Rise of Lightweight Formal Methods

Don’t prove correctness: just find bugs ..
- model checking
- light specification and verification (e.g. ESC, SLAM ..)
- type-checking!

Basic ideas are long established; but industrial attitudes have been softened by the success of model checking in hardware design.

“Formal methods will never have any impact until they can be used by people that don’t understand them” : Tom Melham

What is a Type Systems?

A Type System is a
- tractable syntactic method
- for proving the absence of certain program behaviors
- by classifying phrases according to the kinds of values they compute

Why Type Systems?

Type systems are good for:
- detecting errors
- abstraction
- documentation
- language design
- efficiency
- safety
- etc.. (security, exception, theorem-proving, web-metadata, categorical grammar)
Pure Simply Typed Lambda Calculus

- **t ::=**
  - `x` variable
  - `λ x:T.t` abstraction
  - `t t` application

- **v ::=**
  - `λ x:T.t` abstraction value

- **T ::=**
  - `T → T` type of functions

- **Γ ::=**
  - `∅` empty context
  - `Γ, x:T` type variable binding

Typing

- `x:T ∈ Γ` (T-Var)
  - `Γ ⊢ x : T`

- `Γ, x:T1 ⊢ t2 : T2` (T-Abs)
  - `Γ ⊢ λ x:T1.t2 : T1 → T2`

- `Γ ⊢ t1 : T1 → T2`, `Γ ⊢ t2 : T1` (T-App)
  - `Γ ⊢ t1 t2 : T2`

Where are the Base Types?

- **T ::=**
  - `T → T` type of functions

Extend with uninterpreted base types, e.g.

- **T ::=**
  - `T → T` type of functions
  - `A` base type 1
  - `B` base type 2
  - `C` base type 3

Unit Type

New Syntax:

- `t ::=` terms
  - `unit` constant unit

- `v ::=` values
  - `unit` constant unit

- `T ::=` types
  - `Unit` unit type

Note that `Unit` type has only one possible value.

New Evaluation Rules: None

New Typing Rules:

- `Γ ⊢ unit : Unit` T-Unit
**Sequencing : Basic Idea**

Syntax : \( e_1; e_2 \)

Evaluate an expression (to achieve some side-effect, such as printing), ignore its result and then evaluate another expression.

Examples:

- \((\text{print } x); x+1\)
- \((\text{printcurrenttime}); \text{compute}; (\text{printcurrenttime})\)

**Lambda Calculus with Sequencing**

New Syntax

\[
\begin{align*}
\text{terms} & : \quad \ldots \\
\text{sequence} & : \quad t; t
\end{align*}
\]

New Evaluation Rules:

\[
\frac{t \rightarrow t'}{t_1; t_2 \rightarrow t_1' ; t_2} \quad \text{(E-Seq)}
\]

\[
\frac{t \rightarrow t'}{\text{unit} ; t \rightarrow t} \quad \text{(E-SeqUnit)}
\]

**Sequencing (cont)**

New Typing Rule:

\[
\frac{\Gamma \vdash t_1 : \text{Unit}_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1; t_2 : T_2} \quad \text{(T-Seq)}
\]

**Sequencing (Second Version)**

- Treat \( t_1; t_2 \) as an abbreviation for \( (\lambda x: \text{Unit}. t_2)\ t_1 \).
- Then the evaluation and typing rules for abstraction and application will take care of sequencing!
- Such shortcuts are called derived forms (or syntactic sugar) and are heavily used in programming language definition.
Equivalence of two Sequencing

Let $\lambda^E$ be the simply typed lambda calculus with the Unit type and the sequencing construct.

Let $\lambda^I$ be the simply-typed lambda calculus with Unit only.

Let $e \in \lambda^E \to \lambda^I$ be the elaboration function that translates from $\lambda^E$ to $\lambda^I$.

Then, we have for each term $t$:

- $t \to^E t'$ iff $e(t) \to_I e(t')$
- $\Gamma \vdash^E t : T$ iff $\Gamma \vdash^I e(t) : T$

Ascription : Motivation

Sometimes, we want to say explicitly that a term has a certain type.

Reasons:
- as comments for inline documentation
- for debugging type errors
- control printing of types (together with type syntax)
- casting (Chapter 15)
- resolve ambiguity (see later)

Ascription : Syntax

New Syntax

- $t ::= \ldots$ terms
  - $t$ as $T$ ascription

Example:

$$(f (g (h x y z)))$$

as $\text{Bool}$

Ascription (cont)

New Evaluation Rules:

- $v$ as $T \to v$ (E-Ascribe1)
- $t \to t'$
  - $t$ as $T \to t'$ as $T$ (E-Ascribe2)

New Typing Rules:

- $\Gamma \vdash t : T$
  - $\Gamma \vdash t$ as $T : T$ (T-Ascribe)
**Let Bindings : Motivation**

- Let expression allow us to give a name to the result of an expression for later use and reuse.

- Examples:

  let pi=<long computation> in …pi..pi..pi….

  let square = λ x:Nat. x*x in …(square 2)..(square 4)…

**Lambda Calculus with Let Binding**

New Syntax

```
t ::= … terms
      let x=t in t let binding
```

New Typing Rule:

\[
\frac{
\Gamma \vdash t_1 : T_1 \\
\Gamma, x : T_1 \vdash t_2 : T_2
}{
\Gamma \vdash \text{let } x=t_1 \text{ in } t_2 : T_2
}
\]  \hspace{1cm} (T-Let)

**Let Bindings as Derived Form**

We can consider let expressions as derived form:

In untyped setting:

let x=t_1 in t_2 abbreviates to \((\lambda x. t_2) t_1\)

In a typed setting:

let x=t_1 in t_2 abbreviates to \((\lambda x:\? t_2) t_1\)

How to get type declaration for the formal parameter?

Answer : Type inference (see later).

**Pairs : Motivation**

Pairs provide the simplest kind of data structures.

Examples:

\{9, 81\}

\(\lambda x : \text{Nat. } \{x, x*x\}\)
**Pairs : Syntax**

- \( t ::= \ldots \) terms
  - \( \{t, t\} \) variable
  - \( t.1 \) first projection
  - \( t.2 \) second projection

- \( v ::= \ldots \) value
  - \( \{v, v\} \) pair value

- \( T ::= \ldots \) types
  - \( T \times T \) product type

**Pairs : Typing Rules**

\[
\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \\
\Gamma \vdash \{t_1, t_2\} : T_1 \times T_2 \quad \text{(T-Pair)}
\]

\[
\Gamma \vdash t : T_1 \times T_2 \\
\Gamma \vdash t.1 : T_1 \quad \text{(T-Proj1)}
\]

\[
\Gamma \vdash t : T_1 \times T_2 \\
\Gamma \vdash t.2 : T_2 \quad \text{(T-Proj2)}
\]

---

**Tuples**

Tuples are a straightforward generalization of pairs, where \( n \) terms are combined in a tuple expression.

Example:

\[
\{1, \text{true}, \text{unit}\} : \{\text{Nat}, \text{Bool}, \text{Unit}\} \\
\{1,\{\text{true}, 0\}\} : \{\text{Nat}, \{\text{Bool}, \text{Nat}\}\} \\
\{\} : \{\}
\]

Note that \( n \) may be 0. Then the only value is \( \{\} \) with type \( \{\} \). Such a type is isomorphic to Unit.

---

**Records**

Sometimes it is better to have components labeled more meaningfully instead of via numbers 1..\( n \), as in tuples.

Tuples with labels are called records.

Example:

\[
\{\text{partno}=5524, \text{cost}=30.27, \text{instock }=\text{false}\} \\
\text{has type } \{\text{partno: Nat, cost: Float, instock: Bool}\}
\]

instead of:

\[
\{5524, 30.27, \text{false}\} : \{\text{Nat, Float, Bool}\}
\]
**Sums : Motivation**

Often, we may want to handle values of different structures with the same function.

Examples:

```
PhysicalAddr={firstlast:String, add:String}
VirtualAddr={name:String, email:String}
```

A sum type can then be written like this:

```
Addr = PhysicalAddr + VirtualAddr
```

**Sums : Motivation**

Given a sum type; e.g.

```
K = Nat + Bool
```

Need to use tags `inl` and `inr` to indicate that a value is a particular member of the sum type; e.g.

```
inl 5 : K but not inr 5 : K nor 5 : K
inr true : K
```

**Sums : Syntax**

- **t ::= ...** terms
  - `inl t as T` tagging (left)
  - `inr t as T` tagging (right)
  - `case t of {pi => ti}` pattern matching

- **v ::= ...** value
  - `inl v as T` tagged value (left)
  - `inr v as T` tagged value (right)

- **T ::= ...** types
  - `T + T` sum type
**Sums : Typing Rules**

\[
\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{inl } t_1 \text{ as } T_1 \times T_2 : T_1 \times T_2} \quad \text{(T-Inl)}
\]

\[
\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash \text{inr } t_2 \text{ as } T_1 \times T_2 : T_1 \times T_2} \quad \text{(T-Inr)}
\]

**Variants : Labeled Sums**

Instead of `inl` and `inr`, we may like to use nicer labels, just as in records. This is what *variants* do.

For types, instead of: \( T_1 + T_2 \)
we write: \( <l_1:T_1 + l_2:T_2> \)

For terms, instead of: \( \text{inl } r \text{ as } T_1 + T_2 \)
we write: \( <l_1 = t> \text{ as } <l_1:T_1 + l_2:T_2> \)

**Variants : Example**

An example using variant type:

\( \text{Addr} = <\text{physical:PhysicalAddr, virtual:VirtualAddr}> \)

A variant value:

\( a = <\text{physical=pa}> \text{ as Addr} \)

Function over variant value:

\( \text{getName} = \lambda a : \text{Addr.} \)

\[
\text{case } a \text{ of } \\
<\text{physical=x}> \Rightarrow x.\text{firstlast} \\
<\text{virtual=y}> \Rightarrow y.\text{name}
\]

**Application : Enumeration**

Enumeration are variants that only make use of their labels. Each possible value is unit.

\( \text{Weekday} = <\text{monday:Unit, tuesday:Unit, wednesday:Unit, thursday:Unit, friday:Unit}> \)
**Application : Single-Field Variants**

Labels can be convenient to add more information on how the value is used.

Example, currency denominations (or units):

\[\text{DollarAmount} = \langle \text{dollars:Float} \rangle\]

\[\text{EuroAmount} = \langle \text{euros:Float} \rangle\]

---

**Recursion : Motivation**

Recall the fix-point combinator:

\[\text{fix} = \lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y))\]

Unfortunately, it is not valid (well-typed) in simply typed lambda calculus.

Solution: provide this as a language *primitive* that is hardwired.

---

**Recursion : Syntax & Evaluation Rules**

**New Syntax**

- \( t ::= \ldots \)
- \( \text{fix } t \)
  - fixed point operator

**New Evaluation**

\[\text{fix } (\lambda x : T. t) \rightarrow [x \mapsto \text{fix } (\lambda x : T. t)] t\]  
  (E-FixBeta)

\[t \rightarrow t'\]

\[\text{fix } t \rightarrow \text{fix } t'\]  
  (E-Fix)

---

**Recursion : Typing Rules**

\[\Gamma \vdash t : T \rightarrow T\]

\[\Gamma \vdash \text{fix } t : T\]  
  (T-Fix)

Can you guess the inherent type of fix?
**References : Motivation**

Many languages do not syntactically distinguish between references (pointers) from their values.

In C, we write: \( x = x + 1 \)

For typing, it is useful to make this distinction explicit; since operationally *pointers* and *values* are different.

**Introduce the syntax (ref t), which returns a reference to the result of evaluating t. The type of ref t is Ref T, if T is the type of t.**

Remember that we have many type constructors already:

- Nat × float
- {partno: Nat, cost: float}
- Unit + Nat
- <none: Unit, some: Nat>

**Typing : First Attempt**

\[
\begin{align*}
\Gamma \vdash t : T & \quad (T-Ref) \\
\Gamma \vdash \text{ref } t : \text{Ref } T \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash t : \text{Ref } T & \quad (T-Deref) \\
\Gamma \vdash ! t : T \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash t_1 : \text{Ref } T_1 & \quad \Gamma \vdash t_2 : T_1 & \quad (T-Assign) \\
\Gamma \vdash t_1 := t_2 : \text{Unit} \\
\end{align*}
\]

**What should be the value of a reference?**

**What should the assignment “do”?**

How can we capture the difference of evaluating a dereferencing depending on the value of the reference?

**How do we capture side-effects of assignment?**
**References : Motivation**

Answer:

Introduce *locations* corresponding to references.

Introduce *stores* that map references to values.

Extend evaluation relation to work on stores.

**References : Evaluation**

Instead of:

\[ t \rightarrow t' \]

we now write:

\[ t | \mu \rightarrow t' | \mu' \]

where \( \mu' \) denotes the changed store.

**Evaluation of Application**

\[ (\lambda x : T.t) v | \mu \rightarrow [x \mapsto v] t | \mu \quad \text{(E-AppAbs)} \]

\[ t_1 | \mu \rightarrow t'_1 | \mu' \quad \text{(E-App1)} \]

\[ t_1 t_2 | \mu \rightarrow t'_1 t_2 | \mu' \quad \text{(E-App2)} \]

\[ t_2 | \mu \rightarrow t'_2 | \mu' \]

\[ v t_2 | \mu \rightarrow v t'_2 | \mu' \]

**Values**

The result of evaluating a ref expression is a location

- \( v ::= \)
- \( \lambda x : T.t \)
- \( \text{value} \)
- \( \text{abstraction value} \)
- \( \text{unit} \)
- \( \text{unit value} \)
- \( 1 \)
- \( \text{store location} \)
**Terms**

Below is the syntax for terms.

- $t ::= \text{terms}$
- $x$ variable
- $\lambda x:T.t$ abstraction value
- $\text{unit}$ constant unit
- $t t$ application
- $\text{ref } t$ reference creation
- $! t$ dereference
- $t := t$ assignment
- $l$ store location

**Evaluation of Deferencing**

\[
\begin{align*}
E-\text{AppI} & : t | \mu \rightarrow t' | \mu' \\
E-\text{DeRefLoc} & : ! t | \mu \rightarrow ! t' | \mu'
\end{align*}
\]

**Evaluation of Assignment**

\[
\begin{align*}
E-\text{Assign1} & : t_1 := t_2 | \mu \rightarrow t'_1 := t_2 | \mu' \\
E-\text{Assign2} & : t_2 | \mu \rightarrow t'_2 | \mu' \\
E-\text{Assign} & : l := v_2 | \mu \rightarrow \text{unit } | [l \mapsto v_2] \mu
\end{align*}
\]

**Evaluation of References**

\[
\begin{align*}
E-\text{Ref} & : \text{ref } t | \mu \rightarrow \text{ref } t' | \mu' \\
E-\text{RefV} & : l \not\in \text{dom}(\mu) \\
\end{align*}
\]

\[
\begin{align*}
\text{ref } v | \mu \rightarrow l | \mu, (l \mapsto v)
\end{align*}
\]
Towards a Typing for Locations

\[ \Gamma \vdash \mu(l) : T \]

\[ \Gamma \vdash l : \text{Ref } T \]

(T-Ref)

But where does \( \mu \) come from?

How about adding store to the typing relation

\[ \Gamma \vdash \mu(l) : T \]

\[ \Gamma \vdash l : \text{Ref } T \]

(T-Ref)

..but store is a runtime entity

Idea

Instead of adding stores as argument to the typing relation, we add store typings, which are mappings from locations to types.

Example for a store typing:

\[ \Sigma = (l_1 \mapsto \text{Nat} \rightarrow \text{Nat}, l_2 \mapsto \text{Nat} \rightarrow \text{Nat}, l_3 \mapsto \text{Unit}) \]

Typing: Final

\[ \Sigma(l) = T \]

\[ \Gamma \vdash l : \text{Ref } T \]

(T-Loc)

\[ \Gamma \vdash t : T \]

\[ \Gamma \vdash \text{ref } t : \text{Ref } T \]

(T-Ref)

\[ \Gamma \vdash ! t : T \]

(T-Deref)

\[ \Gamma \vdash t_1 : \text{Ref } T_1 \]

\[ \Gamma \vdash t_2 : T_1 \]

\[ \Gamma \vdash t_1 := t_2 : \text{Unit} \]

(T-Assign)
Exceptions : Motivation

During execution, situations may occur that requires drastic measures such as resetting the state of program or even aborting the program.

- division by zero
- arithmetic overflow
- array index out of bounds
- ...

Errors

We can denote error explicitly:

\[ t ::= \ldots \text{ terms} \]
\[ \text{error} \quad \text{run-time error} \]

Evaluation Rules:

\[ \text{error } t \rightarrow \text{error} \quad (E-\text{AppErr1}) \]
\[ v \text{ error } ightarrow \text{error} \quad (E-\text{AppErr2}) \]

Typing Rule:

\[ \Gamma \vdash \text{error} : T \quad (T-\text{Error}) \]

for any \( T \)

Examples

\((\lambda \ x: \text{Nat}. \ 0) \ \text{error}\)

\((\text{fix} \ (\lambda \ x: \text{Nat}. \ x)) \ \text{error}\)

\((\lambda \ x: \text{Bool}. \ x) \ \text{error}\)

\((\lambda \ x: \text{Bool}. \ x) \ (\text{error} \ \text{true})\)

Error Handling : Motivation

In implementation, the evaluation of error will force the runtime stack to be cleared so that program releases its computational resources.

Idea of error handling : Install a marker on the stack. During clearing of stack frames, the markers can be checked and when the right one is found, execution can resume normally.
**Error Handling**

Provide a try-with (similar to try-catch of Java) mechanism.

\[
\begin{align*}
t ::= & \ldots \text{ terms} \\
\text{try } t \text{ with } t & \quad \text{trap errors}
\end{align*}
\]

New Evaluation Rules:

\[
\begin{align*}
\text{try } v \text{ with } t & \rightarrow v \\
\text{try error with } t & \rightarrow t
\end{align*}
\]

\[
E-TryV)
\]

\[
E-TryError)
\]

\[
E-Try)
\]

**Typing Error Handling**

New Typing Rule:

\[
\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T}{\Gamma \vdash \text{try } t_1 \text{ with } t_2 : T} \quad (E-Try)
\]

**Exception Carrying Values: Motivation**

Typically, we would like to know what kind of exceptional situation has occurred in order to take appropriate action.

*Idea*: Instead of errors, raise exception value that can be examined after trapping. This technique is called exception handling.
### Exception Carrying Values

New syntax:

$$
t ::= \ldots \text{ terms} \\
\text{raise } t \quad \text{raise exception} \\
\text{try } t \text{ with } t \quad \text{handle exception}
$$

New Evaluation Rules:

$$
\text{(raise } v) \ t \rightarrow \text{raise } v \\
[\text{(E-AppRaise1)}] \\
\text{v}_1 \ (\text{raise } v_2) \rightarrow (\text{raise } v_2) \\
[\text{(E-AppRaise1)}] \\
(\text{raise } (\text{raise } v)) \rightarrow (\text{raise } v) \\
[\text{(E-AppRaiseRaise)}] \\
\frac{t \rightarrow t'}{\text{raise } t \rightarrow \text{raise } t'} \\
[\text{(E-Raise)}]
$$

New Evaluation Rules:

$$
\text{try } t_1 \text{ with } t_2 \rightarrow \text{try } t_1 \text{ with } t_2 \\
[\text{(E-Try)}] \\
\text{try } (\text{raise } v) \text{ with } t \rightarrow t \text{ v} \\
[\text{(E-TryError)}] \\
\frac{t_1 \rightarrow t_1'}{\text{try } t_1 \text{ with } t_2 \rightarrow \text{try } t_1' \text{ with } t_2} \\
[\text{(E-Try)}]
$$

New Typing Rules:

$$
\frac{\Gamma \vdash t : \text{T}_{\text{exn}}}{\Gamma \vdash \text{raise } t : T} \\
[\text{(E-Raise)}] \\
\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : \text{T}_{\text{exn}} \rightarrow T}{\Gamma \vdash \text{try } t_1 \text{ with } t_2 : T} \\
[\text{(E-Try)}]
$$
**What Values can serve as Exceptions?**

- $\text{T}_{\text{exn}}$ is Nat as in return codes for Unix system calls.
- $\text{T}_{\text{exn}}$ is String for convenience in printing out messages.
- $\text{T}_{\text{exn}}$ is a certain fixed variant type, such as:
  
  ```
  < divideByZero : Unit,
  overflow : Unit,
  fileNotFound : String
  >
  ```

  ... 

**O’Caml Exceptions**

- Exceptions are a special *extensible* variant type.
- Syntax (exception l of T) does variant extension.
- Syntax (raise l(t)) is short for: raise (\langle l=t \rangle) as $\text{T}_{\text{exn}}$
- Syntax of try is sugar for try and case.

**Motivation : Subtyping**

Typing for application:

\[
\Gamma \vdash t_1 : \text{T}_1 \rightarrow \text{T}_2 \quad \Gamma \vdash t_2 : \text{T}_1
\]

\[\Gamma \vdash t_1 \ t_2 : \text{T}_2\]  

(T-App)

Consider the term:

\[
(\lambda \ r : \{x:\text{Nat}\}. \ r.x) \ \{x=0, y=1\}
\]

But note that:

\[
\{x:\text{Nat}, y:\text{Nat}\} \not\subseteq \{x:\text{Nat}\}
\]

**Subtyping Relation**

Idea : Introduce a subtyping relation $\langle$:

\[\text{S} \langle \text{T} \quad \text{means that every value described by S is also described by T.}
\]

When we view types as *sets of values*, we can say that S is a *subset* of T.
Subsumption Rule

\[ \frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T} \quad \text{(T-Sub)} \]

If we define \(<_\text{S}\) such that \{x: \text{Nat}, y: \text{Nat}\} <: \{x: \text{Nat}\}

We can obtain:

\[ \Gamma \vdash \{x=0, y=1\} : \{x: \text{Nat}, y: \text{Nat}\} \quad \{x: \text{Nat}, y: \text{Nat}\} <: \{x: \text{Nat}\} \]

\[ \Gamma \vdash \{x=0, y=1\} : \{x: \text{Nat}\} \]

General Rules for Subtyping

Subtyping should be a pre-order:

\[ S <: S \text{ for all types } S \quad \text{(S-Refl)} \]

\[ S <: U \quad U <: T \quad \Gamma \vdash t : T \]

\[ S <: T \quad \text{(S-Trans)} \]

Subtyping of Records

\[ \{i_i : T_i\}_{i \in 1..k} <: \{i_i : T_i\}_{i \in 1..n} \quad \text{(S-RedWidth)} \]

\[ S_i <: T_i \quad \forall i \in 1..n \]

\[ \{i_i : S_i\}_{i \in 1..k} <: \{i_i : T_i\}_{i \in 1..n} \quad \text{(S-RedDepth)} \]

Example

\[ \vdash \{a: \text{Nat}, b: \text{Nat}\} <: \{a: \text{Nat}\} \quad \vdash \{m: \text{Nat}\} <: \{\} \quad \text{(S-RedWidth)} \]

\[ \vdash \{x: \{a: \text{Nat}, b: \text{Nat}\}, y: \{m: \text{Nat}\}\} <: \{x: \{a: \text{Nat}\}, y: \{\}\} \quad \text{(S-RedDepth)} \]
**Record Permutation**

Orders of fields in records should be unimportant.

\[ \{b: \text{Bool}, a: \text{Nat}\} \leq \{a: \text{Nat}, b: \text{Bool}\} \]

\[ \{a: \text{Nat}, b: \text{Bool}\} \leq \{b: \text{Bool}, a: \text{Nat}\} \]

Hence \( \leq \) is not a partial-order.

\[ \{k_i : S_i\}_{i \in 1..n} \text{ is a permutation of } \{l_i : T_i\}_{i \in 1..n} \]

\[ \{k_i : S_i\}_{i \in 1..n} \leq \{l_i : T_i\}_{i \in 1..n} \]

(S-RedPerm)

**Subtyping Functions**

Subtyping is *contravariant* in the argument type and *covariant* in the result type.

\[ T_1 \leq S_1 \quad S_2 \leq T_2 \]

\[ S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2 \]

(S-Arrow)

**Contravariance of Argument Types**

Consider a function \( f \) of type: \( S_1 \rightarrow S_2 \)

Consider some type \( T_1 \leq S_1 \). It is clear that \( f \) accepts all elements of \( T_1 \) as argument. Therefore \( f \) should also be of type \( T_1 \rightarrow S_2 \).

\[ f :: T_1 \rightarrow S_2 \]

**Covariance of Result Types**

Consider a function \( f \) of type: \( S_1 \rightarrow S_2 \)

Consider some type \( T_2 \) such that \( S_2 \leq T_2 \). It is clear that \( f \) returns only values of type \( T_2 \). Therefore \( f \) should also be of type \( S_1 \rightarrow T_2 \).

\[ f :: S_1 \rightarrow T_2 \]
**Top**

Introduce a type $\text{Top}$ that is the supertype of every type.

$$S <: \text{Top} \quad \text{for every type } S$$

While $\text{Top}$ is not crucial for typed lambda calculus with subtyping, it has the following advantages:

- corresponds to Object in existing languages
- convenient for subtyping and polymorphism

**Bottom**

Sometimes also useful to add a $\text{Bot}$ type such that:

$$\text{Bot} <: T \quad \text{for every type } T$$

Note that $\text{Bot}$ is empty; as there is no value of this type. If such a value $v$ exist, we would have:

\[
\begin{align*}
\vdash v : \text{Bot} & \quad \text{Bot} <: \text{Top} \\
\vdash v : \text{Bot} & \quad \text{Bot} <: \{\} \\
\vdash v : \text{Top} & \\
\vdash v : \{\} \\
\end{align*}
\]

contradicts canonical form lemma!

**Subtyping of Extensions**

- Ascription and Casting
- Variants
- Lists
- References
- Arrays

**Subtyping and Ascription**

Let us consider expressions of the form $(t \text{ as } T)$

- *Up-casting* means that $T$ is a supertype of the “usual” type of $t$.
- *Down-casting* means that $T$ is a subtype of the “usual” type of $t$.
**Up-Casting**

Up-casting is always safe as implied by subsumption.

\[
\Gamma \vdash t : S \\
S \subseteq T \\
\hline
\Gamma \vdash t : T \\
\hline
\Gamma \vdash t \text{ as } T : T \\
\text{(T-Ascribe)}
\]

**Down-Casting**

Down-casting is to assign a more specific type to a term. The programmer forces the type on the term. The type checker just swallows such claims.

\[
\Gamma \vdash t : S \\
\Gamma \vdash t \text{ as } T : T \\
\text{(T-Downcast)}
\]

Note that stupid-casting is possible.

**Problems with Down-Casting**

With the usual evaluation rule:

\[
v \text{ as } T \rightarrow v
\]

We lose preservation. Need to add a runtime type test as follows:

\[
\vdash v : T \\
v \text{ as } T \rightarrow v \\
\text{(E-DownCast)}
\]

**Variant Subtyping**

Similar to record subtyping, except that the subtyping rule S-VariantWidth is reversed:

\[
<l_i : T_i>_{i \in 1..n} \subset <l_i : T_i>_{i \in 1..n+k} \\
\text{(S-VariantWidth)}
\]

More labels makes the variant bigger in set framework.
**List Subtyping**

List are also co-variant, thus:

\[
\begin{align*}
S & \ll T \\
\text{List } S & \ll \text{ List } T
\end{align*}
\]

**References**

References of the form \( r=\text{ref } v \) are used in two ways:

- for assignment \( r:=t \), similar to arguments of functions:
  
  \[
  := : \text{Ref } T \to T \to ()
  \]

- for dereferencing \( !r \), similar to return values of functions:
  
  \[
  ! : \text{Ref } T \to T
  \]

**References : Assignment**

Let \( r=\text{ref } v \) be of type \( \text{Ref } S \).

Say we have an assignment \( r:=v' \).

We must insist that \( v' \) is a subtype of \( S \), because subsequent dereferencing needs to produce values of type \( S \). Thus:

\[
\begin{align*}
T & \ll S \\
\text{Ref } S & \ll \text{Ref } T
\end{align*}
\]

**References : Dereferencing**

Let \( r=\text{ref } v \) be of type \( \text{Ref } S \).

Say we have a dereferencing \( !r \).

The dereferencing may be used whenever a supertype of \( S \) is required. Thus:

\[
\begin{align*}
S & \ll T \\
\text{Ref } S & \ll \text{Ref } T
\end{align*}
\]
**References: Invariant Typing**

The result is an *invariant subtyping* of references.

\[
\begin{align*}
S \ll T & \quad T \ll S \\
\text{Ref } S & \ll \text{Ref } T
\end{align*}
\]

In other words:
\[\text{contravariance } + \text{covariance } = \text{invariance}\]

**Array Subtyping**

Similar to references since elements of assignment and dereferencing also present.

Invariant subtyping:

\[
\begin{align*}
S \ll T & \quad T \ll S \\
\text{Array } S & \ll \text{Array } T
\end{align*}
\]

**Array Typing in Java**

Java allows covariant subtyping of arrays:

\[
\begin{align*}
S \ll T & \\
\text{Array } S & \ll \text{Array } T
\end{align*}
\]

This is considered to be a design flaw of Java, because it necessitates runtime type checks.

**Java Example**

class Vehicle {int speed;}
class Motorcycle extends Vehicle {int enginecc;}
Motorcycle[] myBikes = new Motorcycle[10]
Vehicle[] myVehicles = myBikes;
myVehicles[0] = new Vehicle();  // ArrayStoreException
**Intersection Types**

The members of intersection type $T_1 \land T_2$ are members of both $T_1$ and of $T_2$. It can be used where either $T_1$ or $T_2$ is expected.

\[ T_1 \land T_2 \Rightarrow T_1 \]
\[ T_1 \land T_2 \Rightarrow T_2 \]
\[
\frac{S \Rightarrow T_1 \quad S \Rightarrow T_2}{S \Rightarrow T_1 \land T_2}
\]

**Intersection Type and Function**

If we know that a term has the function type of both $S \rightarrow T_1$ and $S \rightarrow T_2$, then we can pass it an $S$ and expect to get back a value that is both a $T_1$ and a $T_2$.

\[ S \rightarrow T_1 \land S \rightarrow T_2 \Rightarrow S \rightarrow T_1 \land T_2 \]

**Intersection for Finitary Overloading**

We can use intersection to denote the type of overloaded functions.

For example, the $+$ operator can be applied to a pair of integers and floats, and return corresponding results. Such an overloaded operator can be typed as follows:

\[ \Gamma \vdash + : (Nat \rightarrow Nat \rightarrow Nat) \land (Float \rightarrow Float \rightarrow Float) \]

**Union Types**

Union type $T_1 \lor T_2$ simply denote the *ordinary union* of set of values belonging to both $T_1$ and $T_2$.

This differs from sum/variant types which add tags to identify the origin of a given element. Tagged union is also known as *disjoint union*.

\[ T_1 \Rightarrow T_1 \lor T_2 \]
\[ T_2 \Rightarrow T_1 \lor T_2 \]
\[
\frac{T_1 \Rightarrow S \quad T_2 \Rightarrow S}{T_1 \lor T_2 \Rightarrow S}
\]