Chapter 1

Adding JIT compilation to Montuno: MontunoTruffle

1.1 Introduction

In the first part of this thesis, we introduced the theory of dependent types, specified a small, dependently typed language, and introduced some of the specifics of creating an interpreter for this language, under the name Montuno. The second part is concerned with the Truffle language implementation framework: we will introduce the framework itself and the features it provides to language designers, and use it to build a second interpreter.

To reiterate the goal of this thesis, the intent is to create a vehicle for evaluating whether adding just-in-time compilation produces visible improvements in the performance of dependently typed languages. Type elaboration is often a bottleneck in their performance [**?**], and because it involves evaluation of terms, it should be possible to improve using JIT compilation; as optimizing AST evaluation is a good candidate for JIT compilation. We have designed a language that uses features and constructs that are representative of state-of-the-art proof assistants and dependently typed languages, so that such evaluation may be used as a guideline for further work.

This chapter is concerned with building a second interpreter based on Truffle. First, however, we need to introduce the idea of just-in-time compilation in general, and see how the Truffle implements the concept.

1.2 Just-in-time compilation

Just-in-time compilation (JIT) is an optimization technique that is based on the assumption that, when executing a program, its functions (and the functions in the libraries it uses) are only called in a specific pattern, configuration, or with a specific type of data. While a program is running, the JIT compiler optimizes the parts of it that run often; using an electrical engineering metaphor, such parts are sometimes called *"hot loops"*.

Often, when talking about specific optimizations, we will use the terms *slow path* and *fast path*. The fast path is the one for which the program is currently optimized, whereas the

slow paths are all the other ones, e.g., function calls or branches that were not used during the specific program execution.

There are several approaches to JIT compilation: *meta-tracing* and *partial evaluation* are the two common ones.

Meta-tracing A JIT compiler based on meta-tracing records a *trace* of the path taken during program execution. Often used paths are then optimized: either rewritten, or directly compiled to machine code. Tracing, however, adds some overhead to the runtime of the program, so only some paths are traced. While the programmer can provide hints to the compiler, meta-tracing may result in unpredictable peak performance. This technique has been successfully used in projects like PyPy, that is built using the RPython JIT compiler [**?**], or on GHC with mixed results [**?**].

Partial evaluation The second approach to JIT compilation is called *partial evaluation*, also called the *Futamura projection*. The main principle is as follows: where evaluating (running) an interpreter on a program produces some output, partially evaluating (specializing) the interpreter with regards to a program produces an executable. The specializer assumes that the program is constant and can e.g., eliminate parts of the interpreter that will not be used by the program. This is the approach taken by Truffle [**?**].

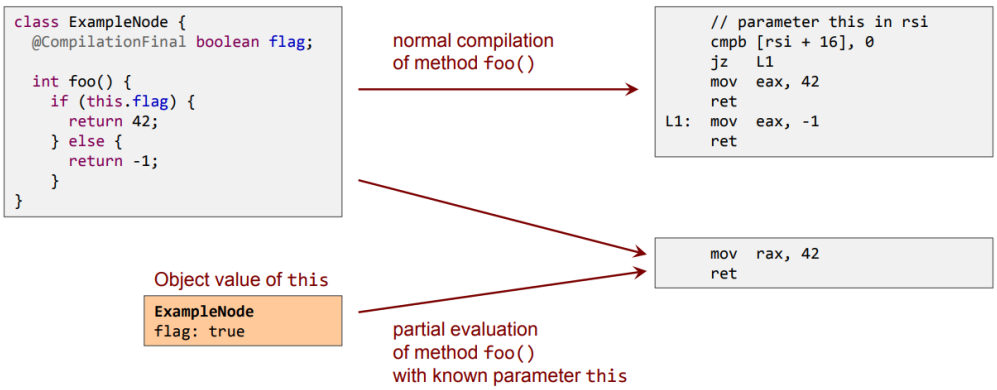

Figure 1.1: Partial evaluation with constant folding (source: oracle.com)

The basic principle is demonstrated in Figure 1.1, on actual code produced by Truffle. In its vocabulary, a `CompilationFinal` value is assumed to be unchanging for a single instance of the program graph node (the field `flag` in the figure), and so the JIT compiler can transform a conditional `if` statement into an unconditional one, eliminating the second branch.

There are, in fact, three Futamura projections, referred to by their ordinals: the *first Futamura projection* specializes an interpreter with regards to a program, producing an executable. The *second Futamura projection* combines the specializer itself with an interpreter, producing a compiler. The third projection uses the specializer on itself, producing a compiler maker. As we will see in later sections, Truffle and GraalVM implement both the first and second projections [**?**].

1.3 Truffle and GraalVM

I have mentioned Truffle several times already in previous chapters. To introduce it properly, we first need to take a look at the Java Virtual machine (JVM). The JVM is a complex platform that consists of several components: a number of compilers, a memory manager, a garbage collector, etc., and the entire purpose of this machinery is to execute `.class` files that contain the bytecode representation of Java, or other languages that run on the JVM platform. During the execution of a program, code is first translated into generic executable code using a fast C1 compiler. When a specific piece of code is executed enough times, it is further compiled by a slower C2 compiler that performs more expensive optimizations, but also produces more performant code.

The HotSpotVM is one such implementation of this virtual machine. The GraalVM project, of which Truffle is a part, consists of several components and the main one is the Graal compiler. It is an Oracle research project that replaces the C2 compiler inside HotSpotVM, to modernize an aging code base written in C++, and replace it with a modern one built with Java [**?**]. The Graal compiler is used in other ways, though, some of which are illustrated in Figure 1.2. We will now look at the main ones.

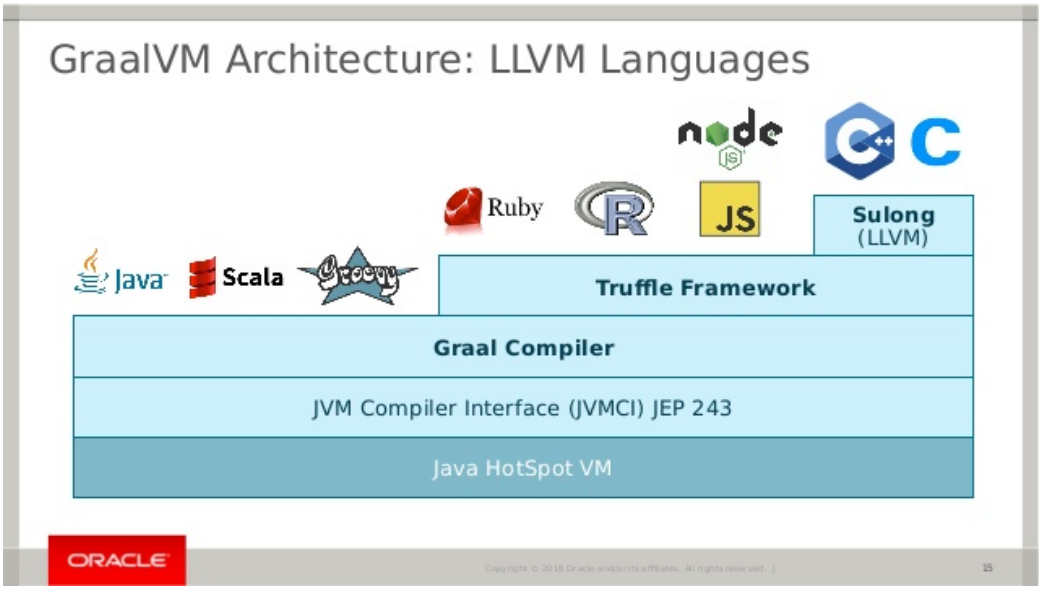

Figure 1.2: GraalVM and Truffle (source: oracle.com)

Graal Graal itself is at its core a graph optimizer applied to program graphs. It processes Java bytecode into a graph of the entire program, spanning across function calls, and reorders, simplifies and overall optimizes it.

It actually builds two graphs in one: a data-flow graph, and an instruction-flow graph. Data-flow describes what data is required for which operation, which can be reordered or optimized away, whereas the instruction-flow graph stores the actual order of instructions as the will happen on the processor: see Figure 1.3.

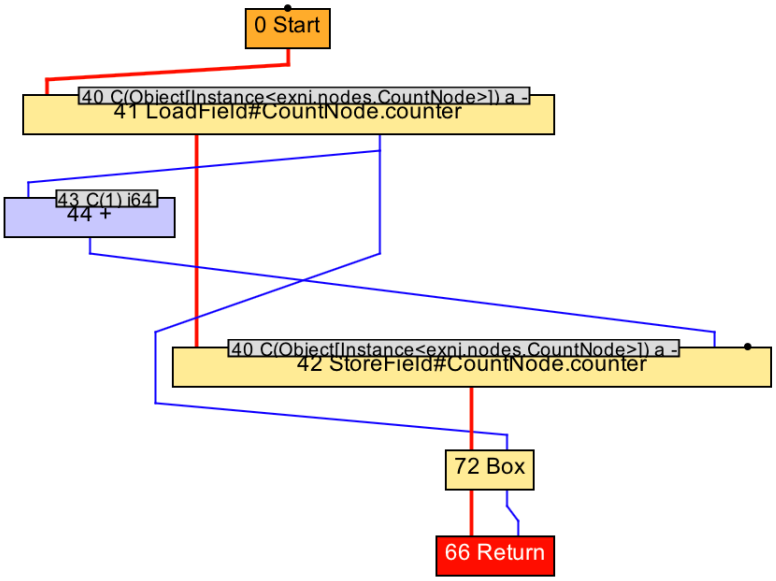

Figure 1.3: Graal program graph, visualized using IGV (source: norswap.com)

SubstrateVM As Graal is a standalone Java library, it can also be used in contexts other than the HotSpotVM. SubstrateVM is an alternative virtual machine that executes Graal-optimized code. It does not perform just-in-time optimizations, though, but uses Graal as an ahead-of-time compiler. The result is a small stand-alone executable file that does not depend on a JVM being installed on a machine, called a *Native Image*. By replacing JIT compilation with ahead-of-time, these binaries start an order-of-magnitude faster than regular Java programs, and can be freely copied between machines, similar to Go or Rust binaries [**?**].

Truffle The Graal program graph, Graal IR, is a directed graph structure in static single assignment form. As it is implemented in Java itself, the graph structure is extensible [**?**], and it is this capability that makes Truffle possible. Truffle is, in essence, a graph manipulation library and a set of utilities for creating these graphs. These graphs are the abstract syntax tree of a language: each node has an `execute` method, calling it returns the result of evaluating the expression it represents.

Interpreter/compiler When creating a programming language, There is a trade-off between writing a interpreter and a compiler. An interpreter is straight-forward to implement and each function in the host language directly encodes the semantics of a language construct, but the result can be rather slow: compared to the language in which the interpreter is written, in can be slower often by a factor to 10x to 100x [**?**]. A compiler, on the other hand, does not execute a program directly, but instead maps its semantics onto the semantics of a different virtual machine, be it the JVM, LLVM, or x86 assembly.

Truffle attempts to side-step this trade-off by making it possible to create an interpreter that can be compiled on-demand via JIT when interpreted or ahead-of-time into a Native

Image; the result should be an interpreter-based language implementation with has the performance of a compiled language and access to all JVM capabilities (e.g. memory management). Instead of running an interpreter inside a host language like Java, the interpreter is embedded one layer lower, into a program graph that runs directly on the JVM and is manipulated by the Truffle runtime that runs next to it.

Polyglot Truffle languages can all run next to one another on the JVM. As a side-effect, communication between languages is possible without the need for usual FFI (foreign function interface) complications. As all values are JVM objects, access to object properties uses the same mechanisms across languages, as does function invocation. In effect, any language from Figure 1.2 can access libraries and values from any other such language.

TruffleDSL Truffle is a runtime library that manages the program graph and a number of other concerns like variable scoping, or the object storage model that allows objects from different languages to share the same layout. TruffleDSL is a user-facing library in the form of a domain-specific language (DSL) that aids in simplifies construction specialized Truffle node classes, inline caches, language type systems, and other specifics. This DSL is in the form of Java *annotations* that give additional information to classes, methods or fields, so that a DSL processor can then use them to generate the actual implementation details.

Instrumentation The fact that all Truffle languages share the same basis, the program graph, means that a shared suite of tooling could be built on top of it: a profiler (VisualVM), a stepping debugger (Chrome Debugger), program graph inspector (IGV), a language server (Graal LSP). We will use some of these tools in further sections.

1.4 Truffle in detail

This concludes the general introduction to Truffle and GraalVM. Now we will look at the specifics of how a Truffle language differs from the type of interpreter we created previously.

The general concept is very similar to the previously created AST interpreter: there is again a tree data structure at the core, where each node corresponds to one expression that can be evaluated. The main differences are in a number of details that were previously implicit, though, like the simple action of "calling a function" which in Truffle involves the interplay of, at a minimum, five different classes.

Figure 1.4 shows the components involved in the execution of a Truffle language. Most of our work will be in the parts labeled "AST", "AST interpreter", and "AST rewriting". All of these involve the contents of the classes that form the abstract syntax tree, as individual graph nodes contain their data, but also their interpretation and rewriting specifics.

Overall, the implementation of a Truffle language can be divided into a few categories. Some of the classes to be sub-classed and methods to be implemented are included in paren-

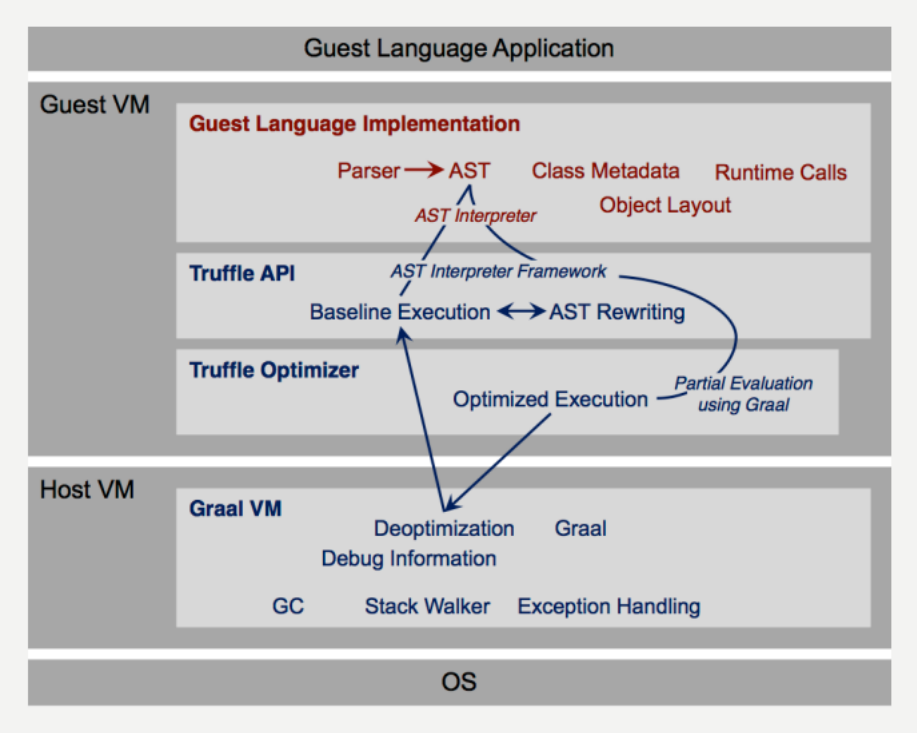

Figure 1.4: Architecture of a Truffle language (source: oracle.com)

theses to give a brief idea of the terminology we will use, although we will expand on each one momentarily. These blocks are:

- language execution (`Launcher`),
- language registration (`Language`, `Context`, `ParsingRequest`),
- program entry point (`RootNode`, `CallTarget`),
- node execution (`VirtualFrame`, `execute`, `call`),
- node specialization (`Specialize`, `Profile`, `Assumption`),
- value types (`TypeSystem`, `ValueType`),
- compiler directives (`transferToInterpreter`, `TruffleBoundary`),
- function calls (`InvokeNode`, `DispatchNode`, `CallNode`),
- object model (`Layout`, `Shape`, `Object`),
- and others (instrumentation, `TruffleLibrary` interfaces, threads).

Launcher The entry point to a Truffle language is a `Launcher` (Listing 1.1). This component handles processing command-line arguments, and uses them to build a language execution context. A language can be executed from Java directly without a `Launcher`, but it handles all GraalVM-specific options and switches, many of which we will use later,

and correctly builds a the language execution environment, including all debugging and other tools that the user may decide to use.

```
class MontunoLauncher : AbstractLanguageLauncher() {
    companion object {
        @JvmStatic fun main(args: Array<String>) = Launcher().launch(args)
    }
    override fun getDefaultLanguages(): Array<String> = arrayOf("montuno");
    override fun launch(contextBuilder: Context.Builder) {
        contextBuilder.arguments(getLanguageId(), programArgs)
        Context context = contextBuilder.build()
        Source src = Source.newBuilder(getLanguageId(), file).build()
        Value returnVal = context.eval(src)
        return returnVal.execute().asInt()
    }
}
```

Listing 1.1: A minimal language `Launcher`

Language registration A language's primary object is a `Language`, whose primary purpose is to answer `ParsingRequest`s with the corresponding program graphs, and to manage execution `Context`s that contain global state of a single language process. It also specifies general language properties like support for multi-threading, or the MIME type and file extension, and decides which functions and objects are exposed to other Truffle languages.

```
@TruffleLanguage.Registration(
    id = "montuno", defaultMimeType = "application/x-montuno"
)
class Language : TruffleLanguage<MontunoContext>() {
    override fun createContext(env: Env) = MontunoContext(this)
    override fun parse(request: ParsingRequest): CallTarget {
        CompilerAsserts.neverPartOfCompilation()
        val parseAST = parse(request.source)
        val nodes = parseAST.map { toNode(it, this) }.toTypedArray()
        return Truffle.getRuntime().createCallTarget(ProgramRootNode(nodes))
    }
}
```

Listing 1.2: A minimal `Language` registration

Program entry point Listing 1.2 demonstrates both a language registration and the creation of a `CallTarget`. A call target represents the general concept of a "callable object", be it a function or a program, and a single call to a call target corresponds to a single stack `VirtualFrame`, as we will see later. It points to the `RootNode` at the entry point of a program graph, as shown in Figure 1.5.

A `CallTarget` is also the basic optimization unit of Truffle: the runtime tracks how many times a `CallTarget` was entered (called), and triggers optimization (partial evaluation) of the program graph as soon as a threshold is reached.

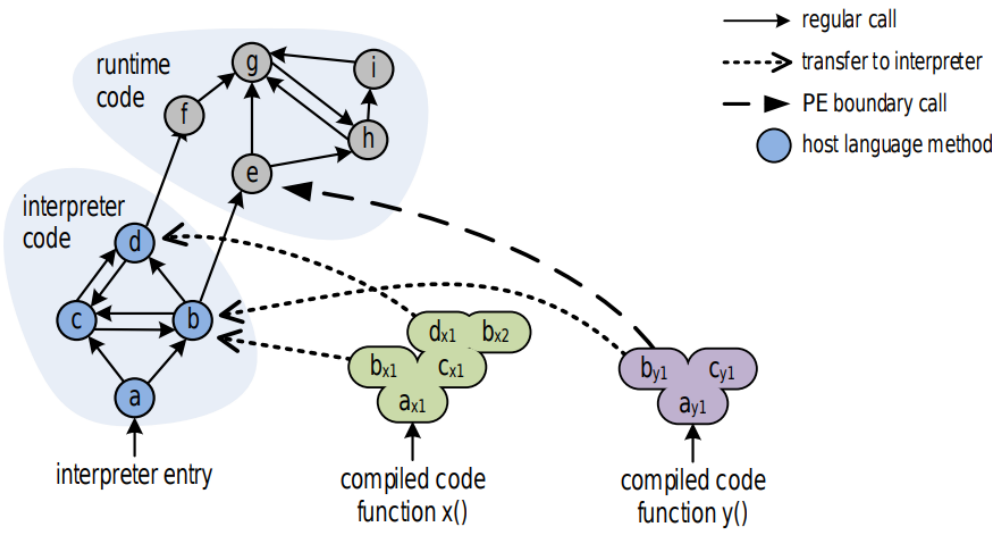

Figure 1.5: Combination of regular and partially-evaluated code (source: oracle.com)

Node execution A `RootNode` is a special case of a Truffle `Node`, the basic building block of the program graph. Each node has a single way of obtaining the result of evaluating the expression it represents, the `execute` method. We may see nodes with multiple `execute` methods later, but they are all ultimately translated by the Truffle DSL processor into a single method: Truffle will pick the most appropriate one based on the methods' return type, arguments types, or user-provided *guard* expressions.

Listing 1.3 contains an example of with two nodes. They share a parent class, `LanguageNode`, whose only method is the most general version of `execute`: one that takes a `VirtualFrame` and returns anything. An `IntLiteralNode` has only one way of providing a result, it returns the literal value it contains. `AddNode`, on the other hand, can add either integers or strings, so it uses another Truffle DSL option, a `@Specialization` annotation, which then generates the appropriate logic for choosing between the methods `addInt`, `addString`, and `typeError`.

```
abstract class LanguageNode : Node() {
  abstract fun execute(frame: VirtualFrame): Any
}
class IntLiteralNode(private val value: Long) : LanguageNode() {
  override fun execute(frame: VirtualFrame): Any = value
}
abstract class AddNode(
  @Child val left: LanguageNode, @Child val right: LanguageNode,
) : LanguageNode() {
  @Specialization fun addInt(left: Int, right: Int) = left + right
  @Specialization fun addString(left: String, right: String) = left + right
  @Fallback fun typeError(left: Any?, right: Any?): Unit
    = throw TruffleException("type error")
}
```

Listing 1.3: Addition with type specialization

Specialization Node specialization is one of the main optimization capabilities of Truffle. The `AddNode` in Listing 1.3 can handle strings and integers both, but if it only ever receives integers, it does not need to check whether its arguments are strings on the *fast path* (the currently optimized path). Using node specialization, the `AddNode` can be in one of four states: uninitialized, integers-only, strings-only, and both generic. Whenever it encounters a different combination of arguments, a specialization is *activated*. Overall, the states of a node form a directed acyclic graph: a node can only ever become more general, as the Truffle documentation emphasize.

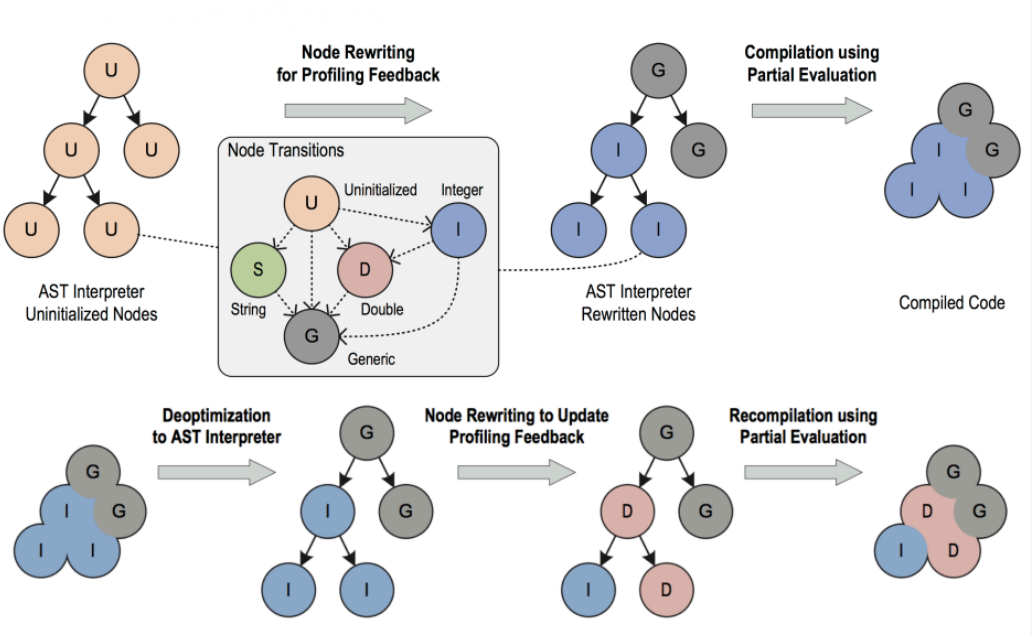

Figure 1.6: Node optimization and deoptimization in Truffle (source: oracle.com)

(De)optimization Node specialization combined with the optimization of a `CallTarget` when called enough times are sufficient to demonstrate the process of JIT compilation in Truffle. Figure 1.6 demonstrates this process on a node type with several more state transitions. When a node reaches a stable state where no more specializations take place, that part of the program graph may be partially evaluated. This produces efficient machine code instead of slow interpreter-based code, specialized for the nodes' current state.

However, this compilation is *speculative*, it assumes that nodes will not encounter different values, and this is encoded in explicit *assumption* objects. When these assumptions are invalidated, the compiled machine code is discarded, and the nodes revert back to their non-optimized form. This process is called *deoptimization* [**?**], and can be explicitly invoked using the Truffle method `transferToInterpreter`.

After a deoptimization, the states of nodes should again stabilize, so that they may be partially evaluated into efficient machine code once more. Often, this (de)optimization process repeats multiple times during the execution of a single program: the period from the start of a program until a stable state is called the *warm-up* phase.

Value types Nodes can be specialized based on various criteria, but the above-mentioned specialization with regards to the type of arguments requires that these types are all declared and aggregated into a `TypeSystem` object and annotation. These are again processed by Truffle DSL into a class that can check the type of a value (`isUnit`, `asBoolean`), and perform implicit conversion between them (`castLong`). Listing 1.4 demonstrates a `TypeSystem` with a custom type `Unit` and the corresponding required `TypeCheck`, and with an implicit type-cast in which an integer is implicitly convertible into a long integer.

```
@CompilerDirectives.ValueType
object Unit

@TypeSystem(Unit::class, Boolean::class, Int::class, Long::class)
open class Types {
  companion object {
    @ImplicitCast
    fun castLong(value: Int): Long = value.toLong()
    @TypeCheck(Unit::class)
    fun isUnit(value: Any): Boolean = value === Unit
  }
}
```

Listing 1.4: A `TypeSystem` with an implicit cast and a custom type

Function invocation An important part of the implementation of any Truffle language consists of handling function calls. A common approach in multiple Truffle is as follows: Given an expression like `fibonacci(5)`. This expression is evaluated in multiple steps: an `InvokeNode` resolves the function that the expression refers to (`fibonacci`) into a `RootNode` and a `CallTarget`, and evaluates its arguments (`5`). A `DispatchNode` creates a `CallNode` for the specific `CallTarget` and stores it in a cache, and finally a `CallNode` what actually performs the switch from one part of the program graph to another, building a stack `Frame` with the function's arguments, and entering the `RootNode`.

```
class ReadLocalVarNode(val name: String) : Node {
  fun execute(frame: VirtualFrame): Any {
    val slot: FrameSlot = frame.getFrameDescriptor().findFrameSlot(name)
    return frame.getValue(slot ?: throw TruffleException("$name not found"));
} }
class WriteLocalVarNode(val name: String, val body: Node) : Node {
  fun execute(frame: VirtualFrame): Unit {
    val slot: FrameSlot = frame.getFrameDescriptor().addFrameSlot(name)
    frame.setObject(slot, body.execute(frame));
} }
```

Listing 1.5: Basic operations with a `Frame`

Stack frames `Frame`s were mentioned several times already: they are Truffle's abstraction of a stack frame. In general, stack frames contain variables and values in the local scope of a function, those that were passed as its arguments and those declared in its body. In Truffle, this is encoded as a `Frame` object, and it is passed as an argument to all `execute` functions. Frame layout is set by a `FrameDescriptor` object, which contains `FrameS-`

`lots` that refer to parts of the frame. Listing 1.5 demonstrates two nodes that interact with a `Frame`: a reference to a local variable, and a local variable declaration.

There are two kinds of a `Frame`, virtual and materialized frames. A `VirtualFrame` is, as its name suggests, virtual, and the values in it can be freely optimized by Truffle, reorganized, or even passed directly in registers without being allocated on the heap (using a technique called Partial Escape Analysis). A `MaterializedFrame` is not virtual, it is an object at the runtime of a program, and it can be stored in program's values or nodes. A virtual frame is preferable in almost all cases, but e.g., implementing closures requires a materialized frame, as it needs to be stored in a `Closure` object. This is shown in Listing 1.6, where `frame.materialize()` captures a virtual frame and stores it in a closure.

```
@CompilerDirectives.ValueType
data class Closure(
  val callTarget: RootCallTarget,
  val frame: MaterializedFrame,
)
class ClosureNode(val root: FunctionRootNode) : Node {
  fun executeClosure(frame: VirtualFrame): Closure = Closure(
    Truffle.getRuntime().createCallTarget(root),
    frame.materialize()
  )
}
```

Listing 1.6: A closure value with a `MaterializedFrame`

Caching These were the main features required for writing a Truffle language, but there are several more tools for their optimization, the first one being *inline caching*. This is an old concept that originated in dynamic languages, where it is impossible to statically determine the call target in a function invocation, so it is looked up at runtime. Most function call sites only use a limited number of call targets, so these can be cached. As the cache is a local one, placed at the call site itself, it is called an *inline cache*. This concept is used for a number of other purposes, e.g., caching the `FrameSlot` in an assignment operator, or the `Property` slot in an object access operation.

In the case of function dispatch, a `DispatchNode` goes through the stages: *uninitialized; /monomorphic*, specialized to a single call target; *polymorphic*, stores a number of call targets small enough, that the cost of searching the cache is smaller than the cost of function lookup; and *megamorphic*, when the number of call targets exceeds the size of the cache, and every function call is looked up again. Figure 1.7 demonstrates this on a DispatchNode, adding a polymorphic cache with size 3, and also demonstrates the Truffle DSL annotations `Cached` and `guards`. The cache key is the provided `CallTarget`, based on which a `DirectCallNode` is created and cached as well. The megamorphic case uses an `IndirectCallNode`: in a `DirectCallNode`, the call target can be inlined by the JIT compiler, whereas in the indirect version it can not.

Guards Figure 1.7 also demonstrates another optimization feature, a generalization of nodes specializing themselves based on types or arguments. A `Specialization` annotation can have arbitrary user-provided *guards*. These are often used in tandem with a cache,

```
abstract class DispatchNode : Node {
  abstract fun executeDispatch(
    frame: VirtualFrame, callTarget: CallTarget, args: Array<Any>): Any

  @Specialization(limit = "3", guards = "callTarget == cachedCallTarget")
  fun doDirect(
    frame: VirtualFrame, callTarget: CallTarget, args: Array<Any>,
    @Cached("callTarget") cachedCallTarget: CallTarget,
    @Cached("create(cachedCallTarget)") callNode: DirectCallNode
  ) = callNode.call(args)

  @Specialization(replaces = "doDirect")
  fun doIndirect(
    frame: VirtualFrame, callTarget: CallTarget, args: Array<Any>,
    @Cached("create()") callNode: IndirectCallNode
  ) = callNode.call(callTarget, args)
}
```

Listing 1.7: Polymorphic and megamorphic inline cache on a `DispatchNode`

or with complex type specializations. In general, using a `Specialization` makes it possible to choose the most optimal node implementation based on its situation or configuration.

Profiles Another tool for optimization are *profiles*. These are objects that the developer can use to track which branch did code execution take: in the implementation of an `if` statement, or when handling an exception. The compiler will use the information collected during optimization, e.g., when the condition in an `if` statement was true every time, and it is tracked in a `ConditionProfile`, the compiler will omit the `else` branch during compilation.

Assumptions *Assumptions* are the last tool that a developer can use to provide more information to the compiler. Unlike profiles and specializations that are local to a node, assumptions are global objects whose value can be changed from any part of a program graph. An assumption is *valid* when created, and it can be *invalidated*, which triggers which triggers deoptimization of any code that relies on it. A typical use of assumptions is shown in Figure 1.8 [**?**], in which TruffleRuby relies on the fact that global variables are only seldom changed and can be cached. A `ReadGlobalVarNode` reads the value of the global variable only the first time, and relies on two assumptions afterwards. These are invalidated whenever the value of the variable changes, and the cached value is discarded.

```
@Specialization(assumptions = [
  "storage.getUnchangedAssumption()",
  "storage.getValidAssumption()"
])
fun readConstant(
  @Cached("getStorage()") storage: GlobalVariableStorage,
  @Cached("storage.getValue()") value: Any
) = value
```

Listing 1.8: Cached reading of a global variable using assumptions [**?**]

Inlining During optimization, the Graal compiler replaces `DirectCallNodes` with the contents of the call target they refer to, performing function *inlining* [**?**]. Often, this is the optimization with the most impact, as replacing a function call with the body of the callee means that many other optimizations can be applied. For example, if a `for` loop contains only a function call and the function is inlined, then the optimizer could further analyze the data flow, and potentially either reduce the loop to a constant, or to a vector instruction.

There are potential drawbacks, and Truffle documentation warns developers to place `TruffleBoundary` annotations on functions that would be expanded to large program graphs, like `printf`, as Graal will not ever inline a function through a `TruffleBoundary`.

Splitting Related to inlining, a call target can also be *split* into a number of *monomorphic* call targets. Previously, we saw an `AddNode` that could add either integers or strings. If this was a global or built-in function that was called from different places with different configurations of arguments, then this node could be split into two: one that only handles integers and one for strings. Only the monomorphic version would then be inlined at a call site, leading to even better possibility of optimizations.

Both of these two techniques, inlining and splitting, are guided by Graal heuristics, and they are generally one on the last optimization techniques to be checked when there are no more gains to be gained from caching or specializations.

```
@Specialization(guards = [
  "addr.key() == keyCached",
  "shapeCached.check(addr.frame())"
], limit = "20")
fun doSetCached(
  addr: FrameAddr, value: Any,
  @Cached("addr.key()") keyCached: Occurrence,
  @Cached("addr.frame().getShape()") shapeCached: Shape,
  @Cached("shapeCached.getProperty(keyCached)") slotProperty: Property
): Unit {
  slotProperty.set(addr.frame(), value, shapeCached)
}
```

Listing 1.9: Accessing an object property using a `Shape` and a Property [**?**]

Object model Truffle has a standard way of structuring data with fixed layout, called the Object Storage Model [**?**]. It is primarily intended for class instances that have a user-defined data layout, but e.g., the meta-interpreter project DynSem [**?**] uses it for variable scopes, and TruffleRuby uses it to make C `structs` accessible from Ruby as if they were objects. Similar to `Frames`, an empty `DynamicObject` is instantiated from a `Shape` (corresponds to a `FrameDescriptor`) that contains several instances of a `Property` (corresponds to a `FrameSlot`). Figure 1.9 shows the main method of a node that accesses an object property, also utilizing a polymorphic cache.

Interop As previously mentioned, it is possible to evaluate *foreign* code from other languages using functions like *eval*, referred to as *polyglot*. However, Truffle also makes it

possible to use other languages' *values*: to define a foreign function and use it in the original language, to import a library from a different language and use it as if it was native. This is referred to as an interoperability message protocol or *interop*, for short.

This is made possible by Truffle *libraries*, that play a role similar to *interfaces* in object-oriented languages [**?**], and describe capabilities of `ValueType`s. A library *message* is an operation that a value type can support, and it is implemented as a special node in the program graph, as a nested class inside the value type. The `ValueType`s of a foreign language then need to be mapped based on these libraries into a language: a value that implements an `ArrayLibrary` can be accessed using array syntax, see Listing 1.10. Libraries are also used for polymorphic operations inside a language if there is a large amount of value types, to remove duplicate code that would otherwise be spread over multiple `Specialization`s.

```
class ArrayReadNode : Node {
  @Specialization(guards = "arrays.isArray(array)", limit = "2")
  fun doDefault(
    array: Object, index: Int, @CachedLibrary("array") arrays: ArrayLibrary
  ): Int = arrays.read(array, index)
}
```

Listing 1.10: Array access using a `Library`[1]

1.5 Mapping concepts to Truffle

1.5.1 How Truffle can help?

While the framework is a general language implementation framework, many concepts and features are based on speculative optimization, which is best applicable in dynamically-typed languages, and first need to be mapped onto the features required by our type elaboration and normalization procedures.

Truffle is not primarily aimed at statically-typed languages or functional languages. Its most easily accessible benefits lie in speculative optimization of dynamically typed code and inline caches, where generic object-oriented code can be specialized to a specific value type. Statically-typed languages have a lot more information regarding the values that will flow through a function, and e.g. GHC has a specific *specialization* compiler pass.

However, there is a lot of overlap between the static optimizations done by e.g. GHC and runtime optimizations done by Graal. An example would be unfolding/inlining, where the compiler needs to make a single decision of whether to replace a call to a function with its definition – a decision that depends on the size of the definition, whether they are in the same module, and other heuristics [**?**]. A Truffle interpreter would be able to postpone the decision until execution time, when the definition could be inlined if the call happened enough times.

Its execution model is a tree of nodes where each node has a single operation `execute` with multiple specializations. The elaboration/evaluation algorithm from the previous chapter,

[1]Source: https://www.graalvm.org/graalvm-as-a-platform/language-implementation-framework/TruffleLibraries/

however, has several interleaved algorithms (infer, check, evaluate, quote) that we first need to graft on to the Truffle execution model.

In dynamic interpreters that Truffle is aimed at, it is easy to think of the interpreter structure as "creating a graph through which values flow".

1.5.2

1.5.3 Functional languages on Truffle

— one paragraph each, specialties

For inspiration, I have looked at a number of other functional languages created using Truffle: a number of theses (TruffleClojure [**?**], TrufflePascal [**?**], Mozart-Oz [**?**]), two Oracle projects (FastR [**?**], TruffleRuby [**?**]), and a few other projects that will be mentioned throughout the text.

- https://github.com/enso-org/enso/
- https://github.com/cesquivias/mumbler
- DynSem, [**?**] - OSM for frames/scopes

While I was finalizing this thesis, the Enso project was publicly released. It is a dependently-typed language that implements many of the same principles that I do in this thesis. I have attempted to incorporate some of its solutions that were better than the solutions that were originally presented here, for the sake of comparison. These will be mentioned whenever relevant.

Evaluate languages on:

- overall project structure and runtime flow
- global/local names and environment handling
- calling convention
- lazy evaluation
- closure implementation
- graph manipulation, TruffleBoundaries, specializations

Truffled PureScript https://github.com/slamdata/truffled-purescript/

Purescript is a derivative of Haskell, originally aimed at frontend development. Specific to Purescript is eager evaluation order, so the Truffle interpreter does not have to implement thunks/delayed evaluation.

Simple node system compared to other implementations:

- types are double and Closure (trivial wrapper around a RootCallTarget and a MaterializedFrame)
- VarExpr searches for a variable in all nested frames by string name
- Data objects are a HashMap
- ClosureNode materializes the entire current frame
- AppNode executes a closure, and calls the resulting function with a { frame, arg }
- CallRootNode copies its single argument to the frame
- IR codegen creates RootNodes for all top-level declarations, evaluates them, stores the result, saves them to a module Frame
- Abstraction == single-argument closure

FastR One of the larger Truffle languages, a replacement for GNU R, which was "made for statistics, not performance". Faster without Fortran than with (no native FFI boundary, allows Graal to optimize through it)

[**?**]

Interop with Python, in particular - scipy + R plots

Node replacement for specializing nodes, or when an assumption gets invalidated and the node should be in a different state (AbsentFrameSlot, ReplacementDispatchNode, CallSpecialNode, GetMissingValueNode, FunctionLookup.

```
val ctx = Context.newBuilder("R").allowAllAccess(true).build();
ctx.eval("R", "sum").execute(arrayOf<Int>(1,2,3));
```

```
benchmark <- function(obj) {
    result <- 0L
    for (j in 1:100) {
       obj2 <- obj$objectFunction(obj)
       obj$intField <- as.integer(obj2$doubleField)
       for (i in 1:250) { result <- obj$intFunction(i, obj$intField) }
    }
    result
}
benchmark(.jnew("RJavaBench"))
```

Special features:

- Promises (call-by-need + eager promises)

Cadenza [**?**]

- FrameBuilder - specialized MaterializedFrame
- Closure - rather convoluted-looking code

Generating function application looks like:

- TLam - creates Root, ClosureBody, captures to arr, arg/envPreamble
- Lam - creates Closure, BuilderFrame from all captures in frame
- Closure - is a ValueType, contains ClosureRootNode
- ClosureRootNode - creates a new VirtualFrame with subset of frame.arguments

TruffleClojure Implemented in a Master's thesis [**?**]

- I might want to implement envs as tries? Not arrays nor linked lists? Need to try
- each method impl is a root node, kept in bundles of callTargets by a ClojureFn
- closures - by a reference to the outer materializedFrame
- macros expanded during parse time, arguments not evaluated
- macroexpand function that expands macros
- separate section with a heading+listing+description of each special form (do we need this?)

1.5.4 Approach

We also have several options with regard to the depth of embedding: The most natural fit for Truffle is term evaluation, where a term could be represented as a value-level Term, and a CallTarget that produces its value with regard to the current environment. We can also embed the bidirectional elaboration algorithm itself, as a mixture of infer/check nodes.

The representation is also quite different from the functional interpreter where we have used functions and data classes, as in Truffle, all values and operations need to be classes.

There are several concerns here:

- algorithmic improvement is asymptotic – the better algorithm, the better we can optimize it
- Truffle's optimization is essentially only applicable to "hot code", code that runs many times, e.g. in a loop
- We need to freely switch between Term and Value representations using eval/quote
- program features - what do we do, and how to map it to Truffle:
  - infer/check - nodes
  - eval/quote - term = graph, value = value
  - function dispatch
  - instantiating based on type arguments
- project structure - package list with brief description?

Specific changes:

- everything is a class, rewrite functions/operations as classes/nodes
- annotations everywhere
- function dispatch is totally different
- lazy values need to be different
- ???
- required restructuring: compiler structure, hard parts, mention other languages throughout and not specifically an info dump
- providing more information - specialization, constants, invalidation

evaluation phases - translate to Code, run typecheck, run eval vs glued, ???

show program graphs: id, const, const id; optimized graphs

- Type system - Fun, Pap, Closure?, U
- Arrays - how much copying?

1.6 Specific changes in implementation

1.6.1 Data structures

We need to use arrays, Collections are not recommended

Arguments copied to the local frame in function preamble, to have unified access to then and not need to duplicate logic

Frames and frame descriptors for local/global variable

References, indices, uninitialized references

dispatch, invoke, call Nodes, argument schema (copying), ?

eta is TailCallException (2 para + example)

Passing arguments - the technical problem of copying arguments to a new stack frame in the course of calling a function.

Despite almost entirely re-using the Enso implementation of function calls, with the addition of implicit type parameters and without the feature of default argument values,

I will nonetheless keep my previous analysis of calling conventions in functional Truffle languages here, as it was an important part of designing an Truffle interpreter and I spent not-insignificant amounts of time on it.

I have discovered Enso only a short while before finishing my thesis, and had to incorporate the technologically-superior solution

Several parts of creating an AST for function calls:

- determining the position of arguments on the original stack - or evaluating and possibly forcing the arguments
- determining the argument's position on the stack frame of the function
- using this position in the process of inferring the new function call
- dispatch, invoke, call nodes???

Value types Data classes with call targets

...depends on what will work

Term and Val are ValueTypes that contain a callTarget - eval/quote(?)

We could use Objects/Shapes/Layouts for dependent sums or non-dependent named coproducts.

1.6.2 Normalization

Evaluation order We need to defer computations as late as possible - unused values that will be eliminated (1 para)

CBPV concepts, thunks with CallTargets (3 paras, example)

Calling convention the need for the distinction - in languages with currying

the eval/apply paper is a recipe for a stack-based implementation of currying and helpful in our case when we need explicitly manage our stack via Frames as opposed to the interpreter where we relied on the host language for this functionality

known/unknown calls, partially/fully/over-saturated calls

[**?**]

push-enter - arguments are pushed onto the stack, the function then takes as many as it requires

eval-apply - the caller sees the arity of the function and then decides whether it is over-applied (evaluates the function and creates a continuation), appllied exactly (EVAL), or under-applied (creates a PAP, a closure-like value)

- function application in languages with currying can be implemented using two evaluation models, push/enter and eval/apply
- compiled implementations should use eval/apply
- push/enter - arguments are pushed onto the stack, fun is entered, fun checks the number of arguments
- eval/apply - caller evaluates the function and applies it to the correct number of arguments

- need to distinguish known and unknown function calls,
- formalism uses heap objects FUN(●≥0), PAP(●(f)≥●≥1), CON(constructor), THUNK, BLACKHOLE
- + unboxed values not wrapped in any of these
- Rules: thunk→blackhole, blackhole→val, exact, over, under, thunkCall, papCall, retCall
- Truffle in theory supports both, but eval/apply plays better to the optimization where a calltarget should be as specialized as possible
- unboxing requires instanceof checks, we want to specialize/split
- push/enter means we need to copy arguments into an array

1.6.3 Elaboration

DerefNode - reads a variable, either blocks or returns an object. A meta-variable is replaced with a Term whenever it is evaluated/unblocked

1.6.4 Polyglot

Demonstrate calling Montuno from other languages

Demonstrate Montuno's eval construct

Demonstrate Montuno's FFI construct - requires projections/accessors

1.6.5 Driver

ParsingRequest/InlineParsingRequest

Unfortunately, Truffle requires that there is no access to the interpreter state during parsing, which means that we need to perform elaboration inside of a `ProgramRootNode` itself, "during runtime" per se.

need to perform elaboration inside a programRootNode (not while parsing)

stmt;stmt;expr -> return a value

1.6.6 Frontend

REPL needs to be implemented as a `TruffleInstrument`, it needs to modify and otherwise interact with the language context.

Language registration in mx/gu

Instrumentation Truffle is not only aimed at language developers but also at developers of language tools. We will specifically need this I would not mention this otherwise, but we will need to implement an instrument to add a REPL to the language. These tools are not specific to a single language, but most of them work for any language [**?**]. The `Instrument` API is based on events, an instrument declares which nodes it wants to receive events from based on source MIME or on tags (an annotation for debugging tools, can be custom), a *probe* is inserted on these places (a program location that emits events) using a *WrapperNode* that replaces a node and contains both this probe and the original, and this probe emits *events* that mark when execution has entered of left a node.