# ELEGANT ELABORATION WITH FUNCTION INVOCATION

TESLA ZHANG

July 6, 2021

ABSTRACT. We present an elegant design of the core language in a dependently-typed lambda calculus with $\delta$-reduction and an elaboration algorithm.

## 1. INTRODUCTION

Throughout this paper, we will use $\boxed{\text{boxes}}$ in the following two cases:

- to clarify the precedences of symbols when the formulae become too large.
  e.g. $\boxed{\Gamma \vdash \boxed{\lambda x.\, M} \;:\; \boxed{(y : A) \to B} \Leftarrow \boxed{\lambda x.\, u}}$.
- to distinguish type theory terms from natural language text.
  e.g. we combine a term $\boxed{a}$ with a term $\boxed{b}$ to get a term $\boxed{a\ b}$.

In the context of practical functional programming languages, functions can be either *primitive* (like arithmetic operations for primitive numbers, cubical primitives [1, §3.2, §3.4, §5.1] such as `transp`, `hcomp`, `Glue`, etc.) or *defined* (that have user-definable reduction rules, e.g. by using pattern matching). The reduction of function applications are known as $\delta$-reduction [2] and the to-be-reduced terms are called $\delta$-redexes (*redex* for *red*ucible *ex*pressions).

In general, function definitions may reduce only when *fully applied*, while they are also *curried*, allowing the flexible partial application. So, from the type theoretical perspective, functions are always *unary* and function application as an operation is *binary*, while in the operational semantics, function application is n-ary and $\delta$-reduction only works when enough arguments are supplied. We may use *elaboration*, a process that transforms a user-friendly *concrete syntax* into a compact, type theoretical *core language* using type information, to deal with this inconsistency between the intuitive concrete syntax and the optimal internal representation.

For defined functions, even when they are sometimes elaborated as a combination of lambdas and case-trees [3] or eliminators [4] (so we can think of the function syntax as a syntactic sugar of lambdas), the elaborated terms are related to the implementation detail of the programming language, which we tend to hide from the users.

To ensure the functions are fully applied, we may check whether sufficient arguments are supplied every time we want to reduce an application to a function Elaboration is also helpful here: we could choose an efficient representation of function application in the core language and elaborate a user-friendly concrete syntax into it. Consider a term $\boxed{\text{max}\ a\ b}$ where max is a function taking two natural numbers and returning the larger one, we present two styles of function applications in the core language.

- *binary application*. The syntax definition of application is like $e ::= e_1\ e_2$. The above term will be structured as $\boxed{\boxed{\max\ a}\ b}$.
- *spine-normal form*. The syntax definition of application is like $e ::= e_1\ \overline{e}$ ($\overline{e}$ is called a *spine*), where arguments are collected as a list in application terms. The above term will be structured as-is.

If we choose the binary representation, checking the number of supplied arguments requires a traversal of the terms, which is an $O(n)$ process where $n$ is the number of parameters of the applied function (accessing the applied function from an application term is also $O(n)$).

This is not a problem in spine-normal form representation, where we only need to check if the size of the arguments is larger than the number of parameters during $\delta$-reduction. Size comparison is an $O(1)$ process for memory-efficient arrays. However, inserting extra arguments to the spine is a list mutation operation, which requires a list reconstruction in a purely functional setting. The list reconstruction creates a new list with a larger size and copies the old list with the new argument appended to the end, which is again an $O(n)$ process.

Similar problems also exist for *indexed types* [5], where we need the types to be fully applied to determine the available constructors.

### 1.1. Contributions.

We discuss an even more elegant design of application term representation where neither traversal of terms during reduction nor mutation of the argument lists during application are needed. We will use *both* binary application *and* spine-normal form to guarantee that the number of supplied arguments always fits the requirement, and we could always assume function applications to have sufficient arguments supplied.

We present the syntax (in §2) and a bidirectional-style elaboration algorithm (in §3) which can be adapted to the implementation of any programming language with $\delta$-reduction.

## 2. Syntax

We will assume the existence of well-typed *function definitions* (will be further discussed in §3), and focus on lambda calculus terms. Inductive types and pattern matching are also assumed and omitted.

### 2.1. Core language.

The syntax is defined in Fig. 2.1.

Core terms have several assumed properties: well-typedness, well-scopedness, and the functions are *exactly-applied* – the number of arguments matches exactly the number of parameters.

We will assume the substitution operation on core terms, written as $\boxed{u\ [\sigma]}$.

**Example 2.1.** For a function f who has two parameters and returns a function with two parameters, the application $\mathrm{f}\ u_1\ u_2\ u_3\ u_4$ is structured as $\boxed{\boxed{\boxed{\mathrm{f}\ u_1\ u_2}\ u_3}\ u_4}$. The innermost subterm $\mathrm{f}\ u_1\ u_2$ is an exactly-applied function and the outer terms are binary applications. ▲

| | | |
|---|---|---|
| $x, y ::=$ | | variable names |
| $A, B, u, v ::=$ | $\mathsf{f}\,\overline{u}$ | exactly-applied function |
| $\mid$ | $x$ | reference |
| $\mid$ | $u\ v$ | binary application |
| $\mid$ | $(x : A) \to B$ | $\Pi$-type |
| $\mid$ | $\lambda x.\ u$ | lambda abstraction |
| $\sigma ::=$ | $\overline{u/x}$ | substitution object |

FIGURE 2.1. Syntax of terms

2.2. **Concrete syntax.** The concrete syntax is defined in Fig. 2.2. We assume the names to be resolved in concrete syntax, so we can assume the terms to be well-scoped and distinguish local references and function references.

The symbol for local references and functions are overloaded because they make no difference between concrete and core.

| | | |
|---|---|---|
| $M, N ::=$ | $\mathsf{f}$ | function reference |
| $\mid$ | $x$ | local reference |
| $\mid$ | $M\ N$ | application |
| $\mid$ | $\lambda x.\ M$ | lambda abstraction |

FIGURE 2.2. Concrete syntax tree

In the concrete syntax, applications are always binary.

We do not have $\Pi$-types in concrete syntax because they are unrelated to application syntax and their type-checking is relevant to the *universe type* which is quite complicated.

## 3. ELABORATION

In this section, we describe the process that type-checks concrete terms against core terms and translates them into well-typed, exactly-applied core terms.

We define the following operation $\mathsf{apply}(u, v)$ to eliminate obviously reducible core terms:

$$\mathsf{apply}(\lambda x.\ u, v) ::= u[v/x]$$
$$\mathsf{apply}(u, v) ::= u\ v$$

We will use a bidirectional elaboration algorithm [6, 7]. It uses the following typing judgments, parameterized by a context $\Gamma$:

- $\Gamma \vdash M : A \Leftarrow u$, normally called a *checking* or an *inherit* rule.
- $\Gamma \vdash M \Rightarrow u : A$, normally called an *inferring* or a *synthesis* rule.
- $x : A \in \Gamma$, that the context $\Gamma$ contains a local binding $x : A$.

Checking judgments are for introduction rules, while synthesis judgments are for the formation and elimination rules. The direction of the arrows is inspired from [7, §6.6.1]. The arrows ($\Leftarrow$ and $\Rightarrow$) separate the inputs and outputs of the

corresponding elaboration procedure: checking judgments take as input a term in concrete form and its expected type in core form, and produce as output an elaborated version of the term in core form; synthesis judgments take as input a term in concrete form and emit its elaborated form and the inferred type as output.

The context $\Gamma$ contains a list of local bindings (pairs of local variables and types, written as $x : A$) and a list of functions.

A function is defined with a name and a signature (and a body that we do not care about in this paper), where the signature tells us about its parameters and the return type.

3.1. **Abstraction and application.** First of all, we have the basic elaboration rules for type conversion and local references:

$$\frac{\Gamma \vdash M \Rightarrow u : A \qquad \Gamma \vdash A =_{\beta\eta} B}{\Gamma \vdash M : B \Leftarrow u}\ \textsc{Conv} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow x : A}\ \textsc{Var}$$

The rules for lambda abstraction and application are quite straightforward:

$$\frac{\Gamma, x : A \vdash M : B[x/y] \Leftarrow u}{\Gamma \vdash \boxed{\lambda x.\, M} : \boxed{(y : A) \to B} \Leftarrow \boxed{\lambda x.\, u}}\ \textsc{Lam}$$

$$\frac{\Gamma \vdash M \Rightarrow u : \boxed{(x : A) \to B} \qquad \Gamma \vdash N : A \Leftarrow v}{\Gamma \vdash M\ N \Rightarrow \mathsf{apply}(u, v) : B[v/x]}\ \textsc{App}$$

3.2. **Functions.** Before continuing to function references, we need to define the notation for function signature. We define *parameters* to be a list of local bindings:

$$\Delta ::= \overline{x : A}$$

Then, we assume two operations for every function symbol $\mathsf{f}$:

- $\mathsf{params}(\Gamma, \mathsf{f})$ that returns a $\Delta$ that represents the parameters of $\mathsf{f}$ in $\Gamma$.
- $\mathsf{typeOf}(\Gamma, \mathsf{f})$ that returns a term $\boxed{u}$ that represents the type (combining parameters and the return type) of $\mathsf{f}$ in $\Gamma$.

We will need a few more operations before introducing the elaboration rules for functions.

$\mathsf{vars}(\Delta)$: extracts the variables (as a list of terms) from $\Delta$:

$$\mathsf{vars}(\emptyset) ::= \emptyset$$
$$\mathsf{vars}(x : A, \Delta) ::= x, \mathsf{vars}(\Delta)$$

$\mathsf{lambda}(u, \Delta)$: generates a lambda abstraction by induction on $\Delta$:

$$\mathsf{lambda}(u, \emptyset) ::= u$$
$$\mathsf{lambda}(u, \boxed{x : A, \Delta}) ::= \lambda x.\, \mathsf{lambda}(u, \Delta)$$

With these two operations, we can define the elaboration rule for functions. Let $\Delta = \mathsf{params}(\Gamma, \mathsf{f})$, we have the synthesis rule for function references as in Fig. 3.1.

$$\frac{\mathsf{f} \in \Gamma}{\Gamma \vdash \mathsf{f} \Rightarrow \mathsf{lambda}(\boxed{\mathsf{f}\ \mathsf{vars}(\Delta)}, \Delta) : \mathsf{typeOf}(\Gamma, \mathsf{f})}$$

FIGURE 3.1. Elaboration of functions

Here are some examples. Consider the max function discussed in §1, we have the following facts:

$$\mathsf{params}(\Gamma, \mathsf{max}) ::= x : \mathsf{Nat}, y : \mathsf{Nat}$$
$$\mathsf{typeOf}(\Gamma, \mathsf{max}) ::= (x : \mathsf{Nat}) \to (y : \mathsf{Nat}) \to \mathsf{Nat}$$

**Example 3.1.** Concrete term $\boxed{\mathsf{max}}$. By the rule in Fig. 3.1,

$$\begin{aligned}
&\mathsf{lambda}(\mathsf{max}\ \mathsf{vars}(\Delta), \Delta) \\
&= \mathsf{lambda}(\boxed{\mathsf{max}\ \mathsf{vars}(x : \mathsf{Nat}, y : \mathsf{Nat})}, \boxed{x : \mathsf{Nat}, y : \mathsf{Nat}}) && \text{Expand } \Delta \\
&= \mathsf{lambda}(\mathsf{max}\ x\ y, \boxed{x : \mathsf{Nat}, y : \mathsf{Nat}}) && \text{Expand vars} \\
&= \lambda x.\ \lambda y.\ \mathsf{max}\ x\ y && \text{Expand lambda}
\end{aligned}$$

Observe that max is exactly-applied. The result type is $\mathsf{typeOf}(\Gamma, \mathsf{max})$, which equals $(x : \mathsf{Nat}) \to (y : \mathsf{Nat}) \to \mathsf{Nat}$. ▲

**Example 3.2.** Concrete term $\boxed{\mathsf{max}\ M}$, assuming $\Gamma \vdash M : \mathsf{Nat} \Leftarrow u$. We already know the elaboration result of max in example 3.1. By the APP rule, since $\mathsf{apply}(\boxed{\lambda x.\ \lambda y.\ \mathsf{max}\ x\ y}, u) \mapsto \boxed{\lambda y.\ \mathsf{max}\ u\ y}$, the elaborated version of $\mathsf{max}\ M$ is $\lambda y.\ \mathsf{max}\ u\ y$, typed $(y : \mathsf{Nat}) \to \mathsf{Nat}$. Observe that max is still exactly-applied. ▲

**Theorem 3.3.** *Functions are always exactly-applied in the core language, as promised in §1.1.*

*Proof.* We only generate exactly-applied functions in Fig. 3.1, and we do not take arguments away or insert new arguments to function application terms in any other rules or operations. □

## 4. CONCLUSION

We have discussed an elegant core language design with $\delta$-redexes with an elaboration algorithm. With this design, wherever in the compiler, we can assume any given $\delta$-redexes to be exactly-applied. Appending extra arguments to an application term results in a binary application term to a non-function (as in example 2.1).

### 4.1. Implementation.

The discussed core language design is implemented in two proof assistants:

- Arend [8]. There is an abstract class DefCallExpression that generalizes all sorts of *definition invocations*, including functions, data types, constructors,

etc. These expressions are always exactly-applied. The source code is hosted on GitHub [1].

- Aya [9]. Similar to Arend, there is an interface CallTerm in Aya. The source code is also hosted on GitHub [2].

In the implementations, there is one extra complication: invocations to constructors of inductive types have access to the parameters of the inductive type.

4.2. **Related work.** The notion of $\delta$-reduction was discussed in [2], but the $\delta$-redexes are represented in binary application form. Exactly-applied $\delta$-redexes are discussed in [10, 3], but they did not discuss elaboration. The idea of separating spine-normal function application and binary application also appeared in an early work on LISP [11, §6], where spines referred to as *rails* and binary application referred to as *pairs*, but they also did not discuss elaboration.

In the elaboration of Lean 2 [12], functions are transformed into lambdas and *recursors* (and they refer to the corresponding redexes as $\beta$-redexes and $\iota$-redexs [12, §3.3], respectively). This design is not friendly for primitive functions that only work when fully applied.

4.3. **Acknowledgments.** We would like to thank Marisa Kirisame, Ende Jin, Qiantan Hong, and Zenghao Gao for their comments and suggestions on the draft versions of this paper.

## REFERENCES

[1] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. "Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types". Preprint available at https://staff.math.su.se/anders.mortberg/papers/cubicalagda2.pdf. 2020.

[2] Werner Kluge. *Abstract Computing Machines: A Lambda Calculus Perspective*. Springer Science & Business Media, 2006.

[3] Jesper Cockx and Andreas Abel. "Elaborating Dependent (Co)Pattern Matching". In: *Proc. ACM Program. Lang.* 2.ICFP (July 2018). DOI: 10.1145/3236770.

[4] Healfdene Goguen, Conor McBride, and James McKinna. "Eliminating Dependent Pattern Matching". In: *Algebra, Meaning, and Computation: Essays dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*. Ed. by Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 521–540. ISBN: 978-3-540-35464-2. DOI: 10.1007/11780274_27.

[5] Tesla Zhang. "A Simpler Encoding of Indexed Types". In: *Proceedings of the 6th ACM SIGPLAN International Workshop on Type-Driven Development*. TyDe '21. Republic of Korea: ACM, 2021. ISBN: 978-1-4503-8616-6. DOI: 10.1145/3471875.3472991. arXiv: 2103.15408.

[6] Thierry Coquand. "An algorithm for type-checking dependent types". In: *Science of Computer Programming* 26.1-3 (1996), pp. 167–177.

[7] Thierry Coquand et al. "A simple type-theoretic language: Mini-TT". In: *From Semantics to Computer Science; Essays in Honour of Gilles Kahn* (2009), pp. 139–164.

[1] See https://github.com/JetBrains/Arend

[2] See https://github.com/aya-prover/aya-dev

[8] Group for Dependent Types and HoTT. *The Arend Proof Assistant*. https://arend-lang.github.io. JetBrains Research, 2015.

[9] Aya developers. *The Aya Proof Assistant*. https://www.aya-prover.org. 2021.

[10] Semyon Aleksandrovich Nigiyan and Tigran Valery Khondkaryan. "On canonical notion of $\delta$-reduction and on translation of typed $\lambda$-terms into untyped $\lambda$-terms". In: *Proceedings of the YSU A: Physical and Mathematical Sciences* 51.1 (242) (2017), pp. 46–52.

[11] Brian Cantwell Smith. "Reflection and Semantics in LISP". In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '84. Salt Lake City, Utah, USA: Association for Computing Machinery, 1984, pp. 23–35. ISBN: 0897911253. DOI: 10.1145/800017.800513.

[12] Leonardo de Moura et al. *Elaboration in Dependent Type Theory*. 2015. arXiv: 1505.04324 [cs.LO].

THE PENNSYLVANIA STATE UNIVERSITY
*Email address*: yqz5714@psu.edu