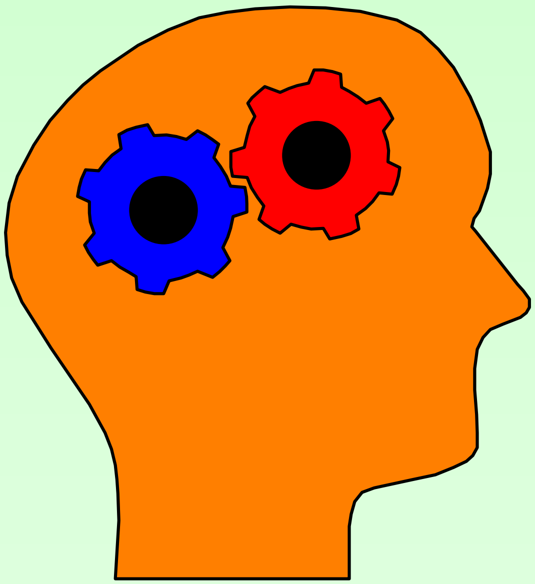
CS6202: Advanced Topics in Programming Languages and Systems

Lecture 4-5 : **Types**



"Types for Extended Lambda Calculus"

Lecturer : Chin Wei Ngan

Email : chinwn@comp.nus.edu.sg

Office : S15 06-01

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 grammer)

Pure Simply Typed Lambda Calculus

• t ::=		terms
	x	variable
	λ x:T.t	abstraction
	t t	application
• v ::=		value
	λ x :T.t	abstraction value
• T ::=		types
	$T \to T$	type of functions
• Γ ::=		contexts
	$\varnothing$	empty context
	Γ, x:T	type variable binding

Typing

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-Var)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\, t_2 : T_2} \quad \text{(T-App)}$$

Where are the Base Types?

- T ::= types
  - $T \rightarrow T$ type of functions

Extend with uninterpreted base types, e.g.

- T ::= types
  - $T \rightarrow T$ 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 :

$\Gamma \vdash$ unit : Unit   T-Unit

Sequencing : Basic Idea

Syntax : e1; e2

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

Examples:

(print x); x+1

(printcurrenttime); compute; (printcurrenttime)

Lambda Calculus with Sequencing

New Syntax

- t ::= … terms
  t ; t sequence

New Evaluation Rules:

$$\frac{t_1 \rightarrow t'_1}{t_1 \,;\, t_2 \rightarrow t'_1 \,;\, t_2} \quad \text{(E-Seq)}$$

$$\text{unit} \,;\, t \;\rightarrow\; t \quad \text{(E-SeqUnit)}$$

Sequencing (cont)

New Typing Rule:

$$\frac{\Gamma \vdash t_1 : \mathrm{Unit}_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 \,;\, t_2 : T_2} \quad \text{(T-Seq)}$$

Sequencing (Second Version)

- Treat $t_1;t_2$ as an abbreviation for $(\lambda x{:}\mathrm{Unit}.\ t_2)\ t_1$.
- 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 λ^E be the simply typed lambda calculus with the Unit type and the sequencing construct.

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

Let $e \in \lambda^E \rightarrow \lambda^I$ be the *elaboration* function that translates from λ^E To λ^I.

Then, we have for each term t:

- $t \rightarrow_E t'$ iff $e(t) \rightarrow_I e(t')$
- $\Gamma \vdash^E t{:}T$ 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 ::=	…	terms
	t as T	ascription

Example:

(f (g (h x y z))) as Bool

Ascription (cont)

New Evaluation Rules:

$$v \text{ as } T \rightarrow v \qquad \text{(E-Ascribe1)}$$

$$\frac{t \rightarrow t'}{t \text{ as } T \rightarrow t' \text{ as } T} \qquad \text{(E- Ascribe2)}$$

New Typing Rules:

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t \text{ as } T : T} \qquad \text{(T-Ascribe)}$$

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:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x{=}t_1 \text{ in } t_2 : T_2} \quad \text{(T-Let)}$$

Let Bindings as Derived Form

We can consider let expressions as derived form:

In untyped setting:

$\text{let } x = t_1 \text{ in } t_2$ abbreviates to $(\lambda\, x.\ t_2)\ t_1$

In a typed setting:

$\text{let } x = t_1 \text{ in } t_2$ 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}

λ x : Nat. {x, x*x}

Pairs : Syntax

• t ::=	…	terms
	{t, t}	variable
	t.1	first projection
	t.2	second projection
• v ::=	…	value
	{v, v}	pair value
• T ::=	…	types
	$T \times T$	product type

Pairs : Typing Rules

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

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

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

Tuples

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

Example:

{1, true, unit}	: {Nat, Bool, Unit}
{1,{true, 0}}	: {Nat, {Bool, 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:

{partno=5524,cost=30.27,instock =false}

has type {partno:Nat, cost:Float, instock:Bool}

instead of:

{5524,30.27,false} : {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 : Motivation

Given the address type:

Addr = PhysicalAddr + VirtualAddr

We can use case construct to analyse the particular value of sum type:

```
getName = λ a : Addr. case a of
                inl x => a.firstlast
                inr y => y.name
```

Sums : Syntax

•	t ::=	…	terms
		inl t as T	tagging (left)
		inr t as T	tagging (right)
		case t of $\{p_i \Rightarrow t_i\}$	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

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{inl } t_1 \text{ as } T_1 \times T_2 : T_1 \times T_2} \qquad \text{(T-Inl)}$$

$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash \text{inr } t_2 \text{ as } T_1 \times T_2 : T_1 \times T_2} \qquad \text{(T-Inr)}$$

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: $\langle l_1{:}T_1 + l_2{:}T_2 \rangle$

For terms, instead of: $\text{inl } r \text{ as } T_1 + T_2$

we write: $\langle l_1 = t \rangle \text{ as } \langle l_1{:}T_1 + l_2{:}T_2 \rangle$

Variants : Example

An example using variant type:

Addr =<physical:PhysicalAddr, virtual:VirtualAddr>

A variant value:

a = <physical=pa> as Addr

Function over variant value:

getName = λ a : Addr.
    case a of
        <physical=x> ⇒ x.firstlast
        <virtual=y> ⇒ y.name

Application : Enumeration

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

Weekday =<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):

DollarAmount =<dollars:Float>

EuroAmount =<euros:Float>

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 ::= … terms
  fix t fixed point operator

t ::=	…	terms
	fix t	fixed point operator

New Evaluation

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

$$\frac{t \rightarrow t'}{\text{fix } t \rightarrow \text{fix } t'} \qquad \text{(E-Fix)}$$

Recursion : Typing Rules

$$\frac{\Gamma \vdash t : T \to T}{\Gamma \vdash \text{fix } t : T} \quad \text{(T-Fix)}$$

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.

References : Motivation

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 $\times$ float	{partno:Nat,cost:float}
Unit+Nat	<none:Unit,some:Nat>

Typing : First Attempt

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \text{ref } t : \text{Ref } T} \qquad \text{(T-Ref)}$$

$$\frac{\Gamma \vdash t : \text{Ref } T}{\Gamma \vdash\ !t : T} \qquad \text{(T-Deref)}$$

$$\frac{\Gamma \vdash t_1 : \text{Ref } T_1 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 := t_2 : \text{Unit}} \qquad \text{(T-Assign)}$$

References : Motivation

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:

$$t \to t'$$

we now write:

$$t \mid \mu \;\to\; t' \mid \mu'$$

where μ' denotes the changed store.

Evaluation of Application

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

$$\frac{t_1 \mid \mu \ \rightarrow\ t'_1 \mid \mu'}{t_1\ t_2 \mid \mu \rightarrow t'_1\ t_2 \mid \mu'} \qquad \text{(E-Appl1)}$$

$$\frac{t_2 \mid \mu \ \rightarrow\ t'_2 \mid \mu'}{v\ t_2 \mid \mu \rightarrow v\ t'_2 \mid \mu'} \qquad \text{(E-Appl2)}$$

Values

The result of evaluating a ref expression is a location

• v ::=		value
	λ x:T.t	abstraction value
	unit	unit value
	l	store location

Terms

Below is the syntax for terms.

• t ::=	terms
x	variable
λ x:T.t	abstraction value
unit	constant unit
t t	application
ref t	reference creation
! t	dereference
t := t	assignment
l	store location

Evaluation of Deferencing

$$\frac{t \mid \mu \;\rightarrow\; t' \mid \mu'}{!\, t \mid \mu \rightarrow\, !\, t' \mid \mu'} \qquad \text{(E-Appl1)}$$

$$\frac{\mu\,(l) = v}{!\, l \mid \mu \rightarrow v \mid \mu} \qquad \text{(E-DeRefLoc)}$$

Evaluation of Assignment

$$\frac{t_1 \mid \mu \;\rightarrow\; t'_1 \mid \mu'}{t_1 := t_2 \mid \mu \rightarrow t'_1 := t_2 \mid \mu'} \qquad \text{(E-Assign1)}$$

$$\frac{t_2 \mid \mu \;\rightarrow\; t'_2 \mid \mu'}{l := t_2 \mid \mu \rightarrow l := t'_2 \mid \mu'} \qquad \text{(E-Assign2)}$$

$$l{:=}v_2 \mid \mu \;\rightarrow \text{unit} \mid [l \mapsto v_2]\,\mu \qquad \text{(E-Assign)}$$

Evaluation of References

$$\frac{t \mid \mu \rightarrow t' \mid \mu'}{\text{ref } t \mid \mu \rightarrow \text{ref } t' \mid \mu'} \qquad \text{(E-Ref)}$$

$$\frac{l \notin dom(\mu)}{\text{ref } v \mid \mu \rightarrow l \mid \mu, (l \mapsto v)} \qquad \text{(E-RefV)}$$

Towards a Typing for Locations

$$\frac{\Gamma \vdash \mu(l) : T}{\Gamma \vdash l : \text{Ref } T} \qquad \text{(T-Ref)}$$

But where does μ come from?
How about adding store to the typing relation

$$\frac{\Gamma \mid \mu \vdash \mu(l) : T}{\Gamma \mid \mu \vdash l : \text{Ref } T} \qquad \text{(T-Ref)}$$

..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:

$$\Sigma = (l_1 \mapsto \mathrm{Nat} \to \mathrm{Nat},\ l_2 \mapsto \mathrm{Nat} \to \mathrm{Nat},\ l_3 \mapsto \mathrm{Unit})$$

Typing : Final

$$\frac{\Sigma(l) = T}{\Gamma \mid \Sigma \vdash l : \text{Ref } T} \qquad \text{(T-Loc)}$$

$$\frac{\Gamma \mid \Sigma \vdash t : T}{\Gamma \mid \Sigma \vdash \text{ref } t : \text{Ref } T} \qquad \text{(T-Ref)}$$

Typing : Final

$$\frac{\Gamma \mid \Sigma \vdash t : \text{Ref } T}{\Gamma \mid \Sigma \vdash \; !t : T} \quad \text{(T-Deref)}$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Ref } T_1 \qquad \Gamma \mid \Sigma \vdash t_2 : T_1}{\Gamma \mid \Sigma \vdash t_1 := t_2 : \text{Unit}} \quad \text{(T-Assign)}$$

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 ::=	…	terms
	error	run-time error

Evaluation Rules:

$$\text{error } t \ \rightarrow\ \text{error} \qquad \text{(E-AppErr1)}$$

$$v \text{ error} \ \rightarrow\ \text{error} \qquad \text{(E-AppErr2)}$$

Typing Rule:

$$\Gamma \vdash \text{error} : T \qquad \text{(T-Error)}$$

for any T

Examples

(λ x:Nat. 0) error

(fix (λ x:Nat. x)) error

(λ x:Bool. x) error

(λ x:Bool. x) (error 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

```
display
  try
    (..complicated calculation..)
  with
    “cannot compute the result”
```

Error Handling

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

t ::=	…	terms
	try t with t	trap errors

New Evaluation Rules:

$$\text{try } v \text{ with } t \;\rightarrow\; v \qquad \text{(E-TryV)}$$

$$\text{try error with } t \rightarrow t \qquad \text{(E-TryError)}$$

$$\frac{t_1 \;\rightarrow\; t'_1}{\text{try } t_1 \text{ with } t_2 \rightarrow \text{try } t'_1 \text{ with } t_2} \qquad \text{(E-Try)}$$

Typing Error Handling

New Typing Rule:

$$\frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash \text{try } t_1 \text{ with } t_2 : T} \qquad \text{(E-Try)}$$

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 Value

New syntax:

t ::=	…	terms
	raise t	raise exception
	try t with t	handle exception

Exception Carrying Values

New Evaluation Rules:

$$\text{(raise v) t} \rightarrow \text{raise v} \qquad \text{(E-AppRaise1)}$$

$$v_1 \text{ (raise } v_2) \rightarrow \text{(raise } v_2) \qquad \text{(E-AppRaise1)}$$

$$\text{(raise (raise v))} \rightarrow \text{(raise v)} \qquad \text{(E-AppRaiseRaise)}$$

$$\frac{t \rightarrow t'}{\text{raise t} \rightarrow \text{raise t'}} \qquad \text{(E-Raise)}$$

Exception Carrying Values

New Evaluation Rules:

$$\text{try v with t} \rightarrow \text{v} \qquad \text{(E-TryV)}$$

$$\text{try (raise v) with t} \rightarrow \text{t v} \qquad \text{(E-TryError)}$$

$$\frac{t_1 \rightarrow t'_1}{\text{try } t_1 \text{ with } t_2 \rightarrow \text{try } t'_1 \text{ with } t_2} \qquad \text{(E-Try)}$$

Exception Carrying Values

New Typing Rules:

$$\frac{\Gamma \vdash t : T_{exn}}{\Gamma \vdash \text{raise } t : T} \quad \text{(E-Raise)}$$

$$\frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T_{exn} \rightarrow T}{\Gamma \vdash \text{try } t_1 \text{ with } t_2 : T} \quad \text{(E-Try)}$$

What Values can serve as Exceptions?

- T_{exn} is Nat as in return codes for Unix system calls.
- T_{exn} is String for convenience in printing out messages.
- T_{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 (<l=t>) as T_{exn}

- Syntax of try is sugar for try and case.

Motivation : Subtyping

Typing for application :

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2} \quad \text{(T-App)}$$

Consider the term:

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

But note that:

$\{x{:}Nat,\ y{:}Nat\} \neq \{x{:}Nat\}$

Subtyping Relation

Idea : Introduce a subtyping relation <:

S <: T 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

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

If we define <: such that $\{x{:}Nat,\ y{:}Nat\} <: \{x{:}Nat\}$

We can obtain:

$$\frac{\Gamma \vdash \{x{=}0,\ y{=}1\} : \{x{:}Nat,\ y{:}Nat\} \qquad \{x{:}Nat,\ y{:}Nat\} <: \{x{:}Nat\}}{\Gamma \vdash \{x{=}0,\ y{=}1\} : \{x{:}Nat\}}$$

General Rules for Subtyping

Subtyping should be a *pre-order*:

S <: S for all types S (S-Refl)

$$\frac{S <: U \qquad U <: T}{S <: T} \quad \text{(S-Trans)}$$

Subtyping of Records

$$\{l_i : T_i\}^{i \in 1..n+k} <: \{l_i : T_i\}^{i \in 1..n} \qquad \text{(S-RcdWidth)}$$

$$\frac{S_i <: T_i \quad \forall\, i \in 1..n}{\{l_i : S_i\}^{i \in 1..n} <: \{l_i : T_i\}^{i \in 1..n}} \qquad \text{(S-RcdDepth)}$$

Example

$$\frac{}{\vdash \{a{:}Nat, b{:}Nat\} <: \{a{:}Nat\}} \qquad \frac{}{\vdash \{m{:}Nat\} <: \{\}} \qquad \text{(S-RcdWidth)}$$

$$\frac{}{\vdash \{x{:}\{a{:}Nat, b{:}Nat\}, y{:}\{m{:}Nat\}\} <: \{x{:}\{a{:}Nat\}, y{:}\{\}\}} \qquad \text{(S-RcdDepth)}$$

Record Permutation

Orders of fields in records should be unimportant.

$$\{b{:}Bool,\ a{:}Nat\} <: \{a{:}Nat,\ b{:}Bool\}$$
$$\{a{:}Nat,\ b{:}Bool\} <: \{b{:}Bool,\ a{:}Nat\}$$

Hence <: is *not* a partial-order.

$$\frac{\{k_i : S_i\}^{i \in 1..n} \text{ is a permutation of } \{l_i : T_i\}^{i \in 1..n}}{\{k_i : S_i\}^{i \in 1..n} <: \{l_i : T_i\}^{i \in 1..n}} \quad \text{(S-RcdPerm)}$$

Subtyping Functions

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

$$\frac{T_1 <: S_1 \qquad\qquad S_2 <: T_2}{S_1 \to S_2 \ <: \ T_1 \to T_2} \qquad \text{(S-Arrow)}$$

Contravariance of Argument Types

Consider a function f of type: $S_1 \to S_2$

Consider some type $T_1 <: S_1$. It is clear that f accepts all elements of T_1 as argument. Therefore f should also be of type $T_1 \to S_2$.

$$\frac{f :: S_1 \to S_2 \qquad \dfrac{T_1 <: S_1}{S_1 \to S_2 <: T_1 \to S_2}}{f :: T_1 \to S_2}$$

Covariance of Result Types

Consider a function f of type: $S_1 \rightarrow S_2$

Consider some type T_2 such that $S_2 <: T_2$. It is clear that f returns only values of type T_2. Therefore f should also be of type $S_1 \rightarrow T_2$.

$$\frac{f :: S_1 \rightarrow S_2 \qquad \dfrac{S_2 <: T_2}{S_1 \rightarrow S_2 <: S_1 \rightarrow T_2}}{f :: S_1 \rightarrow T_2}$$

Top

Introduce a type Top that is the supertype of every type.

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

While 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 Bot type such that:

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

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

$$\frac{\vdash \text{v:Bot} \qquad \text{Bot} <: \text{Top} \to \text{Top}}{\vdash \text{v: Top} \to \text{Top}} \qquad \frac{\vdash \text{v:Bot} \qquad \text{Bot} <: \{\}}{\vdash \text{v: } \{\}}$$

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 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.

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash t : S} \qquad \dfrac{\vdots}{S <: T}}{\Gamma \vdash t : T} \quad \text{(T-Sub)}$$

$$\dfrac{}{\Gamma \vdash t \text{ as } T : T} \quad \text{(T-Ascribe)}$$

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.

$$\frac{\Gamma \vdash t : S}{\Gamma \vdash t \text{ as } T : T} \quad \text{(T-Downcast)}$$

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:

$$\frac{\vdash v : T}{v \text{ as } T \rightarrow v} \qquad \text{(E-DownCast)}$$

Variant Subtyping

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

$$<l_i : T_i>^{i \in 1..n} \quad <: \quad <l_i : T_i>^{i \in 1..n+k} \qquad \text{(S-VariantWidth)}$$

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

List Subtyping

List are also co-variant, thus:

$$\frac{S <: T}{\text{List } S <: \text{List } T}$$

References

References of the form r=ref v are used in two ways:

- for assignment r:=t, similar to arguments of functions:
  contravariant typing
  := : Ref T → T → ()

- for dereferencing !r, similar to return values of functions:
  covariant typing
  ! : Ref T → T

References : Assignment

Let r=ref v be of type 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:

$$\frac{T <: S}{\text{Ref } S <: \text{Ref } T}$$

References : Dereferencing

Let r=ref v be of type Ref S.

Say we have a dereferencing !r.

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

$$\frac{S \ <: \ T}{\text{Ref } S \ <: \ \text{Ref } T}$$

References : Invariant Typing

The result is an *invariant subtyping* of references.

$$\frac{S \;<:\; T \qquad T \;<:\; S}{\text{Ref } S \;<:\; \text{Ref } T}$$

In other words:

contravariance + *covariance* = *invariance*

Array Subtyping

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

Invariant subtyping:

$$\frac{S <: T \qquad T <: S}{\text{Array } S <: \text{Array } T}$$

Array Typing in Java

Java allows covariant subtyping of arrays:

$$\frac{S \;<:\; T}{\text{Array } S \;<:\; \text{Array } T}$$

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 Motorcyle[10]
Vehicle[] myVehicles = myBikes;

myVehicles[0] = new Vehicle();        // ArrayStoreException
```

Intersection Types

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

$$T_1 \wedge T_2 <: T_1$$

$$T_1 \wedge T_2 <: T_2$$

$$\frac{S <: T_1 \qquad S <: T_2}{S <: T_1 \wedge T_2}$$

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.

$$S \rightarrow T_1 \wedge S \rightarrow T_2 \quad <: \quad S \rightarrow T_1 \wedge T_2$$

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:

$$\vdash + : (\text{Nat} \to \text{Nat} \to \text{Nat}) \wedge (\text{Float} \to \text{Float} \to \text{Float})$$

Union Types

Union type $T_1 \vee T_2$ 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*.

$$T_1 <: T_1 \vee T_2$$

$$T_2 <: T_1 \vee T_2$$

$$\frac{T_1 <: S \qquad T_2 <: S}{T_1 \vee T_2 <: S}$$