Minimal Logic
Construction rules

- **And Construction**

  $H \vdash A\&B \ by \ pair(slot_a; slot_b)$
  $H \vdash A \ by \ slot_a$
  $H \vdash B \ by \ slot_b$

- **Exists Construction**

  $H \vdash \exists x.B(x) \ by \ pair(d; slot_b(d))$
  $H \vdash d \in D \ by \ obj(d)$
  $H \vdash B(d) \ by \ slot_b(d)$

- **Implication Construction**

  $H \vdash A \Rightarrow B \ by \ \lambda(x.slot_b(x)) \ new \ x$
  $H, x : A \vdash B \ by \ slot_b(x)$

- **All Construction**

  $H \vdash \forall x.B(x)$ *by* $\lambda(x.slot_b(x))$ *new* x
  $H, x : D \vdash B(x)$ *by* $slot_b(x)$

- **Or Construction**

  $H \vdash A \vee B$ *by* $inl(slot_l)$
  $H \vdash A$ *by* $slot_l$

  $H \vdash A \vee B$ *by* $inr(slot_r)$
  $H \vdash B$ *by* $slot_r$

Decomposition rules

- **And Decomposition**

  $H, x : A\&B,\ H' \vdash G$ *by* $spread(x; l, r.slot_g(l, r))$ *new* l, r
  $H,\ l : A,\ r : B,\ H' \vdash G$ *by* $slot_g(l, r)$

- **Exists Decomposition**

  $H, x : \exists y.B(y),\ H' \vdash G$ *by* $spread(x; d, r.slot_g(d, r))$ *new* d, r
  $H,\ d : D,\ r : B(d),\ H' \vdash G$ *by* $slot_g(d, r)$

- **Implication Decomposition**

  $H, f : A \Rightarrow B,\ H' \vdash G$ *by* $apseq(f; slot_a; v.slot_g[ap(f; slot_a)/v])$ *new* v[9]
  $H, f : A \Rightarrow B,\ H' \vdash A$ *by* $slot_a$
  $H, f : A \Rightarrow B,\ H',\ v : B \vdash G$ *by* $slot_g(v)$

- **All Decomposition**

  $H, f : \forall x.B(x),\ H' \vdash G$ *by* $apseq(f; d; v.slot_g[ap(f; d)/v])$
  $H, f : \forall x.B(x),\ H' \vdash d \in D$ *by* $obj(d)$
  $H, f : \forall x.B(x),\ H', v : B(d) \vdash G$ *by* $slot_g(v)$[10]

- **Or Decomposition**

  $H, y : A \vee B,\ H' \vdash G$ *by* $decide(y; l.leftslot(l);\ r.rightslot(r))$
  $H, l : A, H' \vdash G$ *by* $leftslot(l)$
  $H, r : B, H' \vdash G$ *by* $rightslot(r)$

- **Hypothesis**

  $H, d : D,\ H' \vdash d \in D$ *by* $obj(d)$

  $H, x : A,\ H' \vdash A$ *by* $hyp(x)$

  We usually abbreviate the justifications to *by* d and *by* x respectively.

[9] This notation shows that $ap(f; slot_a)$ is substituted for v in $g(v)$. In the CTT logic we stipulate in the rule that $v = ap(f; slot_a)$ *in* B.
[10] In the CTT logic, we use equality to stipulate that $v = ap(f; d)$ *in* $B(v)$ just before the hypothesis $v : B(d)$.

Intuitionistic Rules

- **False Decomposition**

  $H, f : \mathit{False}, H' \vdash G$ *by* $\mathit{any}(f)$

  This is the rule that distinguishes intuitionistic from minimal logic, called *ex falso quodlibet*. We use the constant *False* for intuitionistic formulas and $\bot$ for minimal ones to distinguish the logics. In practice, we would use only one constant, say $\bot$, and simply add the above rule with $\bot$ for *False* to axiomatize *iFOL*. However, for our results it's especially important to be clear about the difference, so we use both notations.

Note that we use the term d to denote objects in the domain of discourse D. In the classical evidence semantics, we assume that D is non-empty by postulating the existence of some d_0 in it. Also note that in the rule for *False* Decomposition, it is important to use the $\mathit{any}(f)$ term which allows us to thread the explanation for how *False* was derived into the justification for G.

Classical Rules

- **Non-empty Domain of Discourse**

  $H \vdash d_0 \in D$ *by* $\mathit{obj}(d_0)$

- **Law of Excluded Middle (LEM)**

  Define $\sim A$ as $(A \Rightarrow \mathit{False})$
  $H \vdash (A \vee \sim A)$ *by* $\mathbf{magic}(A)$

  Note that this is the only rule that mentions a formula in the rule name.

2.3. Computation rules

Each of the rule forms when completely filled in becomes a term in an applied lambda calculus [24,17,5], and there are *computation rules* that define how to reduce these terms in one step. These rules are given in detail in several papers about Computational Type Theory and Intuitionistic Type Theory, so we do not repeat them here. One of the most detailed accounts is in the book *Implementing Mathematics with the Nuprl Proof Development System* [20,50] and in ITT82 [57]. They are discussed in textbooks on programming languages such as *Types and Programming Languages* [65] and *Type Theory and Functional Programming* [78].

Some parts of the computation theory are needed here, such as the notion that all the terms used in the rules can be reduced to *head normal form*. Defining that reduction requires identifying the *principal argument places* in each term. We give this definition in the next section.

The reduction rules are simple. For $\mathit{ap}(f; a)$, first reduce f, if it becomes a function term, $\lambda(x.b)$, then reduce the function term to $b[a/x]$, that is, substitute the argument a for the variable x in the body of the function b and continue computing. If it does not reduce to a function, then no further reductions are possible and the computation aborts. It is possible that such a reduction will abort or continue indefinitely. But the terms arising from proofs will always reduce to normal form. This fact is discussed in the references.

To reduce $\mathit{spread}(p; x, y.g)$, reduce the principal argument p. If it does not reduce to $\mathit{pair}(a; b)$, then there are no further reductions, otherwise, reduce $g[a/x, b/y]$.

To reduce $decide(d; l.left; r.right)$, reduce the principal argument d until it becomes either $inl(a)$ or $inr(b)$ or aborts or fails to terminate.[11] In the first case, continue by reducing $left[a/l]$ and in the other case, continue by reducing $right[b/r]$.

It is important to see that none of the first-order proof terms is recursive, and it is not possible to hypothesize such terms without adding new computation forms. It is thus easy to see that all evidence terms terminate on all inputs from all models. We state this below as a theorem about valid evidence structures.

Fact Every uniform evidence term for minimal, intuitionistic, and classical logic denotes canonical evidence, and the functional terms terminate on any inputs from any model.

Additional notations. It is useful to generalize the semantic operators to n-ary versions. For example, we will write λ terms of the form $\lambda(x_1, \ldots, x_n.b)$ and a corresponding n-ary application, $f(x_1, \ldots, x_n)$. We allow n-ary conjunctions and n-tuples which we decompose using $spread_n(p; x_1, \ldots, x_n.b)$. More rarely we use n-ary disjunction and the decider, $decide_n(d; case_1.b_1; \ldots; case_n.b_n)$. It is clear how to extend the computation rules and how to define these operators in terms of the primitive ones.

It is also useful to define *True* to be the type $\bot \Rightarrow \bot$ with element $id = \lambda(x.x)$. Note that $\lambda(x.spread(pair(x; x); x_1, x_2.x_1))$ is computationally equivalent to id [41], as is $\lambda(x.decide(inr(x); l.x; r.x))$.[12]