with n and 1, respectively. As long as the *cnt* binding is greater

```
public abstract class ClojureLoopNode extends LetNode {
  private final BranchProfile recurTaken = BranchProfile.create();

  //omitted

  public Object execute(VirtualFrame frame) {
    evalBindings(frame);
    Object result = null;
    boolean recur = true;
    while (recur) {
      try {
        result = evalBody(frame);
        recur = false;
      } catch (RecurException recurException) {
        recurTaken.enter();
        recur = true;
      }
    }
    assert result !itemize= null;
    return result;
  }
}
```

Listing 5.10: *LoopNode* implementation.

than zero, the else branch with the recur expression is executed. Recur will update the values of the local bindings defined by the loop node, i.e. *cnt* will receive the value of the expression `(dec cnt)` and acc will receive the value `(* acc cnt)`. It is important to note that the recur expression must be in tail position, otherwise an error is throw during parsing of the expression.

New:

```
(new Classname args*)
(Classname. args*)
```

Listing 5.13: Grammar of the *new* special form.

```
(def factorial
  (fn [n]
    (loop [cnt n acc 1]
      (if (zero? cnt)
        acc
        (recur

          ;; cnt will take the value (dec cnt)
          ;; acc will take the value (* acc cnt)

          (dec cnt)
          (* acc cnt))))))
```

Listing 5.11: Loop/Recur example

The *new* special form can be used to instantiate arbitrary Java objects. It must be followed by a class name which is resolved during parsing.

The *JavaNewNode* keeps a reference to the already resolved constructor and Java class and an array of child expression which represent the arguments to the constructor. After evaluating the arguments the object is then instantiated by using reflection.

Dot:

```
(. <instance-expr> <member-symbol>)
(. <Classname-symbol> <member-symbol>)
(. <instance-expr> (<method-symbol> <args>*))
(. <instance-expr> <method-symbol> <args>*)
(. <Classname-symbol> (<method-symbol> <args>*))
(. <Classname-symbol> <method-symbol> <args>*)
```

Listing 5.14: Grammar of the *dot* special form.

```
1  public class RecurNode extends ClojureNode {
2    @Children private final ClojureNode[] valueNodes;
3    @Children private final ClojureNode[] bindings;
4
5    //constructor omitted
6
7    @Override
8    public Object execute(VirtualFrame frame) {
9        // first we execute the value nodes, such that intermediate
             results do not influence
10       // expression yet to be evaluated.
11       final Object[] results = executeValues(frame);
12       // after all new values have been evaluated, we can safely
             bind them
13       executeBindings(frame, results);
14       throw RecurException.INSTANCE;
15   }
16   @ExplodeLoop
17   private Object[] executeValues(VirtualFrame frame) {
18       final Object[] results = new Object[valueNodes.length];
19       for (int i = 0; i < valueNodes.length; i++) {
20           results[i] = valueNodes[i].execute(frame);
21       }
22       return results;
23   }
24   @ExplodeLoop
25   private void executeBindings(VirtualFrame frame, Object[] results
         ) {
26       for (int j = 0; j < bindings.length; j++) {
27           ((LetLocalSymbolNode) bindings[j]).executeWrite(frame,
                 results[j]);
28       }
29   }
30 }
```

Listing 5.12: *RecurNode* implementation.

```
1     (.toUpperCase "fred")
2     -> "FRED"
3     (.getName String)
4     -> "java.lang.String"
5     (.-x (java.awt.Point. 1 2))
6     -> 1
7     (System/getProperty "java.vm.version")
8     -> "1.6.0_07-b06-57"
9     Math/PI
10    -> 3.141592653589793
```

Listing 5.15: Dot special form examples.

The *dot* special form can be used to access fields or invoke methods of Java objects. It is important to note that all primitives will be boxed. The dot special form is parsed to either an instance field access, a static field access, an instance method invocation or a static method invocation. Listing 5.15 shows examples of using the dot special form[2].

[2]Source: `http://clojure.org/java_interop`

Chapter 6

Tail Call Optimization

In Clojure, there are no side-effect based control structures like *for- or while-loops* in imperative languages. Instead, recursive function calls are used as looping construct. While both constructs are semantically similar, recursive function calls, like any other function calls, consume space on the stack. Depending on the number of iterations a program may fail with a stack overflow exception. However, if a call is in tail position – i.e., the call is the last thing the function does – then the contents of the current frame are not required anymore after the function returns and can thus be removed prior to performing the call [26]. This way, the stack does not grow when executing call in tail position. Due to the lack of side-effect based loops, most functional programming languages demand support for tail call optimization.

A tail call is a function call performed as the final action of a function. In other words, the return value of the caller is the result of the function call. Thus, the caller frame is not required after the call and can actually removed before performing the call, keeping the stack size constant. Tail calls can be implemented in such a way that they do not consume any space on the stack. Clojure does not support tail call optimization because Java bytecode does not support it. Instead there is a special

Form	Tail Positions
fn	(fn [args] expressions expression=tail)
loop	(loop [bindings] expressions expression=tail)
let	(let [bindings] expressions expression=tail)
do	(do expressions expression=tail)
if	(if test then=tail else=tail)

Table 6.1: This table lists all tail positions in Clojure's special forms.

loop/recur expression for implementing tail recursive calls. During compilation, this special form is translated into a loop. However, tail recursive calls are only a subset of tail calls. In TruffleClojure we offer support for general tail call optimization.

Detecting whether an expression is in tail position is rather easy in Clojure. All possible cases are listed in table 6.1: the last expression in *fn, loop, let and do* forms are always in tail position; the *then and else* expression of an *if* form are also tail positions. To determine whether a call is in tail position it is sufficient to check whether the call expression is in one of these places. This decision is made by the *ClojureParser* which passes a boolean flag to the to the corresponding call node. The call node then performs either a tail call or a regular call.

In TruffleClojure, we base our implementation of tail call optimization on *trampolines*, as described in the paper *Tail Call Optimization on the Java Virtual Machine* [23] by Odersky et. al. Conceptually, a trampoline is an outer function that (repeatedly) invokes an inner function. Each time the inner function intends to perform a call in tail position, it does not call it directly but passes its identity to the trampoline, which then performs the call. In our interpreter, the *outer function* is actually an AST node and the *identity* passed around is a Truffle *call target* to the function to be invoked.

```
(def c
  (fn [z]
    (d z))) ;call to bar in tail position

(def b
  (fn [y]
    (c y))) ;call to c in tail position

(def a
  (fn [x]
    (b x))) ;call to b in tail position

(def foo
  (fn []
    (a 10))) ;call to a in tail position

;;fn d similar to a,b,c...
```

Listing 6.1: Example program for tail call optimization.

Consider the example program in Listing 6.1. All calls are in tail position and thus subject for TCO. The functions defined in the example form a call chain as illustrated in Figure 6.1: function *foo* calls function *a*, function *a* call function *b* and so forth. If TCO is enabled, we throw an exception instead of performing an actual call. The *TrampolinedNode* catches this exception and performs the call. This behavior is shown in Figure 6.2.

In TruffleClojure, the trampoline is a specialization of a *CallNode*, and the "inner function" to invoke is a *CallTarget* to the AST of a function. The implementation is realised by throwing exceptions. Instead of performing a regular call by directly invoking the *CallNode*, we throw a *TailCallException* which contains the next *CallTarget* and the evaluated arguments we want to pass to the function. This exception is then caught by the caller. The caller extracts the next call target and arguments to pass to the function invocation and invokes the function on behalf of the callee. We refer to a *CallNode* that catches these exception as *TrampolinedCallNode* because frames appear to "bounce" off of this node, as is illustrated in Figure 6.3.

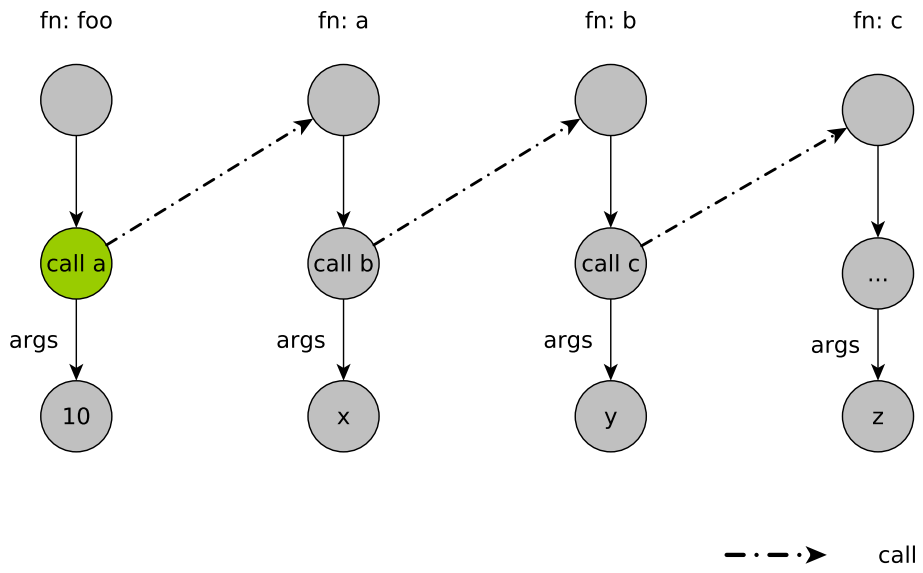


Figure 6.1: The call chain before tail call optimization: Function *foo* calls *A* in tail position, function *A* calls *B* etc.

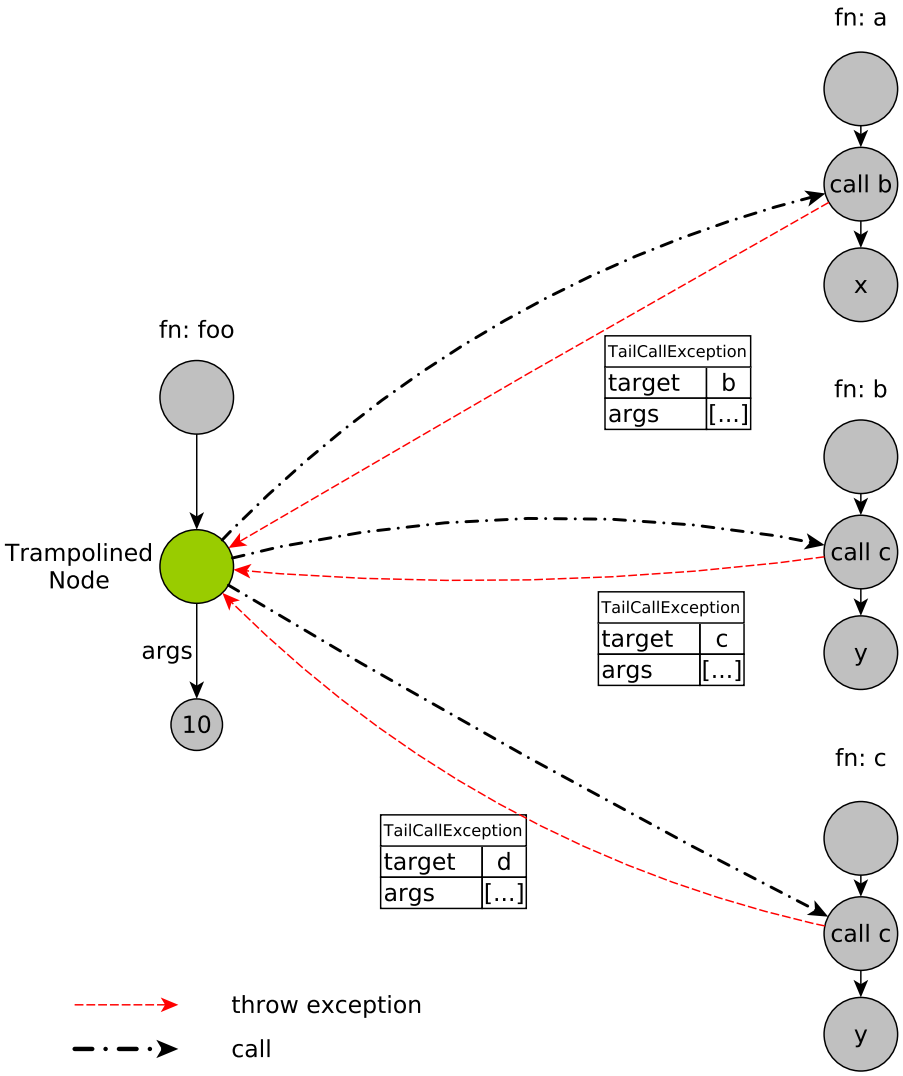
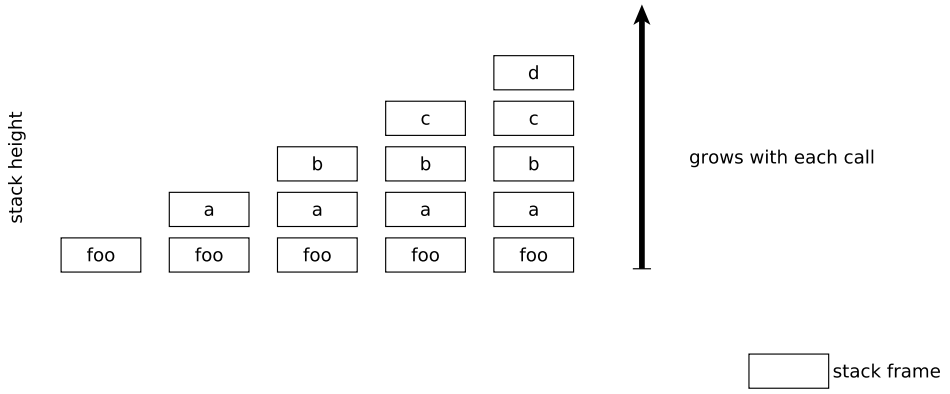
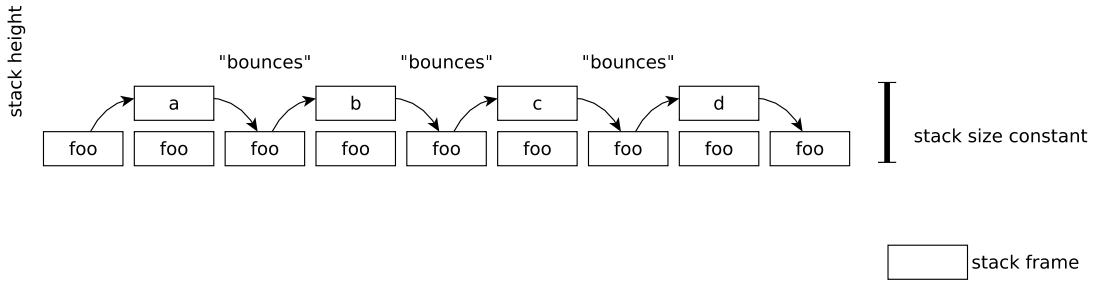


Figure 6.2: The call chain after tail call optimization: Instead of performing a call, function A,B,C throws a *TailCallException*. The *TrampolinedNode* catches the exceptions and performs the actual call



(a) Without TCO enabled, the stack height grows with each function invocation.



(b) If TCO is enabled, calling a function in tail position throws an exception which is caught by the *TrampolinedCallNode* of the caller. The stack size does not grow as in example above: the new frames appear to be bouncing off of the catching node, hence the name *trampoline*.

Figure 6.3: The following examples show the growth of the stack with (a) TCO disabled and (b) enabled. Boxes represent stack frames.

Chapter 7

Case Study

This chapter contains a case study to demonstrate how our interpreter executes a Clojure program. In particular, the example describes macro expansion and the use of builtins instead of the default implementation of Clojure.

The Clojure source for calculating the *nth* factorial is given in Listing 7.1. In this example we define a function called *factorial*. The function has one method implementation that accepts two arguments and calculates the n^{th} factorial by using Clojure's special form *loop/recur*. To define the function, we use the *defn* macro, which is defined by Clojure's core library. Macros are simple Clojure programs that extend the functionality of the parser. Whenever the parser encounters a macro call, it expands the macro: the result of macro expansion for our example is shown in Listing 7.2. Note that the *defn* expression is no longer present: it was replaced by the special forms *def* and *fn*. *Defn* was the only macro present in our example program.

```
(defn factorial   ;defn is a macro
  ([n]
    (loop [cnt n, acc 1]
      (if (<= cnt 1)
        acc
        (recur (- cnt 1) (* acc cnt))))))

(factorial 10)
```

Listing 7.1: Factorial function in Clojure.

```
;the defn macro was expanded into special forms def and fn
(def factorial
  (fn ([n]
      (loop [cnt n, acc 1]
        (if (<= cnt 1)
          acc
          (recur (- cnt 1) (* acc cnt)))))))
```

Listing 7.2: Factorial after macro expansion

After macro expansion, the parser continues parsing the expanded form. Since no further macros are present, the AST is built. The operators in Listing 7.2 are actually regular function calls. The functions named *<=*, *-* and *** are also defined in the core library of Clojure. For example, the implementation of the $*$ operation is given in Listing 7.3. The two-argument overload is defined in terms of a *builtin* expression. This is where our implementation of the core library differs from Clojure: in Clojure, multiplication is implemented as a Java interop call to a utility class, i.e., the two argument case is: `([x y] (. clojure.lang.Numbers (mul x y)))`. This utility method performs type checks to choose the correct action to perform. Instead of an Java interop call, our TruffleClojure AST contains a Truffle node which implements the multiplication operation and is able to perform type specialization (as described in Section 2.2.1).

Once parsing of the first expression is completed, the AST is immediately executed. The `Var` with the name *factorial* is bound to the function *factorial*. The AST of this method is shown in Figure 7.1.

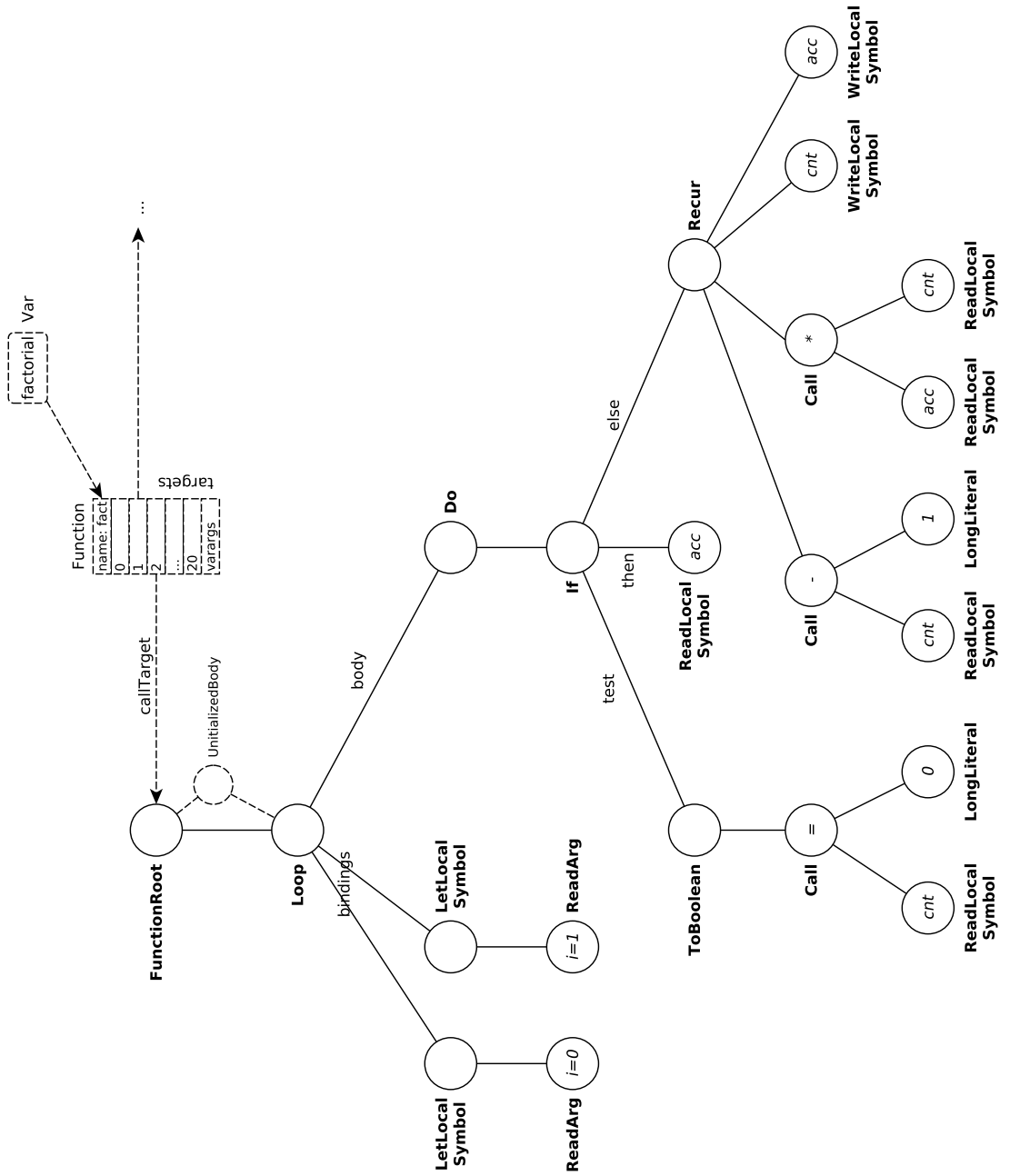


Figure 7.1: AST of factorial method after parsing.

```
1 (defn *
2   "Returns the product of nums. (*) returns 1. Does not auto-promote
3   longs, will throw on overflow."
4   ([] 1)
5   ([x] (cast Number x))
6   ([x y] (builtin (multiply x y)))
7   ([x y & more]
8     (reduce1 * (* x y) more)))
```

Listing 7.3: The core method '*'.

The interpreter continues with evaluating the form in line 8 of Listing 7.1. The result is a function invocation of our *factorial* method. Again, the AST is immediately executed after parsing and returns the result.

Chapter 8

Evaluation

This chapter contains a performance evaluation of our TruffleClojure interpreter compared to the default Clojure implementation.

All benchmark presented in this section were executed on a personal computer with a quad core *Intel(R) Core(TM) i7-2620M CPU @ 2.70GHz* and 8*GB* main memory running the 64*bit* operating system *Ubuntu 14.10 (kernel: 3.16.0-33-generic)*. We compare the performance of our interpreter to several configurations of the official Clojure implementation[1] (Clojure v1.6).

We measure the execution time of our benchmarks and normalize to the default Clojure implementation. All charts in this section display the speedup/slowdown relative to Clojure, i.e. the performance of Clojure is always 1. We perform a *warmup* phase of 50 iterations on all configurations before measuring. The following configurations were measured:

[1] `http://clojure.org/`

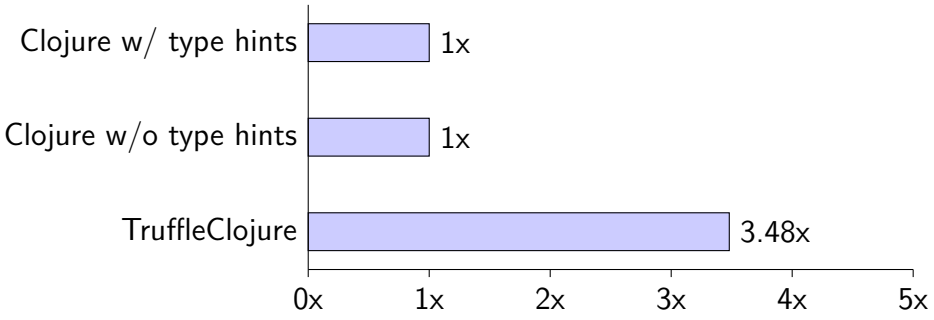


Figure 8.1: *Fibo* performance of TruffleClojure and Clojure.

Clojure1.6 w/ type hints:

The default Clojure implementation (v1.6). No additional flags are used. No changes to type hints. This yields the best performance for regular Clojure.

Clojure1.6 w/o type hints:

The default Clojure implementation (v1.6) without type hints. We removed all type hints from the core libraries and benchmark programs.

Truffle-Clojure:

The default configuration of our interpreter. No additional flags passed. Again, the core libraries and benchmarks do not contain type hints, as they are not necessary for our interpreter.

The *fibo* benchmark calculates the n^{th} Fibonacci number. Figure 8.1 shows an increase in performance by a factor of 2.5 for TruffleClojure. The performance increase is achieved by type specialization: the parameters are assigned to the Truffle frame and operations on these values use the primitive types instead of boxed counter parts as in Clojure. This benchmark nicely demonstrates how Clojure benefits from our Truffle backend. An optimized version of the fibo benchmark (i.e., casting the arguments to primitive types and assigning them to local binding manually) on the default Clojure runtime achieves equivalent performance.

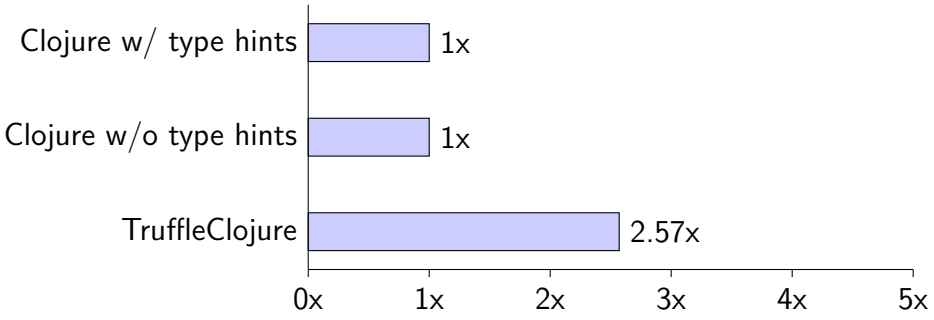


Figure 8.2: *Tailcall* performance of TruffleClojure and Clojure.

The *tailcall* benchmark (Figure 8.2) is a tail-recursive implementation for calculating factorials – without using Clojure's special *loop/recur* construct. Similarly to the *fibo* benchmark, type hints have no effect on this benchmark. With our implementation of tail call optimization we achieve a speedup by a factor of approximately 3. For large enough values Clojure will even terminate with a *StackOverflowException*. Using the *loop/recur* instead of a direct recursive call, the performance of Clojure is similar to TruffleClojure: the Clojure compiler emits bytecode that corresponds to a Java loop instead of a function call.

The *java-io* benchmark collection contains programs that make heavy use of Java interop, i.e., invoking Java methods from within Clojure programs. The results shown in Figure 8.3 are normalized to Clojure's performance. Clojure without type-hints is slower by a factor of approximately 10. With type hints the Clojure compiler is able to statically resolve the methods and emits efficient byte code, e.g. *invokevirtual* or *invokestatic*. Without type hints the methods are invoked via reflection at runtime. In TruffleClojure we speculate that types at call sites stay constant and cache the resolved method handles, which leads to an increase in performance compared to *Clojure w/o type hints* because we do not perform a lookup every time the method is invoked. However, we still rely on reflection to invoke the methods at runtime, which is much slower than using the according bytecode instructions. The core library of Clojure makes heavy use of Java interoperability. This is the reason

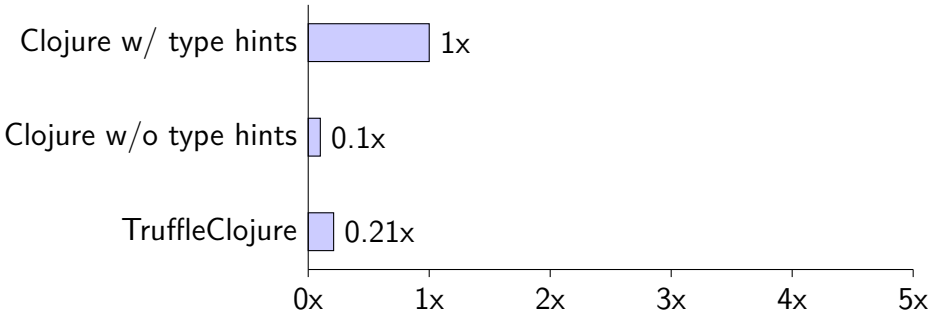


Figure 8.3: *Java-interop* performance of TruffleClojure and Clojure.

why our interpreter is slower for more sophisticated benchmarks that use the core library. Improving the performance of Java interoperability is intentionally left as future work.

Chapter 9

Future Work

So far, we have implemented a subset of Clojure as a proof-of-concept. This chapter discusses potential future work.

Language Completeness:

The current interpreter only implements a subset of the features of Clojure. There is still a lot of work to be done to make the interpreter fully compatible. However, the lack of a formal specification further complicates the completeness effort.

Notable missing features include concurrency support, especially the software transactional memory that is a builtin part of the Clojure language. Furthermore, only parts of the *clojure.core* library were implemented.

Performance:

Peak performance was declared an explicit non-goal for this prototype. Although the performance of *TruffleClojure* is comparable to that of the original Clojure implementation, we believe that it can be further improved. In par-

ticular, the performance of Java interoperability must be improved because Clojure heavily relies on it: a significant amount of core methods perform calls to Java utility classes. Real-life Clojure programs heavily use the core library and are thus slowed down significantly by our current implementation of Java interop.

Concurrency Support:

Another significant feature of Clojure is its approach to concurrency. Clojure is one of the first languages that offer builtin software transactional memory. Concurrency support was declared out of scope for this thesis.

Language Interoperability:

The integration of the Truffle language interoperability project [11, 12] would allow our Clojure prototype to not only call Java but any other language that is already implemented with Truffle (e.g. C, Ruby, JavaScript or R).

Automatic Generation of Type Hints:

The official implementation of Clojure relies on type hints to achieve high performance. Our Truffle-Clojure interpreter infers the types automatically through node specialization. These inferred types could be extracted and used to improve/complete the type annotations of the core libraries.

Chapter 10

Related Work

TruffleJS and other Truffle languages:

TruffleJS [30] is a JavaScript implementation using the Truffle framework. It outperforms other JavaScript engines on top of the JVM [30]. JavaScript was used as Truffle's proof-of-concept. In contrast to other JavaScript implementations on the JVM, TruffleJS does not produce Java bytecodes. Instead, TruffleJS is an AST interpreter that specializes itself during execution. Runtime information is collected during the AST's execution, based on which the AST optimizes itself.

As the Truffle framework becomes more popular, more language implementations start to be developed. There are language implementations for Ruby [25], C [10], R[1] [16] and Python[2] [29].

[1] `https://bitbucket.org/allr/fastr`
[2] `https://bitbucket.org/ssllab/zippy`

ClojureScript and ClojureCLR:

ClojureScript is a variant of Clojure targeting JavaScript instead of Java bytecode [35]. It also runs on the JVM but can be compiled to JavaScript which is intended to be executed in a browser.

ClojureCLR is another derivative of Clojure which runs on the common language runtime (CLR) instead of the JVM. The approach is similar to that of JVM Clojure: namely to compile Clojure programs to platform specific bytecode, in this case the CLR Intermediate Language (IL).

Tail Call Optimization in the Java HotSpot™Virtual Machine:

Schwaighofer et al. describe techniques for supporting tail call optimization in the Java HotSpot VM [24]. They achieve this by modifying the interpreter as well as the client and server compilers of the HotSpot™VM. The programmer may annotate a method call as a tail call and the VM guarantees that the call is optimized.

Their solution complies with the Java access security semantics. Removing a frame from the stack might be in violation of the security policy. Their implementation of TCO detects these violations and will either throw an exception or execute a regular call depending on the flags passed to the VM. If the stack does overflow the deoptimization infrastructure is used to compress the stack in order to guarantee that a series of tail calls does not overflow the stack.

Tail call elimination on the Java Virtual Machine:

Schinz and Odersky [19] describe multiple techniques on how to implement tail call optimization on a JVM that does not offer native support for it. The first technique they describe is to inline all function calls into a big method and using switch statements instead of calls inside this function. However, they conclude this technique is not suitable for big programs, and especially not

for the JVM since methods have a maximum instruction size [9]. The second technique presented is to use *trampolines*. A trampoline is an outer function that repeatedly calls an inner function [19]. Our implementation is inspired by this approach. The third technique presented is to transform the program to continuation passing style, i.e., converting every call into a tail call. Once the stack reaches a certain limit and is about to overflow, the current continuation is passed to a trampoline at the bottom of the stack which then invokes the continuation. The disadvantage of these techniques compared to supporting tail calls natively within the JVM is the added performance overhead and the additional implementation work that is required.

Chapter 11

Conclusions

This thesis describes a novel implementation of the Clojure programming language based on the Truffle interpreter framework. Instead of compiling Clojure programs to Java bytecode, we base our interpreter on an AST which is able to optimize itself during runtime. Optimization incorporates runtime profiling information during the execution of the AST in order to produce efficient code. In addition, we were able to implement tail call optimization which was originally omitted from the Clojure programming language because of the lack of support by the Java Virtual Machine. Although our interpreter is not complete, the benchmarks show promising results and demonstrate that the Clojure programming language can benefit from a complete implementation on top of the Truffle framework. Furthermore, this work also demonstrates the suitability of the Truffle framework to implement functional programming languages.

List of Figures

Listings

Bibliography

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[2] Phil Bagwell. Ideal hash trees. *Es Grands Champs*, 1195, 2001.

[3] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, and Christian Wimmer. Graal IR: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[4] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*, pages 1–10. ACM, 2013.

[5] David Flanagan and Yukihiro Matsumoto. *The Ruby programming language*. " O'Reilly Media, Inc.", 2008.

[6] Michael Fogus and Chris Houser. *The Joy of Clojure: Thinking the Clojure Way*. Manning Publications Co., 2011.

[7] Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[8] Brian Göetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java concurrency in practice*. Addison-Wesley, 2006.

[9] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.

[10] Matthias Grimmer. A Runtime Environment for the Truffle/C VM. Master thesis, Johannes Kepler University Linz, 2013.

[11] Matthias Grimmer. High-performance language interoperability in multi-language runtimes. In *Proceedings of the companion publication of the 2014 ACM SIGPLAN conference on Systems, Programming, and Applications: Software for Humanity*, pages 17–19. ACM, 2014.

[12] Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. Dynamically composing languages in a modular way: supporting c extensions for dynamic languages. In *Proceedings of the 14th International Conference on Modularity*, pages 1–13. ACM, 2015.

[13] Rich Hickey. The Clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 1:1–1:1, New York, NY, USA, 2008. ACM.

[14] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 European Conference on Object-Oriented Programming*, pages 21–38. Springer, 1991.

[15] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A domain-specific language for building self-optimizing AST interpreters. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, pages 123–132. ACM, 2014.

[16] Tomas Kalibera, Petr Maj, Floreal Morandat, and Jan Vitek. A fast abstract syntax tree interpreter for r. In *ACM SIGPLAN Notices*, volume 49, pages 89–102. ACM, 2014.

[17] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine specification*. Pearson Education, 2014.

[18] John McCarthy. History of lisp. In *History of programming languages I*, pages 173–185. ACM, 1978.

[19] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical report, 2004.

[20] Oracle. Graal openjdk project documentation. `http://lafo.ssw.uni-linz.ac.at/javadoc/graalvm/all/index.html`, 2013.

[21] Oracle. OpenJDK: Graal project. `http://openjdk.java.net/projects/graal/`, 2013.

[22] Michel F Sanner et al. Python: a programming language for software integration and development. *J Mol Graph Model*, 17(1):57–61, 1999.

[23] Michel Schinz and Martin Odersky. Tail call elimination on the Java Virtual Machine. *Electronic Notes in Theoretical Computer Science*, 59(1):158–171, 2001.

[24] Arnold Schwaighofer. Tail call optimization in the Java HotSpot VM. 2009.

[25] Chris Seaton, Michael L Van De Vanter, and Michael Haupt. Debugging at full speed. In *Proceedings of the Workshop on Dynamic Languages and Applications*, pages 1–13. ACM, 2014.

[26] Guy Lewis Steele Jr. Debunking the expensive procedure call myth or, procedure call implementations considered harmful or, lambda: The ultimate goto. In *Proceedings of the 1977 annual conference*, pages 153–162. ACM, 1977.

[27] Guido Van Rossum et al. Python programming language. In *USENIX Annual Technical Conference*, volume 41, 2007.

[28] Luke VanderHart, Stuart Sierra, and Christophe Grand. *Practical Clojure*. Springer, 2010.

[29] Christian Wimmer and Stefan Brunthaler. ZipPy on Truffle: a fast and simple implementation of Python. In *Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity*, pages 17–18. ACM, 2013.

[30] Andreas Wöß. Self-optimizing AST Interpreter for JavaScript. Master thesis, Johannes Kepler University Linz, 2013.

[31] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. An object storage model for the Truffle language implementation framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 133–144. ACM, 2014.

[32] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all, 2013.

[33] Clojure. `http://clojure.org/`, 2015.

[34] ClojureCLR. `https://github.com/clojure/clojure-clr`, 2015.

[35] ClojureScript. `https://github.com/clojure/clojurescript`, 2015.

[36] IBM J9. `https://www.ibm.com/developerworks/java/jdk/`, 2015.

[37] OpenJDK. `http://www.openjdk.java.net`, 2015.

Personal Information

Thomas Feichtinger

Born: June 15, 1988 in Gmunden, Austria

Permanent address
Unionstrasse 11a/27
4020 Linz
Austria

Contact information
Phone +43 664 2429061
E-Mail t.feichtinger@gmail.com
LinkedIn tfeichtinger

Professional Experience

Johannes Kepler Universität, Linz, Austria
Student Researcher, Institut für Systemsoftware (SSW) — **07/2014 – current**

Catalysts GmbH, Linz, Austria
Software Engineer — **11/2012 – 03/2015**

Siemens Corporate Research, Princeton, New Jersey, USA
Software Engineering Intern — **02/2012 – 10/2012**

Catalysts GmbH, Linz, Austria
Software Engineer — **03/2011 – 03/2012**

Education

Johannes Kepler Universität, Linz, Austria — **10/2013 – 06/2015**
Software Engineering

Johannes Kepler Universität, Linz, Austria — **10/2008 – 02/2013**
Computer Science

Dublin City University, Dublin, Ireland — **09/2010 – 03/2011**
Computer Applications — *Exchange semester*

Languages
German (native), English (fluent)

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, am xxx

Thomas Feichtinger, BSc.