# 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.

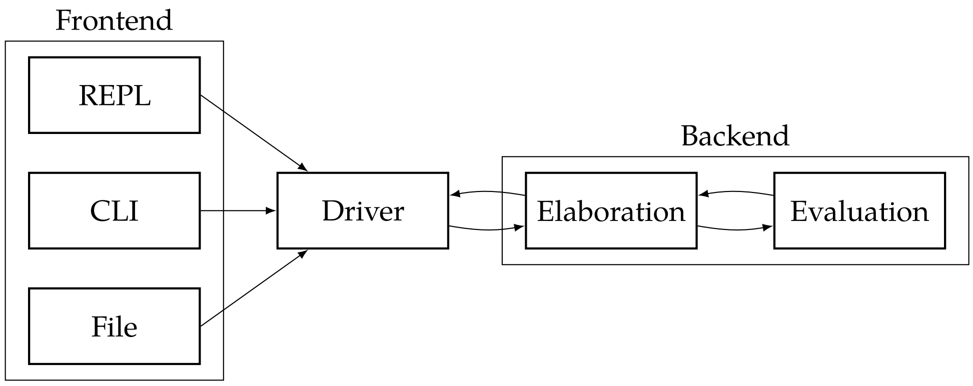

Figure 1.1: Overview of interpreter components

### 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.

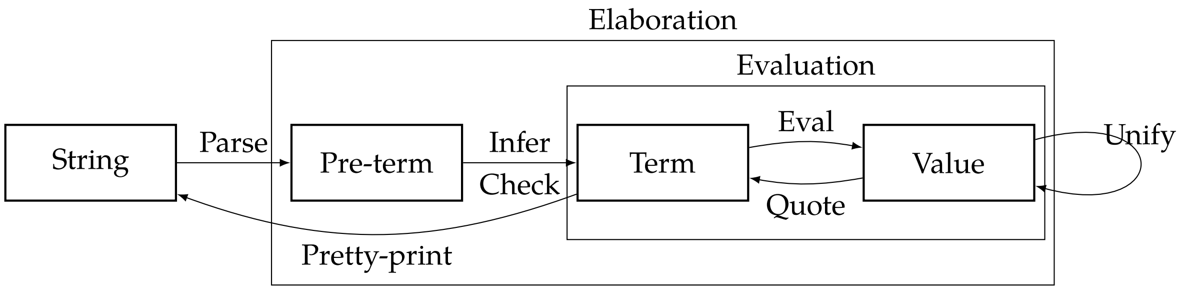

Figure 1.2: Data flow overview

**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 **??**.

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

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 ($\lambda$-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 $five = const 5$, needs to store the value 5 until it is eventually applied to the remaining second argument: $five 15 \longrightarrow 5$.

**HOAS** As Kotlin supports closures on its own, it would be possible to encode $\lambda$-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.

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

Figure 1.7: Neutral values

**Specification** The NbE algorithm is fully formally specifiable, which involves the four operations: the above-mentioned evaluation and quoting, and also reflection of a neutral value (*NeVal*) into a value, and reflection of a value into a normal value (*NfVal*) that includes its type, schematically shown in Figure 1.8. In this thesis, though, we will only describe the relevant parts of the specification in words, and simply say that NbE as we will implement it, is a pair of function $nf = quote(eval(term))$.

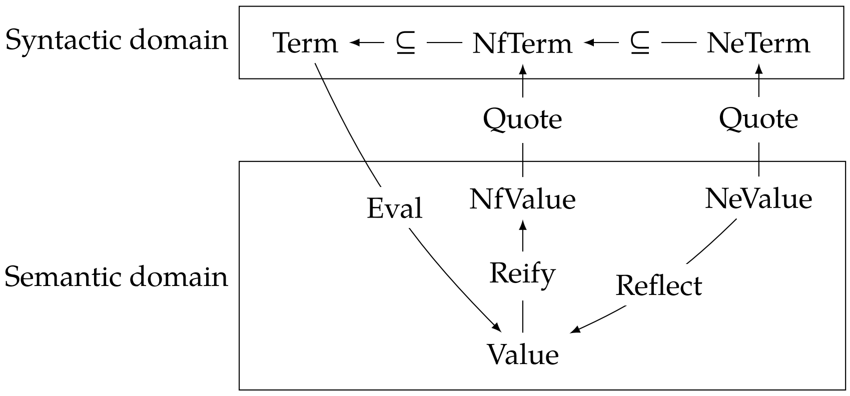

Figure 1.8: Syntactic and semantic domains in NbE [**?**]

### 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 1.9 there are two reduction strategies that we will emulate.

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.

$$x \xrightarrow{name} x$$

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

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

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

(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 1.9: Reduction strategies for λ-calculus [**?**]

**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 [**?**]. In essence, it implements the obvious evaluation algorithm: evaluating a function captures the current environment in a closure, evaluating a variable looks up its value in the environment, and function application inserts the argument into the environment and evaluates the body of the function.

**Environments** The brief algorithm description used a concept we have not yet translated into Kotlin: the environment, or evaluation context. When presenting the $\lambda \rightarrow$-calculus, we have seen the typing context $\Gamma$, to which we will add a value context.

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

The environment, following the above definition, is a stack: defining a variable pushes a pair of a name and a type to the top, and when the variable goes out of scope, it is popped off. An entry is pushed and popped whenever we enter and leave a function context, and the entire environment needs to be captured in its current state whenever we create a closure. When implementing closures in Truffle, we will also need to take care about which variables are actually used in a function and need to be captured, so that we do not capture the entire environment.

**Linked list** The natural translation of the environment definition is a linked list. In a functional language like Haskell, that would be the most efficient implementation, as appending to an immutable list is very cheap there but in Kotlin, we need to take care about not allocating too many objects and need to consider mutable implementations as well.

**Mutable/immutable** In Kotlin and other JVM-based languages, an `ArrayDeque` is the data structure to use for fast stack implementation, which is a mutable implementation of a stack data structure. In general, array-backed data structures are faster than recursive ones on the JVM, which we will use in the Truffle implementation. In this first implementation, however, we can use the easier-to-use immutable linked list implementation, shown in Listing 6. There we can see the value environment, an equivalent structure is also implemented for types.

**Environment operations** We need three operations from an environment data structure: insert (bind) a value, look up a bound value by its level or index, and unbind a variable

```
data class VEnv(val value: Val, val next: VEnv?)

fun VEnv?.len(): Int = if (this == null) 0 else 1 + next.len()
operator fun VEnv?.plus(v: Val): VEnv = VEnv(v, this)
operator fun VEnv?.get(n: Ix): Val
   = if (n.it == 0) this!!.value else this!!.next[n - 1]
```

Listing 6: Environment data structure as an immutable linked list

that leaves the scope. In Listing 6, we see two of them: the operator `plus`, used as `env + value`, binds a value, and operator `get`, used as `env[ix]`, looks a value up. Unbinding a value is implicit, because this is an immutable linked list: the reference to the list used in the outer scope is not changed by any operations in the inner scope. These operations are demonstrated in Listing 7, on the `eval` operations of a variable and a *let* − *in* binding.

There we see also the basic structure of the evaluation algorithm. Careful placement of `lazy` has been omitted, as it splits the algorithm into two, values that need to be evaluated lazily and those that do not, but the basic structure should be apparent. The snippet uses the Kotlin `when-is` construct, which checks the class of the argument, in this case we check if `this` is a `TLocal`, `TLet`, etc.

```
fun eval(ctx: Context, term: Term, env: VEnv): Val = when (term) {
  is TLocal ->
    env[term.ix] ?: VNeutral(HLocal(Lvl(ctx.lvl - term.ix - 1), spineNil))
  is TLet -> eval(ctx, term.body, env + eval(ctx, term.defn, env))
  is TLam -> VLam(term.name, VCl(env, term.body))
  is TApp -> when (fn := eval(ctx, term.lhs, env)) {
    is VLam -> eval(ctx, fn.cl.term, fn.cl.env + eval(ctx, term.rhs, env))
    is VNeutral -> VNeutral(fn.head, fn.spine + term.right)
  }
  // ...
}
```

Listing 7: Demonstration of the `eval` algorithm

**Eval** In Listing 7, a variable is looked up in the environment, and considered a neutral value if the index is bigger than the size of the current environment. In `TLet` we see how an environment is extended with a local value. A λ-abstraction is converted into a closure and function application, if the left-hand side is a `VLam` evaluates the body of this closure. If the left-hand side is a neutral expression, then the result is also neutral, only the spine is extended with another argument. Other language constructs are handled in a similar way,

**Quote** In Listing 8, we see the second part of the algorithm. In the domain of values, we do not have plain variable terms, or *let*−*in* bindings, but unevaluated functions and "stuck" neutral terms. A λ-abstraction, in order to be in normal form, needs to have its body also in normal form, therefore we insert a neutral variable into the environment in place of the argument, and eval/quote the body. A neutral term, on the other hand, has at its head a neutral variable. we convert this variable into a term-level variable, and reconstruct the spine as a tree of nested `TApp` applications.

```
fun quote(ctx: Context, v: Val): Term = when (v) {
  is VNeutral -> {
    x = TLocal(Ix(ctx.depth - v.head - 1))
    for (vSpine in v.spine.reversed()) {
        x = TApp(x, quote(ctx, vSpine))
    }
    x
  }
  is VLam -> TLam(v.name,
      quote(ctx, eval(ctx, v.cl.body, v.cl.env + VNeutral(HLocal(ctx.lvl)))))
  // ...
}
```

Listing 8: Demonstration of the `quote` algorithm

These two operations work together, to fully quote a value, we need to also lazily `eval` its sub-terms. The main innovation of the normalization-by-evaluation approach is the introduction of neutral terms, which have the role of a placeholder value in place of a value that has not yet been supplied. As a result, the expression $quote(eval(term, emptyEnv))$ produces a lazily evaluated normal form of a term in a weak head-normal form, with its sub-terms being evaluated whenever accessed. Printing out such a term would print out the fully normalized normal form.

**Primitive operations** Built-in language constructs like $Nat$ or $false$ that have not been shown in the snippet are mostly inserted into the initial context as values that can be looked up by their name. In general, though, constructs with separate syntax, e.g. $\Sigma$-types, consist of three parts:

- their type is bound in the initial context;
- the term constructor is added to the set of terms and values, and added in `eval()`;
- the eliminator is added as a term and as a spine constructor, i.e., an operation to be applied whenever the neutral value is provided.

The full listing is provided in the supplementary source code, as it is too long to be included in text.

## 1.4 Elaboration

### 1.4.1 Approach

The second part of the internals of the compiler is type elaboration. Elaboration is the transformation of a partially-specified, well-formed program submitted by a user into a fully-specified, well-typed internal representation [?]. In particular, we will use elaboration to infer types of untyped Curry-style $\lambda$-terms, and to infer implicit function arguments that were not provided by the user, demonstrated in Figure 1.10.

|  |  |
|---|---|
| function signature: | $id : \{A\} \to A \to A$ |
| provided expression: | $id\ id\ 5$ |
| elaborated expression: | $(id\ \{Nat \to Nat\}\ id)\ \{Nat\}\ 5$ |

Figure 1.10: Demonstration of type elaboration

**Bidirectional typing** Programmers familiar with statically-typed languages like Java are familiar with type checking, in which all types are provided by the user, and therefore are inputs to the type judgment $\Gamma \vdash e : t$. Omitting parts of the type specification means that the type system not only needs to check the types for correctness, but also infer (synthesize) types, the type $t$ in $\Gamma \vdash e : t$, producing output. In some systems, it is possible to omit all type annotations and only rely on the type constraints of built-in functions and literals. Bidirectional systems that combine both input and output modes of type judgment are now a standard approach [?], often used in combination with constraint solving.

**Judgments** The type system is composed of two additional type judgments we haven't seen yet, that describe the two directions of computation in the type system:

- $\Gamma \vdash e \Rightarrow t$ is "given the context $\Gamma$ and term $e$, infer (synthesize) its type $t$", and
- $\Gamma \vdash e \Leftarrow t$ is "given the context $\Gamma$, term $e$ and type $t$, check that $t$ is a valid type for $t$".

The entire typing system described in Chapter **??** can be rewritten using these type judgments. The main principle is that language syntax is divided into two sets of constructs: those that constrain the type of a term and can be checked against an inferred term, and those that do not constrain the type and need to infer it entirely.

$$\frac{a : t \in \Gamma}{\Gamma \vdash a \Rightarrow t}\ (\textbf{Var}) \qquad \frac{c \text{ is a constant of type } t}{\Gamma \vdash c \Rightarrow t}\ (\textbf{Const})$$

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

$$\frac{\Gamma \vdash a \Rightarrow t \qquad \Gamma \vdash a = b}{\Gamma \vdash a \Leftarrow b}\ (\textbf{ChangeDir}) \qquad \frac{\Gamma \vdash a \Leftarrow t}{\Gamma \vdash (a : t) \Rightarrow t}\ (\textbf{Ann})$$

Figure 1.11: Bidirectional typing rules for the λ→-calculus

**Bidirectional λ→ typing** In Figure 1.11, this principle is demonstrated on the simply-typed λ-calculus with only variables, λ-abstractions and function application. The first four rules correspond to rules that we have introduced in Chapter **??**, with the exception of the constant rule that we have not used there. The two new rules are (**ChangeDir**) and (**Ann**): (**ChangeDir**) says that if we know that a term has an already inferred type, then we can satisfy any rule that requires that the term checks against a type equivalent to this one. (**Ann**) says that to synthesize the type of an annotated term $a : t$, the term first needs to check against that type.

Rules (**Var**) and (**Const**) produce an assumption, if a term is already in the context or a constant, then we can synthesize its type. In rule (**App**), if we have a function with an inferred type then we check the type of its argument, and if it holds then we can synthesize the type of the application $f\ a$. To check the type of a function in rule (**Abs**), we first need to check whether the body of a function checks against the type on the right-hand side of the arrow.

While slightly complicated to explain, this description produces a provably sound and complete type-checking system [**?**] that, as a side effect, synthesizes any types that have not been supplied by the user. Extending this system with other language constructs is not complex: the rules used in Montuno for local and global definitions are in Figure 1.12.

$$\frac{\Gamma \vdash t \Leftarrow \star \qquad \Gamma \vdash a \Leftarrow t \qquad \Gamma, x : t \vdash b \Rightarrow u}{\Gamma \vdash \text{let } x : t = a \text{ in } b \Rightarrow u} \ (\textbf{Let-In})$$

$$\frac{\Gamma \vdash t \Leftarrow \star \qquad \Gamma \vdash a \Leftarrow t}{\Gamma \vdash x : t = a \Rightarrow t} \ (\textbf{Defn})$$

Figure 1.12: Bidirectional typing rules for *let* − *in* and top-level definitions

**Meta-context** One concern was not mentioned in the previous description: when inferring a type, we may not know all its component types: in rule (**Abs**), the type of the function we check may only be constrained by the way it is called. Implicit function arguments $\{A\ B\} \to A \to B \to A$ also only become specific when the function is actually called. The solution to this problem is a *meta-context* that contains *meta-variables*.

These stand for yet undetermined terms [**?**], either as placeholders to be filled in by the user in interactive proof assistants (written as $?\alpha$), or terms that can be inferred from other typing constraints using unification. These meta-variables can be either inserted directly by the user in the form of a hole "_", or implicitly, when inferring the type of a λ-abstraction or an implicit function argument [**?**].

There are several ways of implementing this context depending on the scope of meta-variables, or whether it should be ordered or the order of meta-variables does not matter. A simple-to-implement but sufficiently useful for our purposes is a globally-scoped meta-context divided into blocks placed between top-level definitions.

```
id : {A} → A → A = λx.x
?α = Nat
?β = ?α → ?α
five = (id ?β id) ?α 5
```

Listing 9: Meta-context for the expression `id id 5`

A meta-context a implemented in Montuno is shown in Listing 9. When processing a file, we process top-level expressions sequentially. The definition of the *id* function is processed, and in the course of processing $five$, we encounter two implicit arguments, which are inserted on the top-level as the meta-variables $?\alpha$ and $?\beta$.

### 1.4.2 Unification

Returning to the rule (**ChangeDir**) in Figure 1.12, a critical piece of the algorithm is the equality of two types that this rule uses. To check a term against a type $\Gamma \vdash a \Leftarrow t$, we first infer a type for the term $\Gamma \vdash a \Rightarrow u$, and then test its equivalence to the wanted type $t = u$.

**Conversion checking** The usual notion of equivalence in λ-calculus is *α-equivalence of β-normal forms*, that we discussed in Chapter **??**, which corresponds to structural equality of the two terms. *Conversion checking* is the algorithm that determines if two terms are convertible using a set of conversion rules.

**Unification** As we also use meta-variables in the type elaboration process, these variables need to be solved in some way. This process of conversion checking together with solving meta-variables is called *unification* [**?**], and is a well-studied problem in the field of type theory.

**Pattern unification** In general, solving meta-variables is undecidable [**?**]. Given the constraint $?\alpha\ 5 = 5$, we can produce two solutions: $?\alpha = \lambda x.x$ and $?\alpha = \lambda x.5$. There are several possible approaches and heuristics: first-order unification solves for base types and cannot produce functions as a result; higher-order unification can produce functions but is undecidable; *pattern unification* is a middle ground and can produce functions as solutions, with some restrictions.

**Renaming** In this thesis, I have chosen to reuse the algorithm from [**?**] which, in brief, assumes that a meta-variable is a function whose arguments are all local variables in scope at the moment of its creation. Then, when unifying the meta-variable with another (non-variable) term, it builds up a list of variables the term uses, and stores such a solution as a *renaming* that maps the arguments to a meta-variable to the variables with which it was unified. As the algorithm is rather involved but tangential to the goals of this thesis, I will omit a detailed description and instead point an interested reader at the original source [**?**].

### 1.4.3 Implementation

As with the implementation of normalization-by-evaluation, we will look at the most illustrative parts of the implementation. This time, the comparison can be made directly side-by-side, between the bidirectional typing algorithm and its implementation.

What was not mentioned explicitly is that the type elaboration algorithm has `PreTerms` as its input, and produces `Terms` in the case of type checking, and pairs of `Terms` and `Values` (the corresponding types) in the case of type inference. Unification, not demonstrated here, is implemented as parallel structural recursion over two `Value` objects.

In Figure 1.13, we see the previously described rule that connects the checking and synthesis parts of the algorithm and uses unification. Unification solves meta-variables as a side-effect, here it is only in the role of a guard as it does not produce a value. The code exactly follows the typing rule, the pre-term is inferred, resulting in a pair of a well-typed term and its type, the type is unified with the wanted type and, if successful, the produced term is the return value.

$$\frac{\Gamma \vdash a \Rightarrow t \qquad \Gamma \vdash a = b}{\Gamma \vdash a \Leftarrow b} \; (\textbf{ChangeDir})$$

```
fun LocalContext.check(pre: PreTerm, wanted: Value): Term = when (pre) {
  // ...
  else -> {
    val (t, actual) = infer(pre.term)
    unify(actual, wanted)
    t
  }
}
```

Figure 1.13: Side-by-side comparison of the **ChangeDir** rule

Figure 1.14 shows the exact correspondence between the rule and its implementation, one read left-to-right, the other top-to-bottom. Checking of the type and value are straightforward, translation of $\Gamma, x : t \vdash b \Rightarrow u$ binds a local variable in the environment, so that the body of the $let - in$ expression can be inferred, and the result is a term containing the inferred body and type, wrapped in a `TLet`.

$$\frac{\Gamma \vdash t \Leftarrow \star \qquad \Gamma \vdash a \Leftarrow t \qquad \Gamma, x : t \vdash b \Rightarrow u}{\Gamma \vdash \text{let}\, x : t = a \,\text{in}\, b \Rightarrow u} \; (\textbf{Let-In})$$

```
fun LocalContext.infer(pre: PreTerm): Pair<Term, Value> = when (pre)
  is RLet -> {
    val t = check(pre.type, VStar)
    val a = check(pre.defn, t)
    val (b, u) = localDefine(pre.name, a, t).infer(pre.body)
    Pair(TLet(pre.name, t, a, b), u)
  }
  // ...
}
```

Figure 1.14: Side-by-side comparison of the **Let-in** rule

Lastly, the rule for a term-level λ-abstraction is demonstrated in Figure 1.15. The type produced on the last line of the snippet is a `VPi` unlike the rule, as the rule was written for the λ→-calculus; it is semantically equivalent, however. This rule demonstrates the creation of a new meta-variable as without a placeholder, we are not able to infer the type of the body of the function. This meta-variable might or might not be solved in the course of inferring the body: either way, both the term and the type only contain a reference to a globally-scoped meta-variable and not the solution.

## 1.5 Driver

This concludes the complex part of the interpreter, what follows are rather routine concerns. Next part of the implementation is the driver that wraps the backend, and handles

$$\frac{\Gamma, x : t \vdash e \Leftarrow u}{\Gamma \vdash \lambda x.e \Leftarrow t \to u}\ (\textbf{Abs})$$

```
fun LocalContext.infer(pre: PreTerm): Pair<Term, Value> = when (pre)
  is RLam -> {
    val a = newMeta()
    val (b, t) = localBind(pre.name, a).infer(pre.body)
    Pair(TLam(pre.name, b), VPi(pre.name, a, VCl(env, t.quote())))
  }
  // ...
}
```

Figure 1.15: Side-by-side comparison of the **Abs** rule

its interaction with the surrounding world. In particular, the parser, pretty-printer, and state management.

**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 the parser was a rather simple matter. ANTLR provides two recommended ways of consuming the result of parsing using classical object-oriented design patterns: a listener and a visitor. I used neither as they were needlessly verbose or limiting[8].

Instead of these, a custom recursive-descent AST transformation was used that is demonstrated in Listing 10. This directly transforms the `ParseContext` objects created by ANTLR into our `PreTerm` data type.

```
fun TermContext.toAst(): PreTerm = when (this) {
  is LetContext -> RLet(id.toAst(), type.toAst(), defn.toAst(), body.toAst())
  is LamContext -> rands.foldRight(body.toAst()) { l, r -> RLam(l.toAst(), r) }
  is PiContext -> spine.foldRight(body.toAst()) { l, r -> l.toAst()(r) }
  is AppContext -> operands.fold(oprator.toAst()) { l, r -> r.toAst()(l) }
  else -> throw UnsupportedOperationException(javaClass.canonicalName)
}
```

Listing 10: Parser to `PreTerm` transformation as a depth-first traversal

The data type itself is shown in Listing 11. As with terms and values, it is a recursive data structure, presented here in a slightly simplified manner compared to the actual implementation, as it omits the part that tracks the position of a term in the original source. The grammar that is used as the source for the parser generator ANTLR was already presented once in the conclusion of Chapter **??**, so the full listing is only included in Appendix **??**.

**Pretty-printer** A so-called pretty-printer is a transformation from an internal representation of a data structure to a user-readable string representation. The implementation of

[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 PreTerm
typealias Pre = PreTerm
typealias N = String

sealed class TopLevel
data class RDecl(val n: N, val type: Pre) : TopLevel()
data class RDefn(val n: N, val type: Pre?, val term: Pre) : TopLevel()
data class RTerm(val cmd: Command, val term: Pre) : TopLevel()

object RU : Pre()
object RHole : Pre()
data class RVar(val n: N) : Pre()
data class RNat(val n: Int) : Pre()
data class RApp(val lhs: Pre, val rhs: Pre) : Pre()
data class RLam(val n: N, val body: Pre) : Pre()
data class RPi(val n: N, val type: Pre, val body: Pre) : Pre()
data class RLet(val n: N, val type: Pre, val defn: Pre, val body: Pre) : Pre()
data class RForeign(val lang: N, val eval: N, val type: Pre) : Pre()
```

Listing 11: Data type `PreTerm`

such a transformation is mostly straight-forward, complicated only by the need to correctly handle operator precedence and therefore parentheses.

This part is implemented using the Kotlin library `kotlin-pretty`, which is itself inspired by the Haskell library `prettyprinter` which, among other things, handles correct block indentation and ANSI text coloring: that functionality is also used in error reporting in the terminal interface.

An excerpt from this part of the implementation is included in Listing 12, which demonstrates the precedence enumeration `Prec`, the optionally parenthesizing operation `par`, and other constructions of the `kotlin-pretty` library.

**State management** Last component of the driver code is global interpreter state, which consists mainly of a table of global names, which is required for handling incremental interpretation or suggestions (tab-completion) in the interactive environment. It also tracks the position of the currently evaluated term in the original source file for error reporting.

Overall, the driver receives user input in the form of a string, parses it, expression by expression supplies it to the backend, receiving back a global name, or an evaluated value, which it pretty-prints and returns back to the user-facing frontend code.

## 1.6 Frontend

We will consider only two forms of user interaction: batch processing of a file via a command-line interface, and a terminal environment for interactive use. Later, with the Truffle interpreter, we can also add an option to compile a source file into an executable using Truffle's capability to produce *Native Images*.

```
enum class Prec { Atom, App, Pi, Let }

fun Term.pretty(ns: NameEnv?, p: Prec = Prec.Atom): Doc<Nothing> = when (this) {
  is TVar -> ns[ix].text()
  is TNat -> n.toString().text()
  is TApp -> par(p, Prec.App,
    arg.pretty(ns, Prec.App) spaced body.pretty(ns, Prec.Atom))
  is TLet -> {
    val d = listOf(
      ":".text() spaced ty.pretty(ns, Prec.Let),
      "=".text() spaced bind.pretty(ns, Prec.Let),
    ).vCat().align()
    val r = listOf(
      "let $n".text() spaced d,
      "in".text() spaced body.pretty(ns + n, Prec.Let)
    ).vCat().align()
    par(p, Prec.Let, r)
  }
  // ...
}
```

Listing 12: Pretty-printer written using `kotlin-pretty`

**CLI** We will reuse the entry point of Truffle languages, a `Launcher` class, so that integration of the Truffle interpreter is easier later, and so that we are able to create single executable that is able to use both interpreters.

What is the Launcher, what does it to? args, options

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

Error reporting - mentioned before, 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")

```
Eval, norm, elab
```

Listing 13: Example usage of the CLI interface

**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
- Jupyter kernel

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

```
Evaluation, simplification, elaboration with holes, unification using eqRefl
```

Listing 14: REPL session example

## 1.7 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 $\eta$ reduction/expansion - READ

We do not want to do $\eta$-reduction, as it might introduce non-termination or undecidability (and in general is not compatible with subtyping relations). $\eta$-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 ($\lambda$x: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 $\eta$-rule for $\Pi/\lambda$).