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 [7], 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 [1], or on GHC with mixed results [13].

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 [12].

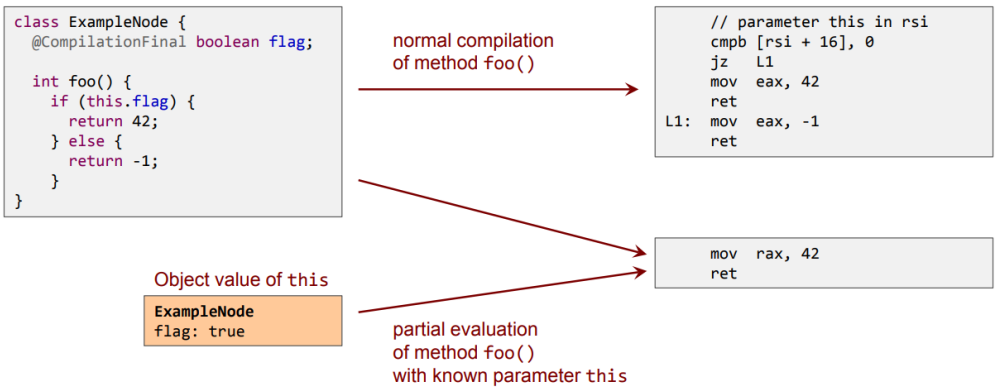


Figure 1.1: Partial evaluation with constant folding (source: oracle.com)

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 [12].

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 [2]. 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.

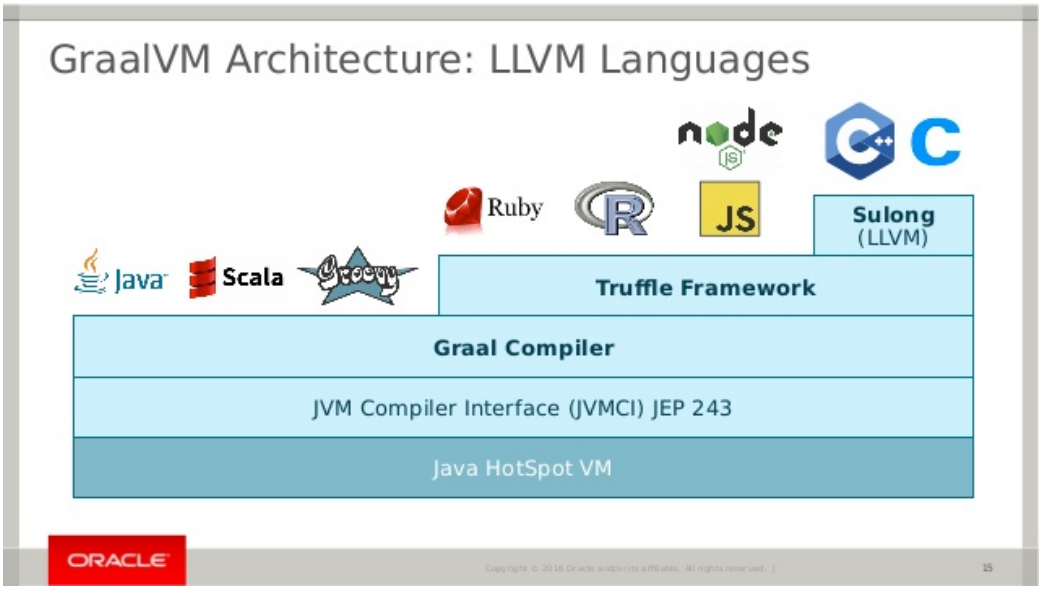


Figure 1.2: GraalVM and Truffle (source: oracle.com)

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.

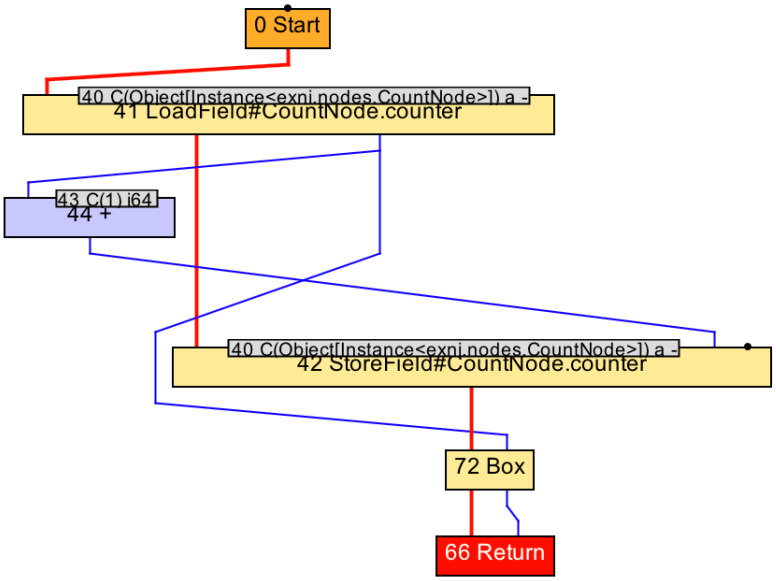


Figure 1.3: Graal program graph, visualized using IGV (source: norswap.com)

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 [19].

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 [2], 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 [19]. 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-

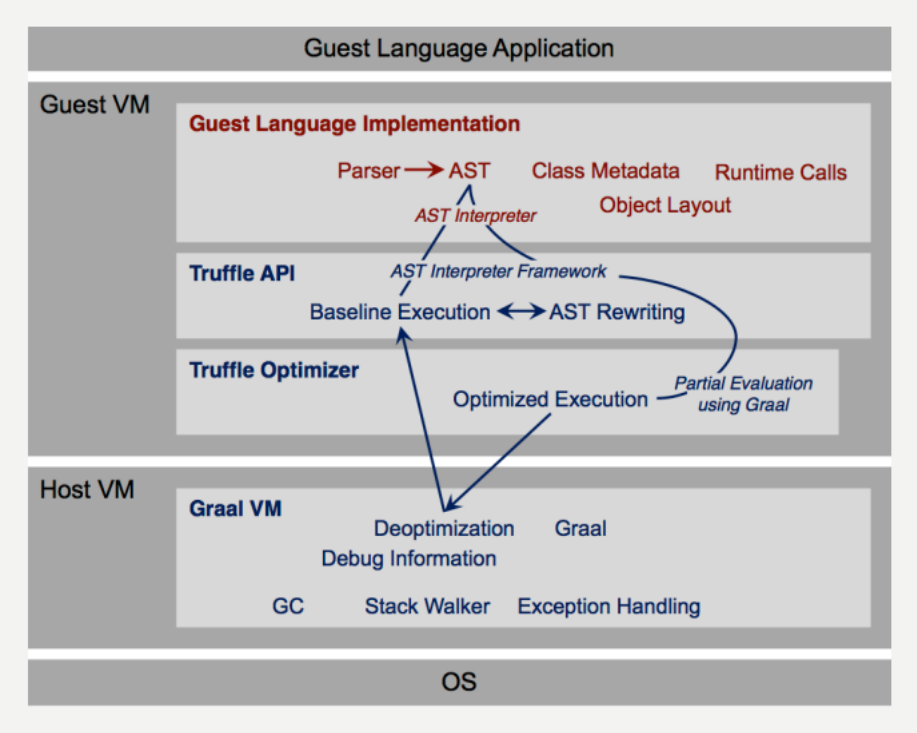


Figure 1.4: Architecture of a Truffle language (source: oracle.com)

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.

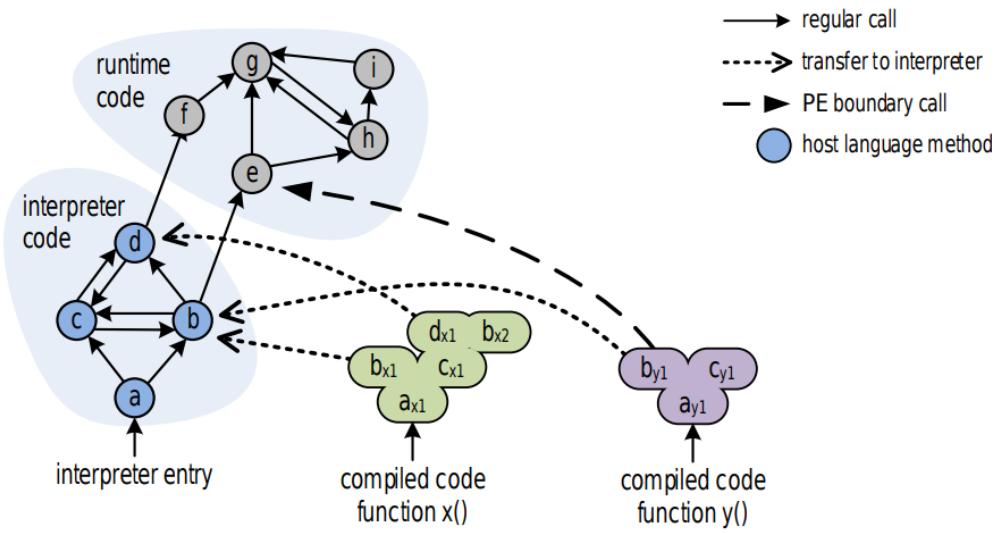


Figure 1.5: Combination of regular and partially-evaluated code (source: oracle.com)

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.

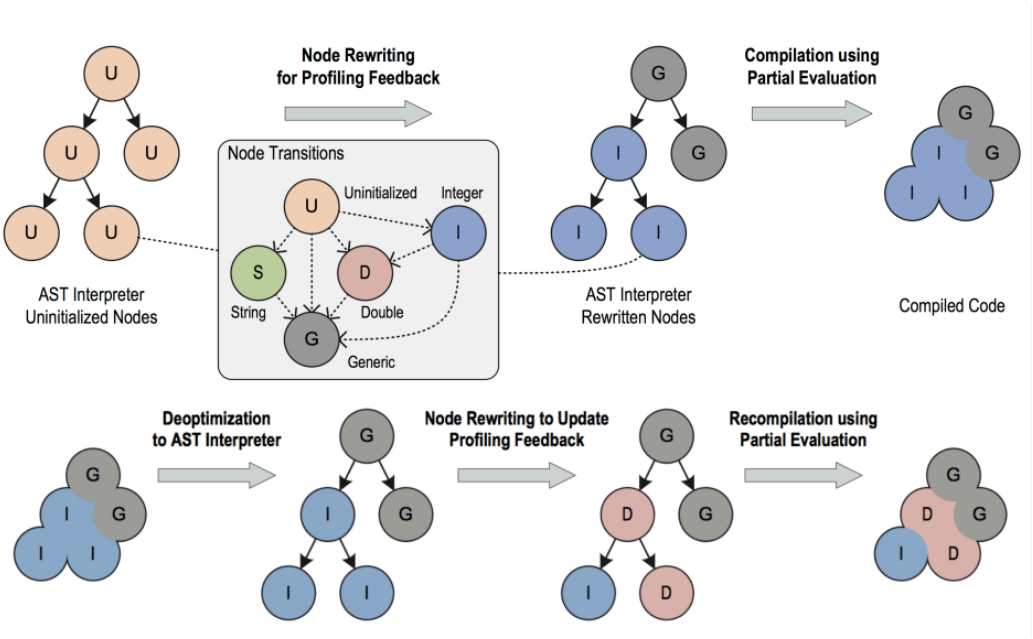


Figure 1.6: Node optimization and deoptimization in Truffle (source: oracle.com)

(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* [17], 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-`

`lot`s 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 1.12, 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 [14], 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 [14]

Inlining During optimization, the Graal compiler replaces `DirectCallNodes` with the contents of the call target they refer to, performing function *inlining* [18]. 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` [16]

Object model Truffle has a standard way of structuring data with fixed layout, called the Object Storage Model [6]. It is primarily intended for class instances that have a user-defined data layout, but e.g., the meta-interpreter project DynSem [16] 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 [6], 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` interface[1]

1.5 Mapping concepts to Truffle

We can now move on to the implementation of the second interpreter itself. Many of the features presented will mostly be used only in Chapter **??**, as this chapter only aims to create a Truffle interpreter that works, as even Truffle documentation recommends to "First, make it work, then make it fast".

Where to use Truffle? Truffle uses JIT compilation, and optimizes repeatedly executed parts of a program. Many parts of the previously implemented interpreter are only one-off computations, though, e.g., the elaboration process itself that processes a pre-term once and produces a corresponding term, discarding the pre-term. Only the evaluation of terms to values runs multiple times, as (top-level) functions are stored in the form of terms.

It is possible that the elaboration process might benefit as well, by implementing *infer*, *check*, and *unify* as Truffle nodes and using those in place of functions, but this chapter will only implement the simpler solution and keep elaboration outside of Truffle evaluation, as many changes will be required nonetheless. This optimization will be evaluated in Chapter **??**.

Inspiration For inspiration, I have looked at a number of other functional languages that use Truffle: a number of theses (TruffleClojure [4], TrufflePascal [5], Mozart-Oz [10]), two Oracle projects (FastR [15], TruffleRuby [14]), and other projects (Cadenza [11], DynSem [16], Mumbler [3], Truffled PureScript [9]).

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

In the last phases of this thesis, the project Enso [8] was released, that also aims to implement a dependently-typed language using Truffle. While time constraints did not allow me to improve on their approach, I have attempted to incorporate and evaluate several of their innovations, especially in Chapter **??**.

1.5.1 Approach

Out of the many changes that are required, the largest is the encoding of functions and closures, replacing data objects with `CallTarget`s. Environments and variable references need to be rewritten to use `Frame`s, and lazy evaluation cannot use Kotlin's `lazy` abstraction, but instead needs to be encoded as an explicit `Thunk` object.

The representation of the evaluation algorithm will also be different, we need to replace a tree transformation algorithm that processes an inert data structure with object-oriented nodes, where each implements its logic in the `execute` method.

(launcher, language, root, elab, eval, unify, context)

Figure 1.7: Program flow of the Truffle interpreter

Figure 1.7 demonstrates the components of the new interpreter. The `Launcher` is the same as in the previous interpreter, only now we use the language `Context` that it prepares based on user-provided options. The `Language` object initializes a different `Context` object, a `MontunoContext`, which is an internal object that contains the top-level variable scope, the meta-variable scope, and other global state variables. `Language` then dispatches parsing requests to the parser, and the pre-terms it produces are then wrapped into a `ProgramRootNode`.

Executing the `ProgramRootNode` starts the elaboration process, where `infer` and `check` build up terms as executable nodes. Any `eval` invocations in the process are then handled by Truffle, producing a `ValueType`. These can be compared, unified, or built back up into a `Term` using `quote`.

Elaboration and evaluation both access the `MontunoContext` object to resolve top-level variables and meta-variables into the corresponding `Term`s. The REPL accesses the context as well in order to produce lists of bound variables, and process REPL commands.

FillIn

The data flow in 1.8 makes the data transformations clear, especially the parts where Truffle is involved .

preterm AST, Term Nodes, Value AST + Nodes

Figure 1.8: Data flow inside the Truffle interpreter

1.5.2 Values

Disregarding constants, there are only two main value types per Figure 1.9: a Π-type (equivalent to a λ-abstraction), and a Σ-type. A Π-type maps onto a closure and will be discussed momentarily. A Σ-type can be expressed as a pair, or a linked list of nested pairs, to use the simplest representation, that we will attempt to optimize in Chapter **??**. Then, there

$$
\begin{array}{rcllll}
\mathit{term} & := & v & | & \mathit{constant} & \\
 & | & a\ b & | & a\ \{b\} & \\
 & | & a \to b & | & (a : A) \to b & | \quad \{a : A\} \to b \\
 & | & a \times b & | & (l : A) \times b & | \quad a.l \\
 & | & \text{let } x = v \text{ in } e & | & [|\ id\ |\ foreign\ |\ type\ |] & \\
 & | & \_ & & & \\
\mathit{value} & := & \mathit{constant} & | & \mathit{neutral} & \\
 & | & \lambda x : A.b & | & \Pi x : A.b & \\
 & | & (a_1, \cdots, a_n) & & & \\
 & | & \_ & & & \\
\mathit{neutral} & := & \mathit{var} & | & \mathit{neutral}\ a_1\ ...a_n & | \quad \mathit{neutral}.l_n
\end{array}
$$

Figure 1.9: Terms and values in Montuno (revisited)

are neutral terms, unresolved variables that accumulate a spine of unapplied operands and projections: these will be expressed as a head containing a variable reference, and a spine with an array of spine values.

Each of these values needs to be a separate class, a `ValueType`, and an entry in the Truffle type system. A snippet in Listing 1.11 shows the `TypeSystem` and two simple value types. Other than the above-mentioned types of values, there is a number of literal types, and a type `Thunk`. We need to have this type explicitly mentioned here, to implement lazy evaluation in Truffle later.

We may need to perform a common set of operations on these values, to have them implement a shared interface: the Truffle way is to use a `Library`, which will be mentioned later, as relevant.

```
@TypeSystem(
  Constant::class, Neutral::class,
  Unit::class, Pi::class, Func::class, Pair::class,
  Thunk::class,
  Boolean::class, Int::class, BigInt::class,
)
class Types {
  @TypeCheck(Unit::class)
  fun isUnit(value: Any) = value === Unit
}

@ValueType
object Unit : TruffleObject
@ValueType
class Pair(val left: Any, val right: Any) : TruffleObject
```

Listing 1.11: A `TypeSystem` and two simple `ValueTypes` from the Truffle interpreter

Closures There are many possible representations of a closure on Truffle. We will explore some later in Chapter **??**, but for now, one simple representation is a simply wrapping a `Term` in a `RootNode`, and storing it alongside the current scope.

A `CallTarget` can only be called with an array of objects, so this is what a `Closure` stores. The `CallTarget` points to a `ClosureRootNodw`, that first copies the array of arguments it was given into the local scope, and executes the body. Variable names in the local scope (`fd.findFrameSlot(i)`) are equal to the de Bruijn levels that are used in values. This can all be see in Listing 1.12.

```
@ValueType
class Closure(val env: Array<Any>, val target: CallTarget) : TruffleObject {
  fun call(arg: Object) = target.call(env.plus(arg))
}
@ValueType
class Thunk(val env: Array<Any>, val target: CallTarget) : TruffleObject {
  fun force() = target.call(env)
}
class ClosureRootNode(@NodeChild val body: Node) : Node {
  @Override
  @ExplodeLoop
  fun executeGeneric(f: VirtualFrame): Any {
    val args = f.getArguments()
    val size = args.size
    val fd = f.getFrameDescriptor()
    for (int i = 0; i < size; i++) {
      f.setObject(fd.findFrameSlot(i), args[i])
    }
    return body.execute(f)
  }
}
```

Listing 1.12: A sketch of the closure implementation

1.5.3 Normalization

Terms are be the nodes of the program graph through which flow the above-defined values. Each term is either fully evaluated, or produces one layer of a value.

The `Terms` share a common super-class, and each defines the correct number of `NodeChildren` and the relevant `execute` method. The super-class `Term` also defines a number of other methods, one for each value type in the `TypeSystem`. These methods serve to specify the expected return type of a node using the type assertions generated by the `TypeSystem`, e.g., `TypesGen.asUnit()`. This is necessary, because the `execute` method of all nodes returns the generic type `Any`, and cannot be further constrained.

Variables This is demonstrated in Listing 1.13, which also includes a local variable node `TLocal`. We will still use de Bruijn indices for terms and de Bruijn levels for values and still receive all the benefits they provide, as mentioned in the previous chapter, but this time any variable references will point to a `Frame`, and not to an array of values.

λ-abstractions The terms `TLam` and `TPi` create a single-argument closure value, shown in Listing 1.14. This closure is created by converting the local scope into an array of objects

```
abstract class Term : Node() {
  abstract fun execute(f: VirtualFrame): Any
  fun executeGeneric(f: VirtualFrame): Any = execute(f)
  fun executeUnit(f: VirtualFrame): Unit = TypesGen.asUnit(executeGeneric(f))
  // ...
}
data class TLocal(val n: Ix) : Term() {
  fun executeGeneric(f: VirtualFrame) {
    val fd = f.getFrameDescriptor()
    return f.getObject(fd.findFrameSlot(fd.getSize() - n - 1))
  }
}
```

Listing 1.13: The super-class `Term`, and a local variable node

(not shown), and including the `CallTarget` that refers to the body of the function. The opposite process, applying a λ-abstraction to an argument requires a `DispatchNode`. The `TApp` node first evaluates the values that make up the function and its argument, and hands them over to a `DispatchNode` that will perform the call.

```
data class TLam(@NodeChild val root: ClosureRootNode) : Term {
  val target = Truffle.getRuntime().createCallTarget(root)
  fun executeClosure(f: VirtualFrame)
    = Closure(frameToEnv(f.materialize()), target)
}
class TApp(@NodeChild val fn: Term, @NodeChild val arg: Term) : Term {
  val dispatchNode = DispatchNode()
  fun executeGeneric(f: VirtualFrame) = dispatchNode.executeDispatch(
    fn.executeClosure(f), arg.executeAny(f)
  )
}
```

Listing 1.14: Demonstration of a λ-abstraction and application in Truffle

Let-in Other constructs follow this pattern. For example, a `let-in` expression first evaluates the value it binds to a variable, assigns it to the `Frame` using a `FrameSlot` that was already computed during the process of elaboration, and then executes the term that contains its body, demonstrated in Listing 1.15.

```
class TLet(
  val fs: FrameSlot,
  @NodeChild val value: Term,
  @NodeChild val body: Term
) : Term() {
  fun executeGeneric(f: VirtualFrame) {
    f.setObject(fs, value.executeAny(f))
    body.execute(f)
  }
}
```

Listing 1.15: `Let-in` expression in the Truffle interpreter

Built-ins Built-in constants and types need to be implemented as special nodes. The resolution of a built-in name to its corresponding node happens during elaboration. Each built-in term has its arity, the number of expected arguments. The elaboration process wraps this node with the correct number of λ-abstractions, and resolves the arguments they will produce to an array of arguments. These are passed to a `BuiltinRootNode` that does not copy them to the local scope, unlike the `ClosureRootNode`, but the built-in node uses them directly.

This is shown on the example of a `Succ` node in Listing 1.16. This node has the arity 1, it expects a single argument, which can be either an already evaluated integer, or a `Thunk` that will produce an integer, which is then *forced*, and coerced to a integer using a function generated by the `TypeSystem`.

```
class Succ : BuiltinTerm(1) {
  @Specialization
  fun doInt(n: Int) = n + 1
  @Specialization
  fun doThunk(t: Thunk) = TypesGen.asInt(t.force()) + 1
}
```

Listing 1.16: A `Succ` node, an example implementation of a built-in term

```
class TEval(val lang: String, val code: String) : Term {
  fun executeGeneric(
    f: VirtualFrame, @CachedContext(MontunoLanguage::class) ctx
  ) {
    val src = Source.newBuilder(lang, code).build()
    val callTarget = ctx.parsePublic(src)
    return callTarget.call()
  }
}
```

Listing 1.17: The implementation of a foreign *eval* term

Polyglot Lastly, the implementation of the foreign evaluation term that was already mentioned in Chapter **??** is included in Listing 1.17. This term has three components, a language identifier, an expression written in that language, and the type of the expression. Truffle makes it straight-forward to implement such a term. The language identifier and the foreign code are first compiled into a Truffle `Source` object, which is then parsed using the Truffle-provided language `Context`. The context acts as a proxy to the foreign `Language`, which then uses the provided source to create a `CallTarget`. Calling it will result in a foreign value.

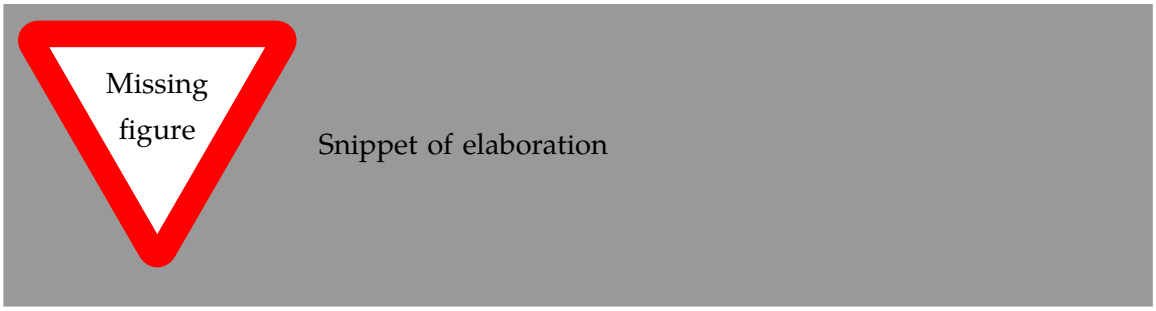
This `TEval` term needs to be wrapped in a node that will map this resulting value to a value that Montuno can use, integers to integers, or arrays to nested Σ pairs, but the result works as expected. Listing 1.18 shows the result of executing a JavaScript equivalent of the `succ` function.

```
Mt> [js|(x) => x + 1|Nat -> Nat|] 5
6
```

Listing 1.18: Calling JavaScript using the *eval* construct

1.5.4 Elaboration

The main change in elaboration is the fact that the functions *infer* and *check* now need to build up the `Term`s as program graphs. This is accomplished by inserting root nodes where necessary, and by keeping a `FrameDescriptor` in the local elaboration context, which is necessary so that we can declare all `FrameSlot`s ahead of time and store them in term nodes, so that no variable lookup needs to take place during evaluation.



Meta-variables are stored in the top-level language context, and meta-variable references use a special node that either looks up the value of a solved meta-variable or forces its evaluation, returning control from a Truffle context back to the external elaboration process.

The changes in the implementation of the driver and frontend were largely described at the start of this section, so they will not be mentioned again.

Bibliography

[1] C. F. Bolz. *Meta-tracing just-in-time compilation for RPython*. PhD thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, 2014.

[2] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck. Graal ir: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[3] C. Esquivias. cesquivias/mumbler, 2016.

[4] T. Feichtinger. *TruffleClojure: A self-optimizing AST-Interpreter for Clojure/submitted by: Thomas Feichtinger*. PhD thesis, Linz, 2015.

[5] J. Flimmel. Pascal with truffle. 2017.

[6] M. Grimmer, C. Seaton, R. Schatz, T. Würthinger, and H. Mössenböck. High-performance cross-language interoperability in a multi-language runtime. In *Proceedings of the 11th Symposium on Dynamic Languages*, pages 78–90, 2015.

[7] J. S. Gross. *Performance Engineering of Proof-Based Software Systems at Scale*. PhD thesis, Massachusetts Institute of Technology, 2021.

[8] N. B. O. Inc. enso-org/enso, 2021.

[9] S. Inc. slamdata/truffled-purescript, 2015.

[10] M. Istasse, P. Van Roy, B. Daloze, and G. Maudoux. An oz implementation using truffle and graal. 2017.

[11] E. Kmett. ekmett/cadenza, 2019.

[12] F. Latifi. Practical second futamura projection: Partial evaluation for high-performance language interpreters. In *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2019, page 29–31, New York, NY, USA, 2019. Association for Computing Machinery.

[13] T. Schilling. *Trace-based just-in-time compilation for lazy functional programming languages*. PhD thesis, University of Kent, 2013.

[14] Shopify. Optimizing ruby lazy initialization in truffleruby with deoptimization, Mar 2020.

[15] L. Stadler, A. Welc, C. Humer, and M. Jordan. Optimizing r language execution via aggressive speculation. *ACM Sigplan Notices*, 52(2):84–95, 2016.

[16] V. Vergu, A. Tolmach, and E. Visser. Scopes and frames improve meta-interpreter specialization. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[17] C. Wimmer, V. Jovanovic, E. Eckstein, and T. Würthinger. One compiler: Deoptimization to optimized code. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, page 55–64, New York, NY, USA, 2017. Association for Computing Machinery.

[18] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 662–676, 2017.

[19] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 187–204, 2013.