"An Oz implementation using Truffle and Graal"

Istasse, Maxime

Abstract

We discuss and improve Mozart-Graal, an experimental implementation of the Oz language on the Truffle language implementation framework. It allows Oz to benefit from the Graal JIT compiler and is seen as an opportunity to achieve an efficient implementation with lower maintenance requirements than Mozart 1 and Mozart 2. We propose a mapping for the key Oz concepts on Truffle. We take advantage of the static analysis to make the implementation garbage collection-friendly and to avoid extra indirections for variables. The dataflow variables, calls, tail call optimization, unification and equality testing are implemented in a JIT compiler-friendly manner. Threads are however mapped to coroutines. We finally evaluate those decisions and the quality of the obtained optimizations and compare the performance achieved by Mozart-Graal with Mozart 1 and Mozart 2.

Document type : *Mémoire (Thesis)*

Référence bibliographique

Istasse, Maxime. *An Oz implementation using Truffle and Graal.* Ecole polytechnique de Louvain, Université catholique de Louvain, 2017. Prom. : Van Roy, Peter ; Daloze, Benoit ; Maudoux, Guillaume.

UCL
Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

An Oz implementation using Truffle and Graal

Dissertation presented by
Maxime Istasse

for obtaining the Master’s degree in
Computer Science and Engineering

Supervisor(s)
Peter Van Roy, Benoit Daloze, Guillaume Maudoux

Reader(s)
Sébastien Combéfis, Hélène Verhaeghe

Academic year 2016-2017

Acknowledgements

I would like to thank Peter Van Roy for his guidance, his constant enthusiasm and his many relevant questions I would not have addressed otherwise.

I would also like to thank Benoit Daloze for having initiated this project, supervised my work closely and made myself a better contributor.

Many thanks to Guillaume Maudoux for his advice and availability as well as for the Dj'OZ project.

I would like to thank Hélène Verhaeghe for providing me more testing material and her and Sébastien Combéfis for having accepted to read this dissertation.

Finally, I would like to thank my friends and family for their unconditional support.

Abstract

We discuss and improve Mozart-Graal, an experimental implementation of the Oz language on the Truffle language implementation framework. It allows Oz to benefit from the Graal JIT compiler and is seen as an opportunity to achieve an efficient implementation with lower maintenance requirements than Mozart 1 and Mozart 2.

We propose a mapping for the key Oz concepts on Truffle. We take advantage of the static analysis to make the implementation garbage collection-friendly and to avoid extra indirections for variables. The dataflow variables, calls, tail call optimization, unification and equality testing are implemented in a JIT compiler-friendly manner. Threads are however mapped to coroutines.

We finally evaluate those decisions and the quality of the obtained optimizations and compare the performance achieved by Mozart-Graal with Mozart 1 and Mozart 2.

Contents

List of Figures

Listings

Acronyms

API Application Programming Interface. 22, 27, 28, 38, 49, 54, 58

AST Abstract Syntax Tree. 19–21, 23, 24, 26, 27, 31, 32, 34–37, 39, 42, 45, 49, 56, 58, 65, 78, 79

DSL Domain-Specific Language. 21–23, 56, 79

EA Escape Analysis. 23

IR Internal Representation. 22, 62, 66

JIT Just-In-Time. 18, 20, 22, 28, 30, 64

JVM Java Virtual Machine. 17, 23–29, 42, 61, 74

OS Operating System. 27, 28

OSR On Stack Replacement. 57, 58, 65–67

PEA Partial Escape Analysis. 23

PIC Polymorphic Inline Cache. 69

SSA Static Single Assignment. 22

SSH Secure Shell. 61

VM Virtual Machine. 17

Chapter 1

Introduction

It has been a long time since other languages than Java [1] are running on the Java Virtual Machine (JVM) [2]. Some languages have directly been designed to take advantage of the VM and its typed system, such as Scala. Several attempts have also been made aiming for dynamically typed languages to be ported on the JVM, such as Python [3] through Jython [4] and Ruby [5] through JRuby [6].

Those languages have proved to be painful to implement [7]. Not only because official implementations or standard libraries provide features that are not implemented on the JVM, but also because it is extremely hard to achieve good performance given the dynamic mechanisms required to respect the semantics of those languages. In order to achieve this, the JVM had to be adapted because of assumptions that had been made about Java. Just like *just-in-time* compiling the language into Java was not viable because not all the generated Java bytecode could get garbage collected [8].

The *Da Vinci Machine* project [9] was the first attempt in order to bridge this gap, bringing more dynamic features on the JVM. It has been quite successful, allowing some JVM implementations to reach the same level of performance as official implementations. This however still required a lot of engineering and expertise from the language implementers.

The *Truffle* and *Graal* projects [10] have allowed to write interpreters for dynamic languages that are competitive with official or high-budget implementations at peak performance. For instance, TruffleRuby surpasses MRI [5] by orders of magnitude [11] and Graal.JS performs close to V8 [12] [13].

In the meantime, the Oz programming language [14] has almost exclusively been executed on dedicated Virtual Machines (VMs). Mozart 1 [15], though highly optimized, now suffers from it because the hardware has evolved. Mozart 2 has been an attempt at making a new dedicated VM adapted to today's hardware, but cannot be claimed as complete yet. We believe relying on a lasting VM to manage architecture evolution could relieve the Oz language from the burden of being continuously reimplemented.

Furthermore, Oz being a multi-paradigm language behaving differently from Java in many ways, we think it provides an interesting use case for *Truffle* and *Graal*. The research efforts put into those project and their promising results make us believe that an implementation using them is probably worth the effort it takes.

Mozart-Graal [16] is a prototype implementation of Oz on top of *Truffle* and *Graal*, on which we base our work. We take this thesis as an occasion to verify and discuss its mechanisms while extending it.

Contributions

Mozart-Graal is relatively new and aims at being an efficient implementation of the Oz language. It has been initiated by Benoit Daloze as an experiment, therefore lacking of some core features and inconsistent to some respects with Mozart 1 and 2. The basics for functors, dataflow variables, procedures, calls, builtins, classes, records, conses, etc were already set up at the beginning of this work. In fact, the Browser and the Panel programs only required some improvements to run at that point.

However, some of the existing mechanisms had undesirable corner cases, were not optimized or were not garbage-collection-friendly. We have thus mainly devoted our efforts to improving the correctness and the performance of the implementation.

Static analysis has thus been extended to optimize tail calls and reduce variable usages and lifetimes. The mechanism for handling external environments has been revamped to become garbage-collection-friendly. Dataflow variables, unification and equality testing have been corrected. Method calls now benefit from polymorphic inline caches and calls to inexistent methods are well redirected to the `otherwise` fallback method and are even optimized. Different strategies have also been explored to allow earlier Just-In-Time (JIT) compilation. Other modifications have occurred in the process, such as the implementation of new builtins or the inlining of operators, and more improvement ideas have been generated.

All the modifications can be found in a GitHub repository [17]. Part of them have been merged in the official Mozart-Graal repository [16] and the others are meant to be in a short future. The total modifications consist of 52 commits providing 3 thousands new lines as a result of 4 thousands additions and 1 thousand deletions spread in more than a hundred files.

In order to evaluate the improvements made, a tool has been written in Python for running Oz programs and aggregating their outputs. It also allows for other implementations to run those. This tool is available in a dedicated GitHub repository [18] along with some benchmark programs and a Jupyter notebook [19] for data visualization.

Structure

First, the main principles about self-optimizing interpreters, Truffle, Graal, their roles and general specificities will be introduced in Chapter 2.

A more refined idea of what has been accomplished in Mozart-Graal is provided right after in Chapter 3. Both in terms of the structure, and in terms of the language that has been implemented.

Chapter 4 then proposes to have a look at the mapping of some key concepts from Oz to Truffle without considering performance. This should allow to grasp the primitives necessary to write languages implementations running with Truffle.

Chapter 5 is then all about specific optimizations that are now performed when running code on Mozart-Graal. The first part details improvements in performance and memory management that rely on information provided by the static analysis of our compiler whereas the second part describes optimizations relying on Graal or making its job easier.

Chapter 6 evaluates the different optimizations and compares Mozart-Graal to Mozart 1 and 2, exposing both quantitative data collected on benchmarks and some of the Graal compiler output. Some limitations of the actual code and tools behind have been encountered and will be detailed.

Chapter 7 sets out some possible future developments.

Chapter 2

Main Principles

In this chapter, we will discuss briefly what Truffle and Graal are, and what are the principles behind them. We will see what self-optimizing Abstract Syntax Tree (AST) interpreters are, how Truffle allows writing them easier. Finally, we will see what makes Graal special and how it can take advantage from such interpreters being written with Truffle.

2.1 Self-Optimizing AST Interpreters

We consider an AST interpreter as a tree of node objects that can be evaluated within some execution environment. AST interpreters are a very common way to implement a language. We will see how they work in their simplest form, and how they can take advantage from runtime execution profiling to make themselves considerably faster.

2.1.1 Classical AST Interpreters

AST intepreters work mostly by traversing the AST in post-order fashion, performing operations specific to each node that will affect the state of the program to emulate the desired semantics.

Here is an example of an AST for a non-trivial expression in Oz:

```
if (B mod 2) == 0 then
    B div 2
else
    (3*B)+1
end
```

Listing 2.1: Oz expression example

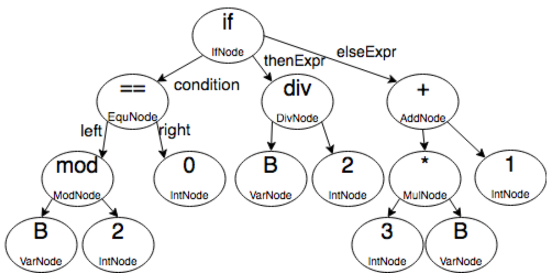


Figure 2.1: AST from expression example at Listing 2.1

In this dissertation, each node kind corresponds to a Java class, and so each node is a Java object. The operations they perform at execution are implemented in their `Object execute(Frame)` method.

The idea behind this signature is that nodes can return some result when evaluated and use the current stack frame to access local variables, but they can also make use of the whole Java language, access their attributes or constants across the VM, ... Every node can thus execute its children, and perform arbitrarily complex logic with their results.

If AST interpreters are considered as being the most straightforward way to implement a language, they are also usually very slow, executing the guest language orders of magnitude slower than the host language. A lot of interpreted languages suffer from poor performances on the JVM because they are basically too rich. Their nodes therefore have to perform a lot of checks and dispatches, even in the most trivial and performance-critical parts, in order to be correct every time.

2.1.2 Self-Optimizing AST Interpreters

Self-optimizing interpreters are interpreters taking advantage of the execution profile in order to optimize themselves [20].

The first idea that allows those interpreters to be optimized is that there are conditions under which for some node instances, the `execute` method could do simplified work. If a "+" node in the code always adds two integers for instance, there is no need to make complex dispatching on object types.

The second idea, the one making self-optimization fit that well into AST interpreters is that we can discover at runtime under which conditions every node is being executed from the beginning of execution. It is therefore possible to avoid hard work as long as the preconditions of lighter variants are fulfilled.

Nodes can also have their own state depending on what they have received in their previous usages, and conditions about it can be expressed in guards. This is especially useful for implementing mechanisms similar to polymorphic inline caches for instance. Those are motivated by the fact some call nodes could observe a small set of types for the whole execution of a program. Information about methods can thus be cached most of the time.

A simple way of implementing self-optimizing interpreters is presented in [20]. The language implementer basically provides several node classes (i.e. specializations) for each single feature, each specialization checking the preconditions it relies on still hold before executing. If this is not the case anymore, it must replace itself with another specialization and let it execute instead.

A code from the paper is provided in LISTING 2.2 in order to illustrate such a node specialization for the "+" operator expecting both terms to be integers. The paper also describes a way to avoid boxing primitive return types by providing more `execute` variants, like `int executeInt(Frame)`, and using exceptions to de-optimize when objects are encountered instead. If this has not yet been implemented in this Mozart-Graal, the Just-In-Time (JIT) compiler already tries to avoid boxing in compiled code.

```java
class IntegerAddNode extends BinaryNode {
        public int execute(Frame frame) {
                Object left = leftNode.execute(frame);
                if (!(left instanceof Integer)) {
                        // Rewrite this node and execute the rewritten
                        // node using already evaluated left.
                }
                Object right = rightNode.execute(frame);
                if (!(right instanceof Integer)) {
                        // Rewrite this node and execute the rewritten
                        // node using already evaluated left and right.
                }
                // Overflow check omitted for simplicty.
                return (Integer) left + (Integer) right;
        }
}
```

Listing 2.2: Example of an AddNode specialization for the Integer type

Note a desirable property for this kind of interpreter is to stabilize with time. It will ensure the AST manipulations become asymptotically negligible with respect to the guest program execution. This will become even more important when compiling the AST to reach higher performance.

This is why [20] suggests a node should become more general on each rewriting, and never go from a more generic specialization to a more specific one. In case of specializations for unrelated types[1], the best way to obtain a more general specialization is to chain them, testing types and using the corresponding specialization. We will see Truffle proposes a practical way of formulating all of this.

[1]i.e. none of the types is a particular case of another. `add(String a, String b)` and `add(int a, int b)` for instance.

```java
@NodeChildren({ @NodeChild("left"), @NodeChild("right") })
public abstract class AddNode extends OzNode {

        @Specialization(rewriteOn = ArithmeticException.class)
        protected long add(long a, long b) {
                return ExactMath.addExact(a, b);
        }

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

        @Specialization
        protected BigInteger add(BigInteger a, BigInteger b) {
                return a.add(b);
        }

}
```

Listing 2.3: Truffle implementation of the Oz "+" operator

2.2 Truffle

Truffle is a Java framework providing the basic machinery for implementing self-optimizing AST interpreters for new languages. It may seem there is very little that can be abstracted if the goal is to stay very general in terms of expressiveness, but the idea is also to guide their design in a way they could be optimized by the Graal compiler.

In order to achieve this, here is a non-exhaustive list of what Truffle provides:

1. A superclass for AST nodes, with the AST rewriting routines.
2. The stack mechanism, along with the `Frame` interface.
3. `RootNode`s and `CallNode`s for functions and calls representation.
4. Compiler directives and profiling instructions for the Graal compiler.
5. A flexible object storage model.
6. A Domain-Specific Language (DSL) avoiding a lot of boilerplate code to the developer
   (a) For writing succinctly specializations of a node along with their guards and state.
   (b) That takes care of implicit conversions when calling them.

Its DSL aim at making the process of writing a self-optimizing interpreter much less repetitive. For instance, it makes it possible to write all the specializations of a node in one class, only by defining annotated methods. The code generator then takes care of describing everything relative to the rewriting of the nodes, the chaining, caching, etc. Annotations to drive it are described in [21] and are rather intuitive so that they will be used here and there without much attention.

Listing 2.3 provides the implementation of our "+" operator. Our implementation uses longs and doubles for handling Oz integer and floating numbers respectively. Java `BigInteger`s provide support for arbitrary precision integers. What Listing 2.4 states is that in our type system, `long` numbers can be cast implicitly to `BigInteger` if no `long` specialization is instantiated. The absence of implicit conversion from `long` to `double` will in contrast avoid Truffle to try passing a `long` argument to a `double` specialization.

Let us consider the "+" node of the `Add` function in Listing 2.5. For its first usage, a specialization for two longs will be instantiated. The second use will create a chain with both the `long` specialization and the one for the `double` type. The third usage will add a `BigInteger` specialization to this chain. The fourth usage will enter the long specialization but trigger its rewriting because of the overflow. The long specialization is discarded and this enables the `BigInteger` specialization to catch them and cast them into `BigInteger`s. Those consecutive rewritings can be visualized in Figure 2.2.

```
@TypeSystem({ long.class, BigInteger.class })
public abstract class OzTypes {

        @ImplicitCast
        @TruffleBoundary
        public static BigInteger castBigInteger(long value) {
                return BigInteger.valueOf(value);
        }

}
```

Listing 2.4: Type system description of our implementation

```
MaxLong = 9223372036854775807
fun {Add A B}
    A + B
end

{Add 1 2} % add(long, long), no overflow
{Add 1.0 2.0} % add(double, double)
{Add MaxLong+1 MaxLong+1} % add(BigInteger, BigInteger)
{Add MaxLong 1} % add(long, long), overflow
% Rewriting => add(new BigInteger(MaxLong), new BigInteger(1))
{Add 1 2} % add(new BigInteger(1), new BigInteger(2))
```

Listing 2.5: Oz scenario around AddNode rewriting

One could wonder where the `Object execute(Frame)` of each specialization are now. In fact, the DSL generates a node class extending `AddNode` for each, along with some other classes for handling uninitialized nodes, polymorphic ones, ... Those specialized classes are generated into `AddNodeGen` and call the specialization methods with some extra code around for implicit casting and exception handling for instance. Specialization methods must therefore be accessible by their subclasses (i.e. be `protected`).

2.3 The Graal Compiler

Graal is a Java JIT compiler written in Java and running on top of HotSpot [22]. It can completely replace the client and server compilers, but reuses the other components such as the interpreter, the garbage collection and the class loading mechanism [23]. This is an optimistic compiler that can take advantage of the Truffle Application Programming Interface (API) to compile self-optimizing interpreters.

Some of its internal mechanism will be described very briefly, as they are mostly hidden to the language implementer. However, this is still interesting to know what optimizations it can perform and how, in order to make sure our implementation does not prevent it to perform them on our code. Understanding its Internal Representation (IR) is also useful in order to later be able to interpret the output of the compiler.

2.3.1 Graal IR

Graal has its very own IR for Java programs that is first described in [24]. It states those programs are represented as directed graphs and are in Static Single Assignment (SSA) form [25], meaning variables are split into *versions* assigned only once. Node types are specified through class definition, every node

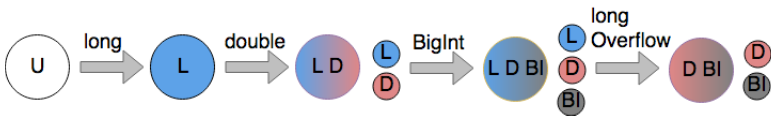


Figure 2.2: Consecutive rewriting of the “+” node in the scenario from Listing 2.5

class extending the `Node` class. (Not the same as Truffle's one)

The graph is created by specifying two types of edges: control flow (successor) edges and data flow (input) edges. Control flow edges only ensure control flow between loops, conditions, etc, whereas data flow edges express the input data of every operation. The latter may as well be used for expressing scheduling dependencies.

The `Node` class provides methods for easily redirecting edges or replace the node in the surrounding graph. As a side note, various interfaces are available on them in order to perform optimizations such as simplification and lowering. This is preferred to the visitor pattern to preserve extensibility.

The order specified by control flow edges is strict. However, nodes with only data flow edges can be moved between their dependencies and nodes depending on them and are said to be *floating* nodes. Flexibility in their scheduling allows to not have to maintain a correct schedule at all time and to determine it later with specific heuristics.

This also allows for code-motion based optimization to be implemented easily. Most floating nodes are for instance candidate to global value numbering [26], which allows for merging nodes that have redundant computations.

In order to run compiled Java *class* files, the Graal compiler first has to convert them into its intermediate representation. For simplicity however, only reducible loops are handled by the conversion. This is legitimated by the fact Java code only generates reducible loops but also means some bytecode obfuscation can render methods to be unusable by the Graal compiler, therefore forcing them to run in interpreter mode.

2.3.2 Optimization

Part of what allows Graal to compile efficiently dynamically typed languages is that it is actually good at optimizing and compiling Java, and therefore also other statically typed JVM languages.

The other part is due to the principles of self-optimizing interpreters and the hints language implementers can provide by annotating them. The former allow simpler routines to be compiled as long as they are sufficient for the execution. The latter helps the compiler acquiring additional knowledge about the code.

2.3.2.1 Optimizing Java Code

Graal still shares a lot with the HotSpot Server Compiler. One of its main improvement over the latter is however an analysis named Partial Escape Analysis (PEA). Escape Analysis (EA) allows a compiler to determine whether an object is accessible outside the allocating method or thread [23]. PEA has better granularity in that it makes the distinction between different branches.

In the HotSpot Server Compiler, EA allows stack allocation, scalar replacement and lock elision to take place [27]. Stack allocation allows objects that are not put into Java fields to be allocated on the stack. scalar replacement allows objects that are only used locally to be replaced by their useful fields allocated on the stack. This is thus stronger than stack allocation. Lock elision avoids unnecessary synchronizations when the object has not yet been able to escape to other threads.

Those optimizations benefit from method inlining, which provides more opportunities to avoid allocating complete objects passed as arguments to inlined function, etc. Refining the escape analysis to the branch level is another opportunity to make them more efficient as well, for instance only allocating objects and acquiring locks on them in the branches they can escape from.

It is however important to note that the compilation process is expensive. It is therefore valuable if the implementer can help the compiler to detect critical parts of programs and to perform some optimizations. Truffle enables this interaction and provides a DSL making it easier to write PEA-friendly code.

2.3.2.2 Truffle Compiler Directives and Annotations

Truffle allows to tag values to be considered as constants within a compilation even though references to them escape. This happens on the AST nodes we write for instance. Node children are considered as constant for compilation, which allows the compiler to directly look through the `execute` calls, consider the methods behind and reason more globally.

The language implementer can also provide other directives to the compiler, stating for instance to never compile some branches, or only if they have never been entered. He can also collect and inject branch probabilities into the AST or specify branches to only be taken in interpreter or in compiled mode. Loops can also be tagged to unroll whenever their number of iteration can be considered as constant.

Finally, all those mechanisms can be coupled with node state to specialize some general routine to the only values or types it has received that can then be considered as constant locally. This is definitely where dynamic languages can get the most from the JVM.

All of this is possible and made easy using Truffle. This will be discussed in SECTION 5.2.

2.3.3 Deoptimization

Such speculative optimization is however only enabled by the concept of deoptimization [28]. Whenever preconditions of compiled code are invalidated it is not safe to execute anymore. When encountering guards evaluating to `false` before such now-invalid sections of code or explicit deoptimization instructions, the interpreter thus has to take over the execution.

This deoptimization phenomenon may happen because of the language implementation using Truffle, for instance when none of the instantiated specializations can handle the actual parameters or when a branch that is not compiled needs to be taken, or that the AST needs to be rewritten. It may however happen for less visible reasons, like when a Java class is loaded, providing a second implementation for a method that was before considered as having only one implementation.

For deoptimization to work, it must be possible to reconstruct the state of the VM from the state of the physical machine at deoptimization points. This is what is achieved through framestates. They basically contain the information about what the interpreter must do in order to catch up what had already been done by the compiled code, without introducing side-effects. This of course includes a bytecode pointer, values in the stack frame, but can also describe objects to be allocated in case they were not.

Chapter 3

Specifications

In this chapter, we first discuss in what Mozart-Graal consists from an architectural point of view. Indeed, the project reuses some parts of Mozart 2, and is based on the JVM. This implies multiple technologies to be used together.

We then state what we expect our implementation to support. The whole standard library [29] has not been reimplemented and running on the JVM has forced us to make trade-offs with respect to the threading model. As far as the semantics is concerned, it should behave in the same way as other implementations.

3.1 Architecture

We discuss here the interactions between our work and the other components that allow it to work.

3.1.1 Project Overview

As illustrated by Figure 3.1, Mozart-Graal of course relies on external work. The parser, static analyzer have been forked from Mozart 2's bootcompiler, which was at first only meant to run the Oz compiler for Mozart 2. A part of the base Oz library also comes from Mozart 2. Truffle, Graal and the JVM are external dependencies and have been left untouched by ourselves.

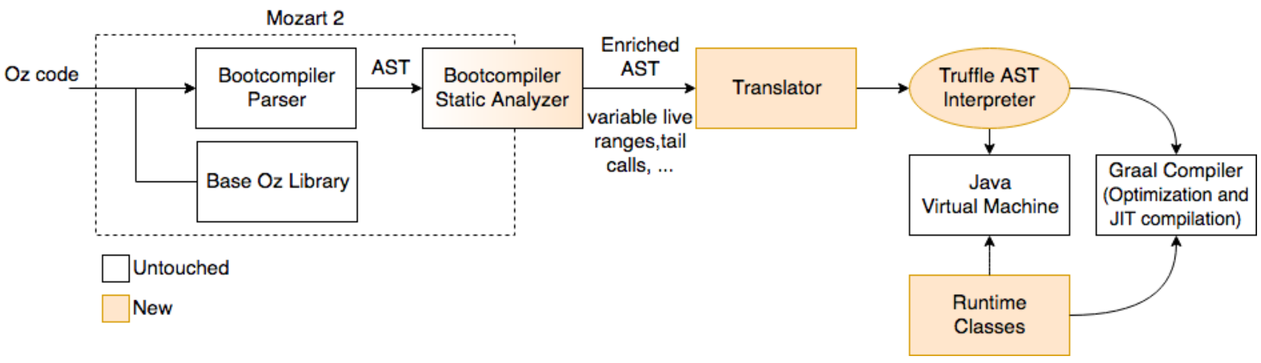


Figure 3.1: Mozart-Graal's flow and dependencies

Nevertheless, a slightly unusual JVM setup is required in order to have our Oz threads to work as expected [30]. The reasons will be discussed in Section 3.2.3.

The Mozart 2 bootcompiler components were written in Scala [31]. As Mozart-Graal targets the JVM and this language seemed more adapted to writing compilation phases than Java, we have kept extending static analysis in this part of the project. Scala is thus the language that will be used in order to discuss adaptations to the static analyzer.

The `Translator` is a Java class that generates Truffle AST nodes (i.e. the structure interpreting the Oz program) from the output of Mozart 2's bootcompiler. Those components are written in Java, so this is the language that will be used to described them in this dissertation.

3.1.2 Parsing and Static Analysis

The parser has mainly been left untouched. The static analyzer has however been modified here and there. Let us have a look at each phase: (–: as-is from Mozart 2, ~: adapted, +: added)

– Identifier resolution, that generates a unique identifier for each variable declaration in the current functor, and that links every occurrence of that variable in the AST to it.

~ Desugaring, that transforms functors, classes, functions more primitives constructs of the language.

  It has been extended in order to push the assignments of result variables as far as possible in the body of procedures in order to make detection of tail calls easier. This will be detailed in Section 4.4.3.

– Constant folding, that replaces occurrences of builtins by direct references.

– Unnesting, that handles the nesting marker `$` in procedure calls.

+ Tail call detection, that tags tail calls in order for the `Translator` to create adequate interpreter nodes.

  This will be discussed in Section 4.4.3 as well.

+ On stack variables detection, tagging local variables that are not required to be instantiated as dataflow variables.

  This optimization will be discussed in Section 5.1.2.

+ Frame slot clearing, that tags statements and expressions before or after which local variables could be forgotten in order to free references as soon as possible.

  This is an optimization and will therefore be discussed in Section 5.1.3.

3.1.3 Tests

Some Oz tests have been kept from Mozart 2 (`platform-test/base/`) for base types and concepts such as numbers, procedures, dictionaries, records, types, exceptions, conversions, threads and laziness. Some others may pass, but as the accent has not been put on completeness of the implementation, most will not. Either because they try to test syntactic structures that are not handled by Mozart 2's bootcompiler parser, or because they make use of unimplemented builtins.

For generic tail calls and cyclic unification, Oz unit tests have also been added there. They should thus lead to stack overflows or block whenever cases are missed. For some additions to static analysis, the behavior is by definition even harder to observe in Oz. Special builtins have been introduced in the `Debug` functor that allow ensuring variables are allocated on stack, or that references have been freed. In order not to influence the static analysis itself, variable names are provided as atoms.

All the tests known to pass are executed when no argument is provided to the `./oz` executable. Furthermore, the implementation should correctly run examples such as the Browser and the Panel located in `examples/`, as well as the 2014 Oz Music Project [32].

3.2 Language Specifications

It has been tried to make the implementation as compliant with existing resources as possible. There are however some differences due to actual limitations of the JVM and the fact implementing the whole standard library would have taken too much time. In this section, we will detail the differences between Oz_T, the implemented language, in contrast to the original Oz language.

3.2.1 The Semantics

Mozart-Graal tries to make Oz_T follow Oz' semantics, provided by [33, chap. 13], and seems to almost achieve it. Some hints about it will be provided in Chapter 4.

The main difference is that Oz_T tries to take advantage from evaluating expressions in the AST to values directly rather than associating them with dataflow variables systematically as the Oz kernel language would. It means that rather than being desugared down to the Oz kernel language, Oz_T programs are rather transformed to some intermediate believed to only have imperceptible differences with Oz, but also to be more efficient in terms of number of instantiated variables. We could call this language Oz_T's kernel language.

This therefore puts some distance between the formal version of the Oz language and Oz_T, since values can now be provided to any operation. We will however help ourselves to close this gap in the implementation, by bringing a unified reasoning on both variables and values. Also, rather than introducing a supplementary stack of variables for the nested expressions of our "kernel language", each of those could mostly be seen as introducing a single-use variable computed before the surrounding operation, and accessed at the place of the expression only.

Finally, if the illusion is not perfect yet, this is because there remains some minor edge cases that are not stated specifically in the transformation, making Oz_T's kernel language stand a little too faithful with respect to the user input compared to Oz's kernel language. This is however a matter of time. Those edge cases are provided with other bugs in Appendix B.

3.2.2 Builtins

Oz_T builtins are decomposed in *intrinsic* and plain-Oz builtins. The former had to be rewritten using Truffle, as discussed later in Section 4.3.9 while the latter have been forked from Mozart 2's standard library. If it has been tried to provide a sufficient kernel some interesting projects to run with the current implementation, it is far from being complete.

In order to make Mozart-Graal maintainers' lives easier, placeholder node classes have been generated for all of them, so when a builtin is missing, an exception is thrown at runtime and specializations can easily be added to the already existing node class.

3.2.3 Threads

For some reason, Oz_T's "threads" are not equivalent to Oz' ones. Let us first define different flavors for concurrency. If there seems to be no consensus as to how every concept should be called in the literature and in APIs, this is how each of those concepts will be referred to in this dissertation:

Threads allow different tasks to run concurrently, and are expected to be scheduled in a fair manner [33]. (they can therefore be suspended at any moment by the scheduler, and are therefore qualified as "preemptive") There is also a notion of priority between different threads.

Most Operating Systems (OSs) provide such a system but limit the number of threads per process. Having to interact through system calls and OS-level context switching induce a large overhead. Language implementations therefore try to provide more adapted and lightweight mechanisms.

(Symmetric) coroutines are basically non-preemptive threads [34]. Scheduling must be specified explicitly within a coroutine by releasing the execution through a *yield* operation. This is why they are said to provide "cooperative concurrency".

Asymmetric coroutines are similar to symmetric coroutines except they must return control to their invoker [34]. It is however possible for them to invoke other coroutines before doing so.

Oz' approach to concurrency is all about preemptive threads running in a single OS thread. On a standard JVM, however, only OS threads are available. Let us first have an overview of how state-of-the-art JVM interpreters and libraries achieve their kind of concurrency.

JRuby directly uses OS threads as asymmetric and symmetric coroutines [35]. This thus means only a limited number of them can be spawned and they induce a large overhead for context switching.

Akka [36] proposes a strong foundation for concurrent and distributed applications, but only makes use of OS threads as well.

Quasar [37] implements lightweight threads by instrumenting bytecode, either at compile-time or at runtime, in order to restore the stack and jump back to the instruction it was executing. [38]

Therefore, those two options would be available:

1. Either spawn an OS thread for every Oz thread, therefore limiting them in number and bringing a huge overhead for context switching. But such threads are preemptive and may benefit from multi-core CPUs.
2. Either simulate lightweight threads via code instrumentation to save the stack, restore it and jump back at the right location.

Luckily for us, a third options exists and consists in a modified JVM with Graal and added support for coroutines, very close to what is described in [39]. It seems better to rely on it.

Indeed, if the first solution seems very appealing, the fact we could only spawn a limited number of threads (around two thousand) is not acceptable. Some Oz programs have a strong tendency to create a lot of them for very lightweight concurrent tasks, which would therefore simply not work.

The second option is not really viable either because such instrumentation would be very intrusive and would most likely interfere with the analysis done by the JIT compiler.

Though the most adapted, the third solution would ideally still imply to add a preemption mechanism. Because the JVM does not provide any API for signals, this would therefore require to add costly timing checks within the program. At each tail call for instance. As it may lead to severe performance degradation and was not necessary in observed programs, it has been decided not to handle preemption for now.

This basically means that a program that never waits on any variable will never yield the execution to any other thread, and therefore that threads may starve. A simple example is for instance a non-lazy integer stream generator, or a function simply calling itself forever.

Such functions are usually not advised, but may be legitimate. In order to provide a temporary solution, Oz_T provides the `{Delay 0}` call. It is optimized to simply cause the current thread to `yield` without more overhead.

Chapter 4

Mapping Oz concepts to Truffle

In this chapter, we will consider how Oz concepts are mapped into Truffle. This mapping aims at being as simple as possible while respecting the semantics and avoiding corner cases.

The variables are first discussed, along with the local environment, and how the dataflow is handled. We then see how procedures are mapped in Mozart-Graal by assembling mechanisms handling their arguments, external environment, code and finally their calls.

Having those allows us to discuss how the base Oz types and the builtin procedures are implemented. The mapping the unification, equality testing and pattern matching is then addressed.

Finally, this chapter ends with the translation of tail calls and their detection at static analysis.

4.1 Oz Dataflow Variables

Dataflow variables do not exist by default on the JVM, but are at the core of the Oz language and its very particular semantics. Those variables can only be assigned once. Indeed, the first unification of a variable with a value will assign it. Further unifications will instead unify its value with the other member. This last mechanism will be discussed in Section 4.4.

Whenever an operation needs a value from an unbound variable, it will have to wait onto it until another thread binds it.

It is also possible to unify two unbound variables. In that case, both variables should be linked so that they become indistinguishable from each other. This means that binding one should bind them both, and any other variable bound with one of them.

```
local A B in
    thread {Wait B} {Show 'B is bound'} end
    A = B
    A = 3 % A and B are now have value "3"
    % "B is bound" is eventually printed
end
```

The natural way of implementing this linking mechanism between two variables is to set one to have the other as value. We will call the variable pointing to the other the *alias*, and the other the *representative*. When an operation is attempted on an alias, it must be performed on its representative in order for the effects to be perceptible by both. Considering the above case, assigning `A` to 3 or waiting on it would otherwise be unnoticeable through `B`.

It is possible to obtain bigger sets of equivalent variables. Because linking two sets of equivalent variables must always link one of the representatives to the other, by construction, any bigger set of equivalent variable only has one representative.

Considering `A` an alias for `B`, `D` a single variable to link with them, a single level of indirection can be achieved by linking `D` to `B`. It is however generally not the case. If `C` is introduced as an alias for `D`, linking `B` to `D` cannot achieve it because either `A` or `C` ends up with two levels of indirection. This is what is represented on Figure 4.2.

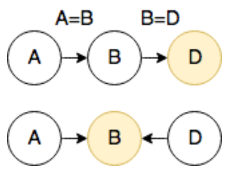


Figure 4.1: Only one level of indirection is possible when at least one variable is single

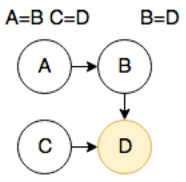


Figure 4.2: More indirection is however necessary for other cases

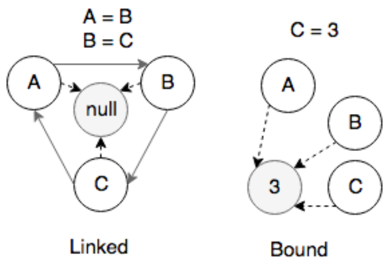


Figure 4.3: Binding linked variables in Mozart-Graal

If it cannot be guaranteed, it could still be valuable to short-circuit indirections when possible. So that when linking `A` to `C` for instance, both could directly get bound to the new representative. Or that when querying `A` after `B` has been linked to `D`, `A` is modified to point to `D` after the query has propagated through the link.

Union-Find [40] is a famous algorithm addressing this kind of problem. It allows merging sets and testing membership using this kind of hierarchy, each set being characterized by a representative. It has improvements for path compression and for deciding which representative should be linked to the other.

This is what is used by Mozart 1 and 2. If its optimized version has very attractive algorithmic properties when it comes to large sets, queries ineluctably end up performing recursive calls that are costly for the JIT compiler to optimize.

This is why in Mozart-Graal, linked variables form a circular linked list as illustrated by FIGURE 4.3 rather than a reversed tree. Because any of them provides references to all the others, all the n variables can get the same value as soon as one is assigned in $O(n)$. Querying a variable for its value or to know whether it is bound or not therefore becomes a simple $\Theta(1)$ test.

Linking two of them consists into merging their circular lists into one. This can be done by exchanging their *next* variables references, as illustrated in LISTING 4.1. Note however that performing this algorithm on two variables that already belong to the same list breaks it into two. This is why a search is first performed in order to ensure the variable are not already linked.

We will now refer to the size of the equivalence class of a variable as its *multiplicity*.

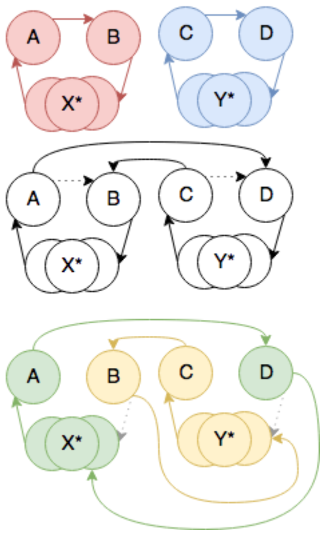
Because of that search to be performed, bringing a variable to a *multiplicity* n therefore has complexity $O(n^2)$. In order to evaluate whether this last constraint is inducing a large performance penalty, numbers have been collected about typical programs [18]. The resulting plot is shown on FIGURE 4.4. We can observe that this is quite rare for variables to have more than a *multiplicity* of 3. This is however possible to write pathological programs creating variables with arbitrary *multiplicity*. The `Flatten` function from the base library is such an example, as the depth of the list to be flattened determines the maximal *multiplicity*. A much simpler pathological program is provided in LISTING 4.2.

4.1.1 Handling the Local Environment

When executing a procedure, every variable declared into it occupies one slot in the current stack frame. Those slots are registered when translating variable declarations. For each unique identifier, Truffle provides a unique `FrameSlot`. Those `FrameSlots` are not shared between variable with disjoint scopes because they may contain type information allowing to store primitive types unboxed.

```
% A-B-X*-A
% C-D-Y*-C
B = A.next
A.next = C.next % A-D-Y*-C-D
C.next = B      % A-D-Y*-C-B-X*-A

% Pathological case, linking D and B already
  linked
% A-D-Y*-C-B-X*-A
Y = D.next
D.next = B.next % D-X*-A-D
B.next = Y      % B-Y*-C-B
```



Listing 4.1: Hazards of linking using circular lists (Colors denote equivalent variables)

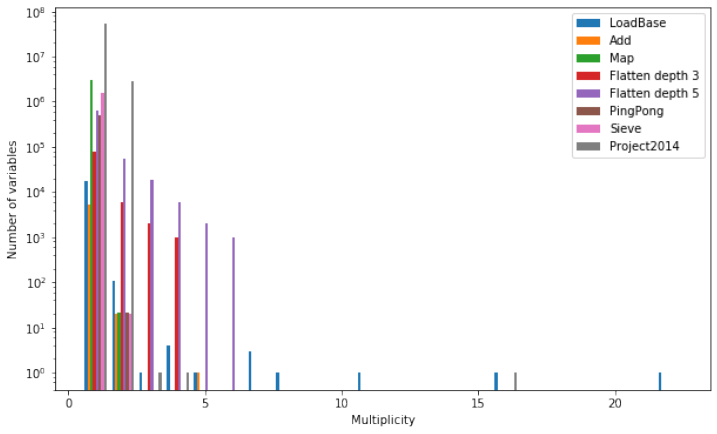


Figure 4.4: Multiplicity frequencies when variables get bound in sample programs

Renaming identifiers is therefore important because `Frames` are specific to an entire procedure execution. Two local variables with the same identifier would otherwise use the same slot, which is not correct in the case where one should be shadowing the other.

Each variable is instantiated and its reference put into the corresponding `FrameSlot` when evaluating its declaration. This is the job of a `LocalNode` implementing the `local ... in ... end` feature. The `LocalNode` can also clear every of its `FrameSlots` at the end of its execution, depending on how variables are captured, which will be discussed in Section 4.2.2.

All occurrences of identifiers in the code that are not declaring a variable will access those slots to get it back. This ends up in local variables being trivially initialized and retrieved using `InitializeVarNodes` and `ReadFrameSlotNodes` respectively, that are as simple as proposed in Listing 4.3.

4.1.2 Handling the Dataflow

There are different behaviors an AST node n evaluating a child providing a variable v can adopt, that are directly related to the semantics of the feature n implements:

- either n requires v to be bound, and uses its value or waits until there is one if it is unbound;
- either n performs different actions depending v being bound or unbound;

```
A = _
for I in 1..N do
   B = _ in
   A = B
end
A = 1
```

Listing 4.2: Pathological case for variable linking with complexity $O(N^2)$

```
public class InitializeVarNode extends OzNode {
    final FrameSlot slot;

    public InitializeVarNode(FrameSlot slot) {
        this.slot = slot;
    }

    @Override
    public Object execute(VirtualFrame frame) {
        frame.setObject(slot, new Variable());
        return unit;
    }
}

public class ReadFrameSlotNode extends OzNode {
    final FrameSlot slot;

    public ReadFrameSlotNode(FrameSlot slot) {
        this.slot = slot;
    }

    @Override
    protected Object execute(Frame frame) {
        return frame.getObject(slot);
    }
}
```

Listing 4.3: Truffle nodes instantiating variables and accessing them

- either n does not depend on v being bound or not and passes it to some other node.

Examples illustrating each case are provided in LISTING 4.4. The most complex case is the last one, where it is not defined beforehand what behavior is expected from the unification. It is important to have a practical way of dealing with those different cases. This is why two kinds of dereferencing nodes have been created:

- `DerefNode` that dereferences a variable, and thus waits if it has no value yet;
- `DerefIfBoundNode`, that dereferences a variable only if it is bound and otherwise lets it go through.

In the case the child evaluates to a value v, both nodes just let it go through, because it is equivalent to a bound variable with value v.

Those nodes must be placed between the feature and its child. Of course, passing a variable or a value around does not need such intermediate treatment. A feature requiring a value from its child just has to wrap it into a `DerefNode` and to provide specializations for value types it could receive. This is thus the `DerefNode` that performs the `Wait`.

For features that behave differently depending on the fact a child might evaluate to a variable that is bound or not, a `DerefIfBoundNode` can be put in-between. Providing a specialization for the `Variable` type will handle the case where it is unbound. All other types will handle the bound case, except the generic `Object` type that has to be guarded with `!isVariable(obj)` in order to make sure the instantiated specialization will not accept a variable accidentally.

The AST corresponding to the last statement of LISTING 4.4 makes use of those nodes and is available in FIGURE 4.5, along with its justification.

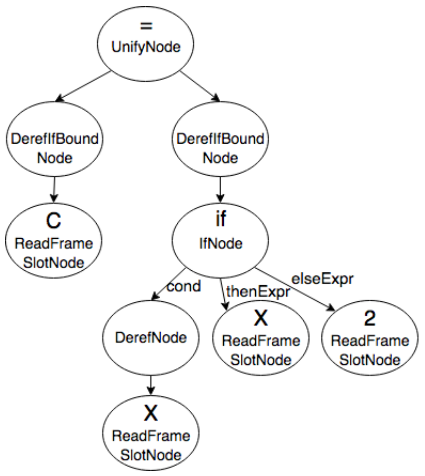


Figure 4.5: AST of last example from LISTING 4.4

`UnifyNode` handles both of its children symmetrically and performs differently depending on each being bound or not, so it needs two `DerefIfBoundNodes`.

`IfNode` waits on the condition in order to choose which child to evaluate, so it needs it to be wrapped into a `DerefNode`. It can then return the result of the adequate child as-is.

```
A = B            % unification performs different operations if A and B are
    bound or not, but never waits
{X A B C}        % waits on X (procedure to call) but provides A, B, C as
    arguments without special attention
{Show A==3}   % equality test will wait until A obtains some value (if a
    variable is unbound, the only way for it not to wait is for the other
    variable to be linked with it and unbound)

C =              % behaves differently whether C and the result of the
    right-hand side are bound or not
     if X then    % waits on X to know which branch to evaluate
        A          % evaluates A or and returns it as-is
     else
        2
     end
```

Listing 4.4: Examples of Oz nodes

In order to demonstrate the usage of this solution, the code of the `IfNode` is provided in LISTING 4.5. The complete `UnifyNode` is too complex to exhibit and takes into account other mechanisms, but a conceptual skeleton to illustrate how the different cases are separated is available in LISTING 4.6.

```
public class IfNode extends OzNode {
    @Child OzNode condition;
    @Child OzNode thenExpr;
    @Child OzNode elseExpr;

    public IfNode(OzNode condition, OzNode thenExpr, OzNode elseExpr) {
        this.condition = DerefNode.create(condition);
        this.thenExpr = thenExpr;
        this.elseExpr = elseExpr;
    }

    @Override
    public Object execute(VirtualFrame frame) {
        if ((boolean) condition.execute(frame)) {
            return thenExpr.execute(frame);
        } else {
            return elseExpr.execute(frame);
        }
    }
}
```

Listing 4.5: Truffle implementation for `IfNode`

```
@NodeChildren({ @NodeChild("left"), @NodeChild("right") })
public abstract class UnifyNode extends OzNode {
    // each of left and right children is wrapped into a DerefIfBoundNode

    /** The DSL will implement this method */
    public abstract Object executeUnify(Object a, Object b);

    // If an argument comes here as a variable, it is unbound
    @Specialization
    protected Object unifyUnboundUnbound(Variable a, Variable b) {
        // ... link variables a and b
        return a;
    }

    @Specialization(guards = "!isVariable(a)")
    protected Object unifyLeftBound(Object a, Variable b) {
        // ... bind variable b to value a
        return a;
    }

    @Specialization(guards = "!isVariable(b)")
    protected Object unifyRightBound(Variable a, Object b) {
        // ... bind variable a to value b
        return b;
    }

    @Specialization(guards = {"!isVariable(a)", "!isVariable(b)"})
    protected Object unifyValueValue(Object a, Object b) {
        // ... unify values further
        return b;
    }

}
```

Listing 4.6: Truffle skeleton for the `UnifyNode`

4.1.2.1 The Wait, WaitNeeded and WaitQuiet Primitives

What happens when an Oz thread "yields" is it basically passes the control to the next thread in the system. If all other threads come to yield or terminate, the control will come back and chances are that the current thread now fulfills the conditions to continue executing. If it is not the case yet, the thread can simply yield again, in the hope that some other thread will get unlocked in the next cycle.

The Oz `Wait`, `WaitNeeded` and `WaitQuiet` primitives are implemented as loops forcing the thread to yield execution until the variable they observe fulfills some condition. Those can be expressed from two booleans: `isBound` and `isNeeded`. `isBound` is set to `true` when unification provides the variable some value.

`WaitQuiet` will simply yield the execution endlessly until it sees `isBound` is set to `true`. `Wait` will first set the `isNeeded` attribute of the variable to `true` and then, just as `WaitQuiet`, yield the execution until it sees `isBound` is set to `true` too.

`WaitNeeded` also yields the execution until it sees `isNeeded` is set to `true`.

4.2 Procedures

Oz procedures are reusable routines that do not provide any return value. When calling one, we are thus more interested into its side-effects. An unusual fact about Oz is that its procedures can actually provide return values by side-effects, binding a return variable provided as argument to some result value.

A procedure basically consists in its own AST, its external environment and its arguments, that are provided at the call site. In Oz, the external environment of a procedure consists of the set of all observable variables at the place this procedure was declared. Here is an example to illustrate this:

```
local MyProc in
   local A B in
      proc{MyProc X Y}
```

```
            {Browse A#B#X#Y} % Use both external and local environments
         end
         B = 2
         A = 1
      end
      % A and B cannot be accessed here, but the procedure conserves them
      % into its external environment.
      {MyProc 3 4} % the call site => Browses 1#2#3#4
   end
```

We can see that A and B are still accessible by the body of the procedure, even though in practice, this code gets called outside of their scope.

4.2.1 Handling the Arguments

Truffle's frame contains the call arguments as an `Object[]` array. In Mozart-Graal, its elements are simply transfered from this array to the local slots at the very beginning of the procedure execution.

This is very convenient because they can then be handled as other local variables in the translation and when performing optimizations, therefore avoiding feature duplication. It also provides a place to store new arguments in the array while computing them in Section 5.1.1. This is thus not wasted space.

As a side note, the first slot of the arguments array will be reserved to the external environment. Because it is specific to procedure instances, it cannot be linked to AST directly.

4.2.2 Implementing the External Environment

The `Translator` maintains a hierarchy of environments while translating the AST generated by the static analyzer. This means it has access to all the information about the variables reachable by the statements it translates, and therefore knows whether a variable was declared in the current procedure or not. In the case a variable cannot be found locally, the `Translator` is able to find the environment it has been declared in.

At runtime, environments are composed of Truffle `Frame`s. If the classical `VirtualFrame` that is passed recursively between `execute` methods cannot be stored, the `MaterializedFrame` can be instantiated and passed around as any Java object. This is thus the perfect candidate for storing external environments.

There is however some freedom as to how to organize those. A first proposition consists in creating a linked list of frames to access the environments the parent procedures have been declared in. Both the depth in the list and the `FrameSlot` to reach any variable are known to the `Translator`, so this can be efficiently implemented.

The second and retained fashion is for each procedure to have a single `MaterializedFrame` that regroups just the required part of the external environment. It will be referred to as "variable selection" in the rest of this dissertation. When a procedure declaration is evaluated, the new frame is created containing all the variables it could need for later calls. The `Translator` must therefore instantiate nodes propagating interesting variables from procedure to procedure. Here is an example of the problem it must solve:

```
proc{X}
    A=1 B Y in
    proc{Y} % Along with this procedure, a frame containing A and B must be
        created (A must be propagated to Z's declaration)
       C=3 D=4 Z
    in
       B=2
       proc{Z E F} % Along with this procedure, a frame containing A and C
           must be created
          {Browse [A C E]} % Browses "[1 3 5]"
       end
       {Z 5 6}
    end
    {Y}
end
```

For comparison and understanding purposes, FIGURE 4.6 provides an illustration for both mechanisms. We are considering just having done one call of `X` so that variables and frames instantiated during this call can be uniquely identified.

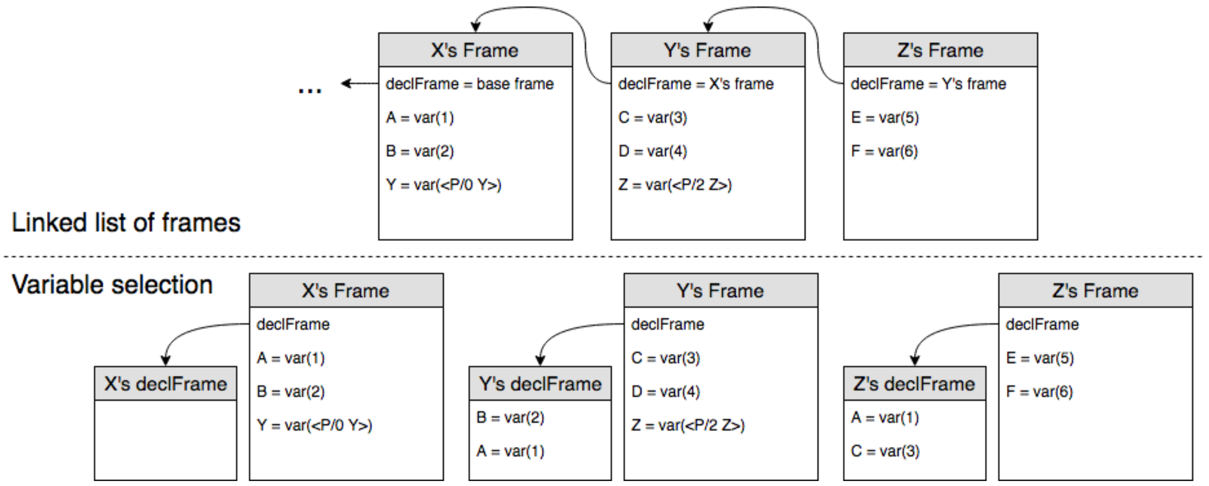


Figure 4.6: Illustration of both frame linked list and environment selection. Frames are represented as containing their arguments and slots for local variables

The second solution brings more control for memory management. For instance, it allows the implementation to clear all variables that are not needed anymore whereas in the other version, even though a procedure might free its local variables that are not captured, all captured variables need to be kept in memory as long as at least one reference to one of them exists.

This is caused by the fact there is no safe way to detect a procedure being garbage collected is the last one having references to some slots of the frame. Indeed, coming back to example FIGURE 4.6, we can see that if procedures `X` and `Y` were garbage collected but not `Z`:

- `D`, `E` and `F` would have been garbage collected because they never occur in any external environment, so the `LocalNodes` could have cleared them.
- `A` and `C` must be retained, because they are required in case of later call to `Z`.
- `B` could be easily collected in the second case, but not in the first because `Z` could still need Y's and X's frame constituting its external environment, as they contain `A` and `C`. The only way to garbage collect `B` would have been to track that `Y` was the last procedure needing it and it was just collected.

For single variables containing integers, the problem would not be major, but it could also happen the developer hopes to loose the head of an infinite stream as soon as possible. Most setups would have motivated this choice, another example being several procedures declared in the same frame and capturing different variables. The only "advantage" of the linked list is in fact that it only requires cloning the current stack frame containing local variables into a `MetarializedFrame`. This however comes at the cost of recursive lookups for accessing them later.

4.2.3 Storing the Code

When the `Translator` encounters a procedure declaration, it translates its body and wraps the resulting AST into a new root node. Truffle provides its own `RootNode` class that allows those independent AST to allocate their own stack frame before being evaluated. Instances of `RootNode` are the nodes Graal uses to delimit compilable units.

Truffle requires its `RootNodes` to be called through the `CallTarget` interface, of which an instance can be easily created using `Truffle.getRuntime().createCallTarget(rootNode)`. This ends up being what is stored in the `OzProc` objects along with the external environment and the arity.

4.2.4 Performing Calls

Oz calls can now be implemented by evaluating the receiver, the arguments, prepending the external environment to them, and perform the actual call using the `CallTarget`'s `call(Object[])` method. A basic implementation is provided in Listing 4.7.

Listing 4.7: A Truffle call node for Oz procedures

```java
@NodeChildren({ @NodeChild("receiver"), @NodeChild("arguments") })
public abstract class CallNode extends CallableNode {
    // The receiver must be dereferenced, the arguments can be passed as-is.

    public abstract Object executeCall(VirtualFrame frame, Object receiver,
            Object[] arguments);

    @Specialization
    protected Object callProc(VirtualFrame frame, OzProc proc, Object[] args) {
        assert args.length == proc.arity;
        Object[] arguments = new Object[args.length+1];
        arguments[0] = proc.declarationFrame;
        System.arraycopy(args, 0, arguments, 1, args.length);
        return proc.callTarget.call(arguments);
    }
    // Other specializations will come later
}
```

4.3 Data Types, Data Structures and Their Builtins

Oz has few primitives to build all kinds of applications. Its data types have been mapped towards Java types without exposing their operations to Oz directly. Instead, those operations are implemented as Oz builtins, base procedures shipped with the implementation. Those will be discussed at the end of the section.

Please note that the type hierarchy from `https://mozart.github.io/mozart-v1/doc-1.4.0/tutorial/node3.html` is not completely respected yet.

4.3.1 Numbers

Oz integer numbers have arbitrary precision and are signed. In Mozart-Graal, they are represented both Java's `long` primitive and the `BigInteger` class.

It is not guaranteed that an integer fitting into 64 bits will always be represented as a `long`. First, because operations on `BigInteger`s always return a `BigInteger` even though it may fit into 64 bits. Second, because once an AST node in the program has been subject to a `long` overflow, it is rewritten in order to always use `BigInteger`s.

Oz floating points numbers are implemented using the Java `double` primitive.

4.3.2 Atoms

Oz atoms are unique symbols that can be compared by reference. They have been mapped to interned Java `String`s.

4.3.3 Booleans

Oz booleans have directly been mapped to the Java `boolean` primitive.

4.3.4 Records

Records are the base data structures to group variables in Oz [33]. Each record consists of a label, a set of features to which contained variables will be associated, and those variables.

They have been mapped to Truffle's `DynamicObjects`, which are described in [41].

To summarize the interesting concepts for Oz, Truffle comes with an API for handling dynamic objects. Those are instances of a traditional Java class having a `Shape` field and some `long` and `Object` placeholder fields. That class, sketched on Listing 4.8, thus serves as a container class. The shape determines what features the dynamic object exposes, and where to find them in the placeholder fields.

Listing 4.8: Excerpt from the `DynamicObjectBasic` placeholder class from Truffle

```java
public class DynamicObjectBasic extends DynamicObjectImpl {
    @DynamicField private long primitive1;
    @DynamicField private long primitive2;
    @DynamicField private long primitive3;
    @DynamicField private Object object1;
    @DynamicField private Object object2;
    @DynamicField private Object object3;
    @DynamicField private Object object4;
    private Object[] objext; // if more than 4 object features
    private long[] primext; // if more than 3 primitive features
    // ...
}
```

Shapes with arbitrary features can be built by adding features one after the other, creating immutable intermediate shapes. Those "feature additions" are kept in memory so that equivalent shapes constructed in the same order always have the same reference. In Mozart-Graal, shapes are constructed by adding features in the lexicographical order. This means arity equivalence can be ensured by shape equality.

Given a shape, a `DynamicObject` can then be created and its features populated and accessed. Adding and removing features directly on the object would be possible, but has no use in Oz where records are immutable. Records are simply implemented as `DynamicObjects` with a label field and specific features.

The benefits for Mozart-Graal are that they are transparent to Truffle, provide an efficient way to handle record arity, and that they have potential in storing unboxed primitives. This last point however requires some more efforts to be effective in Mozart-Graal.

4.3.5 Lists

Oz lists are linked lists, in which each node is:

- either a record of the form `'|'(A B)` where `A` is the value of the node and `B` the next node.
- or the `nil` atom (the empty list).

Storing those "*cons*" records would however cause a lot of space to be wasted. Indeed, most of the placeholder fields of the `DynamicObjects` would be unoccupied since just the "|" label, the value and the *next* pointer need to be stored. In order to deal with this problem, an `OzCons` type has been created with just 2 fields for the value and the *next* pointer, the label being the same for every such record.

4.3.6 Chunks

An Oz chunk is built from a record but only provide restricted access to it. That is, `Arity` and `Label` operations on it are unavailable, and unification between chunks is prohibited. This is implemented through the `OzChunk` class wrapping a record.

4.3.7 Classes and Objects

Oz classes are chunks with specific fields providing the method table and describing attributes and features of its instances. Attributes are stateful cells specific to the instances while features are immutable variables.

Objects are implemented within the `OzObject` class, having record fields for their classes, attributes and features.

The most interesting feature is probably the call that can simply be implemented as an additional specialization to our `CallNode` from Listing 4.7. It is also a good example of how core language features are reused across the interpreter.

Listing 4.9: A specialization for handling calls to Oz object methods

```
@NodeChildren({ @NodeChild("receiver"), @NodeChild("arguments") })
public abstract class CallNode extends CallableNode {
    // ...

    @Specialization
    protected Object callObject(VirtualFrame frame, OzObject self, Object[] args,
            // Nodes specific to this specialization
            @Cached("create()") LabelNode labelNode,
            @Cached("create()") DerefNode derefNode) {
        assert args.length == 1;
        // Get the "methods" record of the class
        DynamicObject methods = (DynamicObject) self.getClazz().get(ooMethUniqueName);
        Object message = derefNode.executeDeref(args[0]); // deref the unique parameter

        // get the associated procedure
        OzProc proc = methods.get(labelNode.executeLabel(message));
        if (proc == null) { // if not found, fallback on otherwise
            proc = methods.get("otherwise");
            message = OTHERWISE_FACTORY.newRecord(message); // 'otherwise'(message) record
        }

        return proc.callTarget.call(proc.declarationFrame, self, message);
    }
}
```

4.3.8 And Much More

Arrays Oz arrays are mapped to our `OzArray` class, embedding both a Java `Object[]` array and an integer specifying the lowest index.

Dictionaries Oz dictionaries have been mapped to our `OzDict` class, that simply extends the Java `HashMap` class.

Cells Oz cells are instances of the `OzCell` class having one non-final field for the current value.

There are still a lot of types to be detailed, but it would not be interesting to enumerate them. Some are only defined in the base library forked from Mozart 2, like Locks that are relying on Cells. The others such as Ports or FailedValues tend to be mapped to trivial Java classes, the only complexity being put into their builtins.

4.3.9 Oz Builtins

Until now, all those types are thus shallow boxes to Oz and would be useless if they could not be operated on. This is were builtins come into account, providing the core features of the language. As for the types, some of them were implemented in plain Oz in Mozart 2 and have been forked as-is. The other part, intrinsic to the virtual machine, has been reimplemented as procedures directly executing a Truffle node written in plain Java. As a side note, operators are implemented as intrinsic builtins. Rather than generating a call however, the `Translator` directly inlines those in the AST.

All the builtins are loaded by the base functor in `lib/main/base/Base.oz`. It loads intrinsic builtins from their `x-oz://boot/*` *boot* URLs. To each such *boot* functor (e.g. `Number`) corresponds a Java class (e.g. `NumberBuiltins`) defining its intrinsic builtins (e.g. the "+" operator) implemented as Truffle AST nodes (e.g. the `AddNode` static class). The base functor then imports all the non-intrinsic builtins files in `lib/main/base/*` and exposes all the core builtins of the language.

An excerpt from the `NumberBuiltins` class showing the code of the `AddNode` is available in Listing 4.10. In order for builtins to be more fluent to specify, the `@Builtin` annotation has been created. It allows to specify which of the arguments should be wrapped into `DerefNodes` or `DerefIfBoundNodes`, using its `deref` and `tryDeref` attributes respectively. The value of the `name` attribute specifies the functor feature it will be accessed at. By default, an extra parameter is also added to the Oz procedure for it to be set to the result of the function. Setting the `proc` attribute of the annotation to `true` will remove such mechanism and therefore ignore the return value.

The rest of the code perfectly reflects what had been discussed in Section 2.2.

Listing 4.10: The `AddNode` builtin, corresponding to the "+" operator

```java
public abstract class NumberBuiltins {
    // ...

    @Builtin(name = "+", deref = ALL)
    @NodeChildren({ @NodeChild("left"), @NodeChild("right") })
    public static abstract class AddNode extends OzNode {
        @Specialization(rewriteOn = ArithmeticException.class)
        long add(long a, long b) {
            return ExactMath.addExact(a, b);
        }

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

        @TruffleBoundary
        @Specialization
        BigInteger add(BigInteger a, BigInteger b) {
            return a.add(b);
        }
    }

    // ...
}
```

4.4 Unification, Equality Testing and Pattern Matching

Unification, pattern matching and equality tests are closely related. Mozart-Graal does however not interleave their code, in order to take advantage of every optimization opportunity. Some specialization of one may however make use of the other concepts.

4.4.1 Unification

Unification between two terms tries to make them equal and binds or links unbound variables in order to do so. The operation is symmetric and can create cyclic structures. Whenever the two terms are not unifiable, an exception is raised. The following listing provides some examples to grasp the ideas:

```
% Simplest case: Binding a variable to a value
A = person(name:X1 age:X2)

% Unification is symmetric
f(X Y) = f(1 2)
f(X 2) = f(1 Y)
f(1 2) = f(X Y)

% Unification can create cyclic structures
A = '|'(a A)

% Unification sometimes simply fails
a = b
person(name:X1 age:26) = person(name:"George" age:25)
```

As stated in [33, p. 101], a failing unification may have side-effects onto its terms because the modifications occurring before the failure do not have to be undone. Considering the last example, `X1` could either end up being bound to `"George"` or not, depending on the order of the unification.

Unification can be implemented as a depth-first traversal of the structures performing the following operations on both terms `A` and `B` recursively:

- if `A` and `B` are two unbound variables, link them;
- if one of them only is unbound, bind it to the value of the other;
- if both are bound to records, ensure they have the same arity and unify the labels and stored variables;
- otherwise, those are other kinds of objects test for equality of both terms.

This algorithm is heavily inspired by what is suggested in [33, p. 104]. The watchful reader could notice that if this is described in terms of variables for the sake of brevity, values are treated equivalently to bound variables by the implementation because of the use of `DerefIfBoundNodes`, as described in Section 4.1.2. The depth-first traversal must be implemented under the form of an explicit stack in order to be able to deal with arbitrarily deep structures, and the cycle detection mechanism from the book must also be added in order to handle the unification of cyclic structures without looping forever in cases as those:

```
% Those cyclic structures are correct to unify
A = '|'(1 '|'(2 A))
B = 1|2|1|2|B
A = B

% Those should fail to unify
C = r(C c)
D = r(D d)
C = D
```

This basically implies to maintain all the pairs of variables that have been or are unified in the process, and not to perform unification again on those. A little adaptation is necessary since values are stored unboxed when possible. All pairs of unified terms are hence stored as long as at least one is a variable, before trying to dereference them. This stays exhaustive because variables are the only way to create cycles.

4.4.2 Testing for Equality

Equality testing can be achieved in a very similar fashion. Both `A` and `B` terms are walked in parallel with an explicit stack, and the following rules are applied:

- if `A` and `B` are bound to records, check if their arities and labels are the same, then test every of their stored variables for equality. If everything is equal, return true;
- if `A` and `B` are unbound, return true only if they are linked, otherwise wait and test again later for equality;
- if only one of `A` and `B` is unbound, wait for it to be bound and then return the equality of both values;
- otherwise, both are bound to non-record objects. Apply some specific specialization depending on their types.

The fact equality may have to wait comes from the fact it must return `true` or `false` as definitive result. A cycle detection mechanism is also required, because the following code is plausible:

```
% Browses true
A = '|'(1 '|'(2 A))
B = 1|2|1|2|B
{Browse A == B}

% Browses false
C = r(C c)
D = r(D d)
{Browse C == D}
```

and would loop forever otherwise. This is done in the very same way as for unification.

4.4.3 Pattern Matching

Pattern matching in Oz allows for succinctly branching depending on the value of some target variable. Let us demonstrate its expressiveness with some fictive function for checking some preconditions on records describing operations:

```
Zero = 0.0
fun {IsErroneous Operation}
   case Operation
      of add(X Y Z ...)          then true % too much arguments (>= 3)
      [] 'div'(X 0)              then true % division by 0
      [] log(!Zero)              then true % log of 0
      [] log(Y) andthen Y < 0.0  then true % log of negative number
      else false
   end
end
```

The whole `case ... of ... end` is made of several clauses and only executes the body of one of them. A clause is composed of a pattern, an optional guard and a body. They are tested sequentially until one has both a pattern matching the target and a guard evaluating to `true`. Its body is then executed, and all subsequent clauses are ignored. If there is no such matching clause, the body of the `else` is evaluated.

Each clause can therefore be translated as an `IfNode` whose condition is made of a *matcher* AST node, followed by the optional guard translated as-is. The only specific features for handling pattern matching thus resides in the *matcher* AST obtained from the *pattern*. The following rules are followed to generate it:

- constants such as booleans, numbers, atoms are translated as nodes testing equality between their value and their target;
- *capture* variables (such as `X` in `'div'(X 0)`) are fresh new variables unified with the value from the equivalent target. This is most used for obtaining the variable and using it in the body or the guard;
- *escaped* variable patterns, with a leading exclamation mark "!", are referring to variables from the local environment. Each is thus translated to a node testing the equality of its variable and its target;
- a record pattern must have constant label and features, and has pattern for the values. It is translated to a node ensuring the label and the arity of the target are the ones expected, and embedding matcher nodes for each feature;
- an *open* record pattern, ending with "...", will work in the same way but accept records having more features than those specified in the pattern.

So pattern matching ends up relying mostly on the equality testing and a bit on unification for capture variables. The construction of cyclic patterns is also achievable, but only through the use of *escaped* variables. Since they are handled by equality testing nodes, nothing more specific has to be done.

4.5 Tail Call Optimization

Tail calls can be optimized in order to avoid growing the stack unnecessarily. The basic idea is that if a call is the last statement of a procedure, then its stack frame can be released before executing the call. There will be no need to access anymore.

In Oz, the tail calls are optimized, and are even the only way to obtain a loopy behavior. On the JVM, in contrast, tail calls are not optimized for tracing reasons, as the designers wanted the stack trace to be explicit and loops are available.

Listing 4.11: Example of how tail calls behave

```
declare
proc{A} {B} 0=0 end     % Not a tail call
proc{B} {C} end         % Well a tail call
proc{C} true=false end
{A}
%***************************** failure *********
%**
%** Tell: true = false
%**
%** Call Stack:
%** procedure 'C' in file "Oz", line 4, column 5
%** procedure 'A' in file "Oz", line 2, column 5
%**---------------------------------------------
```

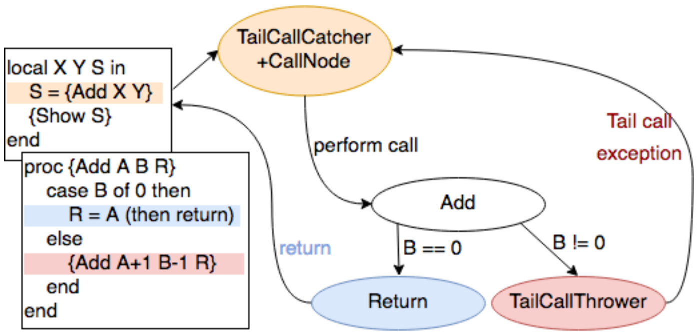


Figure 4.7: Tail call optimization control flow example

The sample code in LISTING 4.11 illustrates how debug information is lost because of this optimization. We can see `A` calls `B`, in a non-tail fashion, and `B` then calls `C` as last statement. This means `B`'s stack frame can be dropped to let the place to `C`. This ends up in `B` not being shown in the failure trace, hence the `C` call seems to come from nowhere. One can therefore perceive the motivation for letting tail calls unoptimized.

4.5.1 Expressing Tail Calls in Java

The first mechanism we have implemented to handle tail calls is very general, using Java exceptions. The principle is to wrap every non-tail `CallNode` (from LISTING 4.7) in the Oz program to be able to perform new calls upon `TailCallExceptions`. Those exceptions must describe the subsequent calls completely, i.e. provide a receiver and arguments. An illustration of how those components interact is provided in FIGURE 4.7.

It is necessary to wrap every non-tail call because we do not know if the original callee could throw an exception, and it would not be correct to let it propagate further since by definition, there are still statements to be executed after the non-tail call node. Tail call nodes, in contrast, will all simply throw a `TailCallException` after having evaluated its receiver and arguments nodes to fill it.

It turns out Truffle provides a class named `ControlFlowException` it has special knowledge of for performance optimizations, and that does not create any stack trace when instantiated. Our `TailCallException` therefore extends it. The following listing provides a way of implementing it:

```java
public class TailCallException extends ControlFlowException {
    public final Object receiver;
    public final Object[] arguments;

    public TailCallException(Object receiver, Object[] arguments) {
        this.receiver = receiver;
        this.arguments = arguments;
    }
}
```

Non-tail `CallNode`s then have to be wrapped into `TailCallCatcherNode`s, and tail `CallNode`s can be replaced by `TailCallThrowerNode`s. A sketch of both classes is provided in Listing 4.12. Now we know how to translate our tail and non-tail calls, the only thing left to do is to tag them at static analysis.

Listing 4.12: Truffle implementation of the tail call mechanism

```
/** Put in place of every non-tail call */
class TailCallCatcherNode extends OzNode {
    @Child CallNode callNode;

    public TailCallCatcherNode(CallNode callNode) {
        this.callNode = callNode;
    }

    Object execute(Frame frame) {
        try {
            return callNode.execute(frame); // First call with its own receiver and arguments
        } catch(TailCallException e) { // First TailCallException comes in
            Object receiver = e.receiver;
            Object[] arguments = e.arguments;
            while(true) {
                try {
                    // Perform calls as long as execptions TailCallException are raised
                    return callNode.executeCall(frame, receiver, arguments);
                } catch(TailCallException e2) {
                    receiver = e2.receiver;
                    arguments = e2.arguments;
                }
            }
        }
        return unit;
    }
}

/** Must be put in place of the tail CallNodes */
class TailCallThrowerNode extends OzNode {
    @Child OzNode receiver;
    @Children OzNodeArray arguments; // will evaluate to an array of values

    Object execute(Frame frame) {
        throw new TailCallException(receiver.execute(frame), arguments.execute(frame));
    }
}
```

4.5.2 Detecting Tail Calls

In this part, we will construct the algorithm that has been implemented in Mozart-Graal's compiler to detect tail calls using the Visitor design pattern in Scala [42]. For conciseness, codes presented here will simply instantiate new nodes implemented using Scala *case classes*.

First, let us consider blocks of several statements and `local` blocks. The idea is simply to allow the algorithm to go inside the last statements recursively, tagging `CallStatement`s as *tail* if some are reached. Other statements are left as-is.

```
proc {X Arg}
   0 = 0 % Some dummy statement
   {SomeProc} % Our tail call
end
```

```
def markTailCalls(statement: Stat): Unit = statement match {
    case call @ CallStatement(receiver, args) =>
        call.kind = CallKind.Tail

    case LocalStatement(decls, stat) =>
        markTailCalls(stat)
    case CompoundStatement(stats) =>
        markTailCalls(stats.last)

    // Insert next cases here

    case _ =>
}
```

A first improvement would be to explore into branches. It turns out their last statements can simply be marked independently since the control flow will only go into one or the other, so any of them would be the last statement if executed.

Lastly, we consider the exception handling nodes among which only the last calls of the `catch` body can be optimized. Indeed, the `try` must remain able to catch exceptions from any statement in its body, including the last one. The `try ... finally ... end` structures are desugared beforehand into handled structures [33, p. 95], hence requiring no special treatment.

```
case IfStatement(condition, trueStatement, falseStatement) =>
    markTailCalls(trueStatement)
    markTailCalls(falseStatement)

case MatchStatement(value, clauses, elseStatement) =>
    clauses.foreach (_ match {
    case MatchStatementClause(pattern, guard, body) =>
        markTailCalls(body)
    })
    markTailCalls(elseStatement)

case TryStatement(body, exceptionVar, catchBody) =>
    markTailCalls(catchBody)
```

However, this is only part of the solution since it does not work at all for functions. Indeed, the initial choice was to fully exploit the object returned by Truffle's `execute` method, therefore translating functions like Listing 4.13 into procedures as showed in Listing 4.14, by simply unifying the result variable with the whole function body. However, call expressions are desugared by introducing a variable, binding it with the call, and then returning it. The problem with this method is that the call is performed first to later unify `R` to its result.

This is not the case for Listing 4.15 that could directly be optimized by the algorithm just described. Pushing the unification of the result variable R as deep as possible will lead to the desired AST and has a nice generalization for nested expressions. In Mozart-Graal, this is the achieved thanks to the `exprToBindStatement` method called while desugaring function declarations into procedure declarations, much earlier than tail call detection. It takes a variable and an expression as parameters and returns a statement equivalent to the unification of both but making sure it happens as late as possible.

Listing 4.13: Tail function code

```
fun {MyFun}
   if true then
      {MyFun}
   else
      b
   end
end
```

Listing 4.14: Output of bootcompiler

```
proc {MyFun R}
   R = if true then
      local R1 in
         {MyFun R1}
         R1
      end
   else
      b
   end
end
```

Listing 4.15: Desirable output

```
proc {MyFun R}
   if true then
      {MyFun R}
   else
      R = b
   end
end
```

Some cases are still missed. It is very frequent for tail-recursive functions to call themselves within records for instance. Furthermore, it also happens for `andthen` and `orelse` expressions to be expected to be optimized. Listing 4.17 provides an example of two functions that should be made tail-recursive.

The expected behavior of the algorithm is rather obvious for the `andthen` and `orelse` operators. The compiler can basically replace them with branches. Extracting tail-calls from records, however, does not seem that well defined. Hopefully, the Core syntax feature of Mozart makes it easier to understand how it should work:

Listing 4.16: exprToBindStatement handling expressions having equivalent statements

```
def exprToBindStatement(expr: Expr, v: Var): Stat = expr match {
    case CallExpression(receiver, arguments) =>
        // case relative to the dollar placeholder is omitted for convenience.
        // in:  R = {A X Y ...}
        // out: {A X Y ... R}
        CallStatement(receive, arguments :+ v)

    case IfExpression(cond, trueExpression, falseExpression) =>
        // in:  R = if A then X else Y
        // out: if A then R=X else R=Y
        IfStatement(cond, exprToBindStatement(trueExpression, v),
                    exprToBindStatement(falseExpression, v))

    // Other cases for StatAndExpression, LocalExpression and MatchExpression
    // More cases to come here

    case _ => BindStatement(v, expr)
}
```

Listing 4.17: Cases of tail functions that are still undetected

```
fun {ShortCircuitOpTailFun X}
   if X then
      true andthen {ShortCircuitOpTailFun {Not X}}
   else
      false orelse {ShortCircuitOpTailFun {Not X}}
   end
end

fun {RecordTailFun}
    someAtom | {RecordTailFun} % '|'(someAtom {RecordTailFun})
end
```

```
fun {SomeTailFun}
   {SomeTailFun}#a
end

fun {SomeNonTailFun}
   {SomeNonTailFun}#local X in a end
end
```

```
proc {SomeTailFun Result1}
   local RecordArg1 in
      Result1 = '#'(RecordArg1 a)
      {SomeTailFun RecordArg1}
   end
end

proc {SomeNonTailFun Result2}
   local RecordArg2 RecordArg3 in
      Result2 = '#'(RecordArg2
          RecordArg3)
      {SomeNonTailFun RecordArg2}
      local X in
         RecordArg3 = a
      end
   end
end
```

The principle we can infer from the experiment is that if there is any suspicion for a node to have side effects, it will be evaluated after the record has been created. The left-to-right order is conserved when evaluating the considered nodes.

In our case, only the last expression with side-effects interests us, so we can leave the others in-place. The following code also provides insights on how nested records are handled. This is indeed interesting to see which call is optimized:

```
fun {SomeTailFun A B}
   a({A} b({B}) someAtom)
end
```

```
proc {SomeTailFun A B R}
   local RecordArg1 RecordArg2 in
      R = a(RecordArg1 b(RecordArg2)
          someAtom)
      {A RecordArg1}
      {B RecordArg2}
   end
end
```

This is thus the rightmost suspicious expression that will become the last statement. This is where `exprToBindStatement` becomes useful again. The result variable `R` will be bound to the main record, and the last suspicious expression in it will be bound to another variable that will become the new result variable for the next statement. This thus provides a recursive way of extracting tail calls from expressions.

Listing 4.18: Example of result record partial unwrapping

```
% last side-effect in gray
Bind R to record a({A} b({B}) someAtom)
=>            R = a({A} R1      someAtom) % replaced by a variable
  Bind R1 to record b({B})
  =>            R1 = b(R2 ) % The process is recursive
    Bind R2 to call {B}
    =>  R2 = {B} => {B R2}
```

Here is thus the missing part of `exprToBindStatement`:

```
case ShortCircuitBinaryOp(left, "andthen", right) =>
    // in:   R = (A andthen B)
    // out: if A then R=B else R=false end
    IfNode(left, exprToBindStatement(right, v), BindStatement(v, False()))

case ShortCircuitBinaryOp(left, "orelse", right) =>
    // in:   R = (A orelse B)
    // out: if A then R=true else R=B end
    IfNode(left, BindStatement(v, True()), exprToBindStatement(right, v))

case Record(label, fields) =>
    var tailExpression: Option[(Variable, Expression)] = None
    // Replace last suspected field if any
    val newFields = fields.reverse.map { field =>
        if (tailExpression.isEmpty && hasSideEffects(field)) {
            val newVar = Variable()
            tailExpression = Some((newVar, field.expression))
            treeCopy.RecordField(field, field.feature, newVar)
        } else {
            field
        }
    }.reverse
    // Bind and try to make tail
    if (tailExpression.isEmpty)
        return BindStatement(v, expr)
    val (newVar, value) = tailExpression.get
    LocalStatement(Seq(newVar), CompoundStatement(Seq(
        BindStatement(v, Record(newFields)),
        exprToBindStatement(newVar, value))))
```

Chapter 5

Specific Optimizations

In the previous chapter, we have mapped Oz concepts to Truffle without paying much attention to performance. We will here propose several specific optimizations for increasing performance and memory efficiency.

The first part will require additions the static analysis to avoid variables, optimize their retention and optimize *self* tail calls.

By contrast, the second part only relies on the Truffle API to help Graal perform inlining and control flow optimization as soon as possible. We also make use of self-optimizing AST interpreters principles to propose more efficient mechanisms for unification and equality tests.

5.1 Static Optimizations

In this section, add information at static analysis to perform some optimizations. We first particularize the mechanism for tail calls to *self* tail calls. We then see how it can help Mozart-Graal avoid instantiating dataflow variables when they are excessive, and free references as soon as possible.

5.1.1 Self Tail Calls Optimization

Some tail calls from a procedure to itself can be statically optimized. Indeed, after having provided every variable a unique identifier, we are sure that calls in a procedure whose receiver is the identifier of the procedure itself are "self" calls. There is nothing to gain in the case of non-tail calls, because we will want a new stack frame to compute things that will be useful when coming back at the statement following the call.

But in the case those calls are also tail, this means the current stack frame could be completely recycled. After having computed the parameters of the call, none of the slots of this execution will be used again. The arguments can be changed in-place in the frame at the call site, and we can jump again at the beginning of the procedure, putting arguments in the `FrameSlot`s, etc.

The mechanism has been implemented as the one for generic tail calls, using a `SelfTailCallException`, but not carrying any data anymore since the receiver is already known, and the arguments are written in the frame by the "call" node before it throws the exception. This exception is then caught by a `SelfTailCallCatcherNode` right under the `RootNode` of the procedure, which simply re-evaluates the latter's body. This is illustrated in Figure 5.1.

5.1.2 On Stack Variables

Not every variable is useful to be instantiated. If this is already the reason why we try to conserve nested expressions as-is, it is possible to go further. Since local variables have their own `FrameSlot` and values could as well be stored in those slots, just as they can flow anywhere in the implementation because indistinguishable from bound variables, there are cases where instantiating a variable and performing

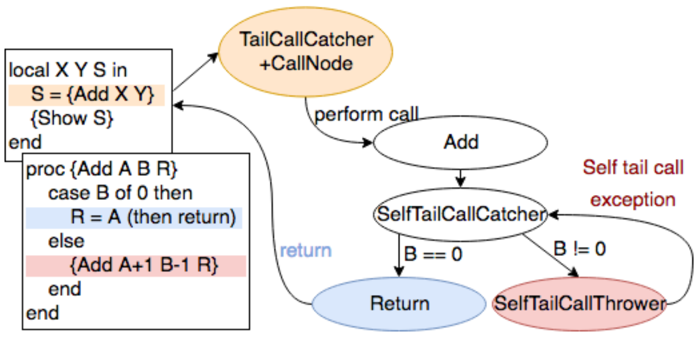


Figure 5.1: Self tail call optimization illustration

unification are avoidable. In those cases, the value can simply be put into its `FrameSlot`. This is what we will call "on stack" variables, because it will not instantiate any variable on the heap.

In order to understand better the potential of this optimization, the first trial was to instantiate variables lazily. The `LocalNode` would thus only have to clear the slots it introduced at the end of its evaluation. The `ReadFrameSlotNode` would instantiate variables when trying to access it in the case it encounters an empty slot, store it into the `FrameSlot` and return it to the node needing it. However, some "pure assignment" cases have to be adapted to truly put values in slots.

First, pattern capture variables can systematically be put on the stack, so this is done by the matcher without any detection mechanism.

Second, a unification expression with a variable on the left is a good candidate for such a "pure assignment", so this is the second case that has been detected and optimized. Again, this could be done dynamically, by having the unification node modified in case its left member was an identifier, looking if its `FrameSlot` was set or not and directly putting the result of the right member evaluation in the slot if not.

One could wonder whether it stays correct when the right member is not bound. In this case, the unbound variable is put in the `FrameSlot`, and it works just as if a unification had been performed. The two variables are definitely "indistinguishable" from each other in later accesses since both slots contain the same reference.

The principle is thus that a variable is instantiated only if it is used before occurring as the left-hand side of a unification. As a side note, this optimization covers all the cases for which we found nested expressions desirable. If for each of those expressions, a variable had been declared and put as left-hand side of a unification with the expression itself as right-hand side, the value/other variable would have been put on the stack every time. It would however allocate a lot of additional `FrameSlots`.

So such a solution seems appropriate and covers a whole set of avoidable variables. Since most of them can be detected at static analysis, it has been attempted by making a choice at the level of the `LocalNode`. The following algorithm has therefore been implemented:

```
For every local node:
    For every statement/expression in it:
        For every variable occurrence, if it is still undecided:
            The variable is being bound (V = value) =>
                decide for it to be put on stack (and tag the bind statement
                    to do so)
            If the variable is used in any other manner =>
                decide for it to be instantiated
    In case of branching, avoid instantiation only if every branch binds it on
        stack.
```

This static solution is not necessarily worse than the dynamic one. Indeed, the latter introduces some more checks at runtime, and uncertainty when accessing the variables can appear. Fine-grained static analysis spawning variables only at their first use can also introduce more checks because of uncertainty. Those cases can be illustrated by the following example, where `DerefIfBoundNodes` and `DerefNodes` would have to specialize for both possibilities:

```
for I in 1..10 do
```

Listing 5.1: Truffle nodes for handling *self* tail calls

```java
/**
* Wraps the body of a procedure in order to re-execute it
* upon SelfTailCallException.
**/
class SelfTailCallCatcherNode extends OzNode {
    @Child OzNode body;

    Object execute(Frame frame) {
        while(true) {
            try {
                return body.execute(frame);
            } catch(SelfTailCallException e) {
                // Nothing to do, just re-execute body
            }
        }
        return unit;
    }
}

class SelfTailCallThrowerNode extends OzNode {
    @Children OzNode[] arguments;

    @ExplodeLoop
    Object execute(Frame frame) {
        Object[] frameArguments = frame.getArguments();

        // It is safe, no node here uses arguments since they are only read at
        // the very beginning of the procedure execution to fill FrameSlots
        // frameArguments[0] is however the declarationFrame of the procedure

        for (int i = 0; i < arguments.length; i++) {
            frameArguments[i+1] = arguments[i].execute(frame);
        }
        throw SelfTailCallException.INSTANCE;
    }
}
```

```
    A B
in
    if {Choose} then
        A = 0 % Binds A on stack
    else
        {Set A} % Requires A to be assigned
    end
    B = A + 3 % Any code using A here will have to be able to be able to
       handle both variables and values
end
```

This is thus all a matter of trade-offs, and the consequences are almost imperceptible. The `LocalNode`-level decision has been implemented prior to those considerations and as there does not seem to be any strictly winning option between them, this is what remains in Mozart-Graal as of this writing.

5.1.3 Frame Slot Clearing

For now, a `LocalNode` introduces its variables at the beginning of its execution and clears their slots after the execution of its body. It seemed useful to try to reset the slots in the frame as soon as their associated variables would not be used by the procedure anymore, allowing the garbage collection to collect them if no other reference is maintained. Let us illustrate that with the following example:

Listing 5.2: Example benefiting from early `FrameSlot` clearing

```
proc{Produce I L}
   {Delay 0} % Because the scheduling is not fair at that time
   case I of 0 then
      L = nil
   else L2 in
      L = I|L2
      {Generate I L2}
   end
end

proc{Sum L Acc0}
   case L of nil then
      Acc0
   [] LH|LT then
      {Sum LT Acc0+LH}
   end
end

local H in
   thread {Generate 100000 H} end
   {Show {Sum H}} % H could be forgotten during the execution of Sum
   {Delay 10000}
end
```

A very long list is introduced whose head is `H`. Normally, `H` will not be forgotten until the end of the local block. However, the thread statement is transformed into a call to the `Thread.create` builtin with an anonymous procedure as parameter.

This anonymous procedure has a reference to `H` in its external environment. If the thread is well implemented enough to forget the reference to its starting procedure, it should not remember any reference to `H` after the first iteration of `Generate` because it is a tail call statement so its original frame is also dropped.

If we had cleared the `FrameSlot` of `H` when providing it to `Sum`, no reference would hold the head of the list in memory. The nodes of the big list could be forgotten successively.

The idea is that some objects can be freed much earlier using this “optimization” than they would by other ways. It therefore requires much less attention when playing with streams for instance, making the language more convenient to use.

In order to achieve this, the basic idea is to tag last occurrences of local variable in a procedure. This can be simplified by walking the tree backward, reducing the problem to finding the first occurrences of local variables.

However, branches have to be taken into account. When a variable is only cleared in one branch, it means it was just not used by the others, because otherwise they would have cleared it as well. The earliest moment at which those variables become useless there is thus right before evaluating their body. Here is an example in which `X?` means the variable `X` is cleared before returning its value, `?X` means it is simply cleared but not accessed:

```
proc{ProcWithBranches A B C}
   {GenerateHugeList B}
   if {Choose} then
      A? = unit % Last occurrence of A in this branch
      {DoSomethingWith B?} % Last occurrence of B
   else
      ?B % B will not be used in this branch, clear it directly
      local X={SomeConstant} in
         A? = X? + 5 % Last occurrences of A and X
      end
   end
   % The first branch, however, does not need to clear X's slot!
   C?=0 % Some other statements not using A or B
end
```

The following code provides an idea of how it has been implemented:

```
case class ClearSite(node: StatOrExpr, before: Boolean)

object VariableClearing extends ReverseWalker {
    var clearSites = HashMap[Symbol, Seq(ClearSite)]()

    /** encapsulates closure execution with another clearSites map */
    def withClearSites[A](newClearSites: HashMap[Symbol, Seq(ClearSite)]
                         )(closure: =>A): A = { /* ... */}
    /** Creates a new map with only local variables not cleared yet */
    def notClearedYet() = { /* ... */ }

    def walk(node: StatOrExpr) = {
        case localNode: LocalCommon =>
            // Symbols are the unique identifiers of each variable.
            local.declarations.foreach(varNode => clearSites.put(varNode.symbol, null))
            walk(local.body)

        case Variable(symbol) =>
            // Only local variables not yet cleared can be cleared at this node
            if (clearSites.get(symbol) == Some(null)) {
                clearSites.put(symbol, ClearSite(node, false)) // Put this node as clearing site
            }

        case IfCommon(cond, thenNode, elseNode) =>
            val thenClearSites = withClearSites(notClearedYet()) {
                walk(thenNode)  // walk this branch with an isolated clearSites map
                clearSites      // return that map
            }
            val elseClearSites = withClearSites(notClearedYet()) {
                walk(elseNode)
                clearSites
            }
            // ...
            // Merge thenClearSites and elseClearSites into the current clearSites map,
            // by keeping clearSites from both. Also add clearSites at the beginning of
            // each branch for each variable cleared in the other branch, but only when
            // considering variables that were already declared.
            walk(cond) // Remember this is a reverse walk

        // ... Very similar case for pattern matching branches

        case _ => super.walk(node)  // otherwise, continue reverse-walking the tree normally
    }
}
```

What then remains to be done then consists in wrapping nodes which are specified into “clearSites” to be translated within `ResetSlotsNode` whose code looks like this:

Listing 5.3: Truffle node for clearing unused slots

```java
public class ResetSlotsNode extends OzNode {
    @CompilationFinal(dimensions = 1) FrameSlot[] before;
    @Child OzNode node;
    @CompilationFinal(dimensions = 1) FrameSlot[] after;

    public ResetSlotsNode(FrameSlot[] before, OzNode node, FrameSlot[] after) {
        this.before = before;
        this.node = node;
        this.after = after;
    }

    @ExplodeLoop
    private void resetSlots(VirtualFrame frame, FrameSlot[] slots) {
        for (int i = 0; i < slots.length; i++) {
            frame.setObject(slots[i], null);
        }
    }

    @Override
    public Object execute(VirtualFrame frame) {
        resetSlots(frame, before);
        Object result = node.execute(frame);
        resetSlots(frame, after);
        return result;
    }
}
```

Finally, the frame argument array elements may be set to null directly as transferring them into the local slots. For the captured variables, procedures cannot free them because they may be useful for future executions.

5.2 Dynamic Optimizations

In this section, we present how using the Truffle API can be used to provide more information to the compiler, allowing it to compile our interpreter more efficiently, performing inlining and transforming tail calls into loops. In the meantime, we also help compilation to potentially happen earlier and more economically. Finally, we propose alleged specializations for unification and equality testing.

5.2.1 Providing More Information to Graal

As introduced in Section 2.2, Truffle provides languages implementers annotations and directives to inform the Graal compiler about information that would be expensive or impossible to collect.

5.2.1.1 Compilation Constants

Truffle provides the `@CompilationFinal(dimensions)` annotation in order to tag class attributes remaining constant for a compilation. The difference with the `final` modifier is `@CompilationFinal` does not enforce the attribute to be constant during the whole execution. Attributes may be modified before any compilation or at de-optimization. It is also more expressive in that it allows array elements to be considered as `@CompilationFinal` as well, up to an arbitrary dimension. Let us consider the following commented code excerpt from `ResetSlotsNode` proposed in Listing 5.3:

```
public class ResetSlotsNode extends OzNode {
    // Both the reference of the array and its elements can be considered as constant
    @CompilationFinal(dimensions = 1) FrameSlot[] before;
    @Child OzNode node; // @Child implies @CompilationFinal semantics
    @CompilationFinal(dimensions = 1) FrameSlot[] after;

    // ...
}
```

In this case, those annotations allows the compiler to use fixed frame offsets when dealing with variables. Having the array constant, therefore also its size, allows using `@ExplodeLoop` in order to unroll the loops iterating over the `FrameSlots` in compiled code.

5.2.1.2 Branch Profiling

At the beginning of their execution, Oz programs run in interpreter mode, using HotSpot. If HotSpot profiles Java branches in the interpreter, those are not differentiated for distinct node instances. The collected profile is therefore not representative at all of the different locations in the Oz code, and would not be of any use for Graal. Truffle instead provides its own classes for profiling branches, loop counts, etc.

BranchProfile allows the language implementer to inform the compiler about branches that are unlikely to be visited. This will prevent it to compile branches that have never been taken.

ConditionProfile is declined into three versions aiming at profiling boolean conditions involved in branches and loops.

- **Binary** which only allows compiling one branch at a time, the latest taken in interpreter mode;
- **Counting** which counts both true and false values while running in interpreter mode, provides the branch probability to the compiler and avoids compiling a branch that has never been taken;
- **Loop** which does the same as the counting profile, but only avoids compiling the body of the loop if never entered.

Those profiles have to be instantiated in the nodes and declared as `@CompilationFinal` as well. Here are some use examples:

```java
// BranchProfile branchProfile = BranchProfile.create()
try {
    // Likely branch - candidate to compilation
} catch(TailCallException tailCall) {
    branchProfile.enter();
    // Unlikely branch will not get compiled unless entered
    // Great to avoid complex unlikely branches to be compiled
    while(/* ... */) {
        // ...
    }
}

// ConditionProfile countingConditionProfile = ConditionProfile.createCountingProfile();
if (countingConditionProfile.profile(someCondition)) {
    // Never compiled if not entered
} else {
    // Never compiled if not entered
}

// LoopConditionProfile loopProfile = LoopConfitionProfile.createCountingProfile();
while(loopProfile.profile(someCondition)) {
    // Not compiled if never entered
}
```

Without those instructions, Graal will compile all the branches of the methods candidate to compilation.

5.2.1.3 Boundaries

In order to avoid compiling code from the Java library, the `@TruffleBoundary` annotation can be used to Java specify methods that should always run in the interpreter. It is indeed important for methods calling the Java standard library in order to prevent the compiler from compiling large portions of it.

5.2.2 Call Optimization: Polymorphic Inline Caches and AST Inlining

In practice, language implementations with dynamic calls try to inline information at call sites in order to speed-up similar next calls.

Let us consider the following Oz code:

Listing 5.4: Trivial Oz example benefiting from polymorphic inline caching and AST inlining

```
proc{Do F}
    {F}          % Either calls A or B
end
proc{A} {Show a} end
proc{B} {Show b} end

for I in 1..100000 do
    {A}          % Always calls A
    {B}          % Always calls B
    {Do A}       % Always calls Do
    {Do B}       % Always calls Do
end
```

Each call site in the loop calls the same procedure on every iteration. The call site in `Do`, however, either targets `A` or `B`.

By default, a call to a procedure consists in several internal lookups before even being able to grow the stack and jump to the `RootNode`. By caching this information into the call site, those lookups could be avoided next times the procedure to be called is known. Small procedures could even get inlined directly into the call site. Truffle exposes the `DirectCallNode` to provide those optimizations on a constant `CallTarget`.

In order to detect and take advantage of such opportunities, a *polymorphic inline cache* is put at each call site. It basically provides every node its finite state machine, initially into the *uninitialized* state. When performing its first call, the node goes into the *monomorphic* state, having one procedure in its cache.

For the subsequent calls, the node checks whether the procedure is the one it knows or not. If not, the node adds it to its cache, and puts itself in the *polymorphic* state, able to store several more procedures to check sequentially. Whenever the size of the cache exceeds some threshold, the node goes into the *megamorphic* state, forgetting the procedures it knew and performing the generic algorithm for every further call.

The *polymorphic* and *megamorphic* states can be seen as optional. The *polymorphic* is only a generalization of the *monomorphic* one for more entries. Without the *megamorphic* state, the node would go back to *uninitialized* when exceeding its capacity, which does not ensure convergence of ASTs.

The vigilant reader will have recognized the same ideas as behind the self-optimizing interpreters. It ends up such mechanism is easily implemented using Truffle and its DSL. Indeed, Truffle's `@Cache` annotation can create several instances of a specialization. The check can then be performed between the cached value and the provided argument. Here is what a call node handling procedures could look like:

Listing 5.5: Truffle `CallNode` for Oz procedures with both polymorphic inline caching and AST inlining

```java
@NodeChildren({ @NodeChild("proc"), @NodeChild("arguments") })
public abstract class CallProcNode extends OzNode {
    // ...

    /** Monomorphic, for 1 procedure instance */
    @Specialization(guards = "proc == cachedProc", limit = "1")
    protected Object callDirectProc(OzProc proc, Object[] arguments,
            @Cached("proc") OzProc cachedProc,
            @Cached("create(cachedProc.callTarget)") DirectCallNode callNode) {
        return callNode.call(OzArguments.pack(cachedProc.declarationFrame, arguments));
    }

    /** Polymorphic, specialization with max X call targets (X = 3 by default) */
    @Specialization(guards = "proc.callTarget == cachedCallTarget",
                    replaces = "callDirectProc")
    protected Object callDirectCallTarget(OzProc proc, Object[] arguments,
            @Cached("proc.callTarget") RootCallTarget cachedCallTarget,
            @Cached("create(cachedCallTarget)") DirectCallNode callNode) {
        return callNode.call(OzArguments.pack(proc.declarationFrame, arguments));
    }

    /** Megamorphic, does not cache anything anymore */
    @Specialization(replaces = { "callDirectProc", "callDirectCallTarget" })
    protected Object callIndirect(OzProc proc, Object[] arguments,
            @Cached("create()") IndirectCallNode callNode) {
        return callNode.call(proc.callTarget,
            OzArguments.pack(proc.declarationFrame, arguments));
    }
}
```

`@Cache` can be used both for creating children nodes specific to a specialization or to store some value in the node's state.

Coming back to our example Listing 5.4. After a certain number of calls, the call target of a `DirectCallNode` will be inlined by Graal. Inlined AST are put back to the *uninitialized* state again, so they can build their own and inline again information specific to this part of the code. It is thus very likely the for loop will get compiled with 4 direct calls to `Show`.

5.2.3 Control-flow Exceptions

The implementation of tail calls from Section 4.4.3 and Section 5.1.1 makes use of Truffle's `ControlFlowException` class. Not only does this class avoid generating the stack trace when instantiated, but it also happens that when both the `throw` and its `catch` sites are visible to the compiler (i.e. the exception does not escape from the considered compilation unit), it is able to to simplify them into pure control flow.

Optimization that are performed on loops can therefore be applied in such routines, and the exception does not need to be allocated anymore.

Limitations

If it constitutes a very desirable optimization, in practice, the procedure in which the `TailCallCatcher` is situated must be compiled in order to perform it. FIGURE 5.2 illustrates the fact even a very large number of iterations of the tail procedure will not allow the control flow to be optimized. This is even worse in the case of self tail calls, where no boundary is crossed, so no compilation will happen at all during the first executions.

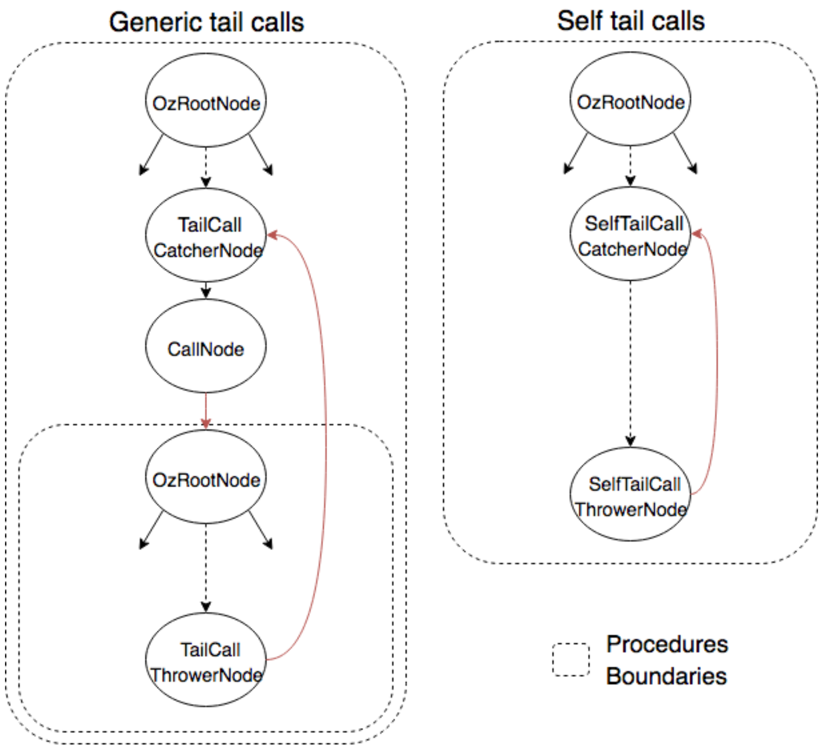


Figure 5.2: Graal's point of view while stuck into tail calls is not ideal

This may look like a minor problem, but a lot of Oz builtins work this way, such as `Map` and `FoldL`. Hopefully, Truffle provides a way to add such compilation boundaries and achieve on stack replacement.

5.2.4 On Stack Replacement

On Stack Replacement (OSR) is the mechanism that enables loops to be optimized during their execution.

Because Graal considers `RootNodes` as compilation unit delimiters and only compiles them after several calls, it may take a lot of time for it to compile tail recursive functions for instance. This is also illustrated in FIGURE 5.2. The `Map` builtin is for instance impacted by this limitation:

```
proc{Map Xs P R}
   case Xs of nil then nil
   [] X|Xr then T in
      R = {P X}|T
      {Map Xr P T} % self tail call
   end
end
```

If `P` will well be candidate to compilation during a very long execution of the `Map` procedure, the `Map` itself will simply loop internally in interpreter mode. Indeed, the self tail call mechanism as described in LISTING 5.1 does not go through any `RootNode`. If a `RootNode` could be added just before the procedure body, the most interesting part of the procedure could be compiled. Setting it right above the `SelfTailCallException` *try-catch* is even better since the compiler is then certain it does not escape and transforms it into a loop.

However, making use of the `RootNode` API that way is not recommended. Truffle provides allows to deal with it through its `LoopNode` class, providing a different way to write the loops so their body constitutes a compilation unit, and allowing them to be optimized during their execution. Figure 5.3 illustrates how Mozart-Graal takes advantage from them, allowing more efficient compilation to happen earlier.

Generic tail calls may benefit a bit less from OSR, as the target procedures were already able to be compiled since their `RootNode` is in the path for every iteration. The call node is however given a chance to inline the call target and to optimize the control flow earlier.

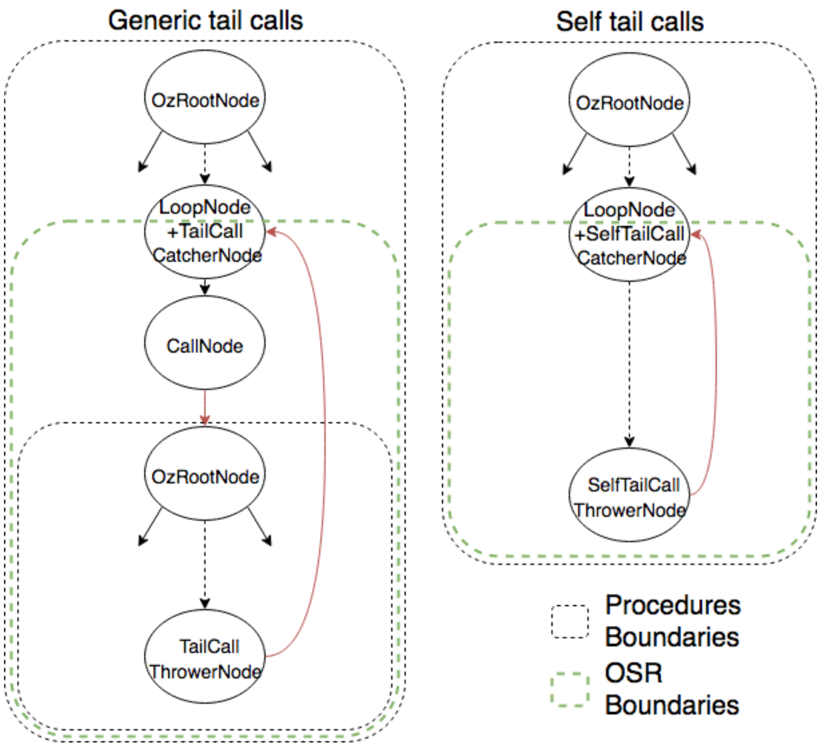


Figure 5.3: On Stack Replacement to allow earlier optimization

5.2.5 Builtin Splitting

It is desirable for functional builtins such as `Map` or `FoldL` to specialize directly to their call sites. Indeed, it allows for early inlining of the functions they call in their own code. More generally, doing so with all builtins avoid using *megamorphic* specializations in early execution and directly collects call site sensitive profiling information.

In order to achieve this, the builtin AST has to be cloned, which is feasible with Truffle `CallDirectNodes` when Graal is enabled. Truffle also allows to inline the cloned call target, which is a good idea since it will not be called from anywhere else.

5.2.6 Lighter Variants for Unification and Equality Testing

Unification as it was presented in Section 4.4 was correct but not very efficient. It was indeed traversing the data structures with an explicit stack and included the cycle detection mechanism in all records unifications, even though those deep or cyclic cases are very rare in practice. This implies a lot of allocations on the heap memory while most unified and compared structures may actually be traversed on the stack and not require cyclic detection.

The idea is to create an optimized specialization of `UnifyNode` without explicit stack and cycle detection. Because unifying two cyclic structures always implies an infinite depth, deoptimization can simply happen when this depth exceeds some threshold, the same we think would suffice to avoid the stack overflow in simple deep cases. On one hand, this threshold must not be too low because nodes that do not require it would make use of cycle detection and this is what we try to avoid. On the other hand, having it too big risks to let stack overflow happen and could waste a lot of time before deoptimization.

Let us consider a cyclic record with N features whose last one is a reference to itself:

```
R = record(1 2 3 ... N-1 R)
Q = record(1 2 3 ... N-1 Q)
R = Q
```

By going depth-first, the complexity of reaching depth d is $O(dN)$. But imagine first features could themselves be complex structures as in the following:

```
SubR = record(1 2 3 ... N)
R = record(SubR SubR ... SubR R)

SubQ = record(1 2 3 ... N)
Q = record(SubQ SubQ ... SubQ Q)

R = Q
```

This would take much more operations ($O(dN^2)$), while cycle detection would have allow to only walk once through `SubR` and `SubQ` and would have remembered them to be equivalent. Arbitrarily pathological cases can be forged.

In order to avoid this potentially huge overhead before deoptimization, this is the number of sub-unifications rather than the depth that is bounded. So a threshold n is always reached in $\Theta(n)$ sub-unifications.

The implementation relies on a base node class `DFSUnifyNode` with different specializations for handling the simple cases in a compiler-friendly manner. It specializes for two unbound variables to link, one unbound variable to bind or two bound variables with values. Some can be compared in a shallow manner, but records are unified recursively, which leads to stack overflows when they are deep circular.

This class is then extended by two other node classes, `DepthLimitedUnifyNode` and `OnHeapUnifyNode`, both only creating some state when unifying two records. The state of the first consists of a number of sub-unifications after which it deoptimizes, while the state of the second consists of the explicit stack for the traversal and the set of already encountered pairs of variables.

A final node class, `GenericUnifyNode`, acts as dispatcher between both, falling back on the `OnHeapUnifyNode` only once the `DepthLimitedUnifyNode` has thrown an exception stating its threshold has been reached. It may happen that unbound variables get affected by the limited depth unification before deoptimization and end up being seen as bound to the cycle detecting unification. This does not create inconsistencies because unification is idempotent.

A very similar mechanism is used for equality testing through the `DFSEqualNode`, `DepthLimitedEqualNode`, `OnHeapEqualNode` and `GenericEqualNode`. Pattern matching can still simply use the `DFSEqualNode` for its constants, since at least the pattern is not circular. A `GenericEqualNode` is however required for the *escaped* variable patterns as they might be circular.

Chapter 6

Evaluation

In this chapter, we first evaluate the improvements brought to Mozart-Graal. We start with the additional information provided from the static analysis allowing to remove part of the dataflow variables and to avoid memory leaks. We then consider the different optimizations around tail calls. As unification and equality tests were efficient but not general enough, the performance degradation for achieving generality will be measured. A section will then be dedicated to the evaluation of inlining and the different features articulated around it.

Finally, we evaluate more globally the current state of Mozart-Graal and compare it with Mozart 1 and 2 on different aspects. We will see that if it can already be competitive in some cases, there is still room for improvement.

6.1 Methodology

Two kinds of programs will help us to obtain results for our implementation: micro-benchmarks made of specific features and a realistic project. Those Oz programs and a tool for running them and collecting their output has been developed and is hosted on GitHub at [18]. It also comprises the Jupyter notebook we have used [19] for data visualization. The project we have considered is the Project 2014 [32] provided by Guillaume Maudoux. All the codes have been instrumented to print clock times and to be configurable by the runner program. As discussed in Appendix A.1, the impact of measuring time as been verified to be negligible with respect to the interpretation we make of those collected times.

The configuration is done through the `bench_*.txt` files of which each line describes a program to run with given parameters. Different categories of parameters are covered:

1. The Oz implementation to run the program with. The benchmarks used in this dissertation are written to run on Mozart 1, Mozart 2 and Mozart-Graal.
2. Options of the JVM or of the Graal compiler when running the program under Mozart-Graal.
3. Options that are defined in Mozart-Graal, for disabling optimizations for instance.
4. Actual variables in the main functor. This may help to tune number of iterations, complexity of structures, ...

All of those programs have been run on a dedicated server through Secure Shell (SSH) connection. Only one has been run at a time, with the minimal amount of concurrent services running. This server runs under Ubuntu Server 14.04 LTS and is equipped of an Intel(R) Core(TM) i5-2400 CPU (quad core) at 3.10GHz and 4x4096MB RAM at 1333MHz.

Mozart 1 is used as of its `1.4.0-2008-07-03-GENERIC-i386` release available at `https://sourceforge.net/projects/mozart-oz/files/v1/`.

Mozart 2 is used as of its `2.0.0-alpha.0 build 4105` release available at `https://sourceforge.net/projects/mozart-oz/files/v2.0.0-alpha.0/`.

Mozart-Graal is used as of this branch: `https://github.com/mistasse/mozart-graal/tree/memoire-final`. Truffle has been used as of this commit: `https://github.com/eregon/truffle/tree/c3ae06dbe4c6008f83c3fab69e22e7095a30f929` and Graal as of this commit: `https://github.com/eregon/graal-core/tree/1236bbd1691733995623f3f42f9ac3984300e437`.

Because we are mostly interested in peak performance, all our programs are running the main code section several times, the exact number depending on the difficulty of the task. Under some circumstances code may get compiled during the first iteration or may not get compiled during the whole execution. This will be discussed in due course.

Uncompiled code and compilation itself also require more memory allocation which implies garbage collection. If this affects a lot early iterations, this usually becomes stable or periodic at some point. The default parameters of the benchmarks are set accordingly in order to ensure each program reaches its stationary performance during at least half the iterations. Modifying some of the parameters may however influence this behavior.

Comparison between optimizations will mostly be performed based on peak performance, which we decided to measure as the mean iteration time, considering only the second half of all the iterations. It end up being insensitive to the early noise but still gets affected by periodic garbage collection. More metrics will also be used here and there such as instantiated variables, early iterations profiles, or total execution times of the programs. Most of the time, iteration times are averaged over several runs. This is achieved by taking the mean of analogous iterations of the different runs.

In some cases, Graal IR graphs are used to discuss whether the compiled code reaches the quality we expect or not. Those are collected and printed using the Ideal Graph Visualizer [43], and can be found in Appendix A.4. If no part of this dissertation will be dedicated to the reading of those graphs, they are annotated in order to provide some insights about what is happening. Part of [24] is dedicated to explaining the nodes more deeply. The very new [44] also provides insights about analyzing such graphs applied to the case of TruffleRuby.

The Benchmark Programs

As stated here before, several programs will be used in order to assess the performance of our implementation. Each consists of some main task of adjustable complexity and a loop executing it for a given number of iterations. This loop also reports the time taken for each iteration, i.e. execution of the task.

Add The Add program consists in adding two numbers by incrementing one of them and decrementing the other in a tail recursive manner. Its complexity parameter is the number to decrement `N`, while the other always starts at 0 for convenience.

Map The Map program consists in applying a small arithmetic function on every element of a list. By default, the arithmetic function is a simple multiplication with the constant 11. The complexity parameter is the length `N` of the input list.

Flatten The Flatten program consists in flattening a list of arbitrary size and depth. The complexity parameter is the number `N` of sublists of depth 5 in that list.

PingPong The PingPong program consists in two threads interacting through a stream, each placing an atom in it one after the other until a certain number of atoms have been placed. The parameter is the number of atoms `N` each thread puts in the stream.

Sieve The Sieve program consists in computing the list of prime numbers under an arbitrary threshold `N` in a concurrent manner, each discovered prime number having its own thread filtering the numbers it can divide and passing others to next threads. The parameters also comprise whether to stop creating threads at $\sqrt{N}$ or not, and whether the functions have to be lazy or not.

Project2014 The Project2014 program consists in executing Project 2014 without writing the final file to the disk. The partition `joie.dj.oz` will be considered by default. It makes heavy use of lists, `Map`, `Flatten` and user-defined functions.

LoadBase The LoadBase program is a simple functor doing nothing. It will however be used to get measures about loading the runtime.

Only the code for the `Add` benchmark is provided in this report, the other being available on [18]. Table 6.1 states the default parameters for each program.

Benchmark	Iterations	`N`	Other
Add	100	2000000	
Map	100	1000000	$f(x) = 11 * x$
Flatten	100	1000	Each sublist is of depth 5 and is made of 3 sublists of depth 4 which are contain 3 sublists of depth 3, etc
PingPong	100	250000	
Sieve	100	10000	Not lazy, creates filters for each prime up to `N` (not $\sqrt{N}$)
Project2014	10		`joie.dj.oz`

Table 6.1: Default benchmarks parameters

As there are a lot of options that can be turned on and off in Mozart-Graal, it has been decided to rather evaluate their impact by only switching the ones of interest on and off and letting the others activated. Turning the other options off usually prevents optimizations of interest to have their maximal impact.

6.2 On Stack Variables

As stated in SECTION 5.1.2, our implementation avoids instantiating variables with two mechanisms. First, it does not add unnecessary variables for nested expressions. Second, it also avoids instantiating declared variables that do not require any dataflow and that respect some assignment pattern. Since the former was "free" and we have no option to desugar all expressions, it will not be evaluated. The latter, however, required some efforts and is optional in Mozart-Graal.

The numbers of variables created under the different configurations have been collected on our benchmarks and are reported in Table 6.2. Only one iteration has been performed, and this comprises the cost of loading the base library. It appears that close to fifty percent of the variables can be avoided this way. The results provided by the left binding optimization may seem disappointing, but this is because benchmarks make heavy use of the functional features of the language. Since functions in Oz require variables to return values, non-functional sections of code are the ones mostly benefiting this optimization.

Also, our most realistic benchmark is the Project 2014. The gain shown there also means that even though variables from nested expressions and pattern captures have been avoided, it was still possible to avoid instantiating approximately 10% of the remaining variables.

Benchmark	Variables instantiated (Relative gain)			
	No optimization	Optimize captures	Optimize `X=v`	Optimize both
Add	53109	21913 (-58.74%)	50564 (-4.79%)	19368 (-63.53%)
Map	5053657	3022142 (-40.20%)	5051091 (-0.05%)	3019576 (-40.25%)
Flatten	2845146	1357947 (-52.27%)	2842593 (-0.09%)	1355394 (-52.36%)
PingPong	1053669	522143 (-50.45%)	1051102 (-0.24%)	519576 (-50.69%)
Sieve	3170702	1578868 (-50.20%)	3168155 (-0.08%)	1576321 (-50.28%)
Project2014	139775829	83288481 (-40.41%)	131305383 (-6.06%)	74818035 (-46.47%)
LoadBase	50450	20589 (-59.19%)	48027 (-4.80%)	18166 (-63.99%)

Table 6.2: Number of instantiated variables for one iteration of every benchmark

Note that those quantities are not taking into account the optimizations performed by the Graal compiler itself because no convenient way of doing so has been found. It is expected that the local variables not escaping from compiled procedures would be removed. As to whether variables are actually escaping or not, it all depends on the scope of the compiler which depends on how inlining is performed.

Conclusion

On stack variables constitute a first step towards a more parsimonious use of dataflow variables. Most of them would be avoided when critical sections get JIT compiled, but the quality of this optimization depends on the compiler's point of view. Making sure we do not rely on the JIT compiler to avoid the majority of non-dataflow variables therefore seems to be a good start.

6.3 Variable Selection

Prior to this master's thesis, the external environment of each procedure was not a "filtered" copy, but rather a pointer to the upper declaring local environment. Just like the alternative solution presented in Section 4.2.2. This means that releasing variables as in Section 5.1.3 was impossible, therefore leading to memory leaks.

An example is that heads of streams were more difficult to drop. Let us take this example:

```
declare
proc {Ping S}
   case S of pong|A then
      A = ping|_
      {Ping A.2}
   end
end
proc {Pong S}
   case S of ping|A then
      A = ping|_
     {Ping A.2}
   end
end

local S = pong|_ in
   thread {Ping S} end
   thread {Pong S.2} end
end
```

Because the variables of the `local` are stored in the frame of the main procedure, the `Ping` and `Pong` were in fact maintaining references to the head directly. It is possible to write programs achieving it without leaks by encapsulating the stream creation in a procedure, but this is uncomfortable as this requires special attention from the developer.

Now, references are dropped correctly thanks to variable selection which provides the dependencies for procedures in a more refined manner. The local block can thus clear its slot while the other threads are able to forget the anonymous procedure they have been started with. No reference to `S` is maintained anymore, the garbage collector can therefore discard the beginning of the stream from memory.

6.3.1 Frame Slot Clearing

We have found that this was however still too limiting. Indeed, developers still have to pay attention to ensure leaving the local block before the stream gets too big. This means the following code is also leaking, even though it is clear the intention was to release the head of the stream:

```
declare
proc {Ping S}
   case S of pong|A then
      A = ping|_
      {Ping A.2}
   end
end
proc {Pong S I}
   case S of ping|A andthen I >= 0 then
      A = ping|_
      {Pong A.2}
```

```
    end
end
T0 T1

local S = _|_ in
    thread {Ping S} end
    T0 = {Time}
    {Pong S.2 2000000} % maintains a reference to S, but could clear it when
        computing S.2
end
T1 = {Time}
{Show T1-T0}
```

This is even more disturbing as if the `Pong` call had been the last statement, the memory leak would not have happened. This is why frame slot clearing has been introduced in order to release references to local variables as early as possible, right after having evaluated their last identifier.

As a side note, clearing the frame slots should not influence the generated code since the compiler is already aware variable occurrences. It is however necessary for the interpreter mode that does not have a complete vision on them.

The impact can be evaluated on our PingPong program by running 1 iteration with $N = 10^6$ with the JVM option `-Xmx100m` for restricting the heap size to 100MB. The version clearing the slots terminates while the other tries to store the whole stream and ends up crashing.

Limitations

Frame slot clearing takes care of freeing all slots, even though it is sometimes obvious the procedure is going to end and no statement is susceptible to block it for a long time. Improvements could reduce the number of clearing nodes by removing those that are negligible in that sense and by merging those that are separated only by non-blocking statements.

Some AST nodes also directly keep references to values into their state because the Graal compiler could not see through weak references when they have been written. It could thus not optimize those nodes as well when using them. As it is not the case anymore, some node classes should be adapted in order for Mozart-Graal to be totally memory-friendly. Classical calls have however already been updated.

Conclusion

Variable selection thus makes Mozart-Graal much more friendly with the garbage collection than the hierarchy of frames by allowing it to dissociate variables in the same frame. This also allows clearing frame slots to free references as soon as they are not used, which relieves the developer from having to think about escaping local blocks declaring a stream before doing computation on it. Those two optimizations make the whole implementation much more predictable when programming.

6.3.2 Tail Calls, Loops and On Stack Replacement

Prior to our work, tail calls were not implemented. This is therefore one of the first contributions realized for this master's thesis. We do not consider benchmarking in absence of tail call optimization because serious tasks making use of them simply crash when doing so. We rather evaluate the different variants of this optimization and their treatment by the compiler.

We will first illustrate the performance improvement brought by self tail calls and OSR using the `Add` micro-benchmark from Listing A.1. We will then evaluate their impact on the Project 2014 from [32].

The peak iteration times are reported in Table 6.3, and Figure 6.1 plots iteration times obtained with those different sets of optimizations. The experiments on the left do not perform *self* tail call detection, which means only the generic tail call optimization is applied, while experiments in the right part considers them. Each of those configuration is considered with and without OSR.

Let us discuss the four curves independently.

Case #1 no self tail calls and no OSR : the `Add` function gets compiled, but the exception mecha-

Self tail calls	OSR	Peak iter. time (ms)
No	No	804.28
No	Yes	2.27
Yes	No	1678.60
Yes	Yes	2.74

Table 6.3: Peak iteration times of `Add` with and without OSR and self tail call optimization

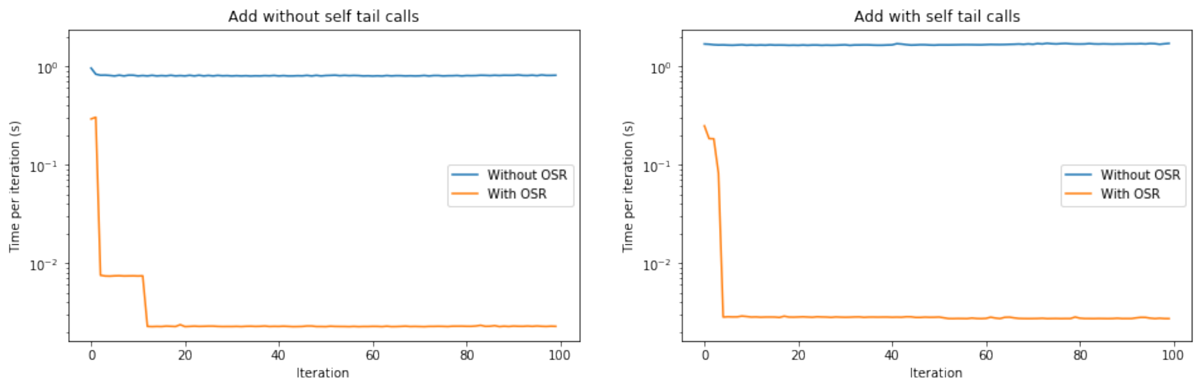


Figure 6.1: Iteration times of `Add` with and without OSR and self tail call optimization

nism is still used. Indeed, the whole `for` loop body constitutes the smallest compilation unit from which the control flow exception does not escape and is not called enough to get compiled.

Case #2 with self tail calls but no OSR : the `Add` function never gets compiled because its `RootNode` is not solicited enough since the loopy mechanism is situated below it. It thus performs even worse than the generic mechanism. Everything stays interpreted and the exception is still allocated, even though lighter.

Case #3 with self tail calls and OSR : the body of the `Add` procedure quickly gets compiled, the control flow is optimized and intermediate boxing and unboxing disappear.

Case #4 no self tail calls but OSR : two plateaus are visible. The first optimization compiles the `TailCallCatcher` above the `Add` call, which happens to be monomorphic. The loop is therefore optimized, but some accesses and checks are still performed at every iteration making them suboptimal.

The second plateau happens because of a second optimization compiling the upstream `for` loop. Thanks to the additional information, the compiler is now able to avoid optimize more every incrementation. Some tens of milliseconds are even gained for each iteration compared to the self call with OSR, which are most likely due to the compilation of the external components of the `for` loop.

6.3.2.1 Graal IR Graphs

We will discuss the Graal IR graphs are reported in Figure A.3 and Figure A.4 for the `Add` procedure in #1 and #3 respectively. The two phases from #4 are provided in Figure A.5 and Figure A.6 but are less readable because external code intervenes. The same substructures as #3 are however observable. For case #2, it simply never gets compiled. There is therefore no Graal IR to discuss.

#1 : The top of the graph takes care of getting the arguments, unboxing the integers and testing `A`'s value. *Floating* `IntegerAddExact` and `IntegerSubExact` represent the incrementation and decrementation. The left branch (B1) boxes the incremented and decremented values and allocates a `TailCallException`. The right branch (B2) boxes the integer, binds the result variable with it and returns. Boxing therefore happens at every incrementation.

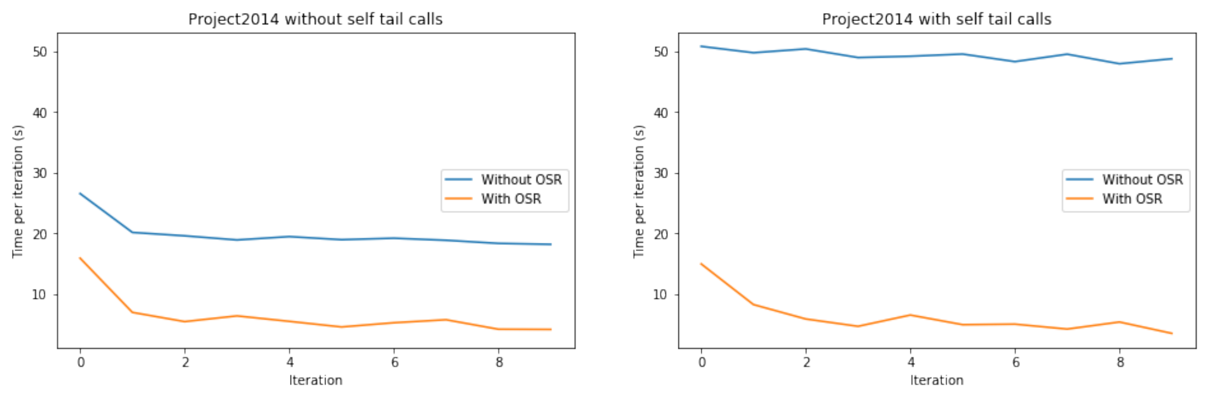


Figure 6.2: Mean iteration times of `Project2014` with and without OSR and self tail call optimization

#3 : The top of the graph still takes care of extracting and unboxing the arguments from their array and testing `A`'s value. In case it is equal to 0, B1 still takes care of binding the result variable. The other case leads to a loop performing iterations as long as `A` is different to 0. Whenever this loop exists, B4 binds the variable and B5 boxes the result. The control from B1 and B1-B5 are then merged for the return.

An important consideration is that the first iteration has been extracted from the loop in order to only unbox arguments at the very beginning, and to box them again at the very end. Every iteration can happen on primitives and end up in a very clear mapping from the Oz code.

As a side note, one could introduce intermediate local variables for computing `A-1` and `B+1` as in Listing A.2. In case the compiler is sure they do not escape, scalar replacement is performed on them. This means that under configuration #3 and #4 the following code leads to the same optimized graph and performance as without them.

Ten iterations of the project have been run 5 times with each of the parameter set. The resulting mean iterations are provided in Figure 6.2. We can attest that the results are very similar to those of the `Add` program. The two tail call mechanisms with OSR perform rather evenly, while without OSR, it is better to disable to let the compiler collect data from calls.

Limitations

The tail calls detection presented in Section 4.4.3 has limitations. Indeed, it does not optimize calls in records assigned at the last statements in procedures. This does however not seem to cause practical problems because people tend to pay more attention when writing procedures, but it would be better to achieve it. A more systematic simplification of assignments to call expressions would provide it for free.

Also, the *self* tail call mechanism only tracks procedures by the name in their definition before desugaring. This means binding a variable to an anonymous procedure will not make it candidate to this optimization. Not all cases can be covered statically, but more could certainly be.

Conclusion

This experiment confirms that the compiler optimizes the control flow from `ControlFlowExceptions`, that our tail calls have the potential to be transformed into loops and that the compiler avoids boxing and instantiating variables between iterations. It also emphasizes the importance of On Stack Replacement (OSR) for both tail call mechanisms.

6.4 Unification and Equality Testing

At the beginning of this master's thesis, unifications and equality tests were performed on the stack and without cycle detection, because neither cyclic nor too deep data-structures had ever been encountered. Other cases of unifications were however already quite well optimized. The challenge has therefore been to integrate the cycle detection and the explicit stack while affecting performance of cases not requiring them as least as possible.

The implemented solution only adds counters on the specializations for records. Whenever more than 20 sub-tasks (unifications or equality tests) are performed, the node deoptimizes itself in order to provide the cycle detection and the explicit stack for all the times it will have to handle records. This threshold is also never reached on typical programs. For instance, executing the browser reaches a maximum of 8 sub-unifications or equality tests.

As this feature is never used in the standard library, only the impact on a synthetic benchmark has been measured. This benchmark unifies or tests equality of 2 equivalent records having a linear structure of depth 15 ten thousands times per iteration. Three scenarios are tested in both cases:

- No cycle detection or explicit stack fallback, which is the base solution we want to stay close to.
- Sub-task counting up to 100 sub-tasks. This one never deoptimizes and should stay as efficient as possible.
- Sub-task counting up to 10 sub-tasks, then fallback on cycle detection and explicit stack traversal. It will directly deoptimize in our benchmark but ensures correctness in all cases.

The benchmark results are provided in Figure 6.3 and the peak iteration times are reported in Table 6.4. We can see that the depth-limited specialization stays much closer to the base algorithm, but that the general one is not that much slower. Indeed, only the traversal of the first level when considering records. Most of the time is therefore spent in interpreter mode. The difference thus resides in the overhead of using complex data-structures and algorithms.

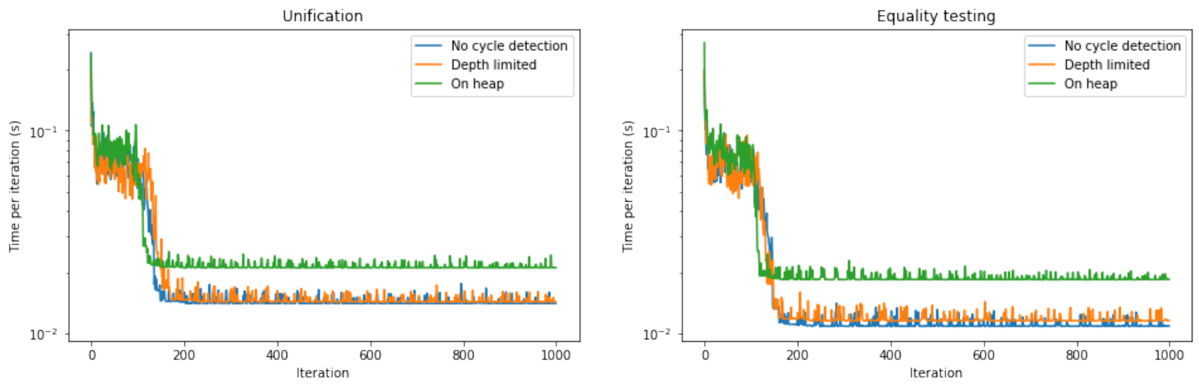


Figure 6.3: Mean iteration times of the different variants of unification and equality tests

Operation	Peak iteration time (ms)		
	Without cycle detection	Depth limited (On stack)	On heap
Unification	14.23	14.51 (+1.98%)	21.33 (+49.86%)
Equality	11.01	11.71 (+6.29%)	18.62 (+69.11%)

Table 6.4: Peak iteration times for the different variants of unification and equality tests

Limitations

The first level of records unification and equality testing is compiled, but not unrolled. An opportunity therefore lies in specializing those nodes to the shapes of the records they encounter during the execution.

Benchmark	Compiled nodes without splitting			Compilation time (ms)		
	no splitting	with splitting		no splitting	with splitting	
Add	4105	3952	(-3.72%)	8518	7219	(-15.25%)
Map	5345	6996	(+30.89%)	10759	12391	(+15.17%)
Flatten	4677	35437	(+657.74%)	9951	22267	(+123.76%)
PingPong	4630	4508	(-2.63%)	11381	11177	(-1.80%)
Sieve	4702	4851	(+3.18%)	10872	10957	(+0.78%)
Project2014	12514	18387	(+46.93%)	24885	36152	(+45.28%)

Table 6.5: Average impact of enabling builtins splitting on the compiler

Conclusion

There is room for improvement here, but we can attest that the depth-limited specializations provides a more desirable performance and allows for further optimizations to have a more significant impact.

6.5 Polymorphic Inline Caches, Inlining and Builtin Splitting

Polymorphic Inline Caches (PICs) with `DirectCallNode`s constitute the mechanism allowing Graal to cache information about call targets and to eventually inline them. This constitutes a core mechanism helping other optimizations to take place.

Our reference programs have been executed with and without `DirectCallNode`s in order to assess their importance. All the resulting iteration times of 5 runs of each complete benchmark ($5N$ iterations) have been reported in Figure 6.4. The self tail call optimization has been disabled to avoid the bias from the fact most of our benchmarks are only performing self tail calls.

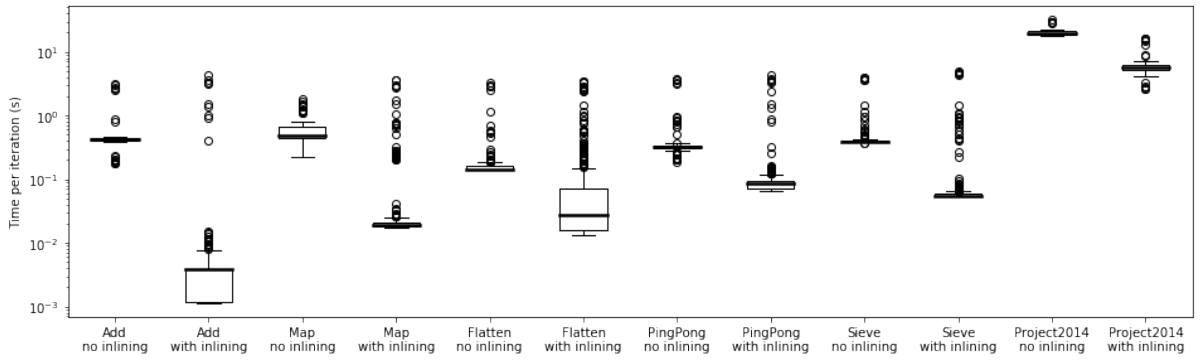


Figure 6.4: Distributions of iteration times for all benchmarks with and without inlining

We can attest that our programs can often benefit from a speedup factor from 10 to 100 by enabling it. The earlier inlining can happen, the better it is.

6.5.1 Builtins Splitting

In Mozart-Graal, as described in Section 5.2.5, an option forces any `DirectCallNode`s calling a builtin to inline it. This allows builtins not to have to be called several times before collecting call site sensitive information. For higher-order functions like `Map` or `FoldL` that are used at a lot of locations with different function parameters, it can be very interesting to bypass the early megamorphic builtins calls.

Figure 6.5 provides the distribution of the time taken by all iterations from executing 5 times complete benchmarks (5*`It` iterations). Self tail call detection has again been disabled to improve the variety of actual call usages.

The impact on our benchmarks is not very significant because either they do not make heavy use of

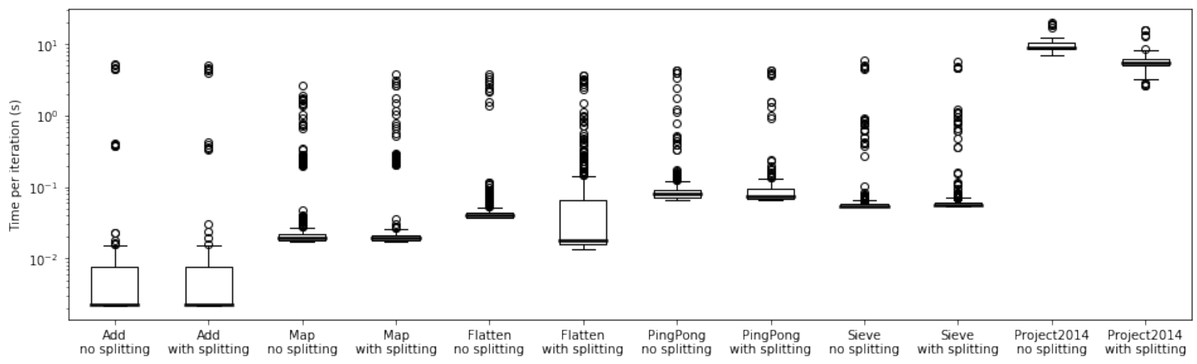


Figure 6.5: Distributions of iteration times for all benchmarks with and without builtins splitting

builtins or they split them quickly. We can however see that the Project 2014 benefits greatly from this option, as it uses `Map` and `Flatten` at different places and with a wider variety of parameters. This early splitting however comes at the cost of more nodes and compilation time, as emphasized in Table 6.5.

Limitations

Forcing builtins splitting is currently only performed on an all-or-none basis. It could probably be interesting to refine it to the interesting ones only, for instance by tagging them manually.

Conclusion

In this section, we have thus seen that inlining is a serious factor to take in consideration when designing a language and that real applications may take significant benefit if it is performed earlier. It may however come at some cost for the compiler so this would probably deserve to be refined.

6.6 Coroutines in Place of Threads: The Bounded Buffer

In producer-consumer applications, laziness is desirable in order to avoid for the producer to be too far ahead. But when the producer is slower than the consumer, one may want to keep him producing items before they are needed, ahead of time, keeping it busy while the consumer processes items before requesting others. The bounded buffer is an elegant way of achieving this. An implementation proposed by [33, p. 295] is provided in Listing 6.1.

Listing 6.1: Code of the Bounded Buffer

```
fun {Buffer1 In N}
   End=thread {List.drop In N} end
   fun lazy {Loop In End}
      case In of I|In2 then
         I|{Loop In2 thread End.2 end}
      end
   end
in
   {Loop In End}
end
```

When instantiated, the bounded buffer of capacity N simulates demand for the N first items. It provides a *mirror* lazy stream such that when item n is requested, item $n + N$ is made *needed*. On one hand, this allows for the *producer* to respond more adequately to spikes in the demand for instance, smoothening its production rate. On the other hand, laziness prevents him to go too far ahead and to cause memory problems for instance.

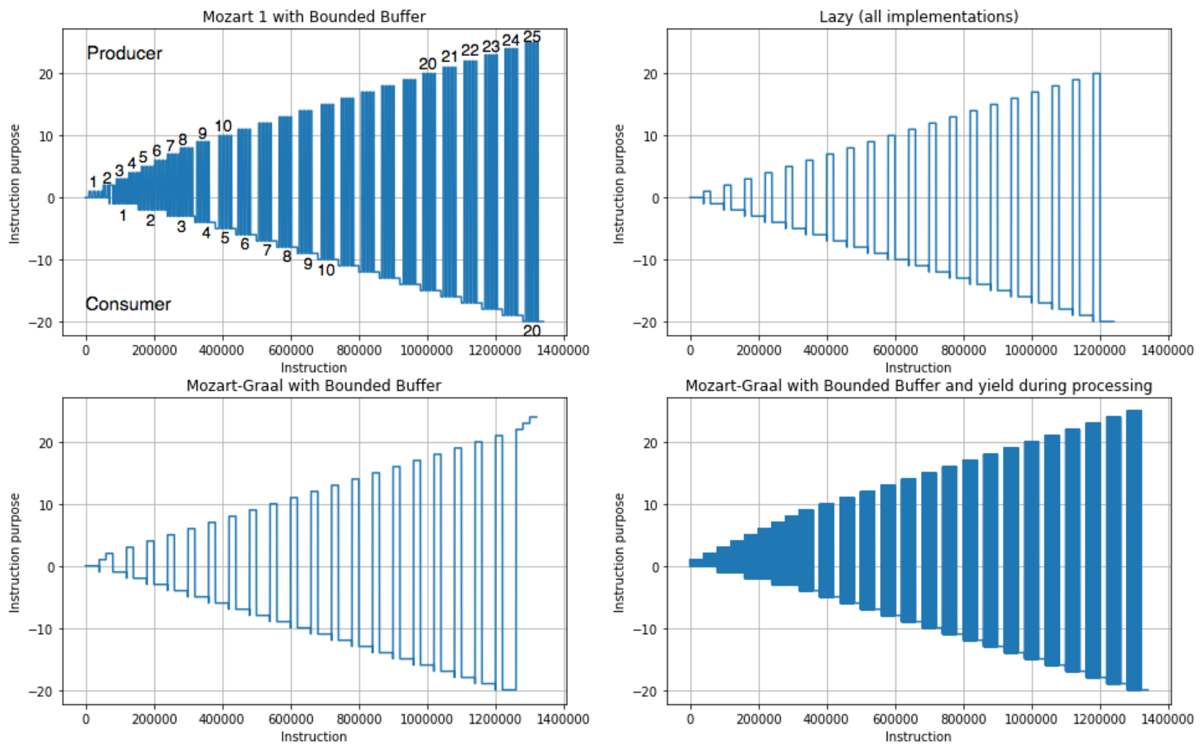


Figure 6.6: Visualization for execution passing in producer-consumer applications

Let us consider a lazy *producer* that generates a stream of items requiring t seconds to produce. Our *consumer* takes $2t$ seconds to process each, and can only do it sequentially.

Without the bounded buffer, the *consumer* will have to request and wait for each item before processing it, which will require $3t$ seconds of its time for each. With the bounded buffer, the *producer* will produce items while the *consumer* processes the previous ones, the consumer will therefore be able to process items in $2t$ seconds each.

The vigilant reader may note that if both threads are simulated or running on the same core, every of their "computation second" is doubled in terms of real time. This makes the bounded buffer a bit less useful in practice in effectively single-threaded applications, but still nicely demonstrates the synchronization introduced by the coroutines while threads can get desynchronized from each other.

The experiment in FIGURE 6.6 provides an overview of the execution of the producer-consumer application described here before. This experiment simulates instructions to be performed by both threads in order to produce or process items. $t = 20000$ is the number of such instructions that the producer has to execute to produce every item, and is high enough so that preemption should happen if it is implemented. The "purpose" axis states whether each instruction happens in the producer (positive sign) or in the consumer (negative sign), and the item it is related to with the absolute value. Vertical lines passing through $y = 0$ thus represent a transition from a thread to another. Rectangles may be formed of those, representing the fact both threads are working in parallel. Simple horizontal lines represent periods during which one thread is working alone.

After creation of the bounded buffer, the consumer first executes a $2t$ instructions routine before beginning to consume, with is represented by the points for which $y = 0$. The consumer simply processes 20 items. The lazy producer without the bounded buffer behaves as expected on all implementations, being asked to produce each item right before the consumer processes it and producing 20 items in total. In the other plots, The bounded buffer has capacity 5. We can see with Mozart 1 that the *producer* takes some advance while its execution is interleaved with the *consumer*. After having produced item 9, it however waits that the *consumer* begins to process item 5 before producing 10 for instance, and so on, and ends up producing 25 items while only 20 are consumed.

With Mozart-Graal, the bounded buffer does not achieve the desired effect. The *consumer* first has

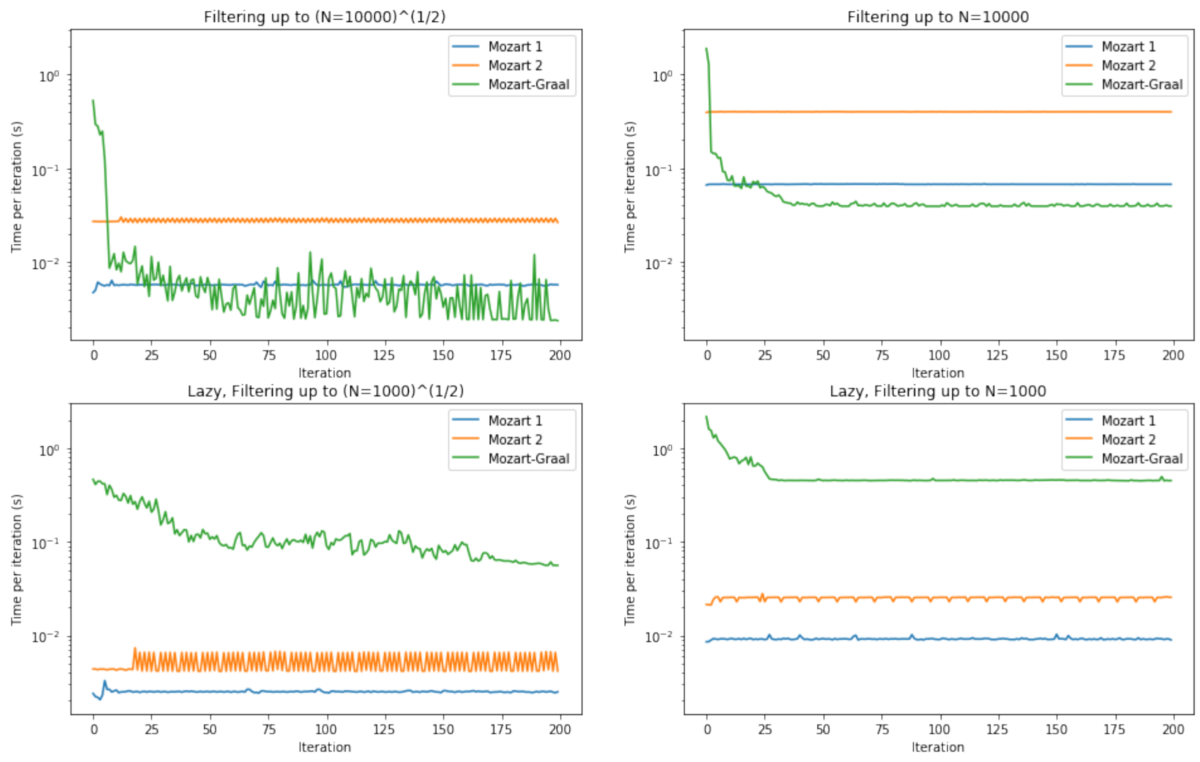


Figure 6.7: Mean iteration times for `Sieve` under different parameter sets and implementations

to ask for the first item before it gets computed along with the second element. When the *consumer* asks the bounded buffer for element n, the $(n+1)^{th}$ gets produced. This is one of the only differences with the lazy version and is most likely caused by the `List.drop` thread asking for one more item every time it gets the execution. The other difference being that once the *consumer* has terminated, the *producer* computes further elements.

The bounded buffer does thus not suffice in order to avoid laziness in our case because coroutines only pass execution when they have to wait on a variable. The bottom-right plot illustrates the fact that if the `Process` operation takes care of yielding the execution, we may obtain a result similar to what is expected, but at the cost of adding explicit synchronizations, which we usually try to avoid.

Conclusion

This is therefore clear that coroutines reintroduce synchronization the programmer may have wished to avoid when using thread, and that concepts such as the bounded buffer loose most of their interest. Indeed, it does not decouple the *producer* and the *consumer* in our case anymore.

6.7 Coroutines Performance: The Sieve of Eratosthenes

The Sieve of Eratosthenes has different variants. Making it lazy will put the accent on cooperation of the threads, and stopping creating filters at the square root of the maximum number will create less threads. The four combinations of those two parameters have been tested on Mozart 1, Mozart 2 and Mozart-Graal in order to extract some tendencies. The resulting *mean executions* appear of FIGURE 6.7.

Each iteration computes the primes numbers under `N`, and the process is repeated 200 times. We can see Mozart-Graal tends to asymptotically outperform Mozart 1 and Mozart 2 in greedy cases. This seems to confirm coroutine creation is not much of a bottleneck as it is able to spawn more than 1200 threads in 40ms (1 iteration) while doing the computation.

When considering the lazy cases, however, `N` has had to be decreased in order for Mozart-Graal to

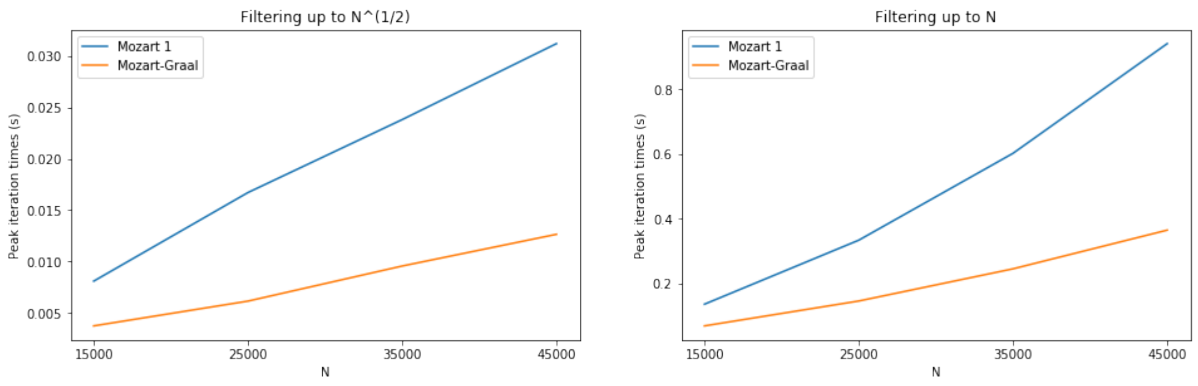


Figure 6.8: Evolution of peak iteration times for the `Sieve` when increasing `N` on Mozart-Graal and Mozart 1

complete in reasonable time. Indeed, because scheduling has not been optimized at all and our coroutines pass execution to each other in a round-robin manner, a full cycle through all the threads is required for each integer number up to `N`. This thus directly points one weakness of our implementation.

Scalability in the greedy case has also been compared for Mozart 1 and Mozart-Graal in Figure 6.8. Average times for late iterations have been measured for different maximal numbers `N`. This shows that the tendency does not inverse as the number of threads to create and execute grows.

Conclusion

The Lazy Sieve of Eratosthenes points out the round-robin execution passing as weakness of our implementation. This could be improved by passing it directly to a thread waiting on a variable that has just been bound, but would also require to take starvation into consideration. Indeed, the current algorithm has the advantage of guaranteeing that if every thread yields or terminate, no thread starves.

Other than that, this first comparison with other implementations seems to reveal Mozart-Graal to be competitive after some warm-up time.

6.8 Performance Comparison With Mozart 1, Mozart 2

In this section, we compare Mozart 1, Mozart 2 and Mozart-Graal on the Project 2014 and on our other benchmarks.

Two partitions from students have been used: the default `joie.dj.oz` and `strugala.dj.oz`. Five runs of ten iterations each have been averaged and reported in Figure 6.9.

We can see that the first execution of the project by Mozart-Graal is already faster that Mozart 2 but quite far from Mozart 1. Mozart-Graal however closes the gap with the latter in the following iterations.

The iteration times for the other benchmarks are provided in figure Figure 6.10. If it is quite exceptional for Mozart-Graal to directly beat Mozart 2, it usually becomes much faster when considering peak performance, sometimes even outperforming Mozart 1.

6.8.0.1 Conclusion

Mozart-Graal does not completely outperform Mozart 1 asymptotically yet, which we have found very interesting. This probably means there are a lot of possibilities for improvements, and confirms the quality of the work behind Mozart 1.

Mozart 2 is less competitive because it was first aiming for maintainability. It should however be faster for small scripts.

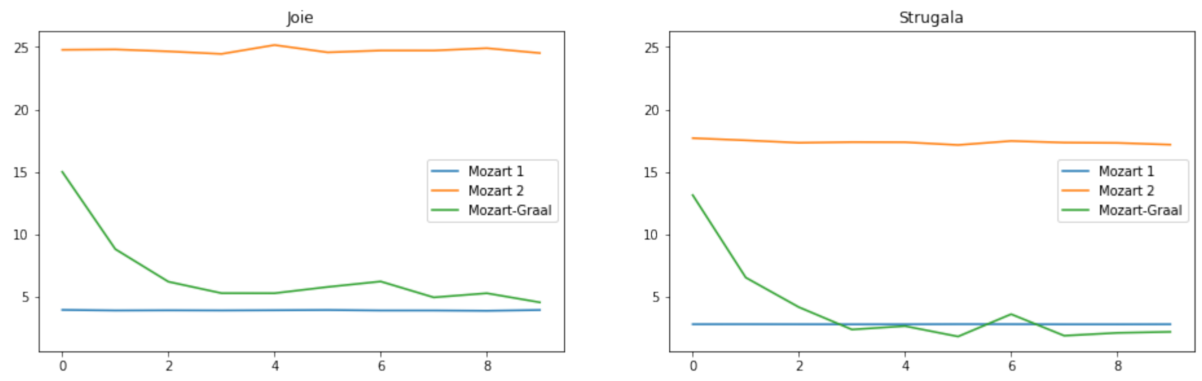


Figure 6.9: Mean iteration times for `Project2014` on Mozart 1, Mozart 2 and Mozart-Graal on the `joie.dj.oz` and `strugala.dj.oz` partitions

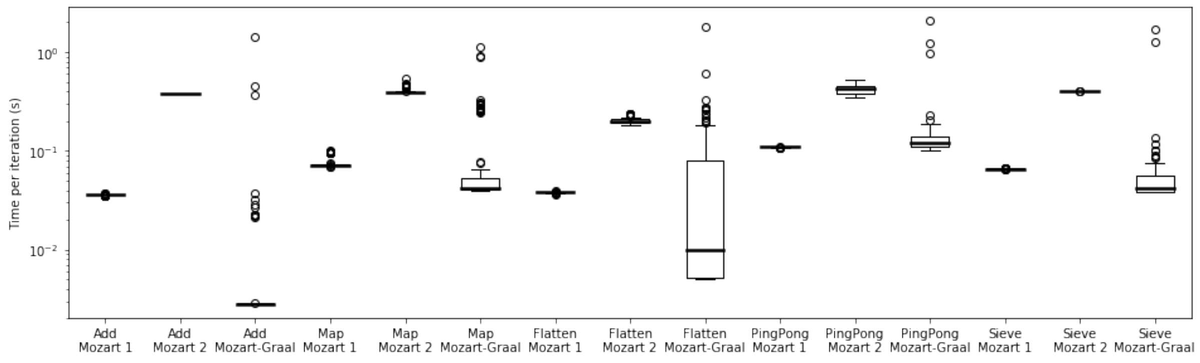


Figure 6.10: Distribution of iteration times on Mozart 1, Mozart 2 and Mozart-Graal for all the benchmarks but `Project2014`

6.9 Memory

In this section, we discuss the memory consumption of Mozart-Graal and compare it with other implementations. We first consider the size of specific data structures, and will compare memory usage between the 3 VMs in general.

We first want to grasp a general idea of the memory consumption for lists and records which are the principal components of Oz applications. The JVM does not propose any way of evaluating the memory occupied by objects. The only measure it provides to that respect is the complete size of the heap, which may change only by executing Oz code. Hopefully, such measures are also provided by Mozart 1 and 2, which allow for a portable estimation.

The idea is to generate a lot of objects between two measures of the heap size, and to divide the total difference by the number of objects generated. To neglect the cost of the execution of the generation, a garbage collection must be performed after the generation while maintaining a reference to the generated objects before the second heap size measure. To keep the heap size difference coherent, a garbage collection must also be performed before the first heap size measure.

We are mainly interested in the sizes of records and conses as they are the most used data structures in the language. Hopefully, they constitute perfect nodes for linked lists in order to maintain references to all generated objects in a light manner. They have been filled with either atoms or small integers. Variables are also included as they are required to build the structure.

The operation has been repeated 1 thousand times for each implementation. The results are provided in Appendix A.3. Once again, the results are very noisy for Mozart-Graal, but taking the median does not seem unreasonable. To assess the stability, measures have also been taken with different numbers of generated objects The resulting sizes appear in Table A.1 which end up surprisingly coherent.

We can see there that it can be expected for Mozart-Graal to consume approximately 5 to 7 times the memory required by Mozart 1, when working on those structures, which is significantly more than the memory required by Mozart 2.

The fact our implementation use the base `DynamicObjectBasic` placeholder for its records is one reason, because it allocates space for primitives that is not used by our implementation.

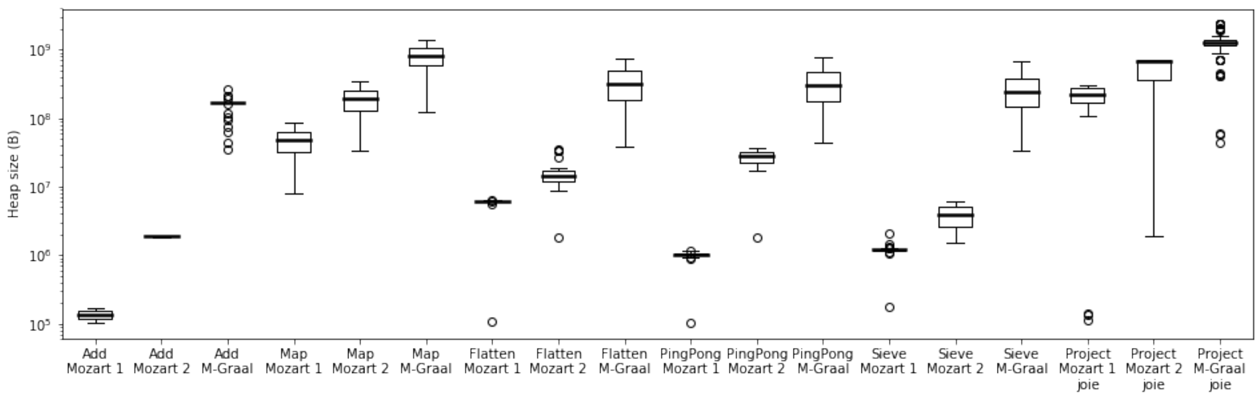


Figure 6.11: Heap size measures for all benchmarks and implementations

Measurements of total heap space have also been realized on the benchmarks between iterations and are reported in Figure 6.11, which tend to confirm this tendency on the Project 2014. For the other benchmarks however, more orders of magnitude separate Mozart-Graal from the others because of the large amount of heap memory allocated for the interpreter and the compiler to get the program running.

Conclusion

Mozart-Graal thus consumes more memory than both Mozart 1 and Mozart 2. It is possible to get closer to Mozart 2 by dropping the placeholder fields for primitives, but the fact Mozart 1 has been written for to run on 32 bits architectures and that it has been very well optimized to this respect make it unapproachable.

6.10 Conclusion

In this chapter, we have illustrated the fact that a non negligible part of the dataflow variables can be avoided thanks to on stack variables. We have also witnessed that the variable selection and the frame slot clearing make Mozart-Graal able to drop most references it will not use anymore, and that this allows a more efficient garbage collection as well as a more enjoyable experience for Oz developers.

We have confirmed that tail calls are well detected and optimized into loops in an efficient manner. The compiler also takes the opportunity to remove the dataflow variables that are not escaping.

We have been able to make unification and equality tests more robust without sacrificing much in performance, using the principles of self-optimizing interpreters.

The importance of inlining has been emphasized, and some strategies to make it happen faster have been proposed based on the performance reached with the systematic splitting of builtins.

The known limitations of the actual implementation with respect to all those subjects have been discussed.

Furthermore, the impact of using coroutines instead of threads has been evaluated, and we have seen the current scheduler can become quite problematic.

Finally, Mozart-Graal has been compared to Mozart 1 and Mozart 2 both in terms of time and memory consumption. It requires more memory than both previous implementation but already comes close to Mozart 1 for asymptotic performance, even though a lot of improvements seem achievable in that sense.

Chapter 7

Future Work

Union-Find for Testing Whether Two Variables are Equivalent

In Section 4.1, the circular list of variables has been introduced as very practical for ensuring $O(1)$ queries on dataflow variables. However, linking two unbound variables requires to test whether both are already linked or not in order to avoid breaking the circular list. This test currently makes a full pass on one list, hence having complexity $O(n)$, n being the size of the biggest equivalence set of both.

This potentially provides creating a set of n equivalent variables a complexity of $O(n^2)$. If pathological cases are rare in practice, using the Union-Find for this test would simply solve the complexity problem while keeping the advantages of the other for queries.

This algorithm could even used in the case of large sets only, to avoid the cost of recursion while sets are small.

Improving Threads

We have seen in Section 6.7 that scheduling in Mozart-Graal is currently far from perfect. Indeed, yielding the execution from a coroutine and coming back to it has linear complexity in the total number of coroutines, even in the case where all other coroutines are waiting on a variable. Optimizing the scheduling, for instance by considering coroutines waiting on just-bound variables, would be a simple improvement. Starvation should ideally be taken in consideration.

Some preemption mechanism could also be added, making the coroutines closer to the authentic Oz threads. This would however require checks to be put in the program in order to prevent routines from being to run indefinitely, which will of course affect performance.

Finally, Mozart-Graal does not currently take advantage of multiple cores for Oz programs. The language could probably be extended to take advantage from them, as it has been made for the Mozart 2 implementation [45].

Broader Variable Optimization Around Call Expressions

For now, call expressions are only optimized when they are explicit tail calls. It would be interesting to apply the technique from Section 4.4.3 to any call expression that ends up being unified with a declared variable. This would as well generalize the tail call detection for all procedures.

Improve Support for Unboxed Primitives

Until now, primitives are only unboxed when stored into stack frames. Values flowing through the graph are not, which implies a lot of boxing and unboxing operations in interpreter mode and could be optimized thanks to the ideas from [20]. The Truffle `DynamicObject`s also have support for storing primitives, but would require to be more careful when comparing shapes for instance.

Smarter Frame Slot Clearing

For now, there are two ideas coming along with Section 5.1.3.

The first is that it would be feasible for thread or lock procedures to clear their environment slots as it is done for local variables everywhere. Because anonymous procedures created by this syntactic sugar are guaranteed to be called only once after their declaration, it would be correct for the static analysis to tag the slots from the external environment as well.

The second idea is that it should be possible to economize nodes performing the slots clearing (i.e. `ResetSlotNodes`). Either because the procedure could be guaranteed to terminate quickly after (only unifications happen after for instance), or because some several `ResetSlotNodes` could be merged when only separated by *quick* operations.

Experimenting With More Dynamic Optimizations

A lot of directions have not been experimented yet. For instance, replacing bound variables by their value on the stack when reading them or trying to dereference variables before performing calls.

Resugaring Oz Kernel Language

Currently, Mozart-Graal takes advantage from the way natural code is written. Detecting self tail calls relies on the name procedures are declared with (before being desugared into a unification), nested expressions directly reduce the need for frame slots. One-time procedures would also most likely be detected when desugared.

The idea is that the Oz kernel language that is very good for bytecode VMs is a bit too low level for Mozart-Graal, so it will render a different and less optimized AST than the human version. It would be great to achieve close equivalence.

Two approaches are therefore possible: Either take the Oz kernel code and generate more human-like code, basically trying to reverse the desugaring, or refine those passes of the static analysis.

Interoperability With the JVM Ecosystem

Allowing Oz to interoperate with existing Java libraries could make Oz applications benefit from features for which no specific builtin would have to be maintained.

Truffle provides its PolyglotEngine for interoperability with Java but also other languages developed on it. Being compliant with it would thus means all of them can be accessed from Oz and that Oz could expose some of its mechanisms to them.

Making use of the Oz Compiler

For now, Mozart-Graal uses Mozart 2's bootcompiler parser. If this works well for the main features of the language, some syntactic constructs encountered here and there are not recognized. In order to avoid duplicating work and to stay in the spirit of previous implementations, the Oz compiler could handle programs down to the Oz kernel language AST or some intermediate, and rely on some other tool to convert it into a Truffle AST.

Chapter 8

Conclusion

This dissertation aimed at discussing Mozart-Graal on its way to become an efficient Oz implementation on Truffle and Graal, while evaluating if the features they propose would suffice to achieve it.

In Chapter 2, we have discussed the main concepts behind self-optimizing AST interpreters and how Truffle provides a convenient way of implementing them. We have also understood in what it constitutes a front-end to the Graal compiler, which is able to recognize and optimize them optimistically. We have seen the basic ideas behind their orchestration to compile dynamic languages efficiently.

In Chapter 3, we have exposed the architecture of Mozart-Graal and have defined its main characteristics. Indeed, we wanted the implemented language to be as close as possible to Oz, up to some limitations such as the quantity of implemented builtins and the concurrency model for threads that is not matched in our case. We have also seen that Mozart-Graal is based on Mozart 2 for the parsing and some parts of the static analysis, the standard library and the tests.

In Chapter 4, we have transposed the key Oz concepts to our interpreter. We have seen how the variables, the dataflow, the environment, the procedures, the different primitive data structures as well as the unification, equality testing, pattern matching and the tail call optimization can be implemented when performance is not a requirement. We have acknowledged the Truffle framework proposes interesting primitives to implement the environment, the procedures, the records and the tail calls, and that its DSL allows to build a system that is easy to reason about.

Chapter 5 has provided improvements in memory usage and performance. First by refining static analysis in order to particularize self tail calls, to avoid indirection for particular cases of declared variables and for releasing references in the stack frames as soon as possible. We have then annotated the interpreter for the compiler to be able to focus on the interesting parts of programs, to perform inlining and to optimize the control flow of tail calls. Furthermore, ways to make those optimization occur faster such as OSR and builtins splitting have been explored. Finally, efficient specializations for unification and equality tests have been proposed.

In Chapter 6, we have attested that part of the dataflow variables can be avoided thanks to the refinements of the static analysis and that this way of implementing the environment and of clearing the slots allows more efficient garbage collections and makes the language less difficult to reason about when it comes to reference retention. We have also confirmed that the Graal compiler takes care of transforming tail calls into loops when possible, and that it removes the overhead of variables that are not escaping from compilation units. We have confirmed that efficient specializations for unification and equality tests bring a significant performance gain in contrast to the general mechanism.

The role of inlining in the general performance has been emphasized, and significant improvements have been obtained with OSR and the builtins splitting. Mozart-Graal has been proved to consume more memory than Mozart 1 and Mozart 2 but could close the gap with Mozart 2. It also seems to have the potential to outperform Mozart 1 on long executions and Mozart 2 on shorter ones. The supplementary synchronization introduced by using coroutines instead of threads has also been illustrated.

Finally, the current limitations of the implementation have been discussed. We propose further developments and are rather confident that with them and the active research made on Truffle and Graal, Mozart-Graal could become a viable option for running all kinds of Oz applications.

Bibliography

[1] Java (programming language). https://en.wikipedia.org/wiki/Java_(programming_language). 17

[2] Java Virtual Machine. https://en.wikipedia.org/wiki/Java_virtual_machine. 17

[3] The Python Programming Language. https://www.python.org/. 17

[4] Jython: Python for the Java Platform. http://www.jython.org/. 17

[5] Ruby Programming Language. https://www.ruby-lang.org/. 17

[6] The Ruby Programming Language on the JVM. http://jruby.org/. 17

[7] Charles Nutter. JRuby: The Pain of Bringing an Off-Platform Dynamic Language to the JVM, 2009. https://www.infoq.com/presentations/nutter-jruby-jvm-lang-summit. 17

[8] Charles Nutter. A First Taste of InvokeDynamic, 2008. http://blog.headius.com/2008/09/first-taste-of-invokedynamic.html. 17

[9] OpenJDK. The Da Vinci Machine Project. http://openjdk.java.net/projects/mlvm/. 17

[10] Oracle. GraalVM - New JIT Compiler and Polyglot Runtime for the JVM. http://www.oracle.com/technetwork/oracle-labs/program-languages/overview/index-2301583.html. 17

[11] Chris Seaton. Ruby Benchmarks Report. http://chrisseaton.com/rubytruffle/deoptimizing/benchmarks. 17

[12] Chrome V8. https://developers.google.com/v8/. 17

[13] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-language compiler benchmarking: Are we fast yet? *SIGPLAN Not.*, 52(2):120–131, November 2016. 17

[14] Oz Programming Language. https://en.wikipedia.org/wiki/Oz_(programming_language). 17

[15] The Mozart Programming System. http://mozart.github.io/. 17

[16] Benoit Daloze. Mozart-Graal repository. https://github.com/eregon/mozart-graal. 17, 18

[17] Modifications to Mozart-Graal. https://github.com/mistasse/mozart-graal/tree/memoire-final. 18

[18] Maxime Istasse. Mozart-Graal Benchmarking repository. https://github.com/mistasse/mozart-graal-benchmarking/tree/memoire-final. 18, 30, 61, 63, 85

[19] Project Jupyter. http://jupyter.org/. 18, 61

[20] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing ast interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM. 20, 77

[21] 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*, GPCE 2014, pages 123–132, New York, NY, USA, 2014. ACM. 21

[22] HotSpot. `https://en.wikipedia.org/wiki/HotSpot`. 22

[23] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial escape analysis and scalar replacement for java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 165:165–165:174, New York, NY, USA, 2014. ACM. 22, 23

[24] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal ir: An extensible declarative intermediate representation. 22, 62

[25] Static single assignment form. `https://en.wikipedia.org/wiki/Static_single_assignment_form`. 22

[26] Cliff Click. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 246–257, New York, NY, USA, 1995. ACM. 23

[27] Oracle. Java HotSpot^{TM} Virtual Machine Performance Enhancements. `http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html`. 23

[28] 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*, VMIL '13, pages 1–10, New York, NY, USA, 2013. ACM. 24

[29] Mozart Documentation. `http://mozart.github.io/mozart-v1/doc-1.4.0/`. 25

[30] Building the JVMCI for Mozart-Graal. `https://github.com/eregon/mozart-graal/tree/master/vm/jvmci`. 25

[31] Scala Programming Language. `https://en.wikipedia.org/wiki/Scala_(programming_language)`. 25

[32] Guillaume Maudoux. Projet Dj'OZ, 2014. `http://perso.uclouvain.be/guillaume.maudoux/dewplayer/`. 26, 61, 65

[33] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* The MIT Press, 1st edition, 2004. 27, 37, 41, 45, 70

[34] Ana Lûcia De Moura and Roberto Ierusalimschy. *Revisiting coroutines.* 2004. 27

[35] Charles Nutter. JRuby - The Hard Parts. `https://www.slideshare.net/CharlesNutter/jruby-the-hard-parts`. 28

[36] Akka. `http://akka.io/`. 28

[37] Parallel Universe. Quasar. `http://www.paralleluniverse.co/quasar/`. 28

[38] Parallel Universe. Implementing, Abstracting and Benchmarking Lightweight Threads on the JVM, 2014. `http://blog.paralleluniverse.co/2014/02/06/fibers-threads-strands/`. 28

[39] Lukas Stadler, Thomas Würthinger, and Christian Wimmer. Efficient coroutines for the java platform. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 20–28, New York, NY, USA, 2010. ACM. 28

[40] Disjoint-set data structure. `https://en.wikipedia.org/wiki/Disjoint-set_data_structure`. 30

[41] 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. 38

[42] Fredrik Skeel Løkke. Scala and Design Patterns-exploring Language Expressivity. *Aarhus Universitet, Datalogisk Institut*, 2009. 44

[43] SSW JKU. Ideal Graph Visualizer. `http://ssw.jku.at/General/Staff/TW/igv.html`. 62

[44] Kevin Menard. A Systematic Approach to Improving TruffleRuby Performance, 2017. `http://nirvdrum.com/2017/05/30/a-systematic-approach-to-improving-truffleruby-performance.html`. 62

[45] Benoit Daloze, Peter Van Roy, Per Brand, and Sébastien Doeraene. Extending Mozart 2 to support multicore processors, 2014. 77

Appendix A

Benchmark Additional Resources

We here provide additional resources for the benchmarks and the methodology applied behind.

Only the code for `Add` is provided because it is the only one discussed extensively. Other codes are however available at [18].

A.1 Negligeability of Time Measures

Figure A.1 provides iteration times for performing simple time measures. This is nothing else than the same noise that appears when we measure the iteration times for our benchmarks.

Ideally, this should be completely negligible with respect to the time taken for each benchmark iteration. We usually try to make the measures at least around to 10 times greater than the noise in order to be able to neglect it.

The only exception is in Figure 6.7 (on the bottom left) where it is not negligible for Mozart 1. In this case, however, Mozart 1 is much faster than Mozart-Graal even with this positive additive noise, and the purpose of this benchmark is all about pointing out a weakness in Mozart-Graal.

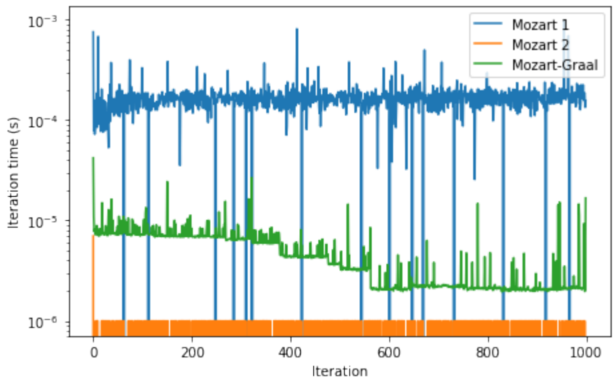


Figure A.1: Noise introduced by time measurements

A.2 Add Program Code

Listing A.1 provides the code for the graph of Appendix A.4. The variant exposed in Listing A.2 yields to the exact same graphs when the loop is made explicit.

Listing A.1: `Add` program

```
functor
import
   Application(exit:Exit)
   MTime(time:Time diff:Diff)
       at 'time.ozf'
   System(showInfo:Show)
define

`$It`= 50
`$N` = 2000000

proc {Add A B R}
   if A == 0 then
      R = A
   else
      {Add A-1 B+1 R}
   end
end

for R in 1..`$It` do
   local
      T0 T1
   in
      {Time T0}
      A := {Add `$N` 0}
      {Time T1}
      {Show " --- "#({Diff T0
          T1})}
   end
end

{Exit 0}
end
```

Listing A.2: Variant of `Add` with declared variables

```
functor
import
   Application(exit:Exit)
   MTime(time:Time diff:Diff)
       at 'time.ozf'
   System(showInfo:Show)
define

`$It`= 50
`$N` = 2000000

proc {Add A B R}
   if A == 0 then
      R = A
   else A1 B1 in
      A-1 = A1
      B+1 = B1
      {Add A1 B1 R}
   end
end

for R in 1..`$It` do
   local
      T0 T1
   in
      {Time T0}
      A := {Add `$N` 0}
      {Time T1}
      {Show " --- "#({Diff T0
          T1})}
   end
end

{Exit 0}
end
```

A.3 Memory Measures

The measures resulting from a run of the `bench_memory` program on all 3 implementations are provided in Figure A.2. The median is taken and normalize per allocated object to obtain Table A.1.

A.4 Graal Graphs

Those graph are imported as vector graphics which allows for zooming indefinitely, which may be required for some in order to distinguish anything.

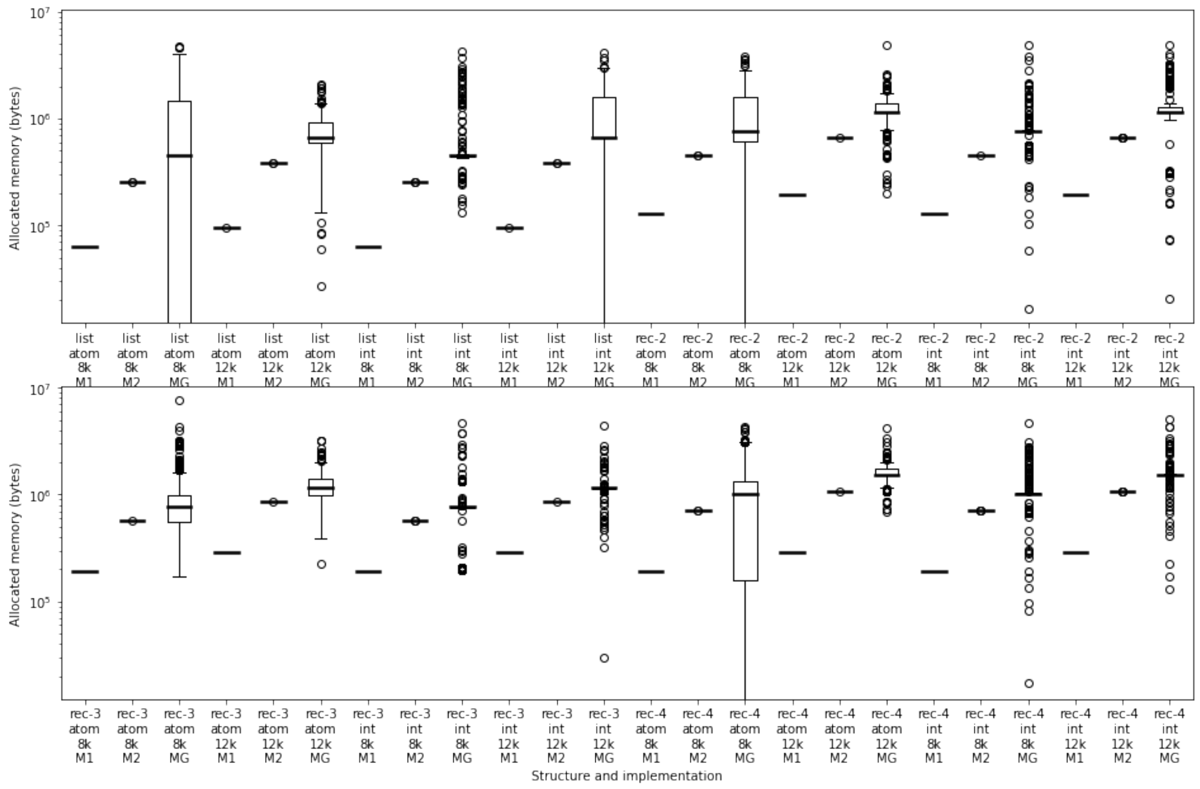
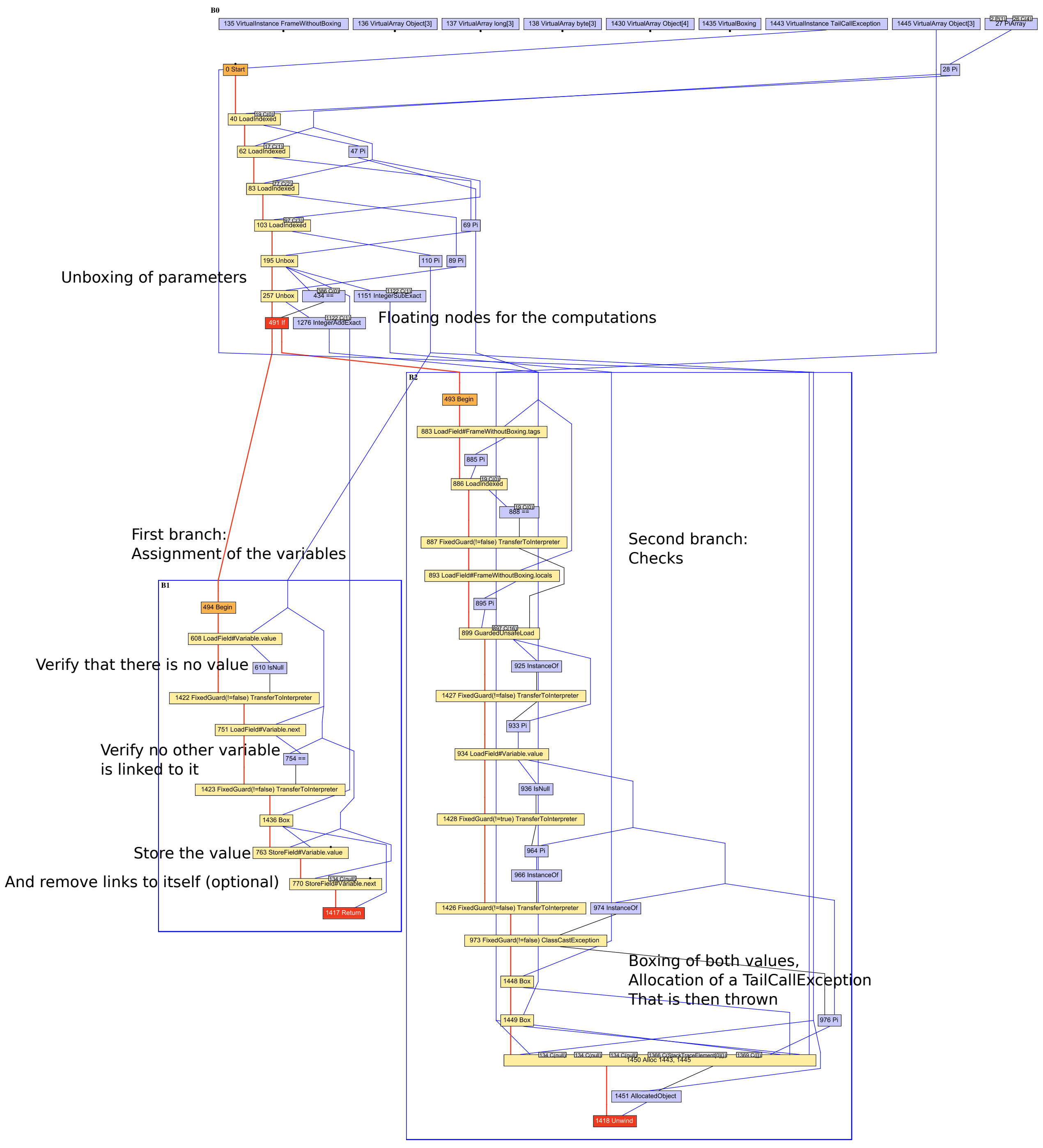


Figure A.2: Difference in heap size around data structure allocation

Node	Item type	Length	Heap allocated per node (Bytes)		
			Mozart 1 (Base)	Mozart 2	Mozart-Graal
Cons	Atom	8000	8	32 (x4.0)	56 (x7.1)
		12000	8	32 (x4.0)	56 (x7.1)
	Int	8000	8	32 (x4.0)	56 (x7.0)
		12000	8	32 (x4.0)	56 (x7.0)
2-features Record	Atom	8000	16	56 (x3.5)	96 (x6.0)
		12000	16	56 (x3.5)	96 (x6.0)
	Int	8000	16	56 (x3.5)	96 (x6.0)
		12000	16	56 (x3.5)	96 (x6.0)
3-features Record	Atom	8000	24	72 (x3.0)	96 (x4.0)
		12000	24	72 (x3.0)	96 (x4.0)
	Int	8000	24	72 (x3.0)	96 (x4.0)
		12000	24	72 (x3.0)	96 (x4.0)
4-features Record	Atom	8000	24	88 (x3.7)	128 (x5.3)
		12000	24	88 (x3.7)	128 (x5.3)
	Int	8000	24	88 (x3.7)	128 (x5.3)
		12000	24	88 (x3.7)	128 (x5.3)

Table A.1: Results for the data structure allocation experiment



B0
135 VirtualInstance FrameWithoutBoxing
136 VirtualArray Object[3]
137 VirtualArray long[3]
138 VirtualArray byte[3]
1430 VirtualArray Object[4]
1435 VirtualBoxing
1443 VirtualInstance TailCallException
1445 VirtualArray Object[3]
27 PiArray
0 Start
28 Pi
40 LoadIndexed
62 LoadIndexed
47 Pi
83 LoadIndexed
103 LoadIndexed
69 Pi
195 Unbox
110 Pi
89 Pi
Unboxing of parameters
257 Unbox
434 ==
1151 IntegerSubExact
491 If
1276 IntegerAddExact
Floating nodes for the computations
B2
493 Begin
883 LoadField#FrameWithoutBoxing.tags
885 Pi
886 LoadIndexed
888 ==
887 FixedGuard(!=false) TransferToInterpreter
893 LoadField#FrameWithoutBoxing.locals
895 Pi
899 GuardedUnsafeLoad
925 InstanceOf
1427 FixedGuard(!=false) TransferToInterpreter
933 Pi
934 LoadField#Variable.value
936 IsNull
1428 FixedGuard(!=true) TransferToInterpreter
964 Pi
966 InstanceOf
1426 FixedGuard(!=false) TransferToInterpreter
974 InstanceOf
973 FixedGuard(!=false) ClassCastException
1448 Box
1449 Box
976 Pi
1450 Alloc 1443, 1445
1451 AllocatedObject
1418 Unwind
Second branch:
Checks
Boxing of both values,
Allocation of a TailCallException
That is then thrown
First branch:
Assignment of the variables
B1
494 Begin
608 LoadField#Variable.value
Verify that there is no value
610 IsNull
1422 FixedGuard(!=false) TransferToInterpreter
751 LoadField#Variable.next
Verify no other variable
is linked to it
754 ==
1423 FixedGuard(!=false) TransferToInterpreter
1436 Box
Store the value
763 StoreField#Variable.value
And remove links to itself (optional)
770 StoreField#Variable.next
1417 Return

Figure A.3: Graal IR obtained for `Add` without self tail calls and without OSR

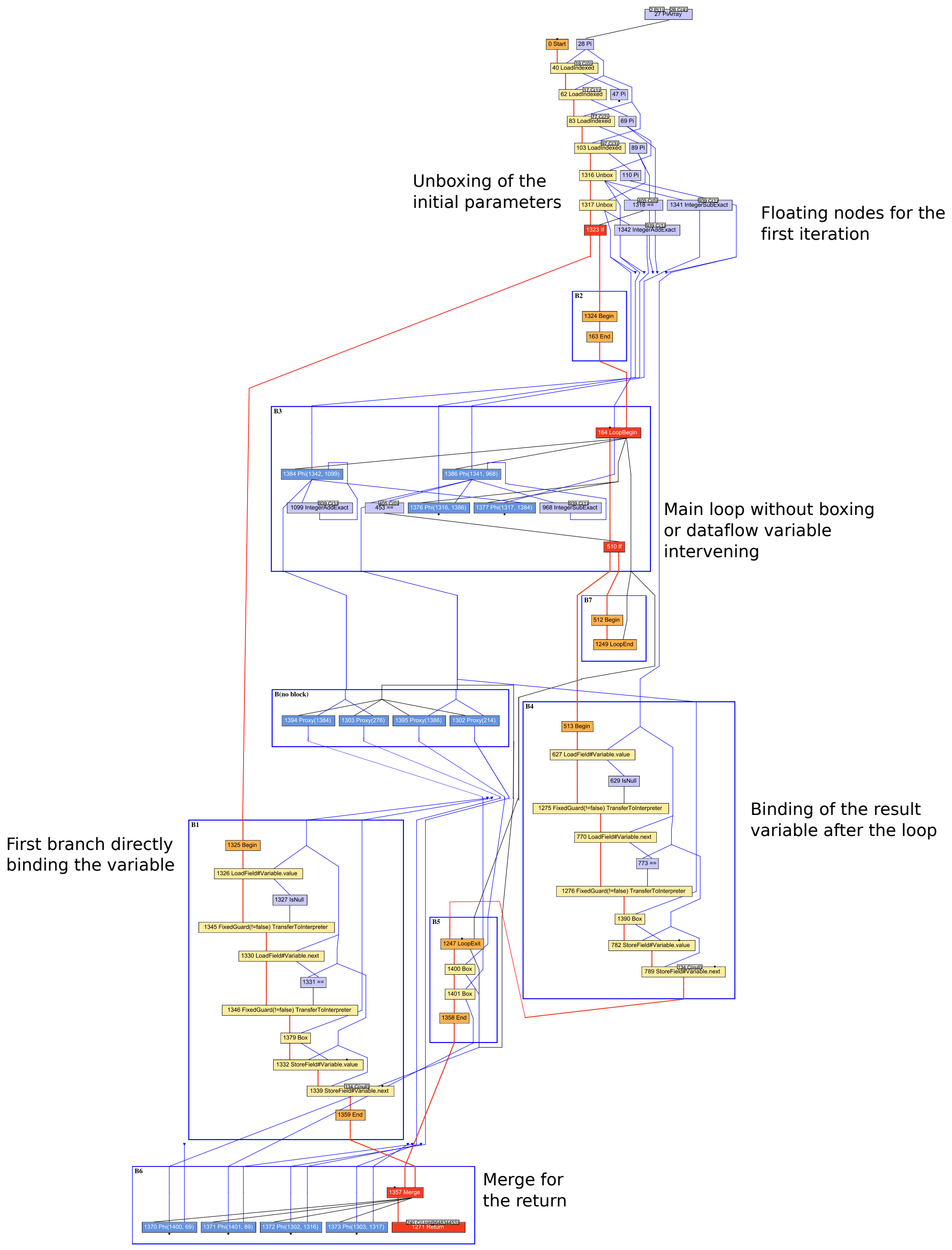


Figure A.4: Graal IR obtained for `Add` with self tail calls and with OSR

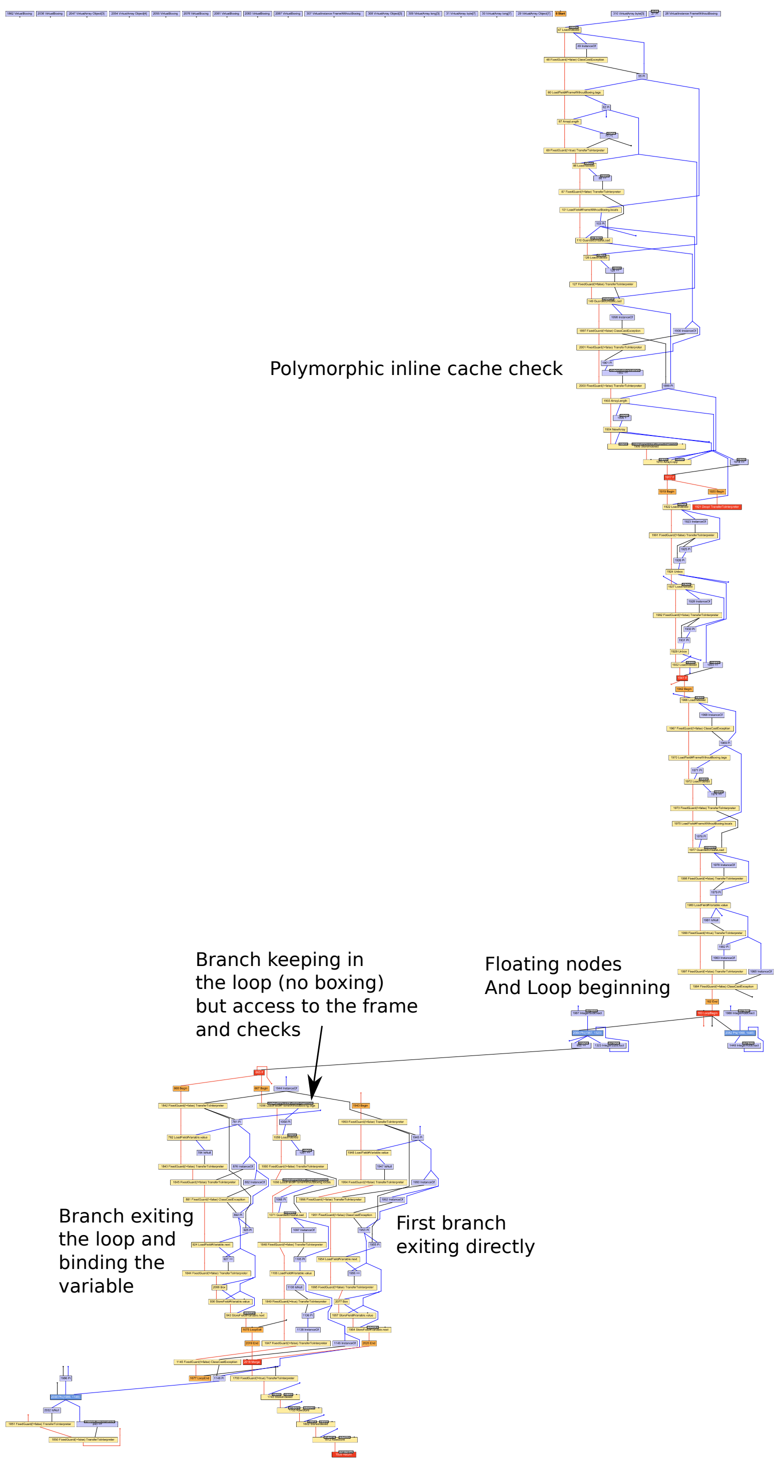

Polymorphic inline cache check
Floating nodes And Loop beginning
Branch keeping in the loop (no boxing) but access to the frame and checks
Branch exiting the loop and binding the variable
First branch exiting directly

Figure A.5: Graal IR obtained for `Add` without self tail calls but with OSR (1st phase)

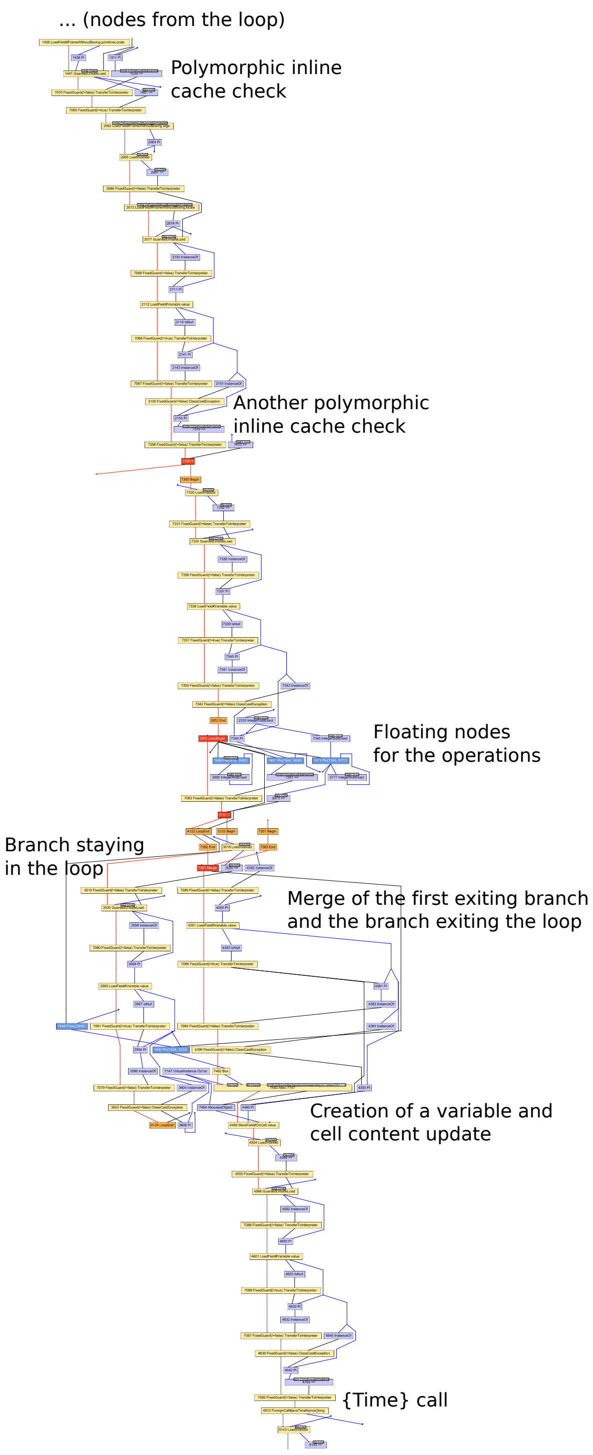

... (nodes from the loop)
Polymorphic inline cache check
Another polymorphic inline cache check
Floating nodes for the operations
Branch staying in the loop
Merge of the first exiting branch and the branch exiting the loop
Creation of a variable and cell content update
{Time} call

Figure A.6: Graal IR obtained for `Add` without self tail calls but with OSR (2^{nd} phase)

Appendix B

Known bugs

- Linking an unbound variable to a view on an unbound variable makes it possible to write through the view.
- Open patterns like `'|'(X Y ...)` would not match `OzCons`
- `A = rec(1 A.1)` does not bind `A` before evaluating `A.1`. The coroutine therefore yields indefinitely.