# The $\theta$-Calculus: Unifying Type Systems through Layered Computation

Draft — Work in Progress

Orti Bazar
orti.bazar@gmail.com

February 2026

**Abstract**

We present the $\theta$-calculus, an extension of the $\lambda$-calculus with two operations—layering ($t{:}t$) and unlayering ($\theta.\, t$)—that provides a unified mechanism for expressing the full spectrum of type system features. We define the syntax, term classification, evaluation semantics, and separation operation, then show that Simply Typed Lambda Calculus, System F polymorphism, and dependent types all arise as configurations of the same mechanism, differing only in the choice of guard combinator.

# 1 Introduction

Most language ecosystems treat the type system as a fundamentally different mechanism from the term language. TypeScript adds types to JavaScript. Mypy adds types to Python. Rust's borrow checker is a separate phase from evaluation. In each case, the type system requires its own distinct formalism, its own rules, its own machinery.

The $\theta$-calculus rejects that dichotomy. It extends the untyped lambda calculus with just two operations—*layering* ($t{:}t$) and *unlayering* ($\theta.\, t$)—and shows that type checking, polymorphism, and dependent types all emerge from the same computational substrate. The layering operation says "this term exists simultaneously at multiple levels," and the *separation* operation decomposes a multi-layer program into its individual layers.

This yields two key results:

- **Unified type system hierarchy.** Simply Typed Lambda Calculus, System F, and dependent types are all encoded using the same mechanism. They are not different type systems—they are different configurations of the same system, differing only in the choice of guard combinator.
- **Dependent types become natural.** When types are terms in another layer, dependent types—where types depend on values—are not a special feature requiring a proof assistant. They are terms that reference values across layers.

The $\theta$-calculus is named in the tradition of the $\lambda$-calculus. Where $\lambda$ denotes abstraction, $\theta$ denotes unlayering—the operation that inspects and decomposes the layered structure of a term. Note: this unlayering operation is useful for dependent types. You will see in the later sections how it is used for solving dependent types.

The remainder of this paper is organized as follows. Section **??** defines the syntax. Section **??** introduces the term classification that drives both evaluation and separation. Sections **??** and **??** present the evaluation and separation semantics. Section **??** demonstrates that STLC, System F, and dependent types are all encodable in the calculus. Section **??** discusses properties of the system. Section **??** concludes. Appendix **??** provides a compact one-page reference of all formal rules.

# 2 Syntax

The $\theta$-calculus extends the untyped lambda calculus with two constructs: layering and unlayering.

**Definition 2.1** (Terms)**.** The set of terms is defined by the following grammar:

| | | |
|---|---|---|
| $t$ ::= | | *terms:* |
| | $x$ | *variable* |
| | $\lambda x.\, t$ | *abstraction* |
| | $t\ t$ | *application* |
| | $t{:}t$ | *layering* |
| | $\theta.\, t$ | *unlayering* |

We adopt the following precedence conventions, from tightest to loosest binding:

1. **Layering** ($t{:}t$) binds tightest: $a\ b{:}c\ d = a\ (b{:}c)\ d$.

2. **Abstraction and unlayering** ($\lambda x.\, t$, $\theta.\, t$) extend as far right as possible: $\lambda x.\, a\ b{:}c = \lambda x.\, (a\ (b{:}c))$.

3. **Application** ($t\ t$) binds loosest and is left-associative: $a\ b\ c = (a\ b)\ c$.

Parentheses are used only when overriding the default precedence, e.g. $(\lambda x.\, t)\ v$ to limit the scope of $\lambda$, or $(t_1\ t_2){:}(t_3\ t_4)$ to layer two applications.

The first three forms are standard lambda calculus. The two new forms are:

- **Layering** ($t_1{:}t_2$) constructs a term that exists simultaneously in multiple computational layers. The term $t_1$ belongs to the first layer and $t_2$ to the second.

- **Unlayering** ($\theta.\, t$) inspects and decomposes the layers of a term, producing a selector that provides access to each layer independently.

# 3 Term Classification

Every term is either *single-layer* or *multi-layer*. This distinction is the engine of the entire system—it drives both evaluation and separation.

**Definition 3.1** (Single-layer and multi-layer terms)**.**

$$t = g \mid h$$

$$\begin{array}{rcll} g & ::= & x \mid \lambda x.\, g \mid g\ g \mid \theta.\, t & \textit{single-layer term} \\ h & ::= & \lambda x.\, h \mid g\ h \mid h\ g \mid h\ h \mid t{:}t & \textit{multi-layer term} \end{array}$$

A single-layer term ($g$) is a conventional lambda calculus term—it lives in one layer. A multi-layer term ($h$) involves layering or has at least one multi-layer sub-term. Multi-layer terms are compact representations of code that *means different things at different layers*.

Note that unlayering ($\theta.\, t$) is classified as a single-layer term even when its body $t$ is multi-layer. This is because $\theta$ is an operator that *decomposes* layers; it does not itself inhabit multiple layers.

## 3.1 Single-Layer Sub-classification

Single-layer terms are further classified into values and non-values:

$$g = v \mid r$$

$$\begin{array}{rcll} v & ::= & \lambda x.\, g & \textit{value} \\ r & ::= & x \mid g\ g \mid \theta.\, t & \textit{non-value} \end{array}$$

Values ($v$) are abstractions over single-layer bodies—they are fully evaluated. Non-values ($r$) are variables, applications, or unlayering forms that may reduce further.

## 3.2 Multi-Layer Sub-classification

Multi-layer terms are classified into *separated* and *separable* terms:

$$h = p \mid s$$

$$\begin{array}{rcll} p & ::= & t{:}t & \textit{separated} \\ s & ::= & \lambda x.\, h \mid g\ h \mid h\ g \mid h\ h & \textit{separable} \end{array}$$

A separated term ($p$) is already decomposed into its layers—it is a layering node $t_1{:}t_2$. A separable term ($s$) still needs the separation operation to decompose it. The distinction between $p$ and $s$ determines which separation rules apply.

# 4 Evaluation

The evaluation function $\epsilon[t] \to t$ reduces a term by one step. We adopt the convention:

$$\frac{a}{a'} := \text{“}a \text{ evaluates to } a'\text{”}$$

When evaluation returns the same term, the term is either a value or stuck [**?**]. A closed term is stuck if it is in normal form but not a value. Multi-layer terms ($h$) are stuck under evaluation; they require explicit unlayering via $\theta$ to make progress.

## 4.1 Variable and Abstraction

Variables and abstractions are already in normal form:

$$\frac{\epsilon[x]}{x} \quad \text{(E-Var)} \qquad\qquad \frac{\epsilon[\lambda x.\, g]}{\lambda x.\, g} \quad \text{(E-Abs)}$$

## 4.2 Application

Application of two single-layer terms ($g\ g$) decomposes into three sub-cases: $r\ g \mid v\ r \mid v\ v$.

When the function is a non-value, evaluate it first:

$$\frac{\epsilon[r\ g]}{\epsilon[r]\ g} \quad \text{(E-App1)}$$

When the function is a value but the argument is not, evaluate the argument:

$$\frac{\epsilon[v\ r]}{v\ \epsilon[r]} \quad \text{(E-App2)}$$

When both are values, perform $\beta$-reduction:

$$\frac{\epsilon[(\lambda x.\, g)\ v]}{[x \mapsto v]\, g} \quad \text{(E-Subst)}$$

## 4.3 Unlayering

Unlayering ($\theta.\, t$) decomposes into three sub-cases: $\theta.\, g \mid \theta.\, s \mid \theta.\, t_1{:}t_2$.

If the body is single-layer, return it directly:

$$\frac{\epsilon[\theta.\, g]}{g} \quad \text{(E-Ground)}$$

A single-layer term has no layered structure to decompose, so unlayering simply returns the term itself.

If the body is separable, apply the separation operation:

$$\frac{\epsilon[\theta.\, s]}{\theta.\, \sigma[s]} \quad \text{(E-Separate)}$$

If the body is a layered term, *squash* it into a selector that recursively unlayers each component:

$$\frac{\epsilon[\theta.\, t_1{:}t_2]}{\lambda x.\, x\ (\theta.\, t_2)\ (\theta.\, t_1)} \quad \text{(E-Squash)}$$

The recursive $\theta$ in the squash result ensures that each sub-term is itself unlayered when the selector is applied. This is necessary because $\theta.\, t$ is classified as a single-layer term ($g$); returning raw $t_1$ or $t_2$ without $\theta$ could yield multi-layer terms ($h$), violating the syntactic category. For single-layer sub-terms, the recursion bottoms out at E-Ground, returning the term directly. For multi-layer sub-terms, squash is applied again, peeling off one level of layering at a time. All paths terminate with no stuck cases.

### 4.4 Multi-Layer Terms

Multi-layer terms do not reduce under evaluation—they must be explicitly unlayered:

$$\frac{\epsilon[h]}{h} \quad \text{(E-Layered)}$$

A multi-layer term represents code that means different things at different layers, and it makes no sense to evaluate it as a single computation. The only way to make progress is to wrap it in $\theta$, which triggers separation and decomposition.

## 5 Separation

The separation function $\sigma[s] \to t$ transforms a separable term into a layered term $t_1{:}t_2$ where each component belongs to a single layer. Separation is the key operation of the $\theta$-calculus: it is the formal mechanism by which a multi-layer program is decomposed into independent single-layer computations.

Separation is defined only on separable terms ($s$). Single-layer terms ($g$) need no separation because they already belong to a single layer. Already-separated terms ($p$) need no separation because they are already decomposed. This restricted domain ensures that each application of $\sigma$ moves a term closer to a fully separated form.

### 5.1 Abstraction

Abstraction over a multi-layer body ($\lambda x.\, h$) has two sub-cases: $\lambda x.\, s \mid \lambda x.\, p$.

If the body is separable, recurse into the body:

$$\frac{\sigma[\lambda x.\, s]}{\lambda x.\, \sigma[s]} \quad \text{(S-Abs)}$$

If the body is already separated, distribute the abstraction over the layers:

$$\frac{\sigma[\lambda x.\, t_1{:}t_2]}{(\lambda x.\, t_1){:}(\lambda x.\, t_2)} \quad \text{(S-Split)}$$

A function whose body spans two layers becomes two functions, one per layer. This is the distribution rule: it pushes abstraction through layering.

### 5.2 Cloning

When a single-layer term interacts with a multi-layer term in an application, the single-layer term is **cloned** into both layers:

$$\frac{\sigma[g\ h]}{g{:}g\ h}\quad \text{(S-Clone1)} \qquad\qquad \frac{\sigma[h\ g]}{h\ g{:}g}\quad \text{(S-Clone2)}$$

This is how ordinary code participates in multiple layers simultaneously. A single-layer term that appears in one layer of the source is duplicated so that it appears in every layer of the separated output.

### 5.3 Application of Multi-Layer Terms

Application of two multi-layer terms $(h\ h)$ decomposes into: $s\ h \mid p\ s \mid p\ p$.

If the function is separable, separate it first:

$$\frac{\sigma[s\ h]}{\sigma[s]\ h}\quad \text{(S-App1)}$$

If the function is separated but the argument is separable, separate the argument:

$$\frac{\sigma[p\ s]}{p\ \sigma[s]}\quad \text{(S-App2)}$$

If both are separated, distribute the application over the layers:

$$\frac{\sigma[t_1{:}t_2\ t_3{:}t_4]}{(t_1\ t_3){:}(t_2\ t_4)}\quad \text{(S-Zip)}$$

Applying a layered function to a layered argument produces a layered result, where each layer's function is applied to that layer's argument.

## 6 Encoding Type System Features

One of the most significant results of the $\theta$-calculus is that multiple type system features emerge from the same layering mechanism. This section defines the combinators or psuedocodes used in the encodings and demonstrates each correspondence.

Since lambda calculus is turing complete, it can simulate memory operations, conditional expressions, and error-signalling keywords. For the sake of simplicity, we use pseudocode to show how type system encodings are constructed.

here are some definitions that are used in the encodings:

here are list of predefined combinators:

$$\begin{aligned} I &:= \lambda x.\, x \\ B &:= \lambda f.\, \lambda g.\, \lambda x.\, f\ (g\ x) \end{aligned}$$

All psuedocode that is not a lambda calculus term are inside curly braces.

Note that the evaluation order in the appendix is call-by-value. Note: Below examples we work on only one or two layers. In reality, you can build a multiple nested layer system.

We all need to raise a type error when the type check fails. so we use the keyword *TypeError* to signal a type error.

we will use literal values for integer and string. Example: 1, 2, ‘hello’, ‘Int’. Types are actually a data behind the scene, so we will use strings as types.

all camelcase symbols are predefined functions. You can imagine them as closed term, like this:

$$(\lambda Print.Print \text{ ‘Hello World!’})\ NativePrintImplementation$$

is for simplicity we will use open term notation like this (both are same)

$$Print \text{ ‘Hello World!’}$$

Let’s see how theta expansion works. we want first run the high layer, then the low layer.

$$(\theta.(\ Print \text{ ‘Evaluate’}):(\ Print \text{ ‘Typecheck’}))(\lambda a.\lambda b.a)$$

$G$ is a type guard that takes two arguments: $a$, the formal parameter type declared in the function signature, and $b$, the type of the argument actually passed. $G$ returns $a$—the declared type—making the type check independent of the argument’s type.

$$G \ := \ \lambda a.\, \lambda b.\ \text{if } Equal(a, b) \text{ then } a \text{ else } TypeError$$

Let’s create a Add function.

$$\begin{aligned} AddImpl &:= \lambda a.\, \lambda b.\, \{a + b\} \\ AddGuard &:= \lambda a.\, \lambda b.\, G \text{ ‘Int’ } a; G \text{ ‘Int’ } b \\ Add &:= AddImpl{:}AddGuard \end{aligned}$$

; is a sequence operator. see TAPL book’s sequence operator section.

$$Add\ 1\ 2$$

this will be parsed as:

$$Add\ 1{:}\text{‘Int’}\ 2{:}\text{‘Int’}$$

Expanding $Add$:

$$= (AddImpl{:}(\lambda a.\, \lambda b.\, G \text{ ‘Int’ } a;\ G \text{ ‘Int’ } b))\ (1{:}\text{‘Int’})\ (2{:}\text{‘Int’})$$

Distribute application over layers ($p\ p$, S-Zip) for the first application:

$$= (AddImpl\ 1){:}((\lambda a.\, \lambda b.\, G \text{ ‘Int’ } a;\ G \text{ ‘Int’ } b) \text{ ‘Int’})\ (2{:}\text{‘Int’})$$

Distribute application over layers ($p\ p$, S-Zip) for the second application:

$$= ((AddImpl\ 1)\ 2){:}(((\lambda a.\, \lambda b.\, G \text{ ‘Int’ } a;\ G \text{ ‘Int’ } b) \text{ ‘Int’}) \text{ ‘Int’})$$

The term is now fully separated into two layers. $\beta$-reduce the type layer; $G$ ‘Int’ ‘Int’ = ‘Int’ at each step (both type checks pass):

$$= ((AddImpl\ 1)\ 2){:}\text{‘Int’}$$

Type layer is a value. $\beta$-reduce the evaluation layer; $AddImpl\ 1\ 2 = \{1 + 2\} = 3$:

$$= 3{:}\text{‘Int’}$$

The term is first fully separated into an evaluation layer and a type-checking layer. The type-checking layer reduces to ‘Int’, confirming both arguments are integers. Only then does the evaluation layer compute $AddImpl\ 1\ 2 = 3$.

## 6.1 Simply Typed Lambda Calculus

In the standard Simply Typed Lambda Calculus (STLC), a typed abstraction is written $\lambda x{:}T.\, t$. In the $\theta$-calculus, this is encoded as:

$$\lambda x{:}T.\, t \quad \Longleftrightarrow \quad B\ (\lambda x.\, t)\ (I{:}(G\ T))$$

The layering $I{:}(G\ T)$ creates a two-layer term: the identity function at the evaluation layer (pass the argument through unchanged) and a type guard at the type-checking layer (check that the argument type is equal to $T$ and return $T$). The composition combinator $B$ threads the argument through both layers.

### Worked example

Consider the STLC term
$(\lambda x{:}Int.\, x + 1)\ 2$. In the $\theta$-calculus:

$$(\lambda x{:}Int.\, x + 1)\ 2 = (B\ (\lambda x.\, x + (1{:}Int))\ (I{:}(G\ Int)))\ (2{:}Int)$$

By $B\ f\ g\ x = f\ (g\ x)$:

$$= (\lambda x.\, x + (1{:}Int))\ ((I{:}(G\ Int))\ (2{:}Int))$$

Distribute application over layers ($p\ p$, S-Zip):

$$= (\lambda x.\, x + (1{:}Int))\ ((I\ 2){:}((G\ Int)\ Int))$$

$I\ 2 = 2$, and $(G\ Int)\ Int = Int$ (since $Int <: Int$):

$$= (\lambda x.\, x + (1{:}Int))\ (2{:}Int)$$

Separate $(\lambda x.\, x + (1{:}Int))$ via S-Split:

$$= ((\lambda x.\, x + 1){:}(\lambda x.\, x + Int))\ (2{:}Int)$$

Distribute application over layers ($p\ p$, S-Zip):

$$= ((\lambda x.\, x + 1)\ 2){:}((\lambda x.\, x + Int)\ Int)$$

$$= (2 + 1){:}(Int + Int) = 3{:}Int$$

The evaluation layer computes 3 and the type-checking layer computes $Int$, confirming that the result is an integer.

## 6.2 System F (Polymorphism)

In System F, the polymorphic identity is $id = \lambda X.\, \lambda x{:}X.\, x$. In the $\theta$-calculus:

$$id = \lambda X.\, B\ (\lambda x.\, x)\ (I{:}(G\ X))$$

Polymorphism is not a separate feature. It arises because $X$ is a variable that can be bound to any type. The same abstraction mechanism that works for terms works for types, because in the $\theta$-calculus types *are* terms in another layer.

### 6.3 Dependent Types

For dependent types, we introduce a type function $T_D = \lambda x.\, t_d$ (a type that depends on a value) and a dependent guard combinator:

$$G_D \coloneqq \lambda a.\, \lambda b.\, \lambda x.\, G_{a|b|D}\ (a\ x)\ (b\ x)$$

$G_D$ is a delayed guard: when type checking depends on a value that is not yet available, we wrap the type check in an abstraction ($\lambda x$). The check is then executed later, once the value is supplied.

A dependent typed abstraction $\lambda x{:}T_D.\, t$ is encoded as:

$$B\ (\lambda x.\, t)\ (I{:}(G_D\ T_D))$$

Since types are terms, a type that depends on a value is simply a function from values to types—no special machinery is required.

### 6.4 Summary

All four type system features share the same structure:

$$B\ (\lambda x.\, t)\ (I{:}(\mathcal{G}\ T))$$

where $\mathcal{G} \in \{G,\, G_D\}$. The layering mechanism is identical; only the guard combinator varies:

| **Feature** | **Guard** | **Returns** |
|---|---|---|
| STLC | $G$ | annotation type $a$ |
| System F | $G$ (with type variable) | annotation type $a$ |
| Dependent | $G_D$ | delayed check |

## 7 Discussion

### 7.1 Conservative Extension

The $\theta$-calculus is a conservative extension of the untyped lambda calculus. Any lambda calculus term is a valid $\theta$-calculus term (classified as single-layer, $g$), and the evaluation rules for variables, abstractions, and applications are identical. The new constructs—layering and unlayering—provide additional expressive power without altering the behavior of existing terms.

### 7.2 Totality of Unlayering

Unlayering ($\theta$) is total: every sub-case of the E-Ground, E-Separate, and E-Squash rules produces a new term. There are no stuck cases in unlayering. For single-layer bodies, E-Ground returns the term directly. For separable bodies, E-Separate delegates to $\sigma$ and recurs. For layered bodies, E-Squash produces a selector with recursive $\theta$ applications that bottom out at E-Ground. This guarantees that any term can be decomposed into its individual layers.

### 7.3 The Role of the Guard Combinator

The encodings in Section **??** reveal that the guard combinator is the single point of variation across type system features. The layering $I{:}(\mathcal{G}\ T)$ is the uniform mechanism; $\mathcal{G}$ alone determines whether the system behaves as STLC ($G$) or handles dependent types ($G_D$). This suggests that the space of type systems is not a taxonomy of distinct formalisms but a parameter space over a single architecture.

### 7.4 Separation Domain

Separation ($\sigma$) is defined only on separable terms ($s$), excluding single-layer terms ($g$) and already-separated terms ($p$). This restricted domain is not a limitation but a design choice: $g$ and $p$ are already in their final separated form, so applying $\sigma$ to them would be a no-op. The restricted domain ensures that each application of $\sigma$ makes meaningful progress toward full separation.

## 8 Related Work

The goal of unifying type systems within a single framework has a long history. Barendregt's *lambda cube* [**?**] organises eight typed lambda calculi along three axes—polymorphism, type operators, and dependent types—and *Pure Type Systems* (PTS) [**?**] generalise the cube by parameterising over an arbitrary set of sorts and axioms. The $\theta$-calculus shares the unification goal but differs in mechanism: PTS varies the sort structure, whereas the $\theta$-calculus keeps a single layering primitive and varies only the guard combinator.

The *Calculus of Constructions* [**?**], at the corner of the lambda cube, also treats types as terms. In CoC, types and terms inhabit a shared universe governed by sort rules; in the $\theta$-calculus, types are literally terms in another layer of the same program, separated by a first-class operation ($\sigma$) rather than by universe hierarchies.

*Multi-stage programming* [**?**] and *two-level type theory* [**?**] separate computation into stages or levels. Davies and Pfenning use modal operators ($\Box$, $\Diamond$) to distinguish code that is closed from code that may contain free variables, enabling safe cross-stage evaluation. Kovács's two-level type theory cleanly separates a meta-level from an object-level for staged compilation. The $\theta$-calculus is related in that a single term can mean different things at different layers, but it does not use modal operators; instead, layering and unlayering are symmetric structural operations that apply uniformly.

*Layered Modal Type Theory* [**?**] is perhaps the closest in terminology: it defines a two-layer system where Layer 0 captures syntactic code and Layer 1 extends it with contextual modal types for intensional analysis. The $\theta$-calculus differs in that layers are not fixed at two, the layering construct is a general term former rather than a modal operator, and the separation operation decomposes an arbitrary number of layers mechanically.

## 9 Conclusion

The $\theta$-calculus demonstrates that a single mechanism—layered computation with separation—is sufficient to express the full spectrum of type system features. Simply Typed Lambda Calculus, System F, and dependent types are not distinct formalisms; they are configurations of the same system, differing only in the choice of guard combinator ($G$ or $G_D$).

The separation rules provide a formal mechanism for decomposing a multi-layer program into independent single-layer computations. The cloning rules ensure that ordinary code participates

in every layer, while the distribution rules ensure that layered structures are correctly projected. Together, these rules define a complete and deterministic procedure for deriving separate computations—one per layer—from a single source program.

*Named in the tradition of the λ-calculus. Inspired by Benjamin C. Pierce's* ***Types and Programming Languages*** *[**?**].*

## References


[1] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[2] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[3] H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992.

[4] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[5] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2–3):95–120, 1988.

[6] R. Davies and F. Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.

[7] A. Kovács. Staged compilation with two-level type theory. *Proceedings of the ACM on Programming Languages*, 6(ICFP):1–30, 2022.

[8] J. Jang, S. Pickard, and B. Pientka. Layered modal type theory. In *European Symposium on Programming (ESOP)*, pages 60–89. Springer, 2024.

# A $\theta$-Calculus Formal Rules

| **Syntax** | | |
|---|---|---|
| $t ::=$ | | *terms* |
| | $x$ | *variable* |
| | $\lambda x.\, t$ | *abstraction* |
| | $t\ t$ | *application* |
| | $t{:}t$ | *layering* |
| | $\theta.\, t$ | *unlayering* |

| **Meta-variables** | | $t = g \mid h$ |
|---|---|---|
| $g ::=$ | $x \mid \lambda x.\, g \mid g\ g \mid \theta.\, t$ | *single layer* |
| $h ::=$ | $\lambda x.\, h \mid g\ h \mid h\ g \mid h\ h \mid t{:}t$ | *multi layer* |
| **Single Layer** | | $g = v \mid r$ |
| $v ::=$ | $\lambda x.\, g$ | *value* |
| $r ::=$ | $x \mid g\ g \mid \theta.\, t$ | *non-value* |
| **Multi Layer** | | $h = p \mid s$ |
| $p ::=$ | $t{:}t$ | *separated* |
| $s ::=$ | $\lambda x.\, h \mid g\ h \mid h\ g \mid h\ h$ | *separable* |

| **Evaluation** | | $\epsilon[t] \to t$ |
|---|---|---|
| $x$ | $\dfrac{\epsilon[x]}{x}$ | E-Var |
| $\lambda x.\, g$ | $\dfrac{\epsilon[\lambda x.\, g]}{\lambda x.\, g}$ | E-Abs |
| $g\ g$ | $r\ g \mid v\ r \mid v\ v$ | |
| | $\dfrac{\epsilon[r\ g]}{\epsilon[r]\ g}$ | E-App1 |
| | $\dfrac{\epsilon[v\ r]}{v\ \epsilon[r]}$ | E-App2 |
| | $\dfrac{\epsilon[(\lambda x.\, g)\ v]}{[x \mapsto v]g}$ | E-Subst |
| $\theta.\, t$ | $\theta.\, g \mid \theta.\, s \mid \theta.\, t{:}t$ | |
| | $\dfrac{\epsilon[\theta.\, g]}{g}$ | E-Ground |
| | $\dfrac{\epsilon[\theta.\, s]}{\theta.\, \sigma[s]}$ | E-Separate |
| | $\dfrac{\epsilon[\theta.\, t_1{:}t_2]}{\lambda x.\ x\ (\theta.\, t_2)\ (\theta.\, t_1)}$ | E-Squash |
| $h$ | $\dfrac{\epsilon[h]}{h}$ | E-Layered |

| **Separation** | | $\sigma[s] \to t$ |
|---|---|---|
| $\lambda x.\, h$ | $\lambda x.\, s \mid \lambda x.\, p$ | |
| | $\dfrac{\sigma[\lambda x.\, s]}{\lambda x.\, \sigma[s]}$ | S-Abs |
| | $\dfrac{\sigma[\lambda x.\, t_1{:}t_2]}{(\lambda x.\, t_1){:}(\lambda x.\, t_2)}$ | S-Split |
| $g\ h$ | $\dfrac{\sigma[g\ h]}{g{:}g\ h}$ | S-Clone1 |
| $h\ g$ | $\dfrac{\sigma[h\ g]}{h\ g{:}g}$ | S-Clone2 |
| $h\ h$ | $s\ h \mid p\ s \mid p\ p$ | |
| | $\dfrac{\sigma[s\ h]}{\sigma[s]\ h}$ | S-App1 |
| | $\dfrac{\sigma[p\ s]}{p\ \sigma[s]}$ | S-App2 |
| | $\dfrac{\sigma[t_1{:}t_2\ t_3{:}t_4]}{(t_1\ t_3){:}(t_2\ t_4)}$ | S-Zip |