STLC with Logic: Poor Man's Axi

August 30, 2024

Intro

These slides propose a basic calculus that could serve as a Poor Man's Axi. The system is Simply Typed Lambda Calculus with a first-order logic on top of it that lets us reason about program equality. At the end, we extend the system with booleans and natural numbers and consider adding some other additional features.

Grammar

Terms:

$e ::=$

$x \mid \lambda x.e \mid e_1\ e_2 \mid$
$(e_1, e_2) \mid \texttt{outl}\ e \mid \texttt{outr}\ e \mid$
$\texttt{inl}\ e \mid \texttt{inr}\ e \mid \texttt{case}\ e\ \texttt{of}\ (e_1, e_2) \mid$
$\texttt{unit} \mid \texttt{elim}_{\texttt{Empty}}\ e$

Types:

$A, B ::= A \to B \mid A \times B \mid A + B \mid \texttt{Unit} \mid \texttt{Empty}$

Propositions:

$P, Q ::=$

$\top \mid \bot \mid \neg P \mid P \vee Q \mid P \wedge Q \mid P \Rightarrow Q \mid P \Leftrightarrow Q \mid$
$\forall x : A.P \mid \exists x : A.P \mid$
$e_1 =_A e_2$

Terms and types

In the base language we have only function types, product types, sum types, *Unit* (the type with one element) and *Empty* (the type with no elements). Later we will add booleans, natural numbers, lists and streams.

Some of the less obvious terms: `outl` e and `outr` e are projections, *unit* is the only value of the *Unit* type and $\texttt{elim}_{\texttt{Empty}}\ e$ is the eliminator of the empty type applied to e. `case` e `of` (e_1, e_2) is the eliminator of the sum type and its arguments e_1, e_2 are supposed to be functions. Note that we don't have a type annotation on the lambda, and no type annotations in the language in general. This is not an oversight – we postpone deciding this detail for later.

Propositions

Our logic is typed (i.e. multi-sorted) first-order logic with equality. We treat negation and biconditional as defined notions ($\neg P$ is $P \Rightarrow \bot$, whereas $P \Leftrightarrow Q$ is $P \Rightarrow Q \wedge Q \Rightarrow P$), so we won't need rules for them.

Note that variables in quantifiers need to be annotated with the their types. Similarly, we need a type annotation in equations between terms.

Judgements

We will present our language using the machinery of judgements and inference rules. Each kind of **judgement** represents a meta-level proposition about objects of our language: what is the type of a term, how do terms compute and so on. To make it easier for beginners to distinguish the various kinds of judgement, we will highlight each kind of judgement with a different color.

Inference rules

An **inference rule** tells us how to derive a given judgement (its conclusion), given derivations of some other judgements and side conditions (the premises). The general form of an inference rule is

$$\frac{\mathcal{J}_1 \dots \mathcal{J}_n}{\mathcal{J}}\textsc{Name}$$

where the premises (the judgements $\mathcal{J}_i$) are written above the bar and the conclusion (the judgement $\mathcal{J}$) is written below the bar. A rule can have many premises, but only a single conclusion. Most of the time it is best to read an inference rule upside down, i.e. to first read the conclusion and only then the premises. Each rule has a name (written in small caps) so that it can be easily referred to, but we will sometimes omit the names.

Typing contexts

Typing contexts:
$\Gamma ::= \cdot \mid \Gamma, x : A$

Our judgements depend on hypotheses of the form $x : A$, which should be read as "x has type A". We gather these hypotheses in a **typing context**, which can be either empty or be an extension of another typing context with a new hypothesis. We treat typing contexts as finite partial maps from variables to types, so that we don't need to worry about order of the hypotheses, repetition of variable names and other unimportant technical details. We will use the Greek capital letter gamma Γ to denote typing contexts.

Judgements – programming language part

Typing judgement:
$\Gamma \vdash e : A$ – in typing context Γ, term e is of type A.

Computational equality judgement:
$\Gamma \vdash e_1 \equiv e_2 : A$ – in typing context Γ, terms e_1 and e_2 are computationally equal.

Sanity checks

The inference rules of the system will guarantee that whenever we derive a judgement, then another simpler judgement (called a **sanity condition**) also holds. For the computational equality $\Gamma \vdash e_1 \equiv e_2 : A$ the sanity conditions are $\Gamma \vdash e_1 : A$ and $\Gamma \vdash e_2 : A$. The typing judgement $\Gamma \vdash e : A$ does not have a sanity condition, but we can think that its sanity condition is the fact that the typing context Γ is a well-formed finite partial map from variables to types.

Typing – basics

The typing judgement tells us which terms have which types. The most important rule of typing is that variables have whatever type the typing context tells us – we will call ths the variable rule. Note that the premise of the rule, $(x : A) \in \Gamma$, does not belong to any judgement – we will call such premises **side conditions** and highlight them in orange.

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var}$$

Typing – basics 2

The remaining rules for the typing judgement are standard. For every language construct, there is only a single rule whose conclusion matches it, so it is always obvious which rule should be used, and the language construct is decomposed in the premises, which hints that an eventual typing algorithm will be terminating.

For every type there are two kinds of rules – introduction rules and elimination rules. Introduction rules are those in which the type appears in the conclusion and the elimination rules are those in which the type appears as the principal (i.e. the first) premise.

We omit names of these rules because it's easier to refer to them by the name of the language construct that they apply to.

Typing – the rules

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B} \qquad \frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B} \qquad \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathtt{outl}\ e : A} \qquad \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathtt{outr}\ e : B}$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \mathtt{inl}\ e : A + B} \qquad \frac{\Gamma \vdash e : B}{\Gamma \vdash \mathtt{inr}\ e : A + B}$$

$$\frac{\Gamma \vdash e : A + B \quad \Gamma \vdash f : A \to C \quad \Gamma \vdash g : B \to C}{\Gamma \vdash \mathtt{case}\ e\ \mathtt{of}\ (f, g) : C}$$

$$\frac{}{\Gamma \vdash \mathtt{unit} : \mathtt{Unit}} \qquad \frac{\Gamma \vdash e : \mathtt{Empty}}{\Gamma \vdash \mathtt{elim}_{\mathtt{Empty}}\ e : A}$$

Computation – basics

We represent computation using the non-directed computational equality relation, which might be a bit unintuitive, but is a common practice. Note that computational equality is typed, i.e. there's a separate computational equality relation for each type. Computational equality draws its meaning from two basic kinds of rules, computation rules and uniqueness rules. Intuitively, two terms of a given type are computationally equal when they compute to the same result, where the meaning of "compute" is specified by the computation rules and the meaning of "the same" is specified by the uniqueness rules. Formally, computational equality is the congruence closure of computation and uniqueness rules, i.e. the least equivalence relation that preserves all term constructors and contains the computation and uniqueness rules.

Substitution

To state the computation rules, we need a substitution operation. Our notation is $e_1\,[x := e_2]$ for a term e_1 in which term e_2 was substituted for the variable x. Unfortunately, I'm too lazy to define substitution here, but it shouldn't be hard for you to define it yourself.

Computation rules

Computation rules describe the most essential computation steps. For example, what happens when we project the first element out of a pair? Note that not all types have computation rules. For example, `Unit` and `Empty` have no computation rules because there's no computation going on in these types.

$$\frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x.b)\ a \equiv b\,[x := a] : B}\textsc{App-Lam}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash \texttt{outl}\ (a, b) \equiv a : A} \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash \texttt{outr}\ (a, b) \equiv b : B}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash f : A \to C \quad \Gamma \vdash g : B \to C}{\Gamma \vdash \texttt{case}\ (\texttt{inl}\ a)\ \texttt{of}\ (f, g) \equiv f\ a : C}\textsc{Case-Inl}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash f : A \to C \quad \Gamma \vdash g : B \to C}{\Gamma \vdash \texttt{case}\ (\texttt{inr}\ b)\ \texttt{of}\ (f, g) \equiv g\ b : C}\textsc{Case-Inr}$$

Uniqueness rules

Uniqueness rules establish that every term of a given type is computationally equal to a constructor of the type. For example, every term of a product type is a pair. The rules for `Unit` and `Empty` are a bit broader – they establish that all terms of these types are equal. Note that there are no uniqueness rules for sums.

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash f \equiv \lambda x.f\ x : A \to B}\textsc{Fun-Uniq}$$

$$\frac{\Gamma \vdash e : A \times B}{\Gamma \vdash e \equiv (\mathtt{outl}\ e, \mathtt{outr}\ e) : A \times B}\textsc{Prod-Uniq}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{Unit} \quad \Gamma \vdash e_2 : \mathtt{Unit}}{\Gamma \vdash e_1 \equiv e_2 : \mathtt{Unit}}\textsc{Unit-Uniq}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{Empty} \quad \Gamma \vdash e_2 : \mathtt{Empty}}{\Gamma \vdash e_1 \equiv e_2 : \mathtt{Empty}}\textsc{Empty-Uniq}$$

Computational equality is an equivalence relation

Computational equality is an equivalence relation, i.e. it is reflexive, symmetric and transitive.

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash e \equiv e : A}\textsc{Comp-Refl}$$

$$\frac{\Gamma \vdash e_1 \equiv e_2 : A}{\Gamma \vdash e_2 \equiv e_1 : A}\textsc{Comp-Sym}$$

$$\frac{\Gamma \vdash e_1 \equiv e_2 : A \quad \Gamma \vdash e_2 \equiv e_3 : A}{\Gamma \vdash e_1 \equiv e_3 : A}\textsc{Comp-Trans}$$

Congruence rules 1

The congruence rules state that computational equality is preserved by every language construct. We omit their names for reasons of space, but they are rather schematic: it would be Lam-Congr for lambda, App-Congr for application and so on.

Congruence rules 2

$$\frac{\Gamma, x : A \vdash e \equiv e' : B}{\Gamma \vdash \lambda x.e \equiv \lambda x.e' : A \to B} \qquad \frac{\Gamma \vdash f \equiv f' : A \to B \quad \Gamma \vdash a \equiv a' : A}{\Gamma \vdash f\ a \equiv f'\ a' : B}$$

$$\frac{\Gamma \vdash a \equiv a' : A \quad \Gamma \vdash b \equiv b' : B}{\Gamma \vdash (a, b) \equiv (a', b') : A \times B}$$

$$\frac{\Gamma \vdash e \equiv e' : A \times B}{\Gamma \vdash \mathtt{outl}\ e \equiv \mathtt{outl}\ e' : A} \qquad \frac{\Gamma \vdash e \equiv e' : A \times B}{\Gamma \vdash \mathtt{outr}\ e \equiv \mathtt{outr}\ e' : B}$$

$$\frac{\Gamma \vdash e \equiv e' : A}{\Gamma \vdash \mathtt{inl}\ e \equiv \mathtt{inl}\ e' : A + B} \qquad \frac{\Gamma \vdash e \equiv e' : B}{\Gamma \vdash \mathtt{inr}\ e \equiv \mathtt{inr}\ e' : A + B}$$

$$\frac{\Gamma \vdash e \equiv e' : A + B \quad \Gamma \vdash f \equiv f' : A \to C \quad \Gamma \vdash g \equiv g' : B \to C}{\Gamma \vdash \mathtt{case}\ e\ \mathtt{of}\ (f, g) \equiv \mathtt{case}\ e'\ \mathtt{of}\ (f', g') : C}$$

$$\frac{}{\Gamma \vdash \mathtt{unit} \equiv \mathtt{unit} : \mathtt{Unit}} \qquad \frac{\Gamma \vdash e \equiv e' : \mathtt{Empty}}{\Gamma \vdash \mathtt{elim}_{\mathtt{Empty}}\ e \equiv \mathtt{elim}_{\mathtt{Empty}}\ e' : A}$$

Computational equality – closing remarks

Note that because of the uniqueness rule for `Unit`, we don't need the congruence rule for `unit`. Moreover, we don't need the congruence rule for `unit` even more, because we already know that comutational equality is reflexive.

Also note that in the congruence rule for `Empty`, the premise is always true because of the uniqueness rule for `Empty`, so it could be replaced with typing rules for e and e'.

However, we keep the superfluous rules in our system to make the presentation more schematic and easier to understand.

Assumption contexts

Assumption contexts:
$\Delta ::= \cdot \mid \Delta, P$

Judgements for the logical part of our system will depend on typing hypotheses too, but they will also depend on another type of hypothesis – that a given proposition P holds. To gather hypotheses of the latter kind together, we introduce the notion of an **assumption context**, which can be either empty or be an extension of another context with the assumption that the proposition P holds. We treat assumption contexts as sets, so that we don't need to worry about order of the hypotheses. We denote these contexts with the Greek capital letter delta Δ.

Athena enjoyers can think of the assumption context as an assumption base.

Judgements for logic

Valid assumption context judgement:
$\Gamma \vdash \Delta$ `valid` – in the typing context Γ, the assumption context Δ is valid.

Well-formed proposition judgement:
$\Gamma \vdash P$ `prop` – in the typing context Γ, proposition P is well-formed.

True proposition judgement:
$\Gamma \mid \Delta \vdash P$ – in typing context Γ and propositional context Δ, proposition P holds.

Judgements for logic – sanity checks

Similarly to the situation for typing and computation judgements, the logical judgements also entail sanity conditions. For $\Gamma \vdash \Delta$ `valid` and $\Gamma \vdash P$ `prop` the sanity condition is that Γ is a well-formed finite partial map. For $\Gamma \mid \Delta \vdash P$ the sanity conditions are $\Gamma \vdash \Delta$ `valid` and $\Gamma \vdash P$ `prop`.

Valid assumption context judgement

The valid assumption context judgement tells us which assumption contexts are valid. The crux of the matter is that we can extend an assumption context only with a well-formed proposition. This judgement depends on a typing context because proposition that we add to the assumption context can contain variables.

$$\frac{}{\Gamma \vdash \cdot \ \texttt{valid}}\ \textsc{Valid-Empty}$$

$$\frac{\Gamma \vdash \Delta \ \texttt{valid} \quad \Gamma \vdash P \ \texttt{prop}}{\Gamma \vdash \Delta, P \ \texttt{valid}}\ \textsc{Valid-Extend}$$

Well-formed proposition judgement

The role of the well-formed proposition judgement is twofold: to ensure that propositions are well-scoped (i.e. they don't contain variables that are not present in the typing context), and that propositional equality is formed only from well-typed terms. This judgement depends only on the typing context.

Well-formed propositions

$$\frac{}{\Gamma \vdash \top \ \mathtt{prop}} \qquad \frac{}{\Gamma \vdash \bot \ \mathtt{prop}}$$

$$\frac{\Gamma \vdash P \ \mathtt{prop} \quad \Gamma \vdash Q \ \mathtt{prop}}{\Gamma \vdash P \vee Q \ \mathtt{prop}} \qquad \frac{\Gamma \vdash P \ \mathtt{prop} \quad \Gamma \vdash Q \ \mathtt{prop}}{\Gamma \vdash P \wedge Q \ \mathtt{prop}}$$

$$\frac{\Gamma \vdash P \ \mathtt{prop} \quad \Gamma \vdash Q \ \mathtt{prop}}{\Gamma \vdash P \Rightarrow Q \ \mathtt{prop}}$$

$$\frac{\Gamma, x : A \vdash P \ \mathtt{prop}}{\Gamma \vdash \forall x : A.P \ \mathtt{prop}} \qquad \frac{\Gamma, x : A \vdash P \ \mathtt{prop}}{\Gamma \vdash \exists x : A.P \ \mathtt{prop}}$$

$$\frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1 =_A e_2 \ \mathtt{prop}}$$

True proposition judgement

The true proposition judgement depends on two contexts – a typing context Γ and an assumption context Δ. This is because propositions can contain variables from Γ and their truth depends on assumptions from Δ.

The basic rule of our logic is that we can use assumptions from the assumption context.

$$\frac{\Gamma \vdash \Delta \ \texttt{valid} \quad P \in \Delta}{\Gamma \mid \Delta \vdash P}\,\textsc{Ass}$$

Rules for connectives

The rules for connectives are entirely standard.

$$\frac{\Gamma \mid \Delta, P \vdash Q}{\Gamma \mid \Delta \vdash P \Rightarrow Q} \qquad \frac{\Gamma \mid \Delta \vdash P \Rightarrow Q \quad \Gamma \mid \Delta \vdash P}{\Gamma \mid \Delta \vdash Q}$$

$$\frac{\Gamma \mid \Delta \vdash P \quad \Gamma \mid \Delta \vdash Q}{\Gamma \mid \Delta \vdash P \wedge Q} \qquad \frac{\Gamma \mid \Delta \vdash P \wedge Q}{\Gamma \mid \Delta \vdash P} \qquad \frac{\Gamma \mid \Delta \vdash P \wedge Q}{\Gamma \mid \Delta \vdash Q}$$

$$\frac{\Gamma \mid \Delta \vdash P}{\Gamma \mid \Delta \vdash P \vee Q} \qquad \frac{\Gamma \mid \Delta \vdash Q}{\Gamma \mid \Delta \vdash P \vee Q}$$

$$\frac{\Gamma \mid \Delta \vdash P \vee Q \quad \Gamma \mid \Delta, P \vdash R \quad \Gamma \mid \Delta, Q \vdash R}{\Gamma \mid \Delta \vdash R}$$

$$\frac{\Gamma \vdash \Delta \ \texttt{valid}}{\Gamma \mid \Delta \vdash \top} \qquad \frac{\Gamma \mid \Delta \vdash \bot}{\Gamma \mid \Delta \vdash P}$$

Substitution in propositions

To express rules for quantifiers, we need the operation of substituting a term for a variable in a proposition. Our notation is $P\,[x := e]$ for proposition P in which variable x was substituted with term e.

Rules for quantifiers

$$\frac{\Gamma, x : A \mid \Delta \vdash P}{\Gamma \mid \Delta \vdash \forall x : A.P}\textsc{Forall-Intro}$$

$$\frac{\Gamma \mid \Delta \vdash \forall x : A.P \quad \Gamma \vdash a : A}{\Gamma \mid \Delta \vdash P\,[x := a]}\textsc{Forall-Elim}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \mid \Delta \vdash P\,[x := a]}{\Gamma \mid \Delta \vdash \exists x : A.P}\textsc{Exists-Intro}$$

$$\frac{\Gamma \mid \Delta \vdash \exists x : A.P \quad \Gamma, x : A \mid \Delta, P \vdash R}{\Gamma \mid \Delta \vdash R}\textsc{Exists-Elim}$$

Rules for equality

Propositional equality is an equivalence relation that can be substitued in proofs. Note that we handle reflexivity by referring to computational equality.

$$\frac{\Gamma \vdash \Delta \ \mathtt{valid} \quad \Gamma \vdash e_1 \equiv e_2 : A}{\Gamma \mid \Delta \vdash e_1 =_A e_2}\ \textsc{Eq-Refl}$$

$$\frac{\Gamma \mid \Delta \vdash e_1 =_A e_2}{\Gamma \mid \Delta \vdash e_2 =_A e_1}\ \textsc{Eq-Sym}$$

$$\frac{\Gamma \mid \Delta \vdash e_1 =_A e_2 \quad \Gamma \mid \Delta \vdash e_2 =_A e_3}{\Gamma \mid \Delta \vdash e_1 =_A e_3}\ \textsc{Eq-Trans}$$

$$\frac{\Gamma \mid \Delta \vdash e_1 =_A e_2 \quad \Gamma, x : A \vdash P \ \mathtt{prop} \quad \Gamma \mid \Delta \vdash P\,[x := e_2]}{\Gamma \mid \Delta \vdash P\,[x := e_1]}\ \textsc{Eq-Subst}$$

Equality of functions

The general rules from the previous slide are not sufficient to prove all equations that we would like. For example, there's nothing we can use to prove function extensionality. We can fix this issue by adding rules that handle equality on a type-by-type basis. For now, the only additional rule we need concerns equality of functions, but with more types we might need more.

$$\frac{\Gamma \mid \Delta \vdash \forall x : A. f\ x =_B g\ x}{\Gamma \mid \Delta \vdash f =_{A \to B} g}\textsc{Funext}$$

Reasoning by cases on terms

Note that so far, we haven't got any rules that allow reasoning by cases on terms. For example, we might want to reason by cases not on a disjunction, but on a term $e : A + B$. To be able to do this, we need to add some more rules. Note that these rules are needed only for positive types (i.e. sums and the empty type), because for negative types the uniqueness rules suffice. Also, there's a slight discrepancy in the presentation between empty and sums, but don't worry about it.

$$\frac{\Gamma \vdash \Delta \ \mathtt{valid} \quad \Gamma \vdash e : \mathtt{Empty}}{\Gamma \mid \Delta \vdash P}\ \textsc{Empty-Ind}$$

$$\frac{\Gamma, a : A \mid \Delta \vdash P\,[x := \mathtt{inl}\ a] \quad \Gamma, b : B \mid \Delta \vdash P\,[x := \mathtt{inr}\ b]}{\Gamma \mid \Delta \vdash \forall x : A + B.P}\ \textsc{Sum-Ind}$$

Classical logic

There are many ways to add classical logic to the system, but we'll have a rule which basically says that we can reason by cases on any proposition.

$$\frac{\Gamma \mid \Delta, P \vdash R \quad \Gamma \mid \Delta, \neg P \vdash R}{\Gamma \mid \Delta \vdash R}\ \textsc{Classic}$$

Adding more types to the language

Our current menagerie of type constructors isn't very expressive. In fact, since the only base types are unit and empty, all we can do is finite types and functions between them.

In the following slides, we will introduce new types and their terms one by one. This will demonstrate the kind of work that needs to be done to extend the base language.

Eliminator for `Unit`

One little silly thing that we don't have is an eliminator for the `Unit` type. It's useless, but why not?

$$\frac{\Gamma \vdash e : A \quad \Gamma \vdash u : \mathtt{Unit}}{\Gamma \vdash \mathtt{elim}_{\mathtt{Unit}}\ e\ u : A}\ \textsc{Unit-Elim}$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \mathtt{elim}_{\mathtt{Unit}}\ e\ \mathtt{unit} \equiv e : A}\ \textsc{Comp-Unit}$$

$$\frac{\Gamma \vdash e \equiv e' : A \quad \Gamma \vdash u \equiv u' : \mathtt{Unit}}{\Gamma \vdash \mathtt{elim}_{\mathtt{Unit}}\ e\ u \equiv \mathtt{elim}_{\mathtt{Unit}}\ e'\ u' : A}\ \textsc{Congr-Unit-Elim}$$

$$\frac{\Gamma \mid \Delta \vdash P\,[x := \mathtt{unit}]}{\Gamma \mid \Delta \vdash \forall x : \mathtt{Unit}.P}\ \textsc{Unit-Ind}$$

Booleans

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}}$$

$$\frac{\Gamma \vdash e : \texttt{Bool} \quad \Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : A}{\Gamma \vdash \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 : A}\textsc{If}$$

$$\frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : A}{\Gamma \vdash \texttt{if true then } e_1 \texttt{ else } e_2 \equiv e_1 : A}\textsc{Comp-If-True}$$

$$\frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : A}{\Gamma \vdash \texttt{if false then } e_1 \texttt{ else } e_2 \equiv e_2 : A}\textsc{Comp-If-False}$$

$$\frac{\Gamma \mid \Delta \vdash P\,[b := \texttt{true}] \quad \Gamma \mid \Delta \vdash P\,[b := \texttt{false}]}{\Gamma \mid \Delta \vdash \forall b : \texttt{Bool}.P}\textsc{Bool-Ind}$$

Booleans – congruence rules

$$\frac{}{\Gamma \vdash \texttt{true} \equiv \texttt{true} : \texttt{Bool}}\ \textsc{True-Congr}$$

$$\frac{}{\Gamma \vdash \texttt{false} \equiv \texttt{false} : \texttt{Bool}}\ \textsc{False-Congr}$$

$$\frac{\Gamma \vdash e \equiv e' : \texttt{Bool} \quad \Gamma \vdash e_1 \equiv e_1' : A \quad \Gamma \vdash e_2 \equiv e_2' : A}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \equiv \texttt{if}\ e'\ \texttt{then}\ e_1'\ \texttt{else}\ e_2' : A}\ \textsc{If-Congr}$$

Natural numbers

$$\frac{}{\Gamma \vdash \mathtt{zero} : \mathbb{N}} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathtt{succ}\ n : \mathbb{N}}$$

$$\frac{\Gamma \vdash z : A \quad \Gamma \vdash s : A \to A \quad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathtt{rec}_{\mathbb{N}}\ z\ s\ n : A}\textsc{Nat-Elim}$$

$$\frac{\Gamma \vdash z : A \quad \Gamma \vdash s : A \to A}{\Gamma \vdash \mathtt{rec}_{\mathbb{N}}\ z\ s\ \mathtt{zero} \equiv z : A}\textsc{Rec-Zero}$$

$$\frac{\Gamma \vdash z : A \quad \Gamma \vdash s : A \to A \quad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathtt{rec}_{\mathbb{N}}\ z\ s\ (\mathtt{succ}\ n) \equiv s\ (\mathtt{rec}_{\mathbb{N}}\ z\ s\ n) : A}\textsc{Rec-Succ}$$

$$\frac{\Gamma \mid \Delta \vdash P\,[n := \mathtt{zero}] \quad \Gamma, n' : \mathbb{N} \mid \Delta, P\,[n := n'] \vdash P\,[n := \mathtt{succ}\ n']}{\Gamma \mid \Delta \vdash \forall n : \mathbb{N}.P}\textsc{Nat-I}$$

Natural numbers – congruence rules

$$\frac{}{\Gamma \vdash \mathtt{zero} \equiv \mathtt{zero} : \mathbb{N}}\textsc{Zero-Congr}$$

$$\frac{\Gamma \vdash n \equiv n' : \mathbb{N}}{\Gamma \vdash \mathtt{succ}\ n \equiv \mathtt{succ}\ n' : \mathbb{N}}\textsc{Succ-Congr}$$

$$\frac{\Gamma \vdash z \equiv z' : A \quad \Gamma \vdash s \equiv s' : A \to A \quad \Gamma \vdash n \equiv n' : \mathbb{N}}{\Gamma \vdash \mathtt{rec}_{\mathbb{N}}\ z\ s\ n \equiv \mathtt{rec}_{\mathbb{N}}\ z'\ s'\ n' : A}\textsc{Nat-Elim-Congr}$$

Lists

$$\frac{}{\Gamma \vdash \texttt{nil} : \mathsf{List}\ A} \qquad \frac{\Gamma \vdash h : A \quad \Gamma \vdash t : \mathsf{List}\ A}{\Gamma \vdash \texttt{cons}\ h\ t : \mathsf{List}\ A}$$

$$\frac{\Gamma \vdash n : A \quad \Gamma \vdash c : A \to B \to B \quad \Gamma \vdash l : \mathsf{List}\ A}{\Gamma \vdash \texttt{fold}\ n\ c\ l : B}\ \textsc{List-Elim}$$

$$\frac{\Gamma \vdash n : B \quad \Gamma \vdash c : A \to B \to B}{\Gamma \vdash \texttt{fold}\ n\ c\ \texttt{nil} \equiv n : B}\ \textsc{Fold-Nil}$$

$$\frac{\Gamma \vdash n : B \quad \Gamma \vdash c : A \to B \to B \quad \Gamma \vdash h : A \quad \Gamma \vdash t : \mathsf{List}\ A}{\Gamma \vdash \texttt{fold}\ n\ c\ (\texttt{cons}\ h\ t) \equiv c\ h\ (\texttt{fold}\ n\ c\ t) : B}\ \textsc{Fold-Cons}$$

$$\frac{\Gamma \mid \Delta \vdash P\,[l := \texttt{nil}] \quad \Gamma, h : A, t : \mathsf{List}\ A \mid \Delta, P\,[l := t] \vdash P\,[l := \texttt{cons}\ h}{\Gamma \mid \Delta \vdash \forall l : \mathsf{List}\ A.P}$$

Lists – congruence rules

$$\frac{}{\Gamma \vdash \texttt{nil} \equiv \texttt{nil} : \mathsf{List}\ A}\ \textsc{Nil-Congr}$$

$$\frac{\Gamma \vdash h \equiv h' : A \quad \Gamma \vdash t \equiv t' : \mathsf{List}\ A}{\Gamma \vdash \texttt{cons}\ h\ t \equiv \texttt{cons}\ h'\ t' : \mathsf{List}\ A}\ \textsc{Cons-Congr}$$

$$\frac{\Gamma \vdash n \equiv n' : A \quad \Gamma \vdash c \equiv c' : A \to B \to B \quad \Gamma \vdash l \equiv l' : \mathsf{List}\ A}{\Gamma \vdash \texttt{fold}\ n\ c\ l \equiv \texttt{fold}\ n'\ c'\ l' : B}\ \textsc{List-Elim-C}$$

Streams

$$\frac{\Gamma \vdash s : \texttt{Stream}\ A}{\Gamma \vdash \texttt{hd}\ s : A} \qquad \frac{\Gamma \vdash s : \texttt{Stream}\ A}{\Gamma \vdash \texttt{tl}\ s : \texttt{Stream}\ A}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash f : A \to A}{\Gamma \vdash \texttt{scons}\ a\ f : \texttt{Stream}\ A}\textsc{Scons}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash f : A \to A}{\Gamma \vdash \texttt{hd}\ (\texttt{scons}\ a\ f) \equiv a : A}\textsc{Hd-Scons}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash f : A \to A}{\Gamma \vdash \texttt{tl}\ (\texttt{scons}\ a\ f) \equiv \texttt{scons}\ (f\ a)\ f : \texttt{Stream}\ A}\textsc{Tl-Scons}$$

Streams – congruence rules

Streams do have all the congruence rules we would expect, but they don't have any uniqueness rules.

$$\frac{\Gamma \vdash s \equiv s' : A}{\Gamma \vdash \mathtt{hd}\ s \equiv \mathtt{hd}\ s' : A}\ \textsc{Hd-Congr}$$

$$\frac{\Gamma \vdash s \equiv s' : A}{\Gamma \vdash \mathtt{tl}\ s \equiv \mathtt{tl}\ s' : \mathtt{Stream}\ A}\ \textsc{Tl-Congr}$$

$$\frac{\Gamma \vdash a \equiv a' : A \quad \Gamma \vdash f \equiv f' : A \to A}{\Gamma \vdash \mathtt{scons}\ a\ f \equiv \mathtt{scons}\ a'\ f' : \mathtt{Stream}\ A}\ \textsc{Scons-Congr}$$

However, for now I have no idea how to formulate the rule of proof by coinduction, so let's omit it.

Record types – intro

The unfortunate reality of product types is that accessing components of nested pairs requires long sequences of `outl` s and `outr` s, which look ugly. To add insult to injury, product types are not strictly associative, so we need to keep track of parentheses in nested pairs.

To free the programmer from these inconveniences, we add record types to the language. Records types are basically finite maps from field names to types, and records are finite maps from field names to values. The inference rules look somewhat ugly because they are rather generic, but everthing works like you'd expect.

The original product types could be represented using record types as $\{\texttt{outl} : A; \texttt{outr} : B\}$. We also don't need the unit type anymore – it gets replaced with the empty record type $\{\}$.

Record types

$$\frac{\forall i \in I. \quad \Gamma \vdash e_i : A_i}{\Gamma \vdash \{\ell_i := e_i\}_{i \in I} : \{\ell_i : A_i\}_{i \in I}} \textsc{Record-Intro}$$

$$\frac{\Gamma \vdash e : \{\ell_i : A_i\}_{i \in I} \quad j \in I}{\Gamma \vdash e.\ell_j : A_j} \textsc{Record-Elim}$$

$$\frac{\Gamma \vdash \{\ell_i := e_i\}_{i \in I} : \{\ell_i : A_i\}_{i \in I} \quad j \in I}{\Gamma \vdash \{\ell_i := e_i\}_{i \in I}.\ell_j \equiv e_j : A_j} \textsc{Record-Comp}$$

$$\frac{\Gamma \vdash r : \{\ell_i : A_i\}_{i \in I}}{\Gamma \vdash r \equiv \{\ell_i := r.\ell_i\}_{i \in I} : \{\ell_i : A_i\}_{i \in I}} \textsc{Record-Uniq}$$

Record types – congruence rules

$$\frac{\forall i \in I. \quad \Gamma \vdash e_i \equiv e_i' : A_i}{\Gamma \vdash \{\ell_i := e_i\}_{i \in I} \equiv \{\ell_i := e_i'\}_{i \in I} : \{\ell_i : A_i\}_{i \in I}}\textsc{Record-Intro-Congr}$$

$$\frac{\Gamma \vdash e \equiv e' : \{\ell_i : A_i\}_{i \in I} \quad j \in I}{\Gamma \vdash e.\ell_j \equiv e'.\ell_j : A_j}\textsc{Record-Elim-Congr}$$

Variant types – intro

Variant improve upon sum types in the same way that records improve upon product types – they can be thought of as named sum types. They are basically finite maps from constructor names to types, and terms of variant types are constructors applied to arguments.

The original sum type can be recovered as $[\texttt{inl} : A; \texttt{inr} : B]$ whereas the empty type can be represented as the empty variant type $[]$.

Variant types

$$\frac{\Gamma \vdash e : A_j \quad j \in I}{\Gamma \vdash c_j\ e : [c_i : A_i]_{i \in I}}\ \textsc{Variant-Intro}$$

$$\frac{\Gamma \vdash e : [c_i : A_i]_{i \in I} \quad \Gamma \vdash r : \{c_i : A_i \to B_i\}_{i \in I}}{\Gamma \vdash \texttt{vcase}\ e\ \texttt{of}\ r : B}\ \textsc{Variant-Elim}$$

$$\frac{\Gamma \vdash e : A_j \quad \Gamma \vdash r : \{c_i : A_i \to B_i\}_{i \in I} \quad j \in I}{\Gamma \vdash \texttt{vcase}\ (c_j\ e)\ \texttt{of}\ r \equiv r.c_j\ e : B}\ \textsc{Variant-Comp}$$

Variant types – congruence rules

$$\frac{\Gamma \vdash e \equiv e' : A_j \quad j \in I}{\Gamma \vdash c_j\ e \equiv c_j\ e' : [c_i : A_i]_{i \in I}}\ \textsc{Variant-Intro-Congr}$$

$$\frac{\Gamma \vdash e \equiv e' : [c_i : A_i]_{i \in I} \quad \Gamma \vdash r \equiv r' : \{c_i : A_i \to B_i\}_{i \in I}}{\Gamma \vdash \mathtt{vcase}\ e\ \mathtt{of}\ r \equiv \mathtt{vcase}\ e'\ \mathtt{of}\ r' : B}\ \textsc{Variant-Elim-Cong}$$

Metatheory

If I didn't screw anything up, the presented system enjoys nice metatheoretical properties:

- Decidability – all judgements are decidable, i.e. we can implement a typechecker, an interpreter/compiler, and we can check whether a given proof is correct.
- Type preservation – computing with a well-typed program does not change its type.
- Termination – well-typed programs terminate.
- Confluence – the final result of computation is unique.
- Canonicity – the final results of computation of a given type correspond to the introduction rules for that type.

Exercises

Just seeing a system like this won't be enough to convince anybody that it makes sense. Therefore, doing some exercises would be advised:

- Assume that A and B are arbitrary types. Define functions $\texttt{swap} : A \times B \to B \times A$ and $\texttt{sweep} : A + B \to B + A$ and prove that they are involutive. Are they computationally involutive, i.e. involutive up to computational equality? Repeat the exercise using record and variant types.
- Can you prove that every term of a product type is a pair?
- Can you prove that every term of a sum type is either $\texttt{inl}\ a$ or $\texttt{inr}\ b$ for some a and b?
- Define addition of natural numbers and prove that it is associative and commutative.
- Write an interesting program and prove an interesting theorem about it.

Additional features – intro

What other features might we add to our language? Let's find out.

Type annotations

A practical programming language certainly needs a way for the programmer to put some helpful type annotations here and there. The rules are simple – an annotated term has the stated type, provided that we can check it does. As for computation, the type annotation simply gets erased. There's also a congruence rule.

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash (e : A) : A}\,\textsc{Annot}$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash (e : A) \equiv e : A}\,\textsc{Comp-Annot}$$

$$\frac{\Gamma \vdash e \equiv e' : A}{\Gamma \vdash (e : A) \equiv (e' : A) : A}\,\textsc{Annot-Congr}$$

Let bindings

Let-bindings are a useful programming language feature, so we could add them. Note that we have some choice there: we could make it a primitive construct, governed by the below rules, or we could simply make $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ an abbreviation for $(\lambda x.e_2)\ e_1$

$$\frac{\Gamma \vdash e_1 : A \quad \Gamma, x : A \vdash e_2 : B}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : B}\textsc{Let}$$

$$\frac{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : A}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \equiv e_2\,[x := e_1] : A}\textsc{Let-Comp}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : A \quad \Gamma \vdash e_2 \equiv e_2' : B}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \equiv \texttt{let}\ x = e_1'\ \texttt{in}\ e_2' : B}\textsc{Let-Congr}$$

Definitions in context – valid context judgement

We can change the definition of typing contexts to allow putting definitions in them. This can be handy for expressing let bindings and also makes fully formal derivations much shorter. However, it also requires adding a new judgement, the well-formed context judgement, because we want to allow adding only well-typed definitions to the context. We must also remember to include these definitions in computational equality.

$\Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, x : A := e$

$$\frac{}{\cdot \ \mathtt{ctx}}\ \textsc{Ctx-Empty} \qquad \frac{\Gamma \ \mathtt{ctx} \quad x \notin \Gamma}{\Gamma, x : A \ \mathtt{ctx}}\ \textsc{Ctx-Extend}$$

$$\frac{\Gamma \ \mathtt{ctx} \quad \Gamma \vdash e : A \quad x \notin \Gamma}{\Gamma, x : A := e \ \mathtt{ctx}}\ \textsc{Ctx-Def}$$

Definitions in context – using variables

We now need two rules for typing variables: one for when the variable is defined and one for when it is not. Moreover, we also need a computation rule which says that variables are computationally equal to the body of their definition.

$$\frac{\Gamma\ \mathtt{ctx} \quad (x : A) \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var-Ext} \qquad \frac{\Gamma\ \mathtt{ctx} \quad (x : A := e) \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var-Def}$$

$$\frac{\Gamma\ \mathtt{ctx} \quad (x : A := e) \in \Gamma}{\Gamma \vdash x \equiv e : A}\ \textsc{Comp-Var}$$

Definitions in context – computation rules

The computation rule for function can (but need not!) look very different when we can put definitions into contexts. This time, instead of substituting a for x in b, we simply extend the context with the definition $x : A := a$ and then proceed with the computation. The rule for `let` also gets simpler (and this time we should prefer this rule, because it makes so much more sense).

$$\frac{\Gamma, x : A := a \vdash b \equiv b' : B}{\Gamma \vdash (\lambda x.b)\ a \equiv b' : B}\ \textsc{App-Lam-Def}$$

$$\frac{\Gamma, x : A := e_1 \vdash e_2 \equiv e_2' : B}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \equiv e_2' : B}\ \textsc{Let-Comp-Def}$$

Refinement types and quotient types – setup

For a moment, let's say that types can depend on propositions. This means that we have a well-formed type judgement which depends on the typing context. We need to modify the valid context judgement to take this into account.

$$\frac{\Gamma\ \mathtt{ctx} \quad \Gamma \vdash A\ \mathtt{type} \quad x \notin \Gamma}{\Gamma, x : A\ \mathtt{ctx}}\ \textsc{Ctx-Extend}$$

With such a setup, $\Gamma\ \mathtt{ctx}$ is a sanity condition for $\Gamma \vdash A\ \mathtt{type}$, whereas $\Gamma \vdash A\ \mathtt{type}$ is a sanity condition for $\Gamma \vdash x : A$.

Refinement types – intro

A refinement type can be thought of a subset of the base type for which some proposition holds. They can be used to express types like “the type of all natural numbers greater than 42”, which can be quite useful. They can also be thought of as a way for the programming layer of the language to access the logic layer.

Refinement types

$$\frac{\Gamma \vdash A\ \mathtt{type} \quad \Gamma, x : A \vdash P\ \mathtt{prop}}{\Gamma \vdash \{x : A \mid P\}\ \mathtt{type}}\ \textsc{Refinement-Form}$$

$$\frac{\Gamma \mid \Delta \vdash e : A \quad \Gamma \mid \Delta \vdash P\,[x := e]}{\Gamma \mid \Delta \vdash \mathtt{refine}\ e : \{x : A \mid P\}}\ \textsc{Refinement-Intro}$$

$$\frac{\Gamma \mid \Delta \vdash e : \{x : A \mid P\}}{\Gamma \mid \Delta \vdash \mathtt{unrefine}\ e : A}\ \textsc{Refinement-Elim}$$

$$\frac{\Gamma \mid \Delta \vdash e : A \quad \Gamma \mid \Delta \vdash P}{\Gamma \vdash \mathtt{unrefine}\ (\mathtt{refine}\ e) \equiv e : A}\ \textsc{Refinement-Comp}$$

$$\frac{\Gamma \mid \Delta \vdash e : \{x : A \mid P\}}{\Gamma \vdash e \equiv \mathtt{refine}\ (\mathtt{unrefine}\ e) : \{x : A \mid P\}}\ \textsc{Refinement-Uniq}$$

$$\frac{\Gamma, e : A \mid \Delta \vdash P\,[x := \mathtt{refine}\ e]}{\Gamma \mid \Delta \vdash \forall x : \{y : A \mid P\}.P}\ \textsc{Refinement-Ind}$$

Refinement types – comments

Let's try to read the above rules:

- $\{x : A \mid P\}$ is a type if A is a well-formed type and P is a well-formed proposition.
- `refine` e is a term of this type if e is of type A and the proposition P holds for e.
- We can project a value of the base from the refinement type.
- Computation and uniqueness rules state the `refine` and `unrefine` are inverses of each other.
- The last rule assures us that we can prove properties of refined terms by considering only the `refine` case.

Quotient types (TODO)

Athena enjoyers can think of quotient types as akin to Athena's structures, except that the underlying type needs not be an algebraic type (which we don't yet have anyway).

$$\frac{\Gamma \vdash A\ \texttt{type} \quad \Gamma, x : A, y : A \vdash R\ \texttt{prop} \quad \texttt{Equivalence}\ R}{\Gamma \vdash A \setminus\setminus R\ \texttt{type}}\ \textsc{Quotient-Form}$$

$$\frac{\Gamma \mid \Delta \vdash e : A \quad \Gamma, x : A, y : A \vdash R\ \texttt{prop} \quad \texttt{Equivalence}\ R}{\Gamma \mid \Delta \vdash \texttt{qin}\ e : A \setminus\setminus R}\ \textsc{Quotient-Intro}$$

$$\frac{\Gamma \mid \Delta \vdash e : A \setminus\setminus R \quad \Gamma \mid \Delta \vdash f : A \to B \quad f\ \texttt{preserves}\ R}{\Gamma \mid \Delta \vdash \texttt{qout}\ f\ e : B}\ \textsc{Quotient-Elim}$$

$$\frac{\Gamma \mid \Delta \vdash e : A \quad \Gamma \mid \Delta \vdash f : A \to B \quad f\ \texttt{preserves}\ R}{\Gamma \vdash \texttt{qout}\ f\ (\texttt{qin}\ e) \equiv f\ e : B}\ \textsc{Quotient-Comp}$$

Where "preserves" is $\Gamma \mid \Delta \vdash \forall x : A. \forall y : A. R\ x\ y \Rightarrow f\ x =_B f\ y$

Additional feature proposals

- Subtyping – some types are now in a subtyping relation $A <: B$ and an $a : A$ can be used in every place where a term of type B is expected.
- Parametric polymorphism – a new type former $\forall \alpha.A$. It can be either explicit (need to manage the type arguments manually) or implicit (type arguments are inferred). The implicit approach is the most common, but it's also harder to implement.
- Algebraic data types – adding them in a clean manner is a bit complicated, because we either need to deal with general recursion or manage a plethora of checks, like coverage and termination.
- Module system – this will be a bit harder to model, but we could borrow it from ML and OCaml.

Additional features – alternative presentations

- The system could be presented using a reduction relation which would cover the "directed" part of computational equality.
- The type system could be presented in a more algorithmic way, which is more amenable to being treated as a reference implementation.
- Is an explicit definition of substitution needed, or nobody cares?