Chapter 1

Language implementation: Montuno

1.1 Introduction

Now with a complete language specification, we can move onto the next step: writing an interpreter. The algorithms involved can be translated from specification to code quite naturally. at least in the style of interpreter we will create at first. The second interpreter in Truffle will require a quite different programming paradigm and deciding on many low-level implementation details, e.g., how to implement actual function calls.

In this chapter we will introduce the algorithms at the core of an interpreter and build a tree-based implementation for the language, elaborating on key implementation decisions. This interpreter will be referred to using the working name Montuno[1].

This interpreter can be called an AST (abstract syntax tree) interpreter, as the principal parts all consist of tree traversals, due to the fact that all the main data structures involved are trees: pre-terms, terms, and values are recursive data structures. The main algorithms to be discussed are: evaluation, normalization, and elaboration, all of them can be translated to tree traversals in a straightforward way.

Language The choice of a programming language is mostly decided by the eventual target platform Truffle, as we will be able to share parts of the implementation between the two interpreters. The language of GraalVM and Truffle is Java, although other languages that run on the Java Virtual Machine can be used[2]. My personal preference lies with more functional languages like Scala or Kotlin, as the code often is cleaner and more concise[3], so in the end, after comparing the languages, I have selected Kotlin due to its multi-paradigm nature: Truffle requires the use of annotated classes, but this first interpreter can be written in a more natural functional style.

[1] Montuno, as opposed to the project Cadenza, to which this project is a follow-up. Both are music terms, *cadenza* being a "long virtuosic solo section", whereas *montuno* is a "faster, semi-improvised instrumental part".

[2] Even though Kotlin seems not to be recommended by Truffle authors, there are several languages implemented in it, which suggests there are no severe problems. "[...] and Kotlin might use abstractions that don't properly partially evaluate." (from https://github.com/oracle/graal/issues/1228)

[3] Kotlin authors claim 40% reduction in the number of lines of code, (from https://kotlinlang.org/docs/faq.html)

Libraries Truffle authors recommend against using many external libraries in the internals of the interpreter, as the techniques the libraries use may not work well with Truffle . Source Therefore, we will need to design our own supporting data structures based on the fundamental data structures provided directly by Kotlin. Only two external libraries would be too complicated to reimplement, and both of these were chosen because they are among the most widely used in their field:

- a parser generator, ANTLR, to process input into an abstract syntax tree,
- a terminal interface library, JLine, to implement the interactive interface.

For the build and test system, the recommended choices of Gradle and JUnit were used.

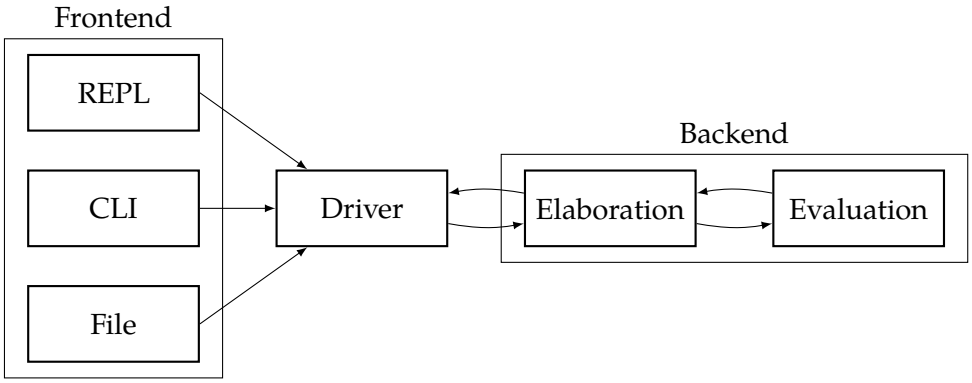


Figure 1.1: Overview of interpreter components

1.1.1 Program flow

A typical interpreter takes in the user's input, processes it, and outputs a result. In this way, we can divide the interpreter into a frontend, a driver, and a backend, to reuse compiler terminology. A frontend handles user interaction, be it from a file or from an interactive environment, a backend implements the language semantics, and a driver connects them, illustrated in Figure 1.1.

Frontend ?!?!?! The frontend will be straight-forward: batch processing mode that reads from a file, and the interactive mode, a REPL (Read-Eval-Print Loop), that receives input from the user and prints out the result of the command. Proof assistants like Agda offer deeper integration with editors like tactics-based programming or others, similar to the refactoring tools offered in development environments for object-oriented languages, but that is unnecessary for the purposes of this thesis.

Backend The components of the backend, here represented as *elaboration* and *evaluation*, implement the data transformation algorithms that are further illustrated in Figure 1.2. In brief, the *elaboration* process turns user input in the form of partially-typed, well-formed *pre-terms* into fully-annotated well-typed *terms*. *Evaluation* converts between a *term* and a *value*: a term can be compared to program data, it can only be evaluated, whereas a value is the result of such evaluation and can be e.g., compared for equality.

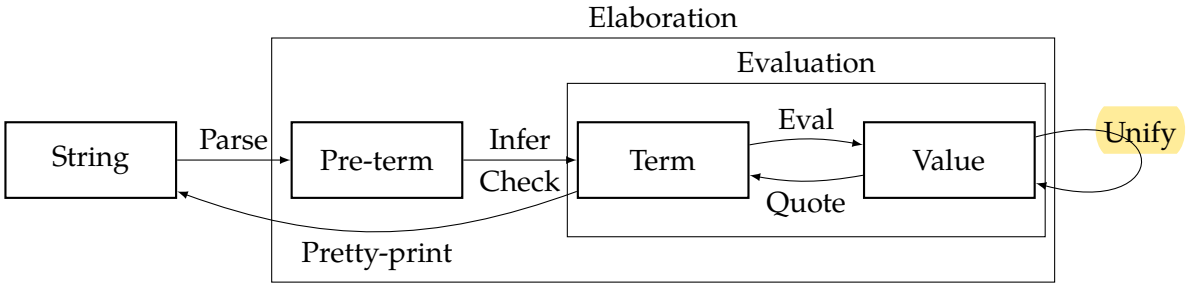


Figure 1.2: Data flow overview

Data flow In Figure 1.2, *Infer* and *Check* correspond to type checking and type inference, two parts of the *bidirectional typing* algorithm that we will use. *Unification* (*Unify*) forms a major part of the process, as that is how we check whether two values are equal. *Eval* corresponds to the previously described βδζι-reduction implemented using the *normalization-by-evaluation* style, whereas *Quote* builds a term back up from an evaluated value. To complete the description, *Parse* and *Pretty-print* convert between the user-readable, string representation of terms and the data structures of their internal representation. For the sake of clarity, the processes are illustrated using their simplified function signatures in Figure 1.

```
fun parse(input: String): PreTerm;
fun pprint(term: Term): String;
fun infer(pre: PreTerm): Pair<Term, Val>;
fun check(pre: PreTerm, wanted: Val): Term;
fun eval(term: Term): Val;
fun quote(value: Val): Term;
fun unify(left: Val, right: Val): Unit;
```

Listing 1: Simplified signatures of the principal functions

In this chapter, we will first define the data types, especially focusing on closure representation. Then, we will specify and implement two algorithms: *normalization-by-evaluation*, and *bidirectional type elaboration*, and lastly finish the interpreter by creating its driver and frontend.

1.2 Data structures

We have specified the syntax of the language in the previous chapter, which we first need to translate to concrete data structures before trying to implement the semantics. Sometimes, the semantics impose additional constraints on the design of the data structures, but in this case, the translation is quite straight-forward.

Properties Terms and values form recursive data structures. We will also need a separate data structure for pre-terms as the result of parsing user input. All of these structures represent only well-formed terms and in addition, terms and value represent the well-typed subset of well-formed terms. Well-formedness should be ensured by the parsing process, whereas type-checking will take care of the second property.

Pre-terms As pre-terms are mostly just an encoding of the parse tree without much further processing, the complete data type is only included in Appendix **??**. The `PreTerm` class hierarchy mostly reflects the `Term` classes with a few key differences, like the addition of compiler directives or variable representation, so in the rest of this section, we will discuss terms and values only.

Location A key feature that we will also disregard in this chapter is term location that maps the position of a term in the original source expression, mostly for the purpose of error reporting. As location is tracked in a field that occurs in all pre-terms, terms, and values, it will only be included in the final listing of classes in Appendix **??**.

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

Figure 1.3: Terms and values in Montuno (revisited)

Revisiting the terms and values specified in Chapter **??** in Figure 1.3, there are a two main classes of terms: those that represent computation (functions and function application), and those that represent data (pairs, records, constants).

Data classes Most *data* terms can be represented in a straight-forward way, as they map directly to features of the host language, Kotlin in our case. Kotlin has a standard way of representing primarily data-oriented structures using `data classes`. These are classes whose primary purpose is to hold data, so-called Data Transfer Objects (DTOs), and are the recommended approach in Kotlin[4]. In Figure 2 we have the base classes for terms and values, and a few examples of structures that map directly from the syntax to a data object.

```
sealed class Term
sealed class Value

data class TLet(val id: String, val bind: Term, val body: Term) : Term()
data class TSigma(val id: String, val type: Term, val body: Term) : Term()
data class TPair(val left: Term, val right: Term) : Term()

data class VPair(val left: Value, val right: Value) : Value()
```

Listing 2: Pair and *let* − *in* representations

[4] https://kotlinlang.org/docs/idioms.html

Terms that encode computation, whether delayed (λ-abstraction) or not (application) will be more involved. Variables *can* be represented in a straight-forward way, but a string-based representations is not the most optimal way. We will look at these three constructs in turn.

1.2.1 Functions

Closure Languages where functions are first-class values and not simply procedures to be called all use the concept of a closure, which is, in brief, a function in combination with the environment in it was created. The reason for that is that the body of a function can refer to variables other than its immediate arguments: the simplest example is the *const* function $λx.λy.x, which, when partially applied to a single argument, e.g., let $five = const5$, needs to store the value 5 until it is eventually applied to the remaining second argument: $five15 \longrightarrow 5$.

HOAS As Kotlin supports closures on its own, it would be possible to encode λ-terms directly as functions in the host language. This is possible, and it is one of the ways of encoding functions in interpreters. This encoding is called the higher-order abstract syntax (HOAS), which means that functions (also called *binders* in literature) in the language are equal to functions in the host language. While representing functions using HOAS produces very readable code and in some cases, e.g., on GHC produces code an order of magnitude faster than using other representations[5]. An example of what it looks like is in Listing 3.

```
data class Closure<T>(val fun: (T) -> T)

val constFive = Closure<Int> { (n) -> 5 }
```

Listing 3: Higher-order abstract syntax encoding of a closure

Explicit closures However, we will need to perform some operations on the AST that need explicit access to environments and the arguments of a function. The alternative to reusing functions of the host language is a *defunctionalized* representation, also called *explicit closure* representation. We will need to use this representation later, when creating the Truffle version: function calls will need to be objects, nodes in the program graph, as we will see in Chapter **??**. In this encoding, demonstrated in Listing 4, we store the term of the function body together with the state of the environment when the closure was created.

```
data class Closure<T>(val fun: Term, val environment: Map<Name,Term>)

val constFive = Closure<Int>(TLocal("x"), mapOf("x" to 5))
```

Listing 4: Defunctionalized function representation

[5] https://github.com/AndrasKovacs/normalization-bench

1.2.2 Variables

Representing variables can be as straight-forward as in Listing 4, a variable can be a simple string containing the name of the variable; this is what the parser produces in the pre-term representation. When describing reduction rules and substitution, we have also referred to variables by their names.

Named Often, when specifying a λ-calculus, the process of substitution $t[x := e]$ is kept vague, as a concern of the meta-theory in which the λ-calculus is encoded. When using variable names (strings), the terms and the code for manipulating them is easily understandable. Function application, however, requires variable renaming (α-conversion), which involves traversing the entire argument term and replacing each variable occurrence with a fresh name that does not occur in the function body. However, this is a very slow process, and is not used in any real implementation of dependent types or λ-calculus.

Nameless An alternative to string-based variable representation is a *nameless* representation, which uses numbers in place of variable names [**?**]. These numbers are indices that point to the current variable environment, offsets from the top or the top of the environment stack. The numbers are assigned, informally, by *counting the lambdas*, as each λ-abstraction corresponds to one entry in the environment, which can be represented as a stack, to which a variable gets pushed with every function application, and popped when leaving a function. These two approaches can be seen side-by-side in Figure 1.4.

	fix	$succ$
Named	$(\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))\ g$	$\lambda x.x\ (\lambda y.x\ y)$
Indices	$(\lambda(\lambda 1\ (0\ 0))\ (\lambda 1\ (0\ 0))\ g$	$\lambda 0\ (\lambda 1\ 0)$
Levels	$(\lambda(\lambda 0\ (1\ 1))\ (\lambda 0\ (1\ 1))\ g$	$\lambda 0\ (\lambda 0\ 1)$

Figure 1.4: Named and nameless variable representations

de Bruijn indices The first way of addressing, de Bruijn indexing, is rather well-known. It is a way of counting from the top of the stack, meaning that the argument of the innermost lambda has the lowest number. It is a "relative" way of counting, relative to the top of the stack, which is beneficial during e.g. δ-reduction in which a reference to a function is replaced by its definition: using indices, the variable references in the function body do not need to be adjusted after such substitution.

de Bruijn levels The second way is also called the "reversed de Bruijn indexing" [**?**], as it counts from the start of the stack. This means that the argument of the innermost lambda has the highest number. In the entire term, one variable is only ever addressed by one number, meaning that this is an "absolute" way of addressing, as opposed to the "relative" indices.

Locally nameless There is a third alternative that combines both named and nameless representations, that has been used in e.g., Lean [**?**]. De Bruijn indices are used for bound

variables and string-based names for free variables. This also avoids any need for bound variable substitution, but free variables still need to be resolved later during the evaluation of a term.

Our choice We will use a representation that has been used in recent type theory implementations [**?**] [**?**], that avoids any need for substitution: de Bruijn indices in terms, and de Bruijn levels in values. Terms that are substituted *into* an existing value do not need to adjust the "relative" indices based on the size of the current environment, whereas using the "absolute" addressing of levels in values means that values can be directly compared. Using this combination of representations, we can doing avoid any substitution at all, as any adjustment of the terms is performed during the evaluation from term to value and back.

Implementation Kotlin makes it possible to construct type-safe wrappers over basic data types that are erased at runtime but that support custom operations. Representing indices and levels as `inline classes` means, that that we can perform arithmetic on them using the natural syntax e.g. `ix + 1`, which we will use when manipulating the environment in the next section. The final representation of variables in our interpreter is in Figure 5.

```
inline class Ix(val it: Int) {
    operator fun plus(i: Int) = Ix(it + i)
    operator fun minus(i: Int) = Ix(it - i)
    fun toLvl(depth: Lvl) = Lvl(depth.it - it - 1)
}

inline class Lvl(val it: Int) {
    operator fun plus(i: Int) = Lvl(it + i)
    operator fun minus(i: Int) = Lvl(it - i)
    fun toIx(depth: Lvl) = Ix(depth.it - it - 1)
}

data class VLocal(val it: Lvl) : Val()
data class TLocal(val it: Ix) : Val()
```

Listing 5: Variable representation

1.2.3 Class structure

Variables and λ-abstractions were the two non-trivial parts of the mapping between our syntax and Kotlin values. With these two pieces, we can fill out the remaining parts of the class hierarchy. The full class listing is in Appendix **??**, here we will show only a direct comparison using the *const* function in Figure 1.5, and the important distinctions in Figure 1.6.

```
PLam("x", Expl,    TLam("x", Expl,    VLam("x", Expl,
  PLam("y", Expl,    TLam("y", Expl,    VCl([valX], VLam("y", Expl,
    PVar("x")))        TLocal(1)))        VCl([valX, valY], VLocal(0)))))
```

Figure 1.5: Direct comparison of `PreTerm`, `Term`, and `Value` objects

	Variables	Functions	Properties
`PreTerm`	String names	`PreTerm` AST	well-formed
`Term`	de Bruijn index	`Term` AST	well-typed
`Value`	de Bruijn level	Closure (`Term` AST + `Values` in context)	normal form

Figure 1.6: Important distinctions between `PreTerm`, `Term`, and `Value` objects

1.3 Normalization

1.3.1 Approach

Normalization-by-evaluation Normalization, as defined in Chapter **??**, is a series of βδζι-reductions. While there are systems that implement normalization as an exact series of reduction rules, it is an inefficient approach that is not common in the internals of state-of-the-art proof assistants. An alternative way of bringing terms to normal form is the so-called *normalization-by-evaluation* (NbE) [**?**]. The main principle of this technique is interpretation from the syntactic domain of terms into a computational, semantic domain of values and back. In brief, we look at terms as an executable program that can be *evaluated*, the result of such evaluation is then a normal form of the original term.

Neutral values If we consider only closed terms that reduce to a single constant, we could simply define an evaluation algorithm over the terms defined in the previous chapter. However, normalization-by-evaluation is an algorithm to bring any term into a full normal form, which means evaluating terms inside function bodies and constructors. NbE introduces the concept of "stuck" values that cannot be reduced further. In particular, free variables in a term cannot be reduced, and any terms applied to a stuck variable cannot be further reduced and are "stuck" as well. These stuck values are called *neutral values*, as they are inert with regards to the evaluation algorithm.

Semantic domain NbE is total and provably confluent [**?**] for any abstract machine or computational domain. Proof assistants use abstract machines like Zinc or STG; any way to evaluate a term into a final value is viable. This is also the reason to use Truffle, as we can translate a term into an executable program graph, which Truffle can later optimize as necessary. In this first interpreter, however, the computational domain will be a simple tree-traversal algorithm.

The set of neutral values in Montuno is rather small (Figure 1.7). An unknown variable, function application with a neutral *head* and arbitrary terms in the *spine*, and a projection eliminator.

$$\mathit{neutral} \quad := \quad \mathit{var} \quad | \quad \mathit{neutral}\ a_1 \ldots a_n \quad | \quad \mathit{neutral}.l_n$$

Figure 1.7: Neutral values

1.3.2 Normalization strategies

Normalization-by-evaluation is, at its core inefficient for our purposes, however [**?**]. The primary reason to normalize terms in the interpreter is for type-checking and inference and that, in particular, needs normalized terms to check whether two terms are equivalent. NbE is an algorithm to get a full normal form of a term, whereas to compare values for equality, we only need the weak head-normal form. To illustrate: to compare whether a λ-term and a pair are equal, we do not need to compare two fully-evaluated values, but only to find out whether the term is a pair of a λ-term, which is given by the outermost constructor, the *head*.

In Chapter **??** we saw an overview of normal forms of λ-calculus. To briefly recapitulate, a normal form is a fully evaluated term with all sub-terms also fully evaluated. A weak head-normal form is a form where only the outermost construction is fully evaluated, be it a λ-abstraction or application of a variable to a spine of arguments.

Reduction strategy Normal forms are associated with a reduction strategy, a set of small-step reduction rules that specify the order in which sub-expressions are reduced. Each strategy brings an expression to their corresponding normal form. Common ones are *applicative order* in which we first reduce sub-expressions left-to-right, and then apply functions to them; and *normal order* in which we first apply the leftmost function, and only then reduce its arguments. In Figure 1.8 there are two reduction strategies that we will emulate.

$$x \xrightarrow{name} x$$

$$\frac{}{(\lambda x.e) \xrightarrow{name} (\lambda x.e)}$$

$$\frac{e_1 \xrightarrow{name} (\lambda x.e) \qquad e[x := e_2] \xrightarrow{name} e'}{(e_1\ e_2) \xrightarrow{name} e'}$$

$$\frac{e_1 \xrightarrow{name} e_1' \not\equiv \lambda x.e}{(e_1\ e_2) \xrightarrow{name} (e_1'\ e_2)}$$

(a) Call-by-name to weak head normal form

$$x \xrightarrow{norm} x$$

$$\frac{e \xrightarrow{norm} e'}{(\lambda x.e) \xrightarrow{norm} (\lambda x.e')}$$

$$\frac{e_1 \xrightarrow{name} (\lambda x.e) \qquad e[x := e_2] \xrightarrow{norm} e'}{(e_1\ e_2) \xrightarrow{norm} e'}$$

$$\frac{e_1 \xrightarrow{name} e_1' \not\equiv \lambda x.e \qquad e_1' \xrightarrow{norm} e_1'' \qquad e_2 \xrightarrow{norm} e_2'}{(e_1\ e_2) \xrightarrow{norm} (e_1''\ e_2)}$$

(b) Normal order to normal form

Figure 1.8: Reduction strategies for λ-calculus [**?**]

In general programming language theory, a concept closely related to reduction strategies is an evaluation strategy. These also specify when an expression is evaluated into a value, but in our case, they apply to our host language Kotlin.

Call-by-value Call-by-value, otherwise called eager evaluation, corresponds to applicative order reduction strategy [**?**]. Specifically, when executing a statement, its sub-expressions are evaluated inside-out and immediately reduced to a value. This leads to predictable program performance (the program will execute in the order that the programmer wrote it,

evaluating all expressions in order), but this may lead to unnecessary computations performed: given an expression `const 5 (ackermann 4 2)`, the value of `ackermann 4 2` will be computed but immediately discarded, in effect wasting processor time.

Call-by-need Call-by-need, also lazy evaluation, is the opposite paradigm. An expression will be evaluated only when its result is first accessed, not when it is created or defined. Using call-by-need, the previous example will terminate immediately as the calculation `ackermann 4 2` will be deferred and then discarded. However, it also has some drawbacks, as the performance characteristics of programs may be less predictable or harder to debug.

Call-by-value is the prevailing paradigm, used in all commonly used languages with the exception of Haskell. It is sometimes necessary to defer the evaluation of an expression, however, and lazy evaluation is emulated using closures or zero-argument functions: e.g., in Kotlin a variable can be initialized using the syntax `val x by lazy { ackermann(4, 2) }`, and it will be initialized only if is ever needed.

Call-by-push-value There is also an alternative paradigm, called call-by-push-value, which subsumes both call-by-need and call-by-value as they can be directly translated to CBPV– in the context of λ-calculus specifically. It does this by defining additional operators *delay* and *force*, one to create a *thunk* that contains a deferred computation, one to evaluate the thunk. Also notable is that it distinguishes between values and computations: values can be passed around, but computations can only be executed, or deferred.

Emulation We can emulate normalization strategies by implementing the full normalization-by-evaluation algorithm, and choosing the evaluation strategy. While evaluation strategy is often an intrinsic property of a language, and Kotlin is by default a call-by-value language, in our case it means inserting `lazy` annotations in the correct place, so that only those values that are actually used are evaluated. In the case of the later Truffle implementation, we will need to implement explicit *delay* and *force* operations of call-by-push-value, which is why we introduced all three paradigms in one place.

1.3.3 Implementation

The basic outline of the implementation is based on Christiansen's [**?**].

Environments Closely related to the topic of variable representation are environments. When presenting the λ→-calculus, we have seen the typing context Γ. We will Γ, and also a value context, to store argument values during evaluation.

Now that we have variables, we need to store them somewhere: environments (Γ). We need two contexts, or one merged, for types and values [**?**] Also, built-ins in the initial context.

$$\Gamma \quad := \quad \bullet \quad | \quad \Gamma, x : t$$

Given the spec, an environment is a stack, implemented as a linked list. The environment is changed whenever we enter or leave a lambda during evaluation or inference, so we need cheap pop/push operations.

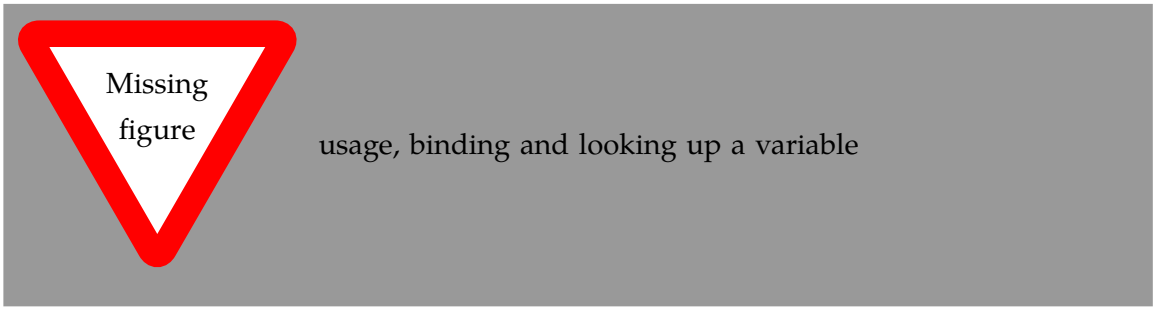
In Java/JVM, an ArrayDeque is the data structure to use for fast stack implementation, but we need to capture the stack every time we create a closure.

Immutable data structures are easy to reason about and easy to translate algorithms to, as whenever we need create a closure we just use the current context, but expensive in JVM world. Mutable means that we need to explicitly add a copy/clone operation every time we capture a context.

In Truffle, but not now, we will also need to care about which variables actually need to be capture in a closure - we do not want expensive dangling references.

For now, let's just implement a custom linked list-based environment, see Listing X.

The two operations an environment needs: bind a variable, and look up a variable by name/index (as unbinding a variable is implicit): in Listing X



An example AST with an environment?

```
const : {A B} → A → B → A      Γ   EnvCons(A, x,
const = λx.λy.x                                EnvCons(B, y, Nil))
              Λ
        ------|
```

– Figure: evaluation with *a* in the context, and without (stuck)

Listing: Key algorithm parts - bind env, call

1.3.4 Primitive operations

Primops can be intrinsified, but we will bind them in the initial context, keeping the core language small

A primop needs:

- a symbol (nullary constructor)
- a neutral head (when it gets stuck)
- evaluation rule in eval (as many VLams as necessary to apply it, so that the computation rule can be tested)

Not projections, though:

- Fst:Term, (`eval env Fst = VLam \$ \v -> vFst v`),
- projections should go to spines, not heads
- neutrals should make access to the Ex that blocks computation as easy as possible,
- see https://github.com/AndrasKovacs/setoidtt/blob/master/proto/Evaluation.hs#L293 for an example

1.4 Elaboration

Why? Ease of use, ... ([**?**] - elaboration, bi-di)

– Figure of a partially and fully typed program

1.4.1 Bidirectional typing

What approaches are possible and widely used? Bidirectional or constraint solving? Or both?

Bidirectional typing (https://www.cl.cam.ac.uk/~nk480/bidir-survey.pdf) = now standard approach, combines type-checking and inference, simpler to implement even if inference is not required

Bidirectional is what we will implement - Truffle is good at optimizing repetitive computation tasks

Two complementary tasks, check and infer. What are they, the gist of their principle.

- [**?**] - full rules, polarized??

$$\frac{a : t \in \Gamma}{\Gamma \vdash a \Rightarrow t} \qquad \frac{c \text{ is a constant of type } t}{\Gamma \vdash c \Rightarrow t}$$

$$\frac{\Gamma, x : t \vdash e \Leftarrow u}{\Gamma \vdash \lambda x.e \Leftarrow t \to u} \qquad \frac{\Gamma \vdash f \Rightarrow t \to u \qquad \Gamma \vdash a \Rightarrow t}{\Gamma \vdash f\ a \Rightarrow u}$$

$$\frac{\Gamma \vdash a \Rightarrow t}{\Gamma \vdash a \Leftarrow t} \qquad \frac{\Gamma \vdash a \Leftarrow t}{\Gamma \vdash (a : t) \Rightarrow t}$$

Figure 1.9: Bidirectional typing rules for the λ→-calculus

1.4.2 Typing rules

Special notation that we haven't seen yet:

- $\Gamma \vdash e \Rightarrow t$ means "given the context Γ and expression e, infer its type t"
- $\Gamma \vdash e \Leftarrow t$ means "given the context Γ, expression e and type t, the type t checks against the expression t".

Our entire typing system can be written in this way, this is a simpler variant:

1.4.3 Meta-context

We will need a meta-context to store not-yet-known types, these are meta-variables. These can be explicit, as holes, or implicit, a not-yet-used variable is mentioned.

Simple-to-implement is a global context - a scoped one is also possible though. It is also possible to freeze/prune the meta-context. Using the globally-scoped meta-context, we will process statements sequentially, adding meta-variables at the top-level. Meta-variables are global, but they depend on local values - they need to be functions.

– Figure of a meta-block, and used meta-variables (A context of "id id id" with three metas?)

1.4.4 Unification

So, two main tasks: infer and check. A large part of these is checking if two types are equal. Unification (instead of conversion checking)

The usual notion of equality in λ-calculus is *α-equivalence of β-normal forms*, structural equality; meaning that we first reduce both expressions as much as possible and then compare the symbols they are made of, ignoring differences in variable naming.

Unification is the same principle, but in the presence of variables, constants, ...

Unification is checking whether two values are the same. Unification solves meta-variables as a side-effect. [**?**] - well-specified unification algorithm, pruning

First-order unification - can solve for base types Higher-order unification - can also produce function types. Pattern unification, as higher-order is indecidable in general - distinct variables, not recursive

1.4.5 Meta-variable solving

Unifying the kind of an expression, if a function application then head and also spine (= list of arguments), is an unsolved meta-variable then attempt to solve the variable

In general, solving is moving variables from the left to the right (= pattern unification???) by a "renaming" - maps variables from the spine to final de Bruijn level in the solution; is partial, requires depth of metacontext, and final context

Also, blocking or non-blocking - a flex does not block (is a meta), can get solved and unblocked; a rigid is a local variable (possibly with a spine), cannot unblock

Also, forcing: brings metas up to date with respect to the environment, up to WHNF (BUT meta solving should be destructive, sharing)

[**?**] - unifying sigma

– Figure: unifying "id" and "λx.x", e.g.

1.4.6 Implementation (or inline, at each part?)

Side-by-side algorithm and implementation = straight-forward translation

1.5 Driver

Flow, modes (decl/defn, commands), global name space

1.5.1 Parser

Lexical and syntactic analysis is not the focus of this work, so simply I chose the most prevalent parsing library in Java-based languages, which seems to be ANTLR [6]. It comes with a large library of languages and protocols from which to take inspiration [7], so creating a parser for our language was not hard, despite me only having prior experience with parser combinator libraries and not parser generators.

ANTLR provides two recommended ways of consuming the result of parsing using classical design patterns: a listener and a visitor. I have used neither as they were needlessly verbose or limiting [8].

I have instead implemented a custom recursive-descent AST transformation that converts `ParseContexts` created by ANTLR into our `Presyntax` data type that we can see in Listing 6. This is actually a slightly simplified version compared to the original as I have omitted the portion that tracks which position of the input file corresponds to each subtree, which is later used for type error reporting.

[6] https://www.antlr.org/

[7] https://github.com/antlr/grammars-v4/

[8] In particular, ANTLR-provided visitors require that all return values share a common super-class. Listeners don't allow return values and would require explicit parse tree manipulation.

```
sealed class TopLevel
data class RDecl(val n: String, val type: PreTerm) : TopLevel()
data class RDefn(val n: String, val type: PreTerm?, val term: PreTerm) : TopLevel()
data class RTerm(val cmd: Command, val term: PreTerm) : TopLevel()

sealed class PreTerm

data class RVar(val n: String) : PreTerm()
data class RNat(val n: Int) : PreTerm()
object RU : PreTerm()
object RHole : PreTerm()

data class RApp(
    val icit: Icit, val rator: PreTerm, val rand: PreTerm
) : PreTerm()
data class RLam(
    val name: String, val icit: Icit, val body: PreTerm
) : PreTerm()
data class RFun(
    val domain: PreTerm, val codomain: PreTerm
) : PreTerm()
data class RPi(
    val name: String, val icit: Icit, val type: PreTerm, val body: PreTerm,
) : PreTerm()
data class RLet(
    val name: String, val type: PreTerm, val defn: PreTerm, val body: PreTerm,
) : PreTerm()
data class RForeign(
    val lang: String, val eval: String, val type: PreTerm
) : PreTerm()
```

Listing 6: The Presyntax data type

We have a `TopLevel` class with three children that represent: definitions that assign a value to a name, optionally with a type; declarations (sometimes called postulates) that only assign a type to a name; and commands like `%normalize` that we will see in later sections.

Their ancestor is a `sealed class` which tells the compiler there will only ever be the subclasses defined in this module. In particular, this means that in any pattern match on the type of a `TopLevel` object we only ever need to handle three cases.

The remaining classes generally map to elements from the language syntax as specified in the previous section: a λ-abstraction, function application, Π-abstraction, a `let` local binding. The `RFun` class is a specialization of the `RPi` class that binds an unnamed, non-dependent type argument.

We will not use these classes immediately–only in Chapter 1.4 will we implement a way to convert this pre-syntax into correct-by-construction `Syntax` objects which can be evaluated and quoted.

— parse recovery

1.5.2 Pretty-printer

The other way around, a way to convert a Term into String, so that it can be printed

A simple AST traversal, only need to track precedence

1.5.3 Error reporting

Needs to be wired-in at many steps - in the parser, piped through the data types, the type-checker, ...

Position tracking in context - "expression under evaluation"

Also will be used in Truffle

1.5.4 State

Global name table

Type context, value context, ...

1.6 Frontend

The user interface we will use is a command-line one: a multi-purpose command that can start an interactive command-line session, execute a file, or pre-compile a source file into a Native Image.

1.6.1 CLI

Reuse Launcher from Truffle, it is the easiest way considering that we will need to use it anyway.

Very simple though, either stdin or a file, parse, execute

1.6.2 REPL

From my research, JLine is the library of choice for interactive command-line applications in Java, so that is what I used for the REPL (Read-Eval-Print Loop). It is a rather easy-to-use library: for the most basic use case, we only need to provide a prompt string, and a callback. We can also add auto-completion, a parser to process REPL commands, custom keybindings, built-in pager or multiplexer, or even a scripting engine using Groovy.

Look at [**?**] for other interaction modes:

- golden tests (`:verbose` as a command, e.g.)
- :verbose = write out after every reduction

- :ski = express in terms of combinators
- interpreter + batch processor that reads REPL commands as well
- Jupyter kernel
- :types on, switches between untyped and simply-typed
- @@(expr) = print out as a proof (type derivation) using typing rules

State keeping - load, reload + querying the global state (1 para)

commands:

- `:l` create a NameTable
- `:r` recreate a NameTable
- `:t` inferVar, print unfolded
- `:nt` inferVar, print folded
- `:n` inferVar type, gQuote term, show
- `:e` print elaboration output including all metas

1.7 Results

Quick look at everything that this toy can do (2-3 examples?)

Evaluation, simplification, elaboration with holes, unification using eqRefl

Error reporting

1.8 Bleh

- [**?**] - trace-based interpreter for GHC, a different approach
- [**?**]

- [**?**], [**?**] - "I had discovered the ΠΣ paper when finishing my thesis: too late, unfortunately"
- [**?**] is an interesting tutorial of a dependent interpreter of dependent languages
- [**?**] - well described contexts + language specification - can I take as inspiration?
- well investigated in [**?**] where there is a comprehensive of NbE techniques as applied to ML
- also in [**?**] there is a treatment of η reduction/expansion - READ

We do not want to do η-reduction, as it might introduce non-termination or undecidability (and in general is not compatible with subtyping relations). η-expansion is sound for producing $\beta\eta$-long normal forms.

If we do not have a $\beta\eta$-long normal form and want to unify/compare (λx:A.M) with N, you unify/compare

$$N\ X$$

and

$$M[x := X]$$

for a fresh X. This is sufficient for $\beta\eta$-equality (for the easy η-rule for Π/λ).