Booleans

Syntax of terms and values

\[
\begin{align*}
\text{terms} & : = \quad \text{constant true} \\
& \quad \text{constant false} \\
& \quad \text{conditional} \\
\text{values} & : = \quad \text{true value} \\
& \quad \text{false value}
\end{align*}
\]

Transition (Evaluation) Relation

The relation \( t \rightarrow t' \) is the smallest relation closed under the following rules:

- if true then \( t_2 \) else \( t_3 \) \( \rightarrow t_2 \)
- if false then \( t_2 \) else \( t_3 \) \( \rightarrow t_3 \)
- \( t_1 \rightarrow t'_1 \)
- if \( t_1 \) then \( t_2 \) else \( t_3 \rightarrow \) if \( t'_1 \) then \( t_2 \) else \( t_3 \)

Terminology

- Computation rules
  - if true then \( t_2 \) else \( t_3 \rightarrow t_2 \)
  - if false then \( t_2 \) else \( t_3 \rightarrow t_3 \)
- Congruence rule
  - \( t_1 \rightarrow t'_1 \)
  - if \( t_1 \) then \( t_2 \) else \( t_3 \rightarrow \) if \( t'_1 \) then \( t_2 \) else \( t_3 \)

Computation rules perform "real" computation steps. Congruence rules guide evaluation order; they determine where computation rules can be next applied.
**IMP: A simple imperative language**

**Syntactic categories:**
- **int** integers
- **bool** boolean
- **L** locations
- **Aexp** Arithmetic expressions
- **Bexp** Boolean expressions
- **Com** Commands
- **Values**
  - \( v ::= n \mid \text{true} \mid \text{false} \)

---

**Abstract syntax (Comm)**

**Commands**

\[
\begin{align*}
  c ::= & \ skip \quad | \ x := e \\
  & \ c_1; c_2 \\
  & \if b \text{ then } c_1 \text{ else } c_2 \\
  & \text{while } b \text{ do } c
\end{align*}
\]

- Typing rules expressed implicitly in the choice of meta-variables
- All side-effects captured within commands
- Do not consider functions, pointers, data structures

---

**Operational Semantics for IMP**

- Unlike the simple language of booleans and conditionals or arithmetic, IMP programs bind variables to locations, and can side-effect the contents of these locations.
- Define \( \sigma \in \Sigma = L \rightarrow Z \) to define the state of program memory.
- Evaluation judgements take one of the following forms:
  - \( c, \sigma \rightarrow \sigma' \)
  - \( t, \sigma \rightarrow t', \sigma' \)

**Values**

\( v ::= n \mid \text{true} \mid \text{false} \)

---

**Natural Semantics for Com**

\[
\begin{align*}
  <e, \sigma> & \Rightarrow n \\
  <x:=e, \sigma> & \Rightarrow \sigma[x \mapsto n]
\end{align*}
\]

\[
\begin{align*}
  <\text{skip}, \sigma> & \Rightarrow \sigma \\
  <c_1,c_2, \sigma> & \Rightarrow \sigma' \\
  <\text{while } b \text{ do } c, \sigma> & \Rightarrow \sigma'
\end{align*}
\]

**Examples:**

\[
\begin{align*}
  <\text{true}, c_1, \sigma> & \Rightarrow \sigma' \\
  <\text{false}, c_2, \sigma> & \Rightarrow \sigma' \\
  <\text{if } b \text{ then } c_1 \text{ else } c_2, \sigma> & \Rightarrow \sigma'
\end{align*}
\]

---

**Values**

\( v ::= n \mid \text{true} \mid \text{false} \)
Redex

A redex is a term that can be transformed in a single step

- A redex has no antecedents
- $r ::= x | x := n | x := n + n | \text{skip} ; c |
  \hspace{1cm} \text{if true then } c_1 \text{ else } c_2 |
  \hspace{1cm} \text{if false then } c_1 \text{ else } c_2 |
  \hspace{1cm} \text{true } \land b | \text{false } \lor b |
  \hspace{1cm} \text{....}

Contexts

- Can define evaluation context via a grammar:
  $E ::= [ ] | n + E | n * E | x := E |
  \hspace{1cm} \text{if } E \text{ then } c_1 \text{ else } c_2 |
  \hspace{1cm} E: c | \text{while } E \text{ do } c$

The grammar fixes the order of evaluation, allowing us to simplify the number and structure of the rules used in the semantics

Operational Semantics: Lambda-Calculus

- Values:
  - $\lambda x. t$
- Computation rule:
  - $((\lambda x. t_1) v) \rightarrow t_1[v/x]$
- Congruence rules
  - $t_1 \rightarrow t'_1$
  - $(t_1 t_2) \rightarrow (t'_1 t'_2)$
  - $t_2 \rightarrow t'_2$
  - $(v t_2) \rightarrow (v t'_2)$
  - $x \text{ not free in } t$
  - $\lambda x. (t x) \rightarrow t$

The first computation rule is referred to as the $\beta$-substitution or $\beta$-conversion rule.
- $((\lambda x. t_1) t_2) \rightarrow t'_1 t_2$ is called a $\beta$-redex.

The last congruence rule is referred as the $\eta$-conversion rule.
- $(\lambda x. (t x)) \rightarrow (v t_2)$
  - where $x$ not in $\text{FV}(t)$ is an $\eta$-redex
  - $\eta$-conversion related to notion of function extensionality. Why?

Normal forms and order of evaluation

- No expression can be converted to two distinct normal forms (Church-Rosser Theorem 1)
- Is there an order of evaluation guaranteed to terminate whenever a particular expression is reducible to normal form?
  - Normal-order: leftmost, outermost reduction: no expression in the argument position of a redex is reduced until the redex is reduced
  - If there is a reduction from $A$ to $B$ and $B$ is in normal form, then there exists a normal order reduction from $A$ to $B$ (Church-Rosser Theorem 2)
Proof Rules: Aximoatic Semantics

- **Skip:**
  \[
  \{ P \} \text{skip} \{ P \}
  \]

- **Assignment:**
  \[
  \{ P[t/x] \} x := t \{ P \}
  \]
  Example: Suppose \( t = x + 1 \)
  Then, \( (x+1 = 2) \times := x + 1 \{ x = 2 \} \)

- **Sequencing:**
  \[
  \frac{(P) \ c_0 \ (Q) \ (Q) \ c_1 \ (Q)}{(P) \ c_0; \ c_1 \ (Q')}
  \]

Proof Rules (cont)

- **Conditionals:**
  \[
  \frac{(P \land b) \ c_0 \ (Q), (P \land \neg b) \ c_1 \ (Q)}{(P) \text{ if } b \text{ then } c_0 \text{ else } c_1 \ (Q)}
  \]

- **Loops:**
  \[
  \frac{(P \land b) \ c \ (P)}{(P) \text{ while } b \text{ do } c \ (P \land \neg b)}
  \]

- **Consequence:**
  \[
  \frac{\Gamma \vdash (P \Rightarrow P'), (P') \ c \ (Q') \ \Gamma \vdash (Q' \Rightarrow Q)}{(P) \ c \ (Q)}
  \]

  if \( \Gamma \vdash P \Rightarrow P' \) then all states \( \sigma \) which satisfy \( P \) also satisfy \( P' \). Rule allows strengthening of \( P \) to \( P' \) and weakening of \( Q' \) to \( Q \)

Syntax: \( F_1 \)

- **Terms**
  \[
  e ::= x \mid \lambda \cdot t . e \mid e_1 \ e_2 \\
  \mid n \mid e_1 + e_2 \mid \text{iszero } e \\
  \mid \text{true} \mid \text{false} \mid \text{not } e \\
  \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3
  \]

- **Types**
  \[
  \tau ::= \text{int} \mid \text{bool} \mid \tau_1 \rightarrow \tau_2
  \]

  \( \rightarrow \) is a function type constructor and associates to the right
  Formal arguments to functions have typing annotations

Static Semantics

- **The typing judgment**
  \[
  \Gamma \vdash e : \tau
  \]

- **Typing rules**
  \[
  \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau . e : \tau \rightarrow \tau'}
  \]
  \[
  \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \ e_2 : \tau_2}
  \]
Static Semantics (cont)

- More typing rules

\[ \Gamma \vdash n : \text{int} \quad \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \]
\[ \Gamma \vdash e_1 + e_2 : \text{int} \]

\[ \Gamma \vdash \text{true} : \text{bool} \quad \Gamma \vdash e : \text{bool} \]
\[ \Gamma \vdash \text{not } e : \text{bool} \]

\[ \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau, \Gamma \vdash e_3 : \tau \]
\[ \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau \]

Operational Semantics of $F_1$

- Evaluation relation

\[ e \Downarrow v \]

- Values

\[ v ::= n \mid \text{true} \mid \text{false} \mid \lambda x : \tau . e \]

- Call-by-value evaluation rules

\[ \lambda x : \tau . e \Downarrow \lambda x : \tau . e \]

\[ \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow [v_2/x]e_1' \Downarrow v \]
\[ e_2 \Downarrow v \]

Operational Semantics (cont)

More rules:

\[ e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n = n_1 + n_2 \]
\[ e_1 + e_2 \Downarrow n \]
\[ n \Downarrow n \]

\[ e_1 \Downarrow \text{true} \quad e_f \Downarrow v \]
\[ \text{if } e_1 \text{ then } e_1 \text{ else } e_f \Downarrow v \]

\[ e_1 \Downarrow \text{false} \quad e_f \Downarrow v \]
\[ \text{if } e_1 \text{ then } e_1 \text{ else } e_f \Downarrow v \]

Small-step contextual semantics

- Define redexes:

\[ r ::= n_1 + n_2 \mid \text{if } v \text{ then } e_1 \text{ else } e_2 \mid (\lambda x : \tau . e) \Downarrow \]

- Define contexts:

\[ E ::= [] \mid E + e_2 \mid n_1 + E \mid \]
\[ \text{if } E \text{ then } e_1 \text{ else } e_2 \mid \]
\[ E e_2 \mid (\lambda x : \tau . e) E \]

- Local reduction rules:

\[ n_1 + n_2 \rightarrow n_1 \Downarrow n_2 \]
\[ \text{if true then } e_1 \text{ else } e_2 \rightarrow e_1 \]
\[ \text{if false then } e_1 \text{ else } e_2 \rightarrow e_2 \]
\[ (\lambda x : \tau . e) v_2 \rightarrow [v_2/x]e_1 \]

- Global reduction rule:

\[ E[r] \rightarrow E[e] \text{ iff } \text{true} \rightarrow e \]
Static Semantics for Product Types: $F_1x$

- Extend the semantics with (binary) tuples:
  
  \[
  e :: = \ldots \mid (e_1, e_2) \mid \text{fst } e \mid \text{snd } e
  \]

- \[
  \tau ::= \ldots \mid \tau_1 \times \tau_2
  \]

- Same typing judgment $\Gamma \vdash e : \tau$

- \[
  \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}
  \]

- \[
  \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{fst } e : \tau_1}
  \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{snd } e : \tau_2}
  \]

Dynamic Semantics for Product Types

- New form of values:
  
  \[
  v :: = \ldots \mid (v_1, v_2)
  \]

- New (big-step) evaluation rules:

\[
\frac{e_1 \downarrow v_1 \quad e_2 \downarrow v_2}{(e_1, e_2) \downarrow (v_1, v_2)}
\]

\[
\frac{e \downarrow (v_1, v_2)}{\text{fst } e \downarrow v_1}
\]

\[
\frac{e \downarrow (v_1, v_2)}{\text{snd } e \downarrow v_2}
\]

Typing Rules for Sum Types

\[
\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \text{injl } e : \tau_1 + \tau_2}
\]

\[
\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \text{injr } e : \tau_1 + \tau_2}
\]

\[
\frac{\Gamma \vdash e_1 : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash e_1 : \tau}{\Gamma, y : \tau_2 \vdash e_2 : \tau}
\]

\[
\frac{\Gamma \vdash \text{case } e_1 \text{ of } \text{injl } x \Rightarrow e_1 \mid \text{injr } y \Rightarrow e_2 : \tau}{\Gamma \vdash \text{injl } e \downarrow \text{injr } v}
\]

\[
\frac{\text{injl } e \downarrow v \quad \text{injr } e \downarrow v}{e \downarrow v}
\]

\[
\frac{e \downarrow \text{injl } v \quad [v/x]e_1 \downarrow v}{\text{case } e \text{ of } \text{injl } x \Rightarrow e_1 \mid \text{injr } y \Rightarrow e_r \downarrow v}
\]

\[
\frac{e \downarrow \text{injr } v \quad [v/y]e_r \downarrow v}{\text{case } e \text{ of } \text{injl } x \Rightarrow e_1 \mid \text{injr } y \Rightarrow e_r \downarrow v}
\]

Dynamic Semantics

- New values:
  
  \[
  v :: = \ldots \mid \text{injl } v \mid \text{injr } v
  \]

- New evaluation rules:

\[
\frac{e \downarrow v}{\text{injl } e \downarrow \text{injr } v}
\]

\[
\frac{\text{case } e \text{ of } \text{injl } x \Rightarrow e_1 \mid \text{injr } y \Rightarrow e_r \downarrow v}{e \downarrow v}
\]
## Typing and Evaluation Rules

\[
e ::= ... | \text{fix } e
\]

\[
\Gamma \vdash e : \tau_1 \rightarrow \tau_2 \\
\Gamma \vdash \text{fix } e : \tau_1 \\
\frac{e \Downarrow \lambda \, x: \tau. e'}{\text{fix } e \Downarrow [(\text{fix } \lambda \, x: \tau. e')/x]e'}
\]

## Arrow Types

\[
\tau_1 \triangleleft \sigma_1 \quad \sigma_2 \triangleleft \tau_2 \\
\sigma_1 \rightarrow \sigma_2 \triangleleft \tau_1 \rightarrow \tau_2
\]

The subtype relation is contravariant in the type of the argument, and covariant in the type of the result.

Intuition: if we have a function \( f \) of type \( \sigma_1 \rightarrow \sigma_2 \), then we know \( f \) accepts elements of any subtype \( \tau_1 \triangleleft \sigma_1 \).

Since \( f \) returns elements of type \( \sigma_2 \), these results belong to any supertype \( \tau_2 \) of \( \sigma_2 \).

---

### Subsumption

\[
\Gamma \vdash e : \sigma, \sigma \triangleleft \tau \\
\frac{}{\Gamma \vdash e : \tau}
\]

This rule tells us that if \( \sigma \triangleleft \tau \), every element \( v \in \sigma \) is also an element of \( \tau \).

Thus, if our subtype relation defined \( \{ x: \text{Nat}, y : \text{Bool} \} \triangleleft \{ x: \text{Nat} \} \), then the subsumption rule would allow us to derive \( \{ x=0, y=true \} : \{ x: \text{Nat} \} \).

---

### Annotations

New syntactic forms:

\[
e ::= ... | \text{injl } e \text{ as } \tau | \text{injr } e \text{ as } \tau |
\]

\[
\text{case } e \text{ of } \text{injl } x \text{ as } \tau \Rightarrow e_1 \\
| \text{injr } y \text{ as } \tau \Rightarrow e_2
\]

\[
\tau ::= ... | \tau_1 + \tau_2
\]

New typing rules:

\[
\Gamma \vdash e : \tau_1 \\
\frac{}{\Gamma \vdash \text{injl } e : \tau_1 + \tau_2 : \tau_1 + \tau_2}
\]

\[
\Gamma \vdash e : \tau_2 \\
\frac{}{\Gamma \vdash \text{injr } e : \tau_1 + \tau_2 : \tau_1 + \tau_2}
\]
Subtyping References

- Try covariance:
  \[ \sigma <: \tau \]
  \[ \sigma \text{ref} <: \tau \text{ref} \]
- Assume \( \sigma <: \tau \):
  - The following holds:
    - \( x : \tau, y : \sigma \text{ref}, f : \sigma \rightarrow \text{int} \rightarrow y := x ; f(l y) \)
    - Unsound: \( f \) is called on \( \sigma \) but is defined only on \( \sigma \)
- If we want covariance of references we can recover type safety with a runtime check for each \( y := x \) assignment
  - The actual type of \( x \) matches the actual type of \( y \)
  - This is not a particularly good design.
  - Note: Java has covariant arrays

Another approach (contravariance):

- Assume \( \sigma <: \tau \):
  - The following holds:
    - \( x : \tau, y : \tau \text{ref}, f : \sigma \rightarrow \text{int} \rightarrow y := x ; f(l y) \)
    - Unsound: \( f \) is called on \( \sigma \) but is defined only on \( \sigma \)
- References can be used in two ways:
  - for reading, context expects a value of type \( \alpha \) but the reference yields a value of type \( \tau \)
  - for writing, the new value provided by the context will have type \( \alpha \). If the actual type of the reference is \( \tau \) then this value may be read and used as \( \alpha \). This will only be safe if \( \sigma <: \tau \).

Static Semantics of Recursive Types

- The typing rules are syntax directed
- Often, for syntactic simplicity, the fold and unfold operators are omitted
  - This makes type checking somewhat harder

Dynamics of Recursive Types

- We add a new form of value
  \[ v ::= \ldots | \text{fold}_{\mu t \cdot \tau} v \]
- The purpose of fold is to ensure that the value has the recursive type and not its unfolding
- The evaluation rules:
  - The folding annotations are for type checking only
  - They can be dropped after type checking
Untyped Programming

- We write $e$ for the conversion of the term $e$ to typed $\lambda$ calculus with recursive types ($F_\mu$)
  - The type of $e$ is $V = \mu t. t \to t$
- The conversion rules
  - $x = x$
  - $\lambda x. e = \text{fold}_V (\lambda x:V. e)$
  - $e_1 e_2 = (\text{unfold}_V e_1) e_2$
- Verify that
  1. $\vdash e : V$
  2. $e \Downarrow v$ if and only if $e \Downarrow v$

New Rules

- fold and unfold operations
  \[
  \sigma <: [X \mapsto \mu X.t] t \\
  \sigma <: \mu X.t \\
  [X \mapsto \mu X.t] t <: \tau
  \]
- Amber Rules
  \[
  \Sigma, X <: Y \vdash S <: T \\
  \Sigma \vdash \mu X.S <: \mu Y.T \\
  (X <: Y) \in \Sigma \\
  \Sigma \vdash X <: Y
  \]

Parametric Polymorphism: Types as Parameters (System F)

- We introduce type variables and allow expressions to have variable types
- We introduce polymorphic types
  \[
  \tau ::= b \mid \tau_1 \to \tau_2 \mid \forall \tau. \tau
  \]
  - $e ::= x \mid \lambda x:e \mid e_1 e_2 \mid \forall \tau. e \mid e[e]\$
  - $\forall \tau. e$ is type abstraction (or generalization)
  - $e[e]$ is type application (or instantiation)
- Examples:
  - $id = \forall t.\lambda x:t. x : \forall t.t \to t$
  - $id[int] = \lambda x:int. x : \text{int} \to \text{int}$
  - $id[bool] = \lambda x:bool. x : \text{bool} \to \text{bool}$
  - "id 5" is invalid. Use "id [int] 5" instead

Impredicative Polymorphism

- The typing rules:
  \[
  \begin{align*}
  x : \tau & \text{ in } \Gamma \\
  \Gamma \vdash x : \tau & \\
  \Gamma, x : \tau \vdash e : \tau' & \\
  \Gamma \vdash \lambda x : \tau. e : \tau \to \tau' & \\
  \Gamma \vdash e_1 : \tau \to \tau' & \\
  \Gamma \vdash e_2 : \tau & \\
  \Gamma \vdash e_1 e_2 : \tau' & \\
  \Gamma \vdash e : \tau & \\
  \Gamma \vdash \forall t.e : \forall t.\tau
  \end{align*}
  \]
Predicative Polymorphism

- Restriction: type variables can be instantiated only with monomorphic types
- This restriction can be expressed syntactically:
  \[ \tau ::= b \mid \tau_1 \times \tau_2 \mid \top \]
  \[ \alpha ::= \tau \mid \forall \tau. \alpha \mid \alpha_1 \rightarrow \alpha_2 \]
  \[ e ::= x \mid e_1 e_2 \mid \lambda \alpha. e \mid \Lambda \tau. e \mid e [\tau] \]
  - Type application is restricted to mono types
  - Cannot apply "id" to itself anymore

- Same typing rules
- Simple semantics and termination proof
- Type reconstruction still undecidable
- Must restrict further!

Prenex Predicative Polymorphism

- Restriction: polymorphic type constructor at top level only
- This restriction can also be expressed syntactically:
  \[ \tau ::= b \mid \tau_1 \times \tau_2 \mid \top \]
  \[ \alpha ::= \tau \mid \forall \tau. \alpha \]
  \[ e ::= x \mid e_1 e_2 \mid \lambda \alpha. e \mid \Lambda \tau. e \mid e [\tau] \]
  - Type application is restricted to mono types (i.e., predicative)
  - Abstraction only on mono types
  - The only occurrences of \( \forall \) are at the top level of a type
    \( (\forall \tau. t \rightarrow t) \rightarrow (\forall \tau. t \rightarrow t) \) is not a valid type

- Same typing rules
- Simple semantics and termination proof
- Decidable type inference!

ML’s Polymorphic Let

- ML solution: slight extension of the predicative \( F_2 \)
  - Introduce "let \( x : \alpha = e_1 \) in \( e_2 \)"
  - With the semantics of "(\( \alpha \mapsto \alpha \). \( e_1 \) )" \( e_2 \)"
  - And typed as "(\( e_1/x \) \( e_2 \) )"

  \[ \Gamma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \tau \]
  \[ \Gamma \vdash \text{let } x : \sigma = e_1 \text{ in } e_2 : \tau \]

- This lets us write the polymorphic sort as
  \[ \text{let } s : \forall \tau . \tau \mapsto \tau, \text{ code for polymorphic sort } \ldots \]
  \[ \text{in } \ldots \text{ s [nat] } \ldots \text{ s [bool] } y \]

- Surprise: this was a major ML design flaw!

The Value Restriction in ML

- A type in a let is generalized only for syntactic values:
  \[ \Gamma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \tau \]
  \[ \Gamma \vdash \text{let } x : \sigma = e_1 \text{ in } e_2 : \tau \]
  \( e_1 \) is a syntactic value or \( \sigma \) is monomorphic

- Since \( e_1 \) is a value, its evaluation cannot have side-effects
- In this case call-by-name and call-by-value are the same
- In the previous example ref (\( \lambda x : t. x \) ) is not a value
- This is not too restrictive in practice!
Existential Types

- Provides two views:
  - From the outside, the actual representation is hidden or opaque
  - From the inside, the object has the type defined by the representation
- Example:
  - The package: abstype p implements
    \[<a : p, f : p \rightarrow p>\]
    \[= \langle p = \text{Nat}, a = 0, f = \text{inc} \rangle\]
  - and has type \(\exists t. <a : t, f : t \rightarrow t>\)
- Note that a given existential type may be associated with many different implementations

Data Abstraction

- It is useful to separate the creation of the abstract type and its use
- Extend the syntax:
  - Terms ::= … | \(<t = \tau, e : \sigma>\) | open \(e_a\) as \(t, x : \sigma\) in \(e_b\)
  - Types ::= … | \(\exists t. \sigma\)
- The expression "\(<t = \tau, e : \sigma>\)" takes the concrete implementation \(e\) and "packs it" as a value of an abstract type \(\tau\) with hidden representation type \(\tau\) and actual type \(\sigma\)
- The "open" expression allows \(e_b\) to access the abstract type expression \(e_a\) using the name \(x\), the unknown type of the concrete implementation \(t\)

Typing Rules for Existential Types

- We add the following typing rules:

  \[\Gamma \vdash [\tau/t]e : [\tau/t]\sigma\]
  \[\Gamma \vdash \langle t = \tau, e : \sigma > : \exists t. \sigma\]
  \[\Gamma \vdash e_a : \exists t. \sigma\]
  \[\Gamma, t, p : \sigma \vdash e_b : t\]

  \[\Gamma \vdash \text{open } e_a \text{ as } t, p : \sigma \text{ in } e_b : t\]

- The restriction in the rule for "open" ensures that \(t\) does not escape its scope

Evaluation Rules for Abstract Types

- We add a new form of value

  \[v ::= \ldots | \langle \tau, v : \sigma >\]

  - This is just like \(v\) but with some type decorations that make it have an existential type
  \[e_a \Downarrow \langle t = \tau, v : \sigma > : \exists t. \sigma\]
  \[\Downarrow [v/x][\tau/t] e_b \Downarrow v'\]

  - At the open \(e_a\) as \(t, x : \sigma\) in \(e_b\) \Downarrow v'

  - If we ignore the type issues "open \(e_a\) as \(t, x : \sigma\) in \(e_b\)" is like "let \(x : \sigma = e_a\) in \(e_b\)"

  - What is different is that \(e_b\) cannot know statically what the concrete type of \(x\) is so it cannot take advantage of it