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 [**?**].

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 [**?**].

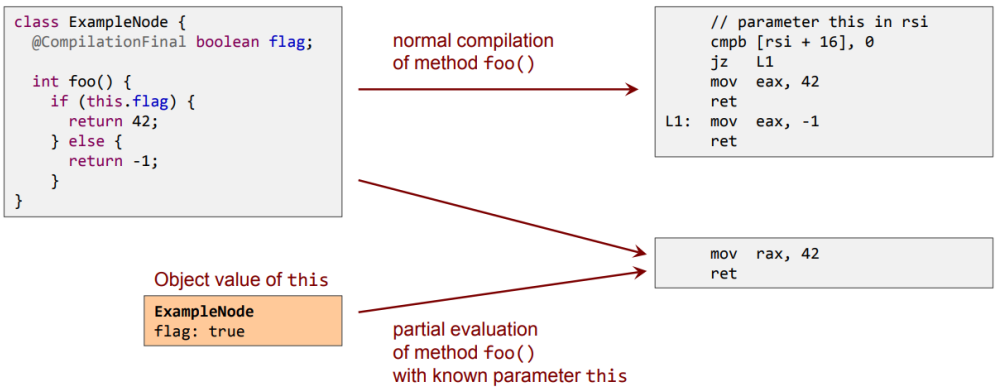

Figure 1.1: Partial evaluation with constant folding (source: oracle.com)

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.

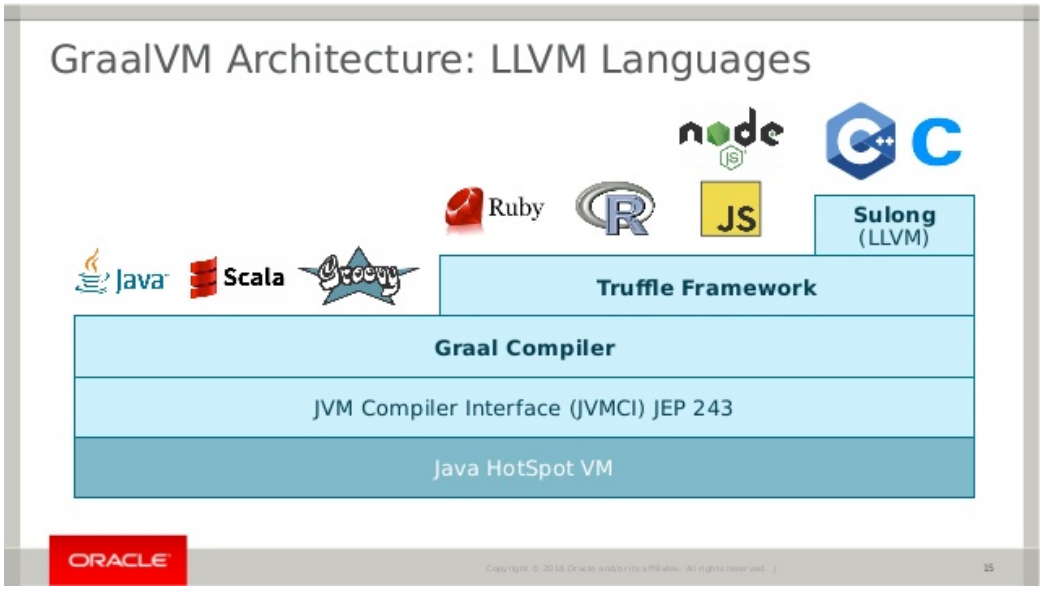

Figure 1.2: GraalVM and Truffle (source: oracle.com)

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.

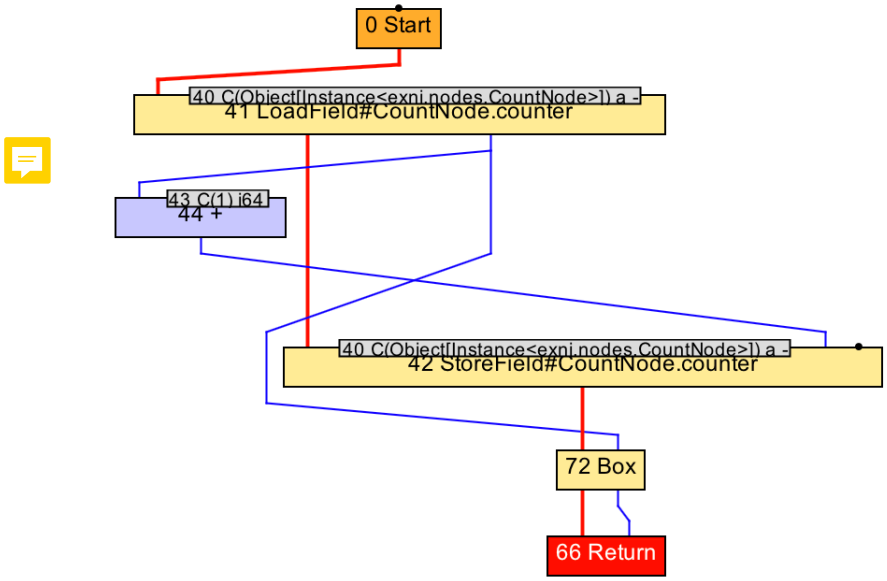

Figure 1.3: Graal program graph, visualized using IGV (source: norswap.com)

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 an 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 capabilities

This concludes the general introduction of Truffle and GraalVM. Now onto the specifics of Truffle languages.

As inspiration, I have looked at a number of other functional languages created using Truffle.

Most of Truffle features, as listed here, are based on *speculative optimization*: if we run a function a few times and see only a certain pattern of arguments and the function's behaviors, we assume that in this run of the program, only this pattern of arguments and behavior will occur, and can optimize for this case.

Overall, the implementation of a Truffle language can be divided into a few parts:

- language registration
- running a language (context, launcher)
- language request (parsing, rootcalltarget)
-

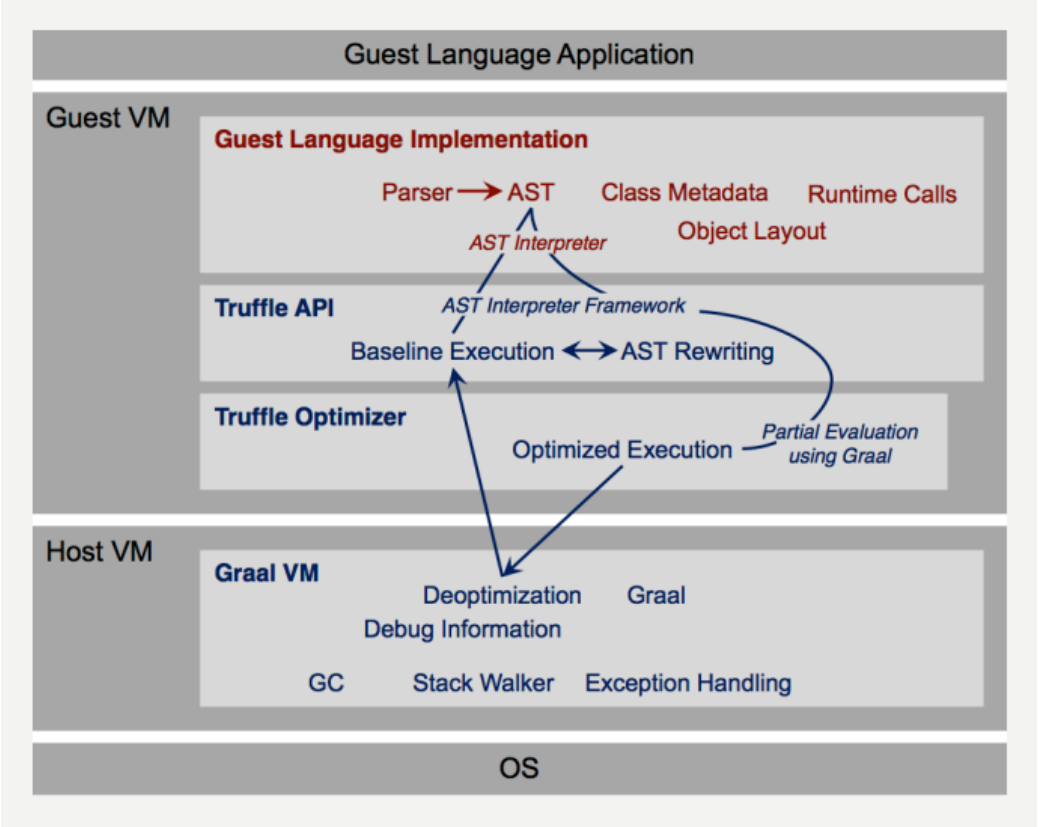

Figure 1.4: Architecture of a Truffle language (source: oracle.com)

- brief list of features:
  - language registration
    - Language, Runtime, Context, ...
  - calling into a language
    - Launcher
    - ParsingRequest
  - function dispatch
    - Frame, FrameDescriptor, Call/Dispatch/Invoke nodes
  - callable things
    - RootNode, CallTarget, CallableNode
  - speculative optimization:
    - profiles, assumptions, specialization
    - inline caching
    - inlining, splitting
  - compiler directives:
    - boundaries, explodeloop
  - libraries: polyglot

- object storage model
  - Object, Shape,
- instrumentation
- threading

—mental model = graph through which values flow, values may contain graphs

General features Common components - Launcher, LanguageRegistration, Nodes, Values, REPL (5 items)

Engine, Context, TruffleLanguage, Instrument

```
N: unbounded
P: N for exclusive, 1 for shared context policy
L: number of installed languages
I: number of installed instruments

- 1 : Host VM Processs
 - N : Engine
   - N : Context
     - L : Language Context
   - P * L : TruffleLanguage
   - I : Instrument
     - 1 : TruffleInstrument
```

Introduce the canonical example - Literal and addition, trivial execute fn (1 para, 1 figure)

Graph manipulations:

- replace() to a more specific variant
- adopt() a new child node
- GraalVM aggressively inlines stable method calls into efficient machine code
- [**?**] - Truffle boundaries
- [**?**] - deoptimization, on-stack replacement

Node optimization In Figure 1.5, this process is demonstrated. Based on profiling feedback (usually the number of calls of a CallTarget invocations), a node may rewrite itself to use the most efficient specialization based on the conditions (usually what values it has been invoked with so far). When the node (tree) reaches a stable stat, it is compiled to optimized machine code using partial evaluation, given the state, and compilation-constant values.

This compilation is conditional, and when the assumptions are invalidated, the node is deoptimized back into its non-compiled form (called "transfer to interpreter"). Nodes will again specialize themselves based on the observed conditions, and when the node reaches a stable state again, the node is partially evaluated once more.

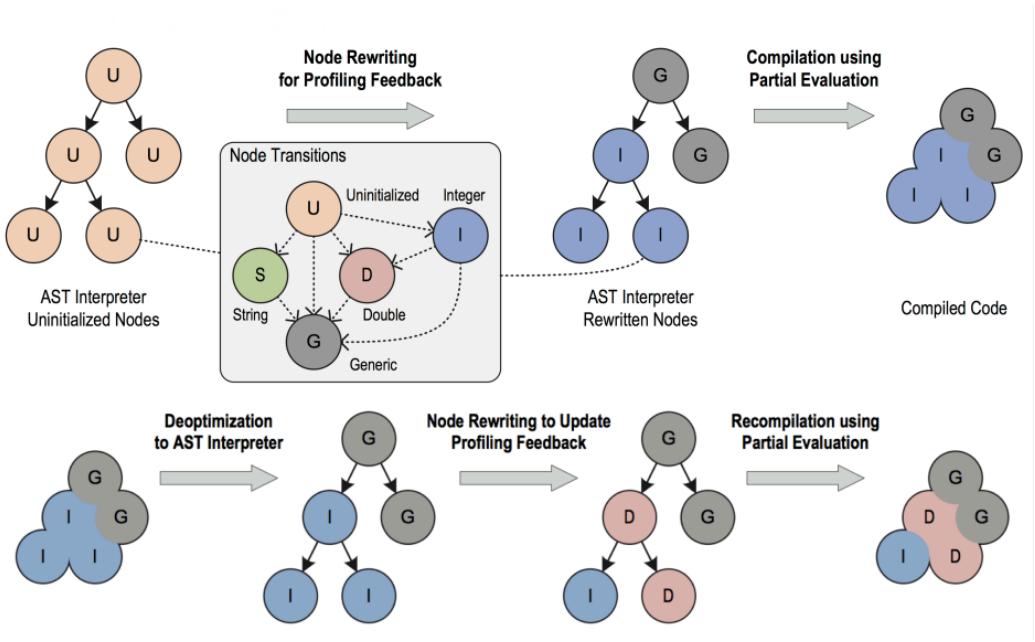

Figure 1.5: Node optimization and deoptimization in Truffle (source: oracle.com)

Type specialization Type system

Basic case is type specialization - when an addition node only encounters integers, there is no need to generate machine code for floats, doubles, or operator overloads - only verified by fast checks. When these fail, the node is de-optimized, and eventually re-compiled again.

Inline caching for e.g. method lookups, virtual method calls are typically only ever invoked on a single class, which can be cached, and the dispatch node can be specialized, perhaps even inline the operation. (uninitialized/monomorphic/polymorphic/megamorphic = working with jumptables + guard conditions)

Specializations are general, though, and nodes can go be specialized on arbitrary conditions, using custom assumptions and *compilation final* values. In general, node states form a directed acyclic graph - "a node can ever become more general".

Using a graph visualizer, we can look at this process on a simple example commonly used to demonstrate this part of the Truffle framework: the + operation.

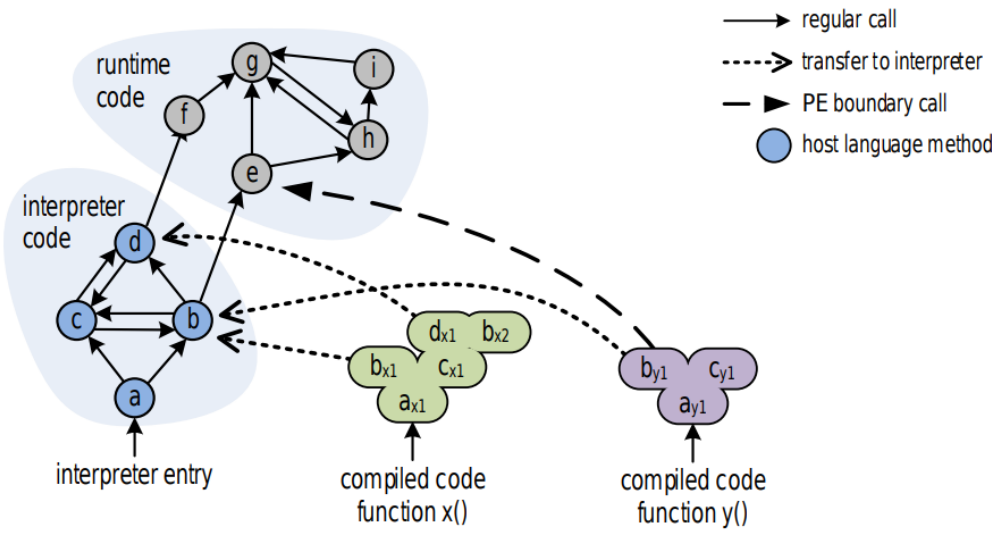

Figure 1.6: Combination of regular and partially-evaluated code (source: oracle.com)

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

The Truffle framework is said to be a domain-specific language, which in this case means a library, a set of annotations, and a code generator. This code generator finds classes that inherit from the `Node` class and generates, among others, the logic behind switching specializations.

The program graph is formed from a tree of Truffle `Nodes` from which we derive our language-specific base class, `LangNode` in this case. We define two classes that inherit from this class, one representing integer literals, and one for the + operator.

The abstract method `execute` in `LangNode` is the evaluation of this node. It takes a `VirtualFrame`, which represents a stack frame, and its return value is also the return value of the node. In addition, methods starting with `execute` are special in Truffle. Truffle will pick the most appropriate one based on return type (with `Any` being the most general) and parameters.

In `IntLiteralNode` we directly override the method `execute`, as there is only one possible implementation. In `AddNode`, however, we keep the class abstract and do not implement `execute`, and instead rely on Truffle to generate the appropriate specialization logic.

Truffle will decide between the specializations based on parameter types, and on user-provided guards (we will see further). Fallback specialization matches in cases where no other one does. Names are irrelevant.

(Maybe show generated code?) Active an inactive specializations: can be multiple active, execute method is based on state first, and only then on type checks - smaller and possibly better optimized result. If no specialization matches, then fall through to `executeAndSpecialize` which invalidates any currently compiled using `CompilerDirectives.transferToInterpreterAndInvalidate` and sets state bits for newly activated specializations.

(@Specialization(Replaces=[“”]))

How to run? Need to wrap in a `RootNode`, which represents executable things like methods, functions and programs. Then create a `CallTarget` using `Truffle.getRuntime().createCall`
Truffle uses CallTargets to record, among others, how often a particular graph is called, and when to compile it. Also it creates a VirtualFrame for this call target out of the provided arguments.

IGV receives JIT compilation output - shows the Graal graphs produced during optimization. Compilation is only triggered after a certain threshold of calls, so we need to run a call target more than just once.

Graal-graph

Another option: a CountNode (public int counter; execute = counter++). Green == state, grey is floating (not flow dependent), blue lines represent data flow (data dependencies), red means control flow (order of operations)

Function dispatch Separation of concerns in function call:

- Invoke node evaluates function and its arguments, calls dispatch
- Dispatch node builds the inline cache
- Direct/Indirect call node (Truffle) calls the CallTarget which calls the RootNode
- VirtualFrame - virtual/optimizable stack frame, “a function’s/program’s scope”, can be eliminated altogether, which results in highly efficient code
- MaterializedFrame - VirtualFrame in a specific form, not optimizable, can be stored in a ValueType

FrameDescriptor - shape of a frame FrameSlot FrameSlotKind

VirtualFrame - can be optimized, reordered

MaterializedFrame - an explicit Java Object on the heap, created from a VirtualFrame by calling `frame.materialize()`

dispatchNode, frames, argument passing, ExecutableNode, RootNode, CallTarget

- RootNode - can be made into a CallTarget. is at the root of a graph, "starting point of a function/program/builtin", "callable AST"

Direct/IndirectCallNode

Frame

- is an object on the heap, allocated in the function prologue
- compiler eliminates the allocation
- FrameDescriptor - layout of the frame
- TCO - maybe describe a trampoline? + a graph

Control flow Inter-node control flow - use exceptions

Caching @Cached()

Profiles, assumptions, specializations profiles, assumptions, specializations:

- similar tools
- profiles for local, monomorphic properties, does not speculate again
- assumptions for non-local speculation, can be triggered from outside
- specializations for local, polymorphic, speculates again

Assumptions can be used to guard important system-wide conditions, with explicit invalidation. Checking is very cheap.

Profiles allow for passing additional information to the compiler - condition profiles, type profiles.

Boundary TruffleBoundary - to avoid inlining large methods: println, database access, ...

Compiler assertions Assertions to check the compiler isCompilationConstant, neverPartOfCompilation

Inlining vs splitting Inlining vs splitting

- inlining replaces a call with a copy of the graph it stands for
- inlining works on the AST level
- splitting produces a fresh callTarget, monomorphic code

Tooling integration Tools:

- tag: an annotation for debugging tools, can be custom
- probe: a program location that emits events
- event: marks when execution has entered of left a node
- instrument: receives events
- WrapperNode: around a node, intercepts events, contains a Node and a ProbeNode
- cite "We can have it all" van de vanter

Object Storage Model [**?**] - Polyglot + OSM intro

- [**?**] - URI language, object-oriented
- OSM (Object Storage Model) - Frame (~typed HashMap)
- Shape = mapping Property ⟶ Location, Shape.newInstance == DynamicObject

Object Storage Model is a common way to organize data with layouts - Objects, Shapes, Layouts.

TruffleRuby uses it for FFI to access Ruby objects directly as if they were C `structs`. DynSem uses these to model frames/scopes instead of Frames, as it is a meta-interpreter and uses Truffle Frames for its own data.

Polyglot ValueTypes, InteropLibrary

- interop: ForeignToLocalTypeNode

[**?**] - Polyglot + OSM intro

Threading

- thread-safety of objects: by default Truffle is not safe, explicit synchronization

1.4.1 Functional languages on Truffle

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/

Old project, but one of the only purely-functional Truffle languages.

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

Mumbler An implementation of a Lisp

https://github.com/cesquivias/mumbler

FastR One of the larger Truffle languages

[**?**]

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

Enso https://github.com/enso-org/enso/

A very late addition to this list, this is a project that originally rejected Truffle (and dependent types in general, if I recall correctly) and used Haskell instead. However, the project Luna was renamed to Enso, and rebuilt from scratch using Truffle and Scala not long before my thesis deadline.

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

- Truffle+Graal means that languages reuse JVM's features: garbage collection, optimization
- API based on Node; Frame stores the transient state of function activation: local variables, arguments passed to the function, intermediate results, shape described by FrameDescriptor, data in the backend stored in Object[] or if primitive types only in long[] (and converted)
- I might want to implement envs as tries? Not arrays nor linked lists? Need to try

- I must have a program execution flow
- I also need a "data shape flow" (src, pre, term, val)
- 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?)

DynSem

- [**?**] - OSM for frames/scopes

1.5 Mapping concepts to Truffle

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

There are also several features that we require that are not a natural fit for it, but where we can find inspiration in other Truffle languages. In particular, lazy evaluation (FastR promises), partial function application (Enso), ???

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.

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".

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

- Truffle
  - De Bruijn in Frames?
  - Eval/apply nucleus (fun –> PAP/THUNK)

  - call-by-need, sharing (fastR)
  - visual VM! (maybe non-EE edition?), Chrome debugger. IGV
  - TCL
  - global frame
  - wired-in primitives
- 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

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