Explicit Casts

The basis for this extension is the simply typed lambda calculus with support for subtyping and exceptions.

Syntax:

\[ e ::= \ldots \mid e \as T \]

Type rule:

\[
\begin{array}{c}
\Gamma \vdash e : S \\
\hline \\
\Gamma \vdash e \as T : T
\end{array}
\]

Evaluation contexts:

\[ E ::= \ldots \mid E \as T \]

Reduction rule:

\[
\begin{array}{c}
\vdash v : T \\
\hline \\
v \as T \rightarrow v
\end{array}
\]

\[
\begin{array}{c}
\forall v : T \\
\hline \\
v \as T \rightarrow \text{raise "cast error"}
\end{array}
\]
Type Tests (alternative to using exceptions)

Syntax:

\[ e ::= \ldots \mid \text{if } e \text{ in } T \text{ then } x \cdot e \text{ else } e \]

Type rule:

\[
\Gamma \vdash e_1 : S \quad \Gamma, x : T_1 \vdash e_2 : T_2 \quad \Gamma \vdash e_3 : T_2
\]

\[
\Gamma \vdash \text{if } e_1 \text{ in } T_1 \text{ then } x \cdot e_2 \text{ else } e_3 : T_2
\]

Evaluation contexts:

\[ E ::= \ldots \mid \text{if } E \text{ in } T_1 \text{ then } x \cdot e_2 \text{ else } e_3 \]

Reduction rules:

\[
\Gamma \vdash v : T_1
\]

\[
\text{if } v \text{ in } T_1 \text{ then } x \cdot e_2 \text{ else } e_3 \rightarrow [x \mapsto v]e_2
\]

\[
\Gamma \vdash v : T_1
\]

\[
\text{if } v \text{ in } T_1 \text{ then } x \cdot e_2 \text{ else } e_3 \rightarrow e_3
\]
Suppose we want \textit{int} <: \textit{float}.

With the semantics given for subtyping so far, this would mean integers would have to be represented in the same way as floats.

For efficiency reasons, we probably want integers and floats to have different representations.

To allow for this, we can associate run-time coercions with uses of the subsumption rule, for example, to convert an integer into a float.
Coercion Semantics for Subtyping

For each subtype derivation, we’ll associate a function that coerces a value of the left-hand type to a value of the right-hand type. Given a derivation $D$, we write $\llbracket D \rrbracket$ for the coercion function.

\[
\begin{align*}
\llbracket \text{Bool} :<: \text{Bool} \rrbracket &= \lambda x : \text{Bool}. x \\
\llbracket \text{Int} :<: \text{Float} \rrbracket &= \text{intToFloat} \\
\llbracket D_1 :: T_1 :<: S_1 \quad D_2 :: S_2 :<: T_2 \rrbracket &= \lambda f : S_1 \rightarrow S_2. \lambda x : T_1. ([D_2] (f ([D_1] x))) \\
\llbracket \text{dom}(R_2) \subseteq \text{dom}(R_1) \quad \forall l \in \text{dom}(R_2). R_1(l) = R_2(l) \rrbracket &= \lambda r : R_1. \{ l = r. l \in \text{dom}(R_2) \} \\
\vdots
\end{align*}
\]
Coercion Semantics for Subtyping

Define a translation from the language with subtyping to the language with explicit coercions:

\[ \Gamma \vdash e \Rightarrow e : T \]

The translation rule corresponding to subsumption inserts a coercion.

\[
\frac{
\Gamma \vdash e \Rightarrow e' : S \quad D :: S <: T
}{
\Gamma \vdash e \Rightarrow ([D] e') : T
} 
\]
Recall the subsumption rule:

\[
\frac{\Gamma \vdash e : S \quad S <: T}{\Gamma \vdash e : T}
\]

This rule can be applied anywhere, anytime. So how do we know when to use it in the type checker?

The problem with this rule is that it is not syntax directed.

Can we get rid of the rule by using \( S <: T \) in the other rules?
The basic idea is to sprinkle uses of $\ll$: in all the places that you would use type equality.

For example, in function application:

\[
\Gamma \vdash e_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash e_2 : T_2 \quad T_2 \ll : T_{11}
\]

\[
\Gamma \vdash (e_1 \ e_2) : T_{12}
\]
There’s also a problem with the definition of subtyping: we don’t know when, or in what order, to apply the record width and depth rules.

\[
\begin{align*}
\text{dom}(R_2) & \subseteq \text{dom}(R_1) & \forall l \in \text{dom}(R_2). R_1(l) = R_2(l) \\
R_1 & \ll R_2
\end{align*}
\]

\[
\begin{align*}
\text{dom}(R_1) = \text{dom}(R_2) & \text{ for } l \in \text{dom}(R_1). R_1(l) \ll R_2(l) \\
R_1 & \ll R_2
\end{align*}
\]

These two rules can be replaced by the following rule:

\[
\begin{align*}
\text{dom}(R_2) & \subseteq \text{dom}(R_1) & \text{for } l \in \text{dom}(R_2). R_1(l) \ll R_2(l) \\
R_1 & \ll R_2
\end{align*}
\]
Suppose for the moment we have defined subtyping by the following syntax-directed rules:

\[
\begin{align*}
&\text{Bool } <: \text{ Bool} \\
&\text{Int } <: \text{ Int} \\
&T \:<: \text{ Top}
\end{align*}
\]

\[
\begin{align*}
&T_1 \:<: S_1 \quad S_2 \:<: T_2 \\
&S_1 \rightarrow S_2 \:<: T_1 \rightarrow T_2
\end{align*}
\]

Can we prove that this subtype relation is transitive?
Proposition

If $R <: S$ and $S <: T$ then $R <: T$.

Proof. by induction on $S$.

Case $S = \text{Int}$. Then $R = \text{Int}$ and $T = \text{Int}$ or $T = \text{Top}$. In either case, $R <: T$.

Case $S = \text{Bool}$. Similar.

Case $S = \text{Top}$. So $T = \text{Top}$ and then $R <: T$. 
Case $S = S_1 \rightarrow S_2$. Then $R = R_1 \rightarrow R_2$ with $S_1 <: R_1$ and $R_2 <: S_2$, and either (a) $T = T_1 \rightarrow T_2$ with $T_1 <: S_1$ and $S_2 <: T_2$ or else (b) $T = Top$. The following diagram shows the proof for case (a):

![Diagram showing subtyping relationships]

For case (b), because $T = Top$ we immediately have $R <: T$. 
What type should the following expression have?

```plaintext
if true then \{x=true,y=false\} else \{x=false,z=true\}
```

Should it be \{x : Top\}, \{x : Bool\}, or just \}\}? In some sense, any type that is supertype of both branches will work. But which is best?

We don’t want to throw information away, so the best type is the **least upper bound** of the types of the two branches.

I.e., if the two branches have type $S$ and $T$, we want the type $R$ such that $S <: R$ and $T <: R$, and for any other type $R'$, $R <: R'$.

Does such a type always exist?