Updates: Dependent Combinator Calculus

Alexandra Aiello

February 11, 2026

Current Iteration

AST

```
abbrev Level := ℕ

inductive Expr where
  | app   : Expr  → Expr → Expr
  | ty    : Level → Expr
  | judge : Level → Expr
    -- (α : Type m) → α → Prop
  | Pi    : Expr
    -- (Prop → Prop) → (Prop → Prop → Prop) → (Type 0)
  | vdash : Expr -- combine judgements into a judgment of an app
  | prop  : Expr -- only inhabited by ⊢ and :
  -- fst, snd, comp - traversal combinators for Prop
  | fst   : Expr
  | snd   : Expr
  | comp  : Expr
  /-
    SK + I + C combinator from BCKW
  -/
  | const' : Level → Level → Expr
  | const  : Level → Level → Expr
  | flip   : Level → Level → Level → Expr
  | both   : Level → Level → Level → Expr
  | id     : Level → Expr
```

Step Relation

```lean
inductive IsStep : Expr → Expr → Prop
| id     : IsStep ⟪(Expr.id m) _α x⟫ x
| both   : IsStep ⟪(both m n o) _α _β _γ x y z⟫ ⟪⟪x z⟫ ⟪y z⟫⟫
| flip   : IsStep ⟪(Expr.flip m n o) _α _β _γ x y z⟫ ⟪x z y⟫
| const' : IsStep ⟪(const' m n) _α _β x y⟫ x
| comp   : IsStep ⟪(f ∘ g) x⟫ ⟪f ⟪g x⟫⟫
| fst    : IsStep ⟪fst ⟪⊢ t_app judge_f judge_x⟫⟫ judge_f
| snd    : IsStep ⟪snd ⟪⊢ t_app judge_f judge_x⟫⟫ judge_x
| left   : IsStep f f'
  → IsStep ⟪f x⟫ ⟪f' x⟫
| right  : IsStep x x'
  → IsStep ⟪f x⟫ ⟪f x'⟫
```

Inference Rules

```
inductive ValidJudgment : Expr → Prop
  | app  : ValidJudgment ⟪(: 1) ⟪Pi t_in t_out⟫ f⟫
    → ValidJudgment ⟪(: n) t_x x⟫
    → DefEq ⟪t_in ⟪(: 1) ⟪Pi t_in t_out⟫ f⟫⟫ ⟪(: n.succ) (Ty n) t_x⟫
    -- t_out decides what to to do with the context and make a new judgment
    → ValidJudgment ⟪t_out
                     ⟪(: 1) ⟪Pi t_in t_out⟫ f⟫
                     ⟪(: n) t_x x⟫⟫

  | parapp : ValidJudgment ⟪⊢ ⟪(: 1) (Ty 0) ⟪Pi t_in t_out⟫⟫ judge_inner_f judge_inner_x⟫
    → ValidJudgment ⟪(: n) t_x x⟫
    → DefEq ⟪t_in ⟪⊢ ⟪(: 1) (Ty 0) ⟪Pi t_in t_out⟫⟫ judge_inner_f judge_inner_x⟫⟫ ⟪(: n.succ) (Ty n) t_x⟫
    → ValidJudgment ⟪t_out
      ⟪⊢ ⟪(: 1) (Ty 0) ⟪Pi t_in t_out⟫⟫ judge_inner_f judge_inner_x⟫
      ⟪(: n) t_x x⟫⟫
  -- ... on next slide
```

Inference Rules (cont...)

```
-- ... ValidJudgment
/-
  Unclear how suspicious this is. See DefEq on the next slide.
  This could be easily refactored by matching the (: T e) e
  and only defeq-ing the type.
-/
| defeq   : ValidJudgment j₁
  → DefEq j₁ j₂
  → ValidJudgment j₂
/-
  Base combinator types. All point-free. Too many to fit on one slide.
-/
| const : ValidJudgment ⟪(: 1) (const.type m n) (const m n)⟫
| both    : ValidJudgment ⟪(: 1) (both.type m n o) (both m n o)⟫
... -- more base combinator types
```

DefEq

```
inductive DefEq : Expr → Expr → Prop
| refl    : DefEq a a
| step    : IsStep e e' → DefEq e e'
| trans   : DefEq e₁ e₂ → DefEq e₂ e₃ → DefEq e₁ e₃
| left    : DefEq f f'  → DefEq ⟪f x⟫ ⟪f' x⟫
| right   : DefEq x x'  → DefEq ⟪f x⟫ ⟪f x'⟫
| vdash   : DefEq judge_app ⟪(: m.succ) (Ty m) t_fx⟫
  → DefEq judge_f ⟪(: n) t_f f⟫
  → DefEq judge_x ⟪(: o) t_x x⟫
  → DefEq ⟪⊢ judge_app judge_f judge_x⟫ ⟪(: m) t_fx ⟪f x⟫⟫
```

Combinator Types: Dependent K

```
def const.type (m n : Level) : Expr :=
  let α := mk_assert_in (Ty m) m.succ
  -- takes α, makes a new (α → Type n)
  let β.α := Expr.snd
  let β.const := (⟪(const' 0 0) Prp Prp⟫ ∘ Expr.snd)
  let β.const_out := (mk_assert_out (Ty n) n.succ)
  let β := (⟪(: 2) (Ty 1)⟫ ∘ (⟪flip_pi β.const_out⟫ ∘ β.const))

  let βx := ⟪(both 0 0 0)
      Prp
      ⟪(const' 0 1) (Ty 0) Prp Prp⟫
      ⟪(const' 0 1) (mk_arrow Prp Prp 0 0) Prp ⟪(const' 0 1) (Ty 0) Prp⟫⟫
      (⟪⊢ ⟪(: n.succ.succ) (Ty n.succ) (Ty n)⟫⟫ ∘ (snd ∘ fst))
      snd⟫

  -- Inserts our type in (: T (const α β x y)) this position.
  let cpy := ⟪(both 0 0 0)
    Prp
    ⟪(const' 1 0) (Ty 0) Prp Prp⟫
    ⟪(const' 1 0) (mk_arrow Prp (Ty 0) 0 1) Prp
      ⟪(const' 1 0) (Ty 0) Prp
        (mk_arrow Prp (mk_arrow Prp Prp 0 0) 0 1)⟫⟫⟫
  let out := ⟪cpy (⊢ ∘ (snd ∘ fst ∘ fst)) ⟪(id 0) Prp⟫⟫

  ⟪Pi ⟪(const' 0 0) Prp Prp ⟪(: m.succ.succ) (Ty m.succ) (Ty m)⟫⟫
    ⟪⊢ ⟪(: 1) (Ty 0)
      ⟪Pi β
        ⟪⊢ ⟪(: 1) (Ty 0) ⟪Pi (Expr.snd ∘ Expr.fst) ⟪⊢ ⟪(: 1) (Ty 0) ⟪Pi βx out⟫⟫⟫⟫⟫⟫⟫⟫⟫⟫
```

The Proof

```
theorem const_well_typed : ValidJudgment ⟪(: m.succ) (Ty m) α⟫
  → ValidJudgment ⟪(: 1) (mk_arrow α (Ty n) m n.succ) β⟫
  → ValidJudgment ⟪(: m) α x⟫
  → ValidJudgment ⟪(: n) ⟪β x⟫ y⟫
  → ValidJudgment ⟪(: m) α ⟪(const m n) α β x y⟫⟫ := by
  intro h_t_α h_t_β h_t_x h_t_y
  judge defeq, parapp, defeq, parapp, defeq, parapp, defeq, app, const
  exact m
  exact n
  exact h_t_α
  defeq step
  step const'
  defeq refl
  exact h_t_β
  unfold mk_arrow
  simp
  defeq trans, step
  step comp
  defeq trans, right, step
  step comp
  defeq right, trans, step
  ... -- too many steps to list on screen
```

The S Type

```
def both.type (m n o : Level) : Expr :=
  /-
    Same as in const. α : Type, β : α → Type
  -/
  let α := mk_assert_in (Ty m) m.succ

  -- takes α, makes a new (α → Type n)
  let β.α := Expr.snd
  let β.const := (⟪(const' 0 0) Prp Prp⟫ ∘ β.α)
  let β.const_out := (mk_assert_out (Ty n) n.succ)
  let β := (⟪(: 2) (Ty 1)⟫ ∘ (⟪flip_pi β.const_out⟫ ∘ β.const))

  let γ := ⟪(both_nondep Prp (mk_arrow Prp Prp 0 0) (Ty 0) 0 1 1)
    (Pi ∘ (snd ∘ fst))
      (⟪⊢ ⟪(: 1) (Ty 0)⟫⟫ ∘
        ⟪flip_pi
          (mk_assert_out (Ty o) o.succ)
          (⟪flip_comp snd⟫ ∘ (⟪⊢ ⟪(: n.succ.succ) (Ty n.succ) (Ty n)⟫⟫ ∘ Expr.snd))⟫)⟫

  let x.mk_γ_xy := ((⟪comp ⟪⊢ ⟪(: o.succ.succ) (Ty o.succ) (Ty o)⟫⟫⟫ ∘
    (⟪flip_comp snd⟫ ∘ ⟪⊢ ⟪(: o.succ.succ) (Ty o.succ) (Ty o)⟫⟫)) ∘ Expr.snd)
  let x := ⟪(both_nondep Prp (mk_arrow Prp Prp 0 0) (Ty 0) 0 1 1)
    (Pi ∘ (snd ∘ fst ∘ fst))
    (⟪⊢ ⟪(: 1) (Ty 0)⟫⟫ ∘
      ⟪(both_nondep Prp (mk_arrow Prp Prp 0 0) (Ty 0) 0 1 1)
        (⟪flip_comp snd⟫ ∘ (⟪⊢ ⟪(: n.succ.succ) (Ty n.succ) (Ty n)⟫⟫ ∘ (Expr.snd ∘ fst)))
        x.mk_γ_xy⟫)⟫

  let y := ⟪(both_nondep Prp (mk_arrow Prp Prp 0 0) (Ty 0) 0 1 1)
    (Pi ∘ (snd ∘ fst ∘ fst ∘ fst)) -- (x : α)
      (⟪⊢ ⟪(: n.succ.succ) (Ty n.succ) (Ty n)⟫⟫ ∘ (Expr.snd ∘ fst ∘ fst))⟫

  ⟪Pi α (ret_pi
    ⟪Pi β (ret_pi
      ⟪Pi γ (ret_pi
```

Summary of Past Iterations

Distinct Iterations of the Calculus

Name	Main Feature	Meta Combinators?*	Uncurried Types?	Types are Well-Typed?
SKM	Reflection	Yes	No	No
$SK\Pi$	Π Combinator	Yes	No	No
$SK\Gamma$	(Γ, Δ) registers	No	Yes	Barely. Couldn't handle random edge cases.
List Calculus	Extremely minimal kernel	No	Barely well-typed	Barely well-typed
Sigma interpretation	Sigma type is data encoding Π	No	No	Probably

Figure: *Meta combinators result in a huge tree. Each meta combinator has a type.

The Ideal Dependent Combinator Calculus

- Use (Γ, Δ) registers.
- Condense π, `next`, `read` into one rule (equivalence proven):
  - Very small kernel
- Pair interpretation: $\Gamma[n]$ is a nested pair. Same eval rules as in (Γ, Δ), but a new well-typed meaning.
- `nil` combinator: downgrades a term to a type.
  - Useful for arguments like α : Type n
- Core calculus is the typical *SK* combinators

Central Thesis: The Sigma-Curry Correspondence

- Combinator types are much easier to form with all arguments in scope (“uncurried”)
- I demonstrate ▸ here that `Sigma.snd` projection is equivalent to function application
- Treating the future application as *data* makes forming types much simpler
- We can capture projection of `fst`, `snd`, application, and many more with a single reduction rule

Research Questions: Sigma-Curry Correspondence

Hypotheses: Sigma-Curry Correspondence

- ~~Should we internalize π projection in $::[a, b]$, or should we have $::[fst, snd]$ combinators?~~ **Yes, internalize projection!**
  - ~~Can we derive `fst`?~~ **Yes!**
  - ~~Can we derive `snd`?~~ **Yes!**
  - ~~Can we fully emulate the old π combinator with a projector agrument?~~ **Yes!**
- ~~Can we derive application from π projection?~~ **Yes!**
- ~~Can we derive S from both + $\pi(id)$?~~ **Yes!**
- ~~Choose between `fst` + `snd` or π list projection combinator.~~ **Answered above.**
  - ~~Intuition says `fst` and `snd`, since they would have simpler types.~~ **Can derive `snd` and `fst`.**
- Can we derive `nil` from $::$`[x, xs] f`?

Research Questions AST

```
inductive Expr where
  | app : Expr → Expr → Expr
  | cons : Expr → Expr → Expr
  | π : Expr
  | fst : Expr | snd : Expr
  | both : Expr
  | const : Expr | const' : Expr
  | id : Expr | nil : Expr | ty : Expr

inductive IsStepStar { rel : Expr → Expr → Prop } : Expr → Expr → Prop
  | refl  : IsStepStar e e
  | trans : rel e₁ e₂
    → IsStepStar e₂ e₃
    → IsStepStar e₁ e₃
```

Figure: I have omitted universe levels for our research question proofs.

Research Questions: Sigma-Curry Correspondence

`fst` and `snd` can be condensed into one rule

Single-Step Reduction for Projection Derivations

```
inductive IsStep : Expr → Expr → Prop
  | sapp   : IsStep (.app ::[x, f], fn) (.app (.app fn f) x)
  | fst    : IsStep (\$ fst, _α, _β, fn, ::[x, f]) (\$ fn, x)
  | snd    : IsStep (\$ snd, _α, _β, fn, ::[x, f]) (\$ fn, f, x)
  | nil    : IsStep (\$ nil, α, x) α
  | id     : IsStep (\$ Expr.id, _α, x) x
  | const' : IsStep (\$ const', _α, _β, x, y) x
  | left   : IsStep f f'
    → IsStep (\$ f, x) (\$ f', x)
  | right  : IsStep x x'
    → IsStep (\$ f, x) (\$ f, x')
```

Deriving the Explicit `fst` Combinator

```
/-
  fst α β fn ::[head, tail] = fn head =
    ::[head, tail] fn =* fn head
-/
theorem fst_der (head tail fn : Expr) : IsStep
  (\$ fst, _α, _β, fn, ::[head, tail]) (\$ fn, head) ↔
  (@IsStepStar IsStep) (\$ ::[head, tail],
    (\$ const', ::[β, (\$ nil, β)], α, fn)) (\$ fn, head) := by
  constructor
  intro h_step; cases h_step
  apply IsStepStar.trans; apply IsStep.sapp
  apply IsStepStar.trans; apply IsStep.left
  apply IsStep.const'; apply IsStepStar.refl
  intro h_step; cases h_step
  case mpr.trans e₂ h_step h_trans ⇒
    cases h_trans; apply IsStep.fst
    apply IsStep.fst
```

Deriving the Explicit snd Combinator

```
/-
  snd α β fn ::[head, tail] = ::[head, tail] fn
  = fn tail head
-/
theorem snd_der (head tail fn : Expr) : IsStep
  (\$ snd, _α, _β, fn, ::[head, tail]) (\$ fn, tail, head) ↔
  (@IsStepStar IsStep) (\$ ::[head, tail], fn) (\$ fn, tail, head) := by
  constructor
  intro h_step; cases h_step
  apply IsStepStar.trans; apply IsStep.sapp
  apply IsStepStar.refl
  case mp.right a ⇒
    cases a
  intro h_step
  cases h_step
  apply IsStep.snd
```

Research Questions: Sigma-Curry Correspondence

Sigma projection is equivalent to application

Single-Step Reduction for Sigma Equivalence Proofs

```
inductive IsStep : Expr → Expr → Prop
  | sapp   : IsStep ($ ::[x, f], fn) ($ fn, f, x)
  | nil    : IsStep ($ nil, α, x) α
  | id     : IsStep ($ Expr.id, _α, x) x
  | const' : IsStep ($ const', _α, _β, x, y) x
  | const  : IsStep ($ const, _α, _β, x, y) x
  /- f and g order is flipped here compared to S.
     both f g x = ::[(f x), (g x)]
     both f g x id = id (g x) (f x) -/
  | both   : IsStep ($ both, _α, _β, _γ, f, g, x)
    ::[($f, x), ($ g, x)]
  | left   : IsStep f f'
    → IsStep ($ f, x) ($ f', x)
  | right  : IsStep x x'
    → IsStep ($ f, x) ($ f, x')
```

All Function Applications have corresponding Sigma Projections

```
/-
  (f x) = e' implies (::[x, f] (id t_f)) = e'
-/
theorem app_imp_proj (t_f f x : Expr) : (@IsStepStar IsStep)
  (\$ f, x) e' → (@IsStepStar IsStep)
  (\$ ::[x, f], (\$ id, t_f)) e' := by
  intro h_step
  cases h_step
  apply IsStepStar.trans; apply IsStep.sapp
  apply IsStepStar.trans; apply IsStep.left
  apply IsStep.id; apply IsStepStar.refl
  apply IsStepStar.trans; apply IsStep.sapp
  apply IsStepStar.trans; apply IsStep.left
  apply IsStep.id; apply IsStepStar.trans
  repeat assumption
```

All Function Applications are $=_\beta$ Sigma Projection

```
/-
  (f x) β= (::[x, f] (id t_f))
-/
theorem apps_are_proj (t_f f x : Expr) : (@IsBetaEq IsStep)
  (\$ f, x) (\$ ::[x, f], (\$ id, t_f)) := by
  apply IsBetaEq.symm; apply IsBetaEq.trans
  apply IsBetaEq.rel; apply IsStep.sapp
  apply IsBetaEq.trans; apply IsBetaEq.rel
  apply IsStep.left; apply IsStep.id
  apply IsBetaEq.refl
```

Research Questions: Sigma-Curry Correspondence

Deriving the S combinator from `both` + projection

S combinator = `id ∘ both`

```
theorem s_both_app_beq (α β γ f g x : Expr') : (@IsBetaEq IsStep)
  ($' s, α, β, γ, f, g, x)
  ($' ($' both, α, β, γ, g, f, x), ($' id, ($' β, z))) := by
  apply IsBetaEq.trans; apply IsBetaEq.rel
  apply IsStep.s; apply IsBetaEq.symm
  apply IsBetaEq.trans; apply IsBetaEq.rel
  apply IsStep.left; apply IsStep.both
  apply IsBetaEq.trans; apply IsBetaEq.rel
  apply IsStep.sapp; apply IsBetaEq.trans
  apply IsBetaEq.rel; apply IsStep.left
  apply IsStep.id; apply IsBetaEq.refl
```

Figure: This proof uses an extended AST with the S combinator for the purposes of this equivalence. Note that the order of f and g are flipped between S and `both`, since `both` is sigma-native.

Research Questions: (Γ, Δ) Contexts

Very Dependent Types

- `::` $[x, xs]$ represents a term. It is computationally relevant. What is the equivalent for types?
- ~~$x : F\ x$: very dependent types, such as this one featured in Altenkirch et al. [Alt+23] might be useful.~~ Not helpful—use Π with clever inference and reduction rules.
- ~~Since our sigma terms encode application as data, we can easily traverse the “context”.~~ List context is also unnecessary, seemingly. See ▸ here.
- ~~`Σ t_in t_out : Type`~~
- ~~To infer domain / codomain:~~ ~~`(((f : Σ Tα Tβ) (x : α)) : ((T ::[x, f]) π))`~~
  - ~~Problem: to project either component, we must know α and β.~~
- ~~Can we do better with `(((f : Σ T) (x : α)) : ((T ::[x, Σ T]) snd))`?~~
- ~~Since `Σ T` is a `Type`, the user cannot force evaluation. Only the kernel can.~~

AST

```
inductive Expr where
  | app    : Expr → Expr → Expr
  /- List-like objects
     They come with built-in projection.
     They are the mirror image of application "as data". -/
  | cons   : Expr → Expr → Expr
  /-
    ::[x, xs] lists are a special case. They are the mirror
    image of application as data. They internalize a projector
    argument π.
  -/
  | Prod   : Expr → Expr → Expr
  /-
    Our representation of curried function types.
    Π t_in t_out
  -/
  | Pi     : Expr → Expr → Expr
  | both   : Expr
  | const  : Expr
  | const' : Expr
  | id     : Expr
  -- downgrades a term to a type
  | nil    : Expr
  | ty     : Expr
```