1 A small calculus, generated from Lean

This paper presents a simply-typed λ-calculus whose definitions and results are in Lean. Everything in a `lean...` environment below is generated from the Lean development by imprimatur; the rest is written by hand, and the hand-written statements are checked against Lean. The examples are Lean theorems about particular terms, stated by imprimatur as sentences; their names `x`, `y`, `z` and `f` are set in typewriter, to tell them from the metavariables x and y that stand for any name.

1.1 Syntax

Terms are name-carrying: a variable is a name, and an abstraction binds a name. The grammar is generated from the inductive types. The descriptions on the right are the authors': imprimatur keeps them when it regenerates the grammar, and uses the constructors' docstrings only for new ones.

$\tau ::=$	b	the base type
$\mid$	$\tau_1 \to \tau_2$	functions

↗

$e ::=$	x	a variable
$\mid$	$\lambda x{:}\tau.\, e$	an abstraction, binding x in e
$\mid$	$e_1\ e_2$	an application
$\mid$	$()$	the unit value

↗

Terms carry names, so two abstractions that differ only in the name they bind are different terms; α-equivalence, in the next section, identifies them. A name is bound by the abstraction that carries it, and free otherwise.

Example 1 (Names matter)**.** $(\lambda \mathsf{x}{:}\mathsf{b}.\, \mathsf{x}) \neq (\lambda \mathsf{y}{:}\mathsf{b}.\, \mathsf{y})$.

Example 2 (Bound and free names)**.** *(1)* $\mathsf{x} \notin \mathit{fv}(\lambda \mathsf{x}{:}\mathsf{b}.\, \mathsf{x}\ \mathsf{y})$ *and (2)* $\mathsf{y} \in \mathit{fv}(\lambda \mathsf{x}{:}\mathsf{b}.\, \mathsf{x}\ \mathsf{y})$.

1.2 Names, α-equivalence and substitution

We handle names as Pitts does [Pitts(2006)]. The basic operation is *swapping*: $(a\ b) \cdot e$ exchanges the names a and b everywhere in e, bound, binding and free occurrences alike. Unlike renaming, swapping needs no side condition at a binder, and it is its own inverse; its definition is structural:

$$
\begin{aligned}
(a\ b) \cdot x &= (a\ b)(x) \\
(a\ b) \cdot (\lambda x{:}\tau.\, e) &= \lambda (a\ b)(x){:}\tau.\, (a\ b) \cdot e \\
(a\ b) \cdot (e_1\ e_2) &= ((a\ b) \cdot e_1)\ ((a\ b) \cdot e_2) \\
(a\ b) \cdot () &= ()
\end{aligned}
$$

↗

Two abstractions are α-equivalent when swapping each binder's name with a name c fresh for both, written $c \# \ldots$, makes their bodies α-equivalent:

$$\frac{}{x =_\alpha x}\ (\alpha\text{-Var}) \qquad \frac{c \# a, b, e, e' \qquad (a\ c) \cdot e =_\alpha (b\ c) \cdot e'}{(\lambda a{:}\tau.\, e) =_\alpha (\lambda b{:}\tau.\, e')}\ (\alpha\text{-Lam})$$

$$\frac{e_1 =_\alpha e_1' \qquad e_2 =_\alpha e_2'}{e_1\ e_2 =_\alpha e_1'\ e_2'}\ (\alpha\text{-App}) \qquad \frac{}{() =_\alpha ()}\ (\alpha\text{-Unit})$$

The two abstractions below bind their names in opposite orders. Swapping each binder with the fresh name `z` makes the bodies equal, so they are α-equivalent.

Example 3 (Two α-equivalent terms)**.** $(\lambda \mathtt{x}{:}\mathtt{b}.\, \lambda \mathtt{y}{:}\mathtt{b}.\, \mathtt{x}) =_\alpha (\lambda \mathtt{y}{:}\mathtt{b}.\, \lambda \mathtt{x}{:}\mathtt{b}.\, \mathtt{y})$.

Substitution $e[u/x]$ is then defined by recursion on e. At an abstraction $\lambda a{:}\tau.\, e$ it first swaps a with a name c fresh for x, a, e and u, so that no free name of u is captured, and continues in $(a\ c) \cdot e$. Swapping preserves the size of a term, so the recursion terminates, although $(a\ c) \cdot e$ is not a subterm of the abstraction. The renaming matters when the substituted term has a free name that the binder would capture: here `y`, so the binder becomes the fresh name `z`.

Example 4 (Substitution avoids capture)**.** $(\lambda \mathtt{y}{:}\mathtt{b}.\, \mathtt{x})[\mathtt{y}/\mathtt{x}] = (\lambda \mathtt{z}{:}\mathtt{b}.\, \mathtt{y})$.

1.3 Typing

The typing rules are generated from the constructors of `HasType`. Their labels come from the constructors' docstrings. The well-formedness side conditions of the Lean rules are left implicit, by a convention declared in the notation module.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \text{T-Var} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x{:}\tau_1.\, e) : \tau_1 \to \tau_2}\ \text{T-Abs}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \text{T-App} \qquad \frac{}{\Gamma \vdash () : \mathtt{b}}\ \text{T-Unit}$$

In the first example `f` is a function in the context, applied to the bound variable, so every rule but T-Unit takes part in the derivation. The second is a non-example: the unit value has only the base type, so T-App cannot apply it, and the term has no type at all.

Example 5 (A typing derivation)**.** $\mathtt{f{:}b} \to \mathtt{b} \vdash (\lambda \mathtt{x{:}b.\ f\ x}) : \mathtt{b} \to \mathtt{b}$.

Example 6 (An ill-typed term)**.** $\neg(\exists \tau,\ \vdash () () : \tau)$.

1.4 Reduction

Reduction is call by value. A premise that a term is a value is left implicit: the value is written v instead. Each rule is labelled from its constructor's name, and can be referred to: rule (⟶-App Left) reduces the function of an application.

$$\frac{}{(\lambda x{:}\tau.\ e)\ v \longrightarrow e[v/x]}\ (\longrightarrow\text{-Beta}) \qquad \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ (\longrightarrow\text{-App Left})$$

$$\frac{e_2 \longrightarrow e_2'}{v\ e_2 \longrightarrow v\ e_2'}\ (\longrightarrow\text{-App Right})$$

↗

The identity applied to the unit value β-reduces at once. When the argument is itself an application, rule (⟶-App Right) reduces it first: the outer β-reduction waits until the argument is a value.

Example 7 (A β-reduction)**.** $(\lambda \mathtt{x{:}b.\ x})\ () \longrightarrow ()$.

Example 8 (Call by value)**.** $(\lambda \mathtt{x{:}b.\ x})\ ((\lambda \mathtt{y{:}b.\ y})\ ()) \longrightarrow (\lambda \mathtt{x{:}b.\ x})\ ()$.

1.5 Results

A result can be generated, as a sentence; its key can be a title, declared in the notation module:

Theorem 1 (Swapping is involutive)**.** *Let $a, b \in$ Name and $e \in$ Term. Then $(a\ b) \cdot ((a\ b) \cdot e) = e$.*

Theorem 2 (Values are final)**.** *Let $v \in$ Val and $e \in$ Term. Then $\neg(v \longrightarrow e)$.*

Or as a rule, from a theorem whose hypotheses become its premises:

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}$$

↗

Or the prose can be written by hand and checked: the unit value has only the base type, that is, if $\Gamma \vdash () : \tau$ then $\tau = \mathtt{b}$↗. Should the Lean statement of `unit_type` change, `imprimatur check` reports this sentence.

Instances of these results are examples too. Swapping renames the binder along with the free occurrence, and swapping again (Theorem 1) gives the term back. The unit value's only type is $\mathtt{b}$, so it has no function type.

Example 9 (Swapping a binder)**.** $(\mathtt{x}\ \mathtt{y}) \cdot (\lambda \mathtt{x}{:}\mathtt{b}.\, \mathtt{y}) = (\lambda \mathtt{y}{:}\mathtt{b}.\, \mathtt{x})$.

Example 10 (The unit value is not a function)**.** $\neg(\vdash () : \mathtt{b} \to \mathtt{b})$.

1.6 Type safety

The substitution lemma is proved as Pitts's paper suggests. Typing is equivariant: swapping two names throughout a term and its context preserves its type.

Theorem 3 (Equivariance)**.** *Let* $a, b \in \mathit{Name}$, $\Gamma \in \mathit{Context}$, $e \in \mathit{Term}$ *and* $\tau \in \mathit{Type}$. *Assume* $\Gamma \vdash e : \tau$. *Then* $(a\ b) \cdot \Gamma \vdash (a\ b) \cdot e : \tau$.

Applied to Example 5, swapping `f` and `x` throughout the context and the term, the function is now called `x` and the bound variable `f`.

Example 11 (Equivariance at work)**.** $\mathtt{x}{:}\mathtt{b} \to \mathtt{b} \vdash (\lambda \mathtt{f}{:}\mathtt{b}.\, \mathtt{x}\ \mathtt{f}) : \mathtt{b} \to \mathtt{b}$.

Typing also depends on the context only at the term's free names, which gives weakening and shadowing at once↗. So substitution preserves types: if $\Gamma, x{:}\tau_1 \vdash e : \tau$ and $\Gamma \vdash u : \tau_1$, then $\Gamma \vdash e[u/x] : \tau$. At an abstraction, substitution renames the binder to a fresh name by swapping, and equivariance carries the typing of its body across the renaming; the induction is on the size of e, which swapping preserves↗. Type safety follows.

Theorem 4 (Preservation)**.** *Let* $e, e' \in \mathit{Term}$, $\Gamma \in \mathit{Context}$ *and* $\tau \in \mathit{Type}$. *Assume* $e \longrightarrow e'$ *and* $\Gamma \vdash e : \tau$. *Then* $\Gamma \vdash e' : \tau$.

A closed, well-typed application of a function to the identity reduces by rule ($\longrightarrow$-Beta), and the result has the same type.

Example 12 (A step preserves the type)**.** *(1)* $\vdash (\lambda \mathtt{f}{:}\mathtt{b} \to \mathtt{b}.\, \mathtt{f}\ ())\ (\lambda \mathtt{x}{:}\mathtt{b}.\, \mathtt{x}) : \mathtt{b}$, *(2)* $(\lambda \mathtt{f}{:}\mathtt{b} \to \mathtt{b}.\, \mathtt{f}\ ())\ (\lambda \mathtt{x}{:}\mathtt{b}.\, \mathtt{x}) \longrightarrow (\lambda \mathtt{x}{:}\mathtt{b}.\, \mathtt{x})\ ()$ *and (3)* $\vdash (\lambda \mathtt{x}{:}\mathtt{b}.\, \mathtt{x})\ () : \mathtt{b}$.

Theorem 5 (Progress)**.** *Let* $e \in \mathit{Term}$ *and* $\tau \in \mathit{Type}$. *Assume* $\vdash e : \tau$. *Then* $e \in \mathit{Val} \vee (\exists e',\ e \longrightarrow e')$.

Progress needs the term to be closed. An open term can be well typed and yet stuck: neither a value nor able to reduce, since no rule applies to a variable.

Example 13 (An open term may be stuck)**.** *(1)* $\mathtt{x}{:}\mathtt{b} \to \mathtt{b} \vdash \mathtt{x}\ () : \mathtt{b}$, *(2)* $\mathtt{x}\ () \notin \mathit{Val}$ *and (3)* $\forall e : \mathit{Term},\ \neg(\mathtt{x}\ () \longrightarrow e)$.

References

[Pitts(2006)] Andrew M. Pitts. Alpha-structural recursion and induction. *Journal of the ACM*, 53(3):459–506, 2006. https://doi.org/10.1145/1147954.1147961.

A Index of notation

Metavariables range over the types below; a variable with a subscript or a prime ranges over the same type. imprimatur generates the table from the categories the notation declares, and checks that every variable printed with a metavariable has its type.

Variable	*Type*		*Shown in*
τ	*Type*	types	*Ty*, p. 1
e	*Term*	terms, with names	*Tm*, p. 1
v	*Val*	Values	*Step*

A variable with a prime or a subscript (τ', τ_1) ranges over the same type.

The notation, grouped by where it is defined: the paper's own (*Stlc*) and Lean's. Each is written as the paper writes it, with the Lean constant it stands for.

Lean		
$\Gamma, x{:}\tau$	`List.cons`	The list whose first element is `head`, where `tail` is the rest of the list
Stlc		
$e =_\alpha e'$	`Alpha`	α-equivalence
$x{:}\tau$	`Atom`	Names of variables
Name	`Atom`	Names of variables
Context	`Ctx`	Contexts: names with their types, the most recent first
$(a\ b) \cdot \Gamma$	`Ctx.swap`	Swapping two names in a context
$\Gamma \vdash e : \tau$	`HasType`	Typing
$\vdash e : \tau$	`HasType`	Typing
$v \in \mathit{Val}$	`IsValue`	Values
$v \notin \mathit{Val}$	`IsValue`	Values
Val	`IsValue`	Values
`f`	`Names.f`	The name `f`
`x`	`Names.x`	The name `x`
`y`	`Names.y`	The name `y`
`z`	`Names.z`	The name `z`, fresh for `x` and `y`
$e \longrightarrow e'$	`Step`	Small-step reduction, call by value
$e_1\ e_2$	`Tm.app`	an application
$c \# a, b, e, e'$	`Tm.atoms`	The names that occur in a term, bound or free
$a \notin \mathit{fv}(e)$	`Tm.fv`	The free names of a term
$\mathit{fv}(e)$	`Tm.fv`	The free names of a term
$\lambda x{:}\tau.\, e$	`Tm.lam`	an abstraction, binding x in e
$(a\ b) \cdot e$	`Tm.swap`	Swapping two names throughout a term: bound, binding and free occurrences alike

$()$	`Tm.unit`	the unit value
x	`Tm.var`	a variable
$\tau_1 \to \tau_2$	`Ty.arrow`	functions
b	`Ty.base`	the base type
$\Gamma(x) = \tau$	`lookup`	The type of a name in a context
$e[u/x]$	`subst`	Capture-avoiding substitution of `u` for `x` in `e`
$(a\ b)(c)$	`swapAtom`	Swapping two names: `a` becomes `b`, `b` becomes `a`, and other names are unchanged