A Dependent Type Theory with Strictly Associative and Unital Sigma Types

MWPLS '26

Kelton OBrien & Carlo Angiuli

November 9, 2026

A 10,000 foot view of (Martin-Löf) dependent type theory

- “Just” Simply typed lambda calculus plus some new types

A 10,000 foot view of (Martin-Löf) dependent type theory

- "Just" Simply typed lambda calculus plus some new types
- A hierarchy of universe types $\mathcal{U}_i$, whose terms are themselves types, so e.g $\mathbb{N} : \mathcal{U}_0$.

A 10,000 foot view of (Martin-Löf) dependent type theory

- "Just" Simply typed lambda calculus plus some new types
- A hierarchy of universe types $\mathcal{U}_i$, whose terms are themselves types, so e.g $\mathbb{N} : \mathcal{U}_0$.
- Replace the usual function type $A \to B$ with a dependent function type $\Pi_{a:A} B(a)$, where the output type can *depend* on what the input is.

A 10,000 foot view of (Martin-Löf) dependent type theory

- "Just" Simply typed lambda calculus plus some new types
- A hierarchy of universe types $\mathcal{U}_i$, whose terms are themselves types, so e.g $\mathbb{N} : \mathcal{U}_0$.
- Replace the usual function type $A \to B$ with a dependent function type $\Pi_{a:A} B(a)$, where the output type can *depend* on what the input is.
- Replace the usual pair type $A \times B$ with a dependent type $\Sigma_{a:A} B(a)$ where the type of the second element in the pair can *depend* on the first

The timeless battle between Σ and records

- Real proof assistants also have some notion of *record*

The timeless battle between Σ and records

- Real proof assistants also have some notion of *record*
- We want to use these a lot of the time: they're more efficient and easier to work with

The timeless battle between Σ and records

- Real proof assistants also have some notion of *record*
- We want to use these a lot of the time: they're more efficient and easier to work with
- But we don't get some of the key benefits of sigma types: no general theorems about records

A concrete example: equality issues

record $rec : \mathcal{U}$
  $foo : A$
  $bar : B$
  $baz : C$

A concrete example: equality issues

record $rec : \mathcal{U}$
  $foo : A$
  $bar : B$
  $baz : C$

$rec' : \mathcal{U}$
$rec' = \Sigma_{foo:A}(\Sigma_{bar:B} C)$

A concrete example: equality issues

record $rec : \mathcal{U}$
$foo : A$
$bar : B$
$baz : C$

$rec' : \mathcal{U}$
$rec' = \Sigma_{foo:A}(\Sigma_{bar:B} C)$
$bar : rec' \to B$
$bar\ m = \mathtt{fst}(\mathtt{snd}(m))$

A concrete example: equality issues

- Clearly the record is nicer in some ways, but say we want to prove two *rec*s are equal

A concrete example: equality issues

- Clearly the record is nicer in some ways, but say we want to prove two *rec*s are equal
- Easy enough to characterize for Σ:

A concrete example: equality issues

- Clearly the record is nicer in some ways, but say we want to prove two *rec*s are equal
- Easy enough to characterize for Σ:
- $\forall p, p' : \Sigma_{a:A} B, p = p' \cong (\mathbf{fst}(p) = \mathbf{fst}(p') \times \mathbf{snd}(p) = \mathbf{snd}(p'))$

A concrete example: equality issues

- Clearly the record is nicer in some ways, but say we want to prove two *rec*s are equal
- Easy enough to characterize for Σ:
- $\forall p, p' : \Sigma_{a:A} B, p = p' \cong (\mathbf{fst}(p) = \mathbf{fst}(p') \times \mathbf{snd}(p) = \mathbf{snd}(p'))$
- No such luck in the record case!

A concrete example: equality issues

- Clearly the record is nicer in some ways, but say we want to prove two *rec*s are equal
- Easy enough to characterize for Σ:
- $\forall p, p' : \Sigma_{a:A} B, p = p' \cong (\mathbf{fst}(p) = \mathbf{fst}(p') \times \mathbf{snd}(p) = \mathbf{snd}(p'))$
- No such luck in the record case!
- Our approach: try to give Σ some of the advantages of records, starting with something like an N-ary Σ type.

A concrete example: equality issues

- Clearly the record is nicer in some ways, but say we want to prove two *rec*s are equal
- Easy enough to characterize for Σ:
- $\forall p, p' : \Sigma_{a:A} B, p = p' \cong (\mathbf{fst}(p) = \mathbf{fst}(p') \times \mathbf{snd}(p) = \mathbf{snd}(p'))$
- No such luck in the record case!
- Our approach: try to give Σ some of the advantages of records, starting with something like an N-ary Σ type.
- One way towards that same goal: associative and unital Σ.

What does “Associative Σ” mean?

- In type theory, we consider type/terms up to *judgmental equality*, usually $\beta(-\eta)$ equality.

What does "Associative Σ" mean?

- In type theory, we consider type/terms up to *judgmental equality*, usually $\beta(-\eta)$ equality.
- Even if equal terms are syntactically different, we still need to treat them the same during typechecking

What does "Associative Σ" mean?

- In type theory, we consider type/terms up to *judgmental equality*, usually $\beta(-\eta)$ equality.
- Even if equal terms are syntactically different, we still need to treat them the same during typechecking
- $[1, 2] : \mathbf{Vec}\, 2$ and $[1, 2] : \mathbf{Vec}\, (1 + 1)$, because $\Gamma \vdash \mathbf{Vec}\, 2 = \mathbf{Vec}\, (1 + 1)$ type

What does "Associative Σ" mean?

- In type theory, we consider type/terms up to *judgmental equality*, usually $\beta(-\eta)$ equality.
- Even if equal terms are syntactically different, we still need to treat them the same during typechecking
- $[1, 2] : \mathbf{Vec}\, 2$ and $[1, 2] : \mathbf{Vec}\, (1 + 1)$, because $\Gamma \vdash \mathbf{Vec}\, 2 = \mathbf{Vec}\, (1 + 1)$ type
- Add $\Gamma \vdash (A \times B) \times C = A \times (B \times C)$ type to the theory

What goes wrong: typechecking

- *Bidirectional* typechecking: terms *synthesize* or *check*

What goes wrong: typechecking

- *Bidirectional* typechecking: terms *synthesize* or *check*
- Usually: (a, b) checks against C when...

What goes wrong: typechecking

- *Bidirectional* typechecking: terms *synthesize* or *check*
- Usually: (a, b) checks against C when...
  - $C = A \times B$

What goes wrong: typechecking

- *Bidirectional* typechecking: terms *synthesize* or *check*
- Usually: (a, b) checks against C when...
  - $C = A \times B$
  - a checks against A

What goes wrong: typechecking

- *Bidirectional* typechecking: terms *synthesize* or *check*
- Usually: (a, b) checks against C when...
  - $C = A \times B$
  - a checks against A
  - b checks against B

What goes wrong: typechecking

- *Bidirectional* typechecking: terms *synthesize* or *check*
- Usually: (a, b) checks against C when...
  - $C = A \times B$ ($A \times B$ **needs to be unique if it exists!**)
  - a checks against A
  - b checks against B

What goes wrong: typechecking

- *Bidirectional* typechecking: terms *synthesize* or *check*
- Usually: (a, b) checks against C when...
  - $C = A \times B$ ($A \times B$ **needs to be unique if it exists!**)
  - a checks against A
  - b checks against B
  - But now $A \times (B \times D) = (A \times B) \times D$. How do we know what to check against?

What goes wrong: typechecking

- *Bidirectional* typechecking: terms *synthesize* or *check*
- Usually: (a, b) checks against C when...
  - $C = A \times B$ ($A \times B$ **needs to be unique if it exists!**)
  - a checks against A
  - b checks against B
  - But now $A \times (B \times D) = (A \times B) \times D$. How do we know what to check against?
- Usually: $\mathbf{fst}(p)$ synthesizes to a type A when ...

What goes wrong: typechecking

- *Bidirectional* typechecking: terms *synthesize* or *check*
- Usually: (a, b) checks against C when...
  - $C = A \times B$ ($A \times B$ **needs to be unique if it exists!**)
  - a checks against A
  - b checks against B
  - But now $A \times (B \times D) = (A \times B) \times D$. How do we know what to check against?
- Usually: $\mathbf{fst}(p)$ synthesizes to a type A when ...
  - p synthesizes to a type C

What goes wrong: typechecking

- *Bidirectional* typechecking: terms *synthesize* or *check*
- Usually: (a, b) checks against C when...
  - $C = A \times B$ ($A \times B$ **needs to be unique if it exists!**)
  - a checks against A
  - b checks against B
  - But now $A \times (B \times D) = (A \times B) \times D$. How do we know what to check against?
- Usually: $\mathbf{fst}(p)$ synthesizes to a type A when ...
  - p synthesizes to a type C
  - $C = A \times B$ for some B

What goes wrong: typechecking

- *Bidirectional* typechecking: terms *synthesize* or *check*
- Usually: (a, b) checks against C when...
  - $C = A \times B$ ($A \times B$ **needs to be unique if it exists!**)
  - a checks against A
  - b checks against B
  - But now $A \times (B \times D) = (A \times B) \times D$. How do we know what to check against?
- Usually: $\mathbf{fst}(p)$ synthesizes to a type A when ...
  - p synthesizes to a type C
  - $C = A \times B$ for some B
  - Same problem: can't invert C!

Checking associative pairs

- Don't have a unique inverse but do (appear to) have some sort of "left-cancellation" property: Can see if A is a prefix of some type C.

Checking associative pairs

- Don't have a unique inverse but do (appear to) have some sort of "left-cancellation" property: Can see if A is a prefix of some type C.
- Can define a function $\texttt{unTimes}(C, A) = B$, where $\Gamma \vdash C = A \times B\,\text{type}$

Checking associative pairs

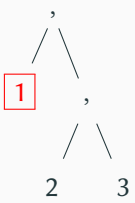
- Don't have a unique inverse but do (appear to) have some sort of "left-cancellation" property: Can see if A is a prefix of some type C.
- Can define a function $\texttt{unTimes}(C, A) = B$, where $\Gamma \vdash C = A \times B\,\text{type}$
- This is enough: we don't need the type values and the syntax to have the same shape

Elaboration Example: (1,(2,3))



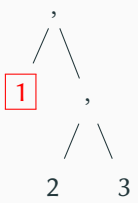
Checking against $(\mathbb{N} \times \mathbb{N}) \times \mathbb{N}$

Elaboration Example: (1,(2,3))



Checking against $(\mathbb{N} \times \mathbb{N}) \times \mathbb{N}$

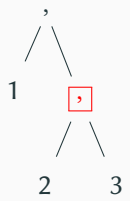
Elaboration Example: (1,(2,3))



Checking against $(\mathbb{N} \times \mathbb{N}) \times \mathbb{N}$

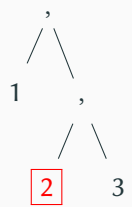
$\texttt{unTimes}((\mathbb{N} \times \mathbb{N}) \times \mathbb{N}, \mathbb{N}) = \mathbb{N} \times \mathbb{N}$

Elaboration Example: (1,(2,3))



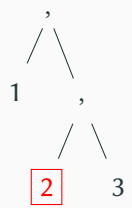
Checking against $\mathbb{N} \times \mathbb{N}$

Elaboration Example: (1,(2,3))



Checking against $\mathbb{N} \times \mathbb{N}$

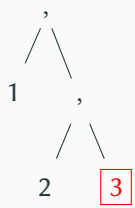
Elaboration Example: (1,(2,3))



Checking against $\mathbb{N} \times \mathbb{N}$

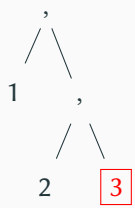
$\texttt{unTimes}(\mathbb{N} \times \mathbb{N}, \mathbb{N}) = \mathbb{N}$

Elaboration Example: (1,(2,3))



Checking against $\mathbb{N}$:

Elaboration Example: (1,(2,3))



Checking against $\mathbb{N}$:
$\texttt{unTimes}(\mathbb{N}, \mathbb{N}) = \textbf{Unit}$

Elaborating eliminators

Let $\Gamma \vdash p : (A \times B) \times C$:

Elaborating eliminators

Let $\Gamma \vdash p : (A \times B) \times C$:

$$\Gamma \vdash \mathbf{fst}(p) : A \times B$$

Elaborating eliminators

Let $\Gamma \vdash p : (A \times B) \times C$:



Elaborating eliminators

Let $\Gamma \vdash p : (A \times B) \times C$:

$\Gamma \vdash \mathbf{fst}(p) : A \times B$?
$\Gamma \vdash \mathbf{fst}(p) : A$?

$\boxed{\Gamma \vdash (A \times B) \times C = A \times (B \times C)\ \mathrm{type}}$

Elaborating eliminators

Let $\Gamma \vdash p : (A \times B) \times C$:

$\Gamma \vdash \mathbf{fst}(p) : A \times B\,?$
$\Gamma \vdash \mathbf{fst}(p) : A?$
$\Gamma \vdash \mathbf{fst}(A, p) : A$

$\boxed{\Gamma \vdash (A \times B) \times C = A \times (B \times C)\ \mathrm{type}}$

Elaborating eliminators

Let $\Gamma \vdash p : (A \times B) \times C$:

$\Gamma \vdash \mathbf{fst}(p) : A \times B\,?$

$\Gamma \vdash \mathbf{fst}(p) : A?$

$\Gamma \vdash \mathbf{fst}(A, p) : A$ ⟵ $\texttt{unTimes}((A \times B) \times C, A) = B \times C$

$\Gamma \vdash (A \times B) \times C = A \times (B \times C)\,\text{type}$

Elaborating eliminators

Let $\Gamma \vdash p : (A \times B) \times C$:

$\Gamma \vdash \mathbf{fst}(p) : A \times B$?
$\Gamma \vdash \mathbf{fst}(p) : A$?
$\Gamma \vdash \mathbf{fst}(A, p) : A$
$\Gamma \vdash \mathbf{snd}(A, p) : B \times C$

$\boxed{\Gamma \vdash (A \times B) \times C = A \times (B \times C)\,\mathsf{type}}$

$\boxed{\mathtt{unTimes}((A \times B) \times C, A) = B \times C}$

What about Σ?

- Focused so far on the non-dependent case, but how do we talk about Σ?

What about Σ?

- Focused so far on the non-dependent case, but how do we talk about Σ?
  - $\Gamma \vdash \Sigma_{a:A}\Sigma_{b:B}C = \Sigma_{(z:\Sigma_{a:A}B)}C$ type ???

What about Σ?

- Focused so far on the non-dependent case, but how do we talk about Σ?
  - $\Gamma \vdash \Sigma_{a:A}\Sigma_{b:B}C = \Sigma_{(z:\Sigma_{a:A}B)}C$ type ???
  - Need $\Gamma.(a : A).(b : B) \vdash C$ type and $\Gamma.(z : \Sigma_{a:A}B) \vdash C$ type...

What about Σ?

- Focused so far on the non-dependent case, but how do we talk about Σ?
  - $\Gamma \vdash \Sigma_{a:A}\Sigma_{b:B}C = \Sigma_{(z:\Sigma_{a:A}B)}C$ type ???
  - Need $\Gamma.(a : A).(b : B) \vdash C$ type and $\Gamma.(z : \Sigma_{a:A}B) \vdash C$ type...
  - $\vdash \Gamma.(a : A).(b : B) = \Gamma.(z : \Sigma_{a:A}B)$ ctx ??????

What about Σ?

- Focused so far on the non-dependent case, but how do we talk about Σ?
  - $\Gamma \vdash \Sigma_{a:A}\Sigma_{b:B}C = \Sigma_{(z:\Sigma_{a:A}B)}C$ type ???
  - Need $\Gamma.(a : A).(b : B) \vdash C$ type and $\Gamma.(z : \Sigma_{a:A}B) \vdash C$ type...
  - $\vdash \Gamma.(a : A).(b : B) = \Gamma.(z : \Sigma_{a:A}B)$ ctx ??????
- Equations make more sense if we switch to a nameless/de bruijn style:

What about Σ?

- Focused so far on the non-dependent case, but how do we talk about Σ?
  - $\Gamma \vdash \Sigma_{a:A}\Sigma_{b:B}C = \Sigma_{(z:\Sigma_{a:A}B)}C$ type ???
  - Need $\Gamma.(a : A).(b : B) \vdash C$ type and $\Gamma.(z : \Sigma_{a:A}B) \vdash C$ type...
  - $\vdash \Gamma.(a : A).(b : B) = \Gamma.(z : \Sigma_{a:A}B)$ ctx ??????
- Equations make more sense if we switch to a nameless/de bruijn style:
- $\Gamma \vdash \Sigma(A, \Sigma(B, C)) = \Sigma(\Sigma(A, B), C)$ type

What about Σ?

- Focused so far on the non-dependent case, but how do we talk about Σ?
  - $\Gamma \vdash \Sigma_{a:A}\Sigma_{b:B}C = \Sigma_{(z:\Sigma_{a:A}B)}C$ type ???
  - Need $\Gamma.(a : A).(b : B) \vdash C$ type and $\Gamma.(z : \Sigma_{a:A}B) \vdash C$ type...
  - $\vdash \Gamma.(a : A).(b : B) = \Gamma.(z : \Sigma_{a:A}B)$ ctx ??????
- Equations make more sense if we switch to a nameless/de bruijn style:
- $\Gamma \vdash \Sigma(A, \Sigma(B, C)) = \Sigma(\Sigma(A, B), C)$ type
- $\vdash \Gamma.A.B = \Gamma.\Sigma(A, B)$ ctx

Problem 2: Variables

Usually $\Gamma.A.B.C \vdash \mathbf{var}_1 : B$

Problem 2: Variables

Usually $\Gamma.A.B.C \vdash \mathbf{var}_1 : B$ $\longleftarrow$ $\vdash \Gamma.A.B.C = \Gamma.A.\Sigma(B, C)\ \mathrm{ctx}$

Problem 2: Variables

Usually $\Gamma.A.B.C \vdash \mathbf{var}_1 : B$ $\longleftarrow$ $\vdash \Gamma.A.B.C = \Gamma.A.\Sigma(B, C)\ \mathrm{ctx}$

$\Gamma.A.B.C \vdash \mathbf{var}_1 : A$?

Problem 2: Variables

Usually $\Gamma.A.B.C \vdash \mathbf{var}_1 : B$ $\longleftarrow$ $\vdash \Gamma.A.B.C = \Gamma.A.\Sigma(B, C)\ \mathrm{ctx}$

$\Gamma.A.B.C \vdash \mathbf{var}_1 : A$?

Could make unambiguous: $\Gamma.A.B.C \vdash \mathbf{var}_{1,\Gamma.A,B} : B$

Elaboration

- Translate from a surface language without these context equations to a core calculus with them

Elaboration

- Translate from a surface language without these context equations to a core calculus with them
- In Type theory: typechecking and translating to a core calculus known as *elaboration*

Elaboration

- Translate from a surface language without these context equations to a core calculus with them
- In Type theory: typechecking and translating to a core calculus known as *elaboration*
- In our elaboration uses a *rigid context*

Elaboration

- Translate from a surface language without these context equations to a core calculus with them
- In Type theory: typechecking and translating to a core calculus known as *elaboration*
- In our elaboration uses a *rigid context*
- $\Theta, (x : A), (y : B), (z : C) \vdash (\mathbf{var}\, y) \Rightarrow B \rightsquigarrow \mathbf{var}_{1,\lfloor \Theta \rfloor . A,B}$

Example

$$\mathit{timesAssoc} : (\mathbb{N} \times \mathbb{N}) \times \mathbb{N} \equiv \mathbb{N} \times (\mathbb{N} \times \mathbb{N})$$
$$= \texttt{refl}\ ((\mathbb{N} \times \mathbb{N}) \times \mathbb{N})$$

Example

$$
\begin{aligned}
&\mathit{timesAssoc} : (\mathbb{N} \times \mathbb{N}) \times \mathbb{N} \equiv \mathbb{N} \times (\mathbb{N} \times \mathbb{N}) \\
&\quad = \texttt{refl}\ ((\mathbb{N} \times \mathbb{N}) \times \mathbb{N}) \\
&\mathit{sigmaAssoc} : \Pi_{A:\mathcal{U}} \Pi_{B:A \to \mathcal{U}} \Pi_{C:A \to B \to \mathcal{U}} \\
&\qquad \Sigma_{a:A} \Sigma_{b:B\,a} C\ a\ b \equiv \Sigma_{z:\Sigma_{a:A} B\,a} C\ (\mathbf{fst}\ A\ z)\ (\mathbf{snd}\ A\ z) \\
&\quad = \texttt{refl}\ (\Sigma_{a:A} \Sigma_{b:B\,a} C\ a\ b)
\end{aligned}
$$

Not discussed/Work in progress

- Other big hurdle we (seem to have) solved: Normalization & Evaluation

Not discussed/Work in progress

- Other big hurdle we (seem to have) solved: Normalization & Evaluation
- In progress: important metatheorems (Soundness and Completeness of Normalization, Decidability of typechecking)

Takeaways

- An associative Σ gives us the benefits of an N-ary Σ type

Takeaways

- An associative Σ gives us the benefits of an N-ary Σ type
- We can still implement and typecheck a type theory with associative Σ despite

Takeaways

- An associative Σ gives us the benefits of an N-ary Σ type
- We can still implement and typecheck a type theory with associative Σ despite
  - not having invertibility of Σ

Takeaways

- An associative Σ gives us the benefits of an N-ary Σ type
- We can still implement and typecheck a type theory with associative Σ despite
  - not having invertibility of Σ
  - having difficult context equations

Takeaways

- An associative Σ gives us the benefits of an N-ary Σ type
- We can still implement and typecheck a type theory with associative Σ despite
  - not having invertibility of Σ
  - having difficult context equations

Thanks!