

Type Systems for Programming Languages (15-814)

Lecture Notes, Fall 2006, Week 6

Kami Vaniea

October 17 and 19, 2006

1 Previously . . .

Remember from last time that we often only want to look at equations up to isomorphism. In order to do just that we use recursive types. The functions `roll` and `unroll` allow us to introduce and eliminate recursive types. A good example of this is linked lists where each link can be unrolled to show the next link. An empty list is the `inl` of nothing and a non-empty list is the `inr` of the new head of the list and a link `l` to the rest of the list which can be unrolled.



$$\begin{aligned} nil &= roll(inl \ \langle \rangle) \\ cons(m, l) &= roll(inr \ \langle m, l \rangle) \end{aligned}$$

Also an unroll can undo what was done by a roll.

$$\frac{roll(e) : value}{unroll(roll(e)) \mapsto e}$$

2 Recursive Types

Using recursive types we can derive the rules for general recursion. To begin with recall that the fixed point operator was defined as:

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash fix \ x : \tau. e : \tau}$$

and evaluated like:

$$fix \ x.e \mapsto [fix \ x.e/x]e$$

In the untyped λ calculus we defined the fix point:

$$Y := \lambda f. (\lambda x. f(x \ x)) (\lambda x. f(x \ x))$$

If we add in the recursive typing to this definition we get something very similar but it includes several roll and unroll statements to accommodate the use of the recursive types.

$$Y := \lambda f : \tau \rightarrow \tau. (\lambda x : \mu Y. Y \rightarrow \tau. f((\text{unroll } x)x)(\text{roll } (\lambda x : \mu Y. Y \rightarrow \tau. f((\text{unroll } x)x))))$$

Dynamically typed languages are a mode of use of statically typed languages. For example let us look at a fragment of the language Scheme. Here we define the language we will be using:

$$u ::= x \mid (\lambda(x)u) \mid (\text{apply } u_1 \ u_2) \mid (\text{cons } u_1 \ u_2) \mid (\text{car } u) \mid (\text{cdr } u) \mid \text{nil} \mid (\text{ifnil } u \ u_1 \ u_2)$$

In our language cons is the same thing as a pair, car is the elimination form for the left side of the cons and cdr is the elimination for the right side. This is very similar to the fst and snd we used previously.

Define the types of Scheme operations in our language.

$$S := \mu X. [\text{nil} : 1, \text{cons} : X \times X, \text{lambda} : X \rightarrow X]$$

The value space of Scheme is very similar but we can think of it as replacing the X with a S to denote that the recursive types are Scheme types.

$$S \cong [\text{nil} : 1, \text{cons} : S \times S, \text{lambda} : S \rightarrow S]$$

If we interpret the u's in the language S as certain e's we can define like a compiler mapping scheme to ML.

For example as we have seen previously lambda, nil and cons can be defined using the roll and unroll methods. This allows us to create a mapping between our version of Scheme and PCF.

$$\begin{aligned} \overline{(\text{lambda } (x) \ u)} &= \text{roll}[\text{lambda} = \lambda x : S. \overline{u}] \\ \overline{(\text{nil})} &= \text{roll}[\text{nil} = \langle \rangle] \\ \overline{(\text{cons } u_1 \ u_2)} &= \text{roll}[\text{cons} = \langle \overline{u_1}, \overline{u_2} \rangle] \end{aligned}$$

We can then use these definitions to help us case analyze.

$$\begin{aligned} \overline{(\text{apply } u_1 \ u_2)} &:= \\ &\text{case } \text{unroll}(\overline{u_1}) \{ \\ &\quad [\text{nil} = _] \Rightarrow \text{error} \\ &\quad [\text{cons} = _] \Rightarrow \text{error} \\ &\quad [\text{lambda} = f] \Rightarrow f(\overline{u_2}) \\ &\quad \} \end{aligned}$$

In the case of apply we can see that both the nil and cons cases can't be applied and so attempting to do so produces an error. Since any of the three types could be present we have to test for all three even though we know that only the lambda case will produce good output. Occasionally this is erroneously called a "run-time type check" since we are checking at run time to see that the typing is correct.

$$\overline{(car\ u)} :=$$

$$\text{case unroll}(\overline{u})\{$$

$$\quad [nil = _] \Rightarrow \text{error}$$

$$\quad [cons = \langle a, _ \rangle] \Rightarrow a$$

$$\quad [lambda = _] \Rightarrow \text{error}$$

$$\}$$

And similarly for cdr:

$$(cdr\ u) :=$$

$$\text{case unroll}(\overline{u})\{$$

$$\quad [nil = _] \Rightarrow \text{error}$$

$$\quad [cons = \langle _ , b \rangle] \Rightarrow b$$

$$\quad [lambda = _] \Rightarrow \text{error}$$

$$\}$$

$$\overline{(ifnil\ u\ u_1\ u_2)} :=$$

$$\text{case unroll}(\overline{u})\{$$

$$\quad [nil = _] \Rightarrow \overline{u_1}$$

$$\quad [cons = _] \Rightarrow \overline{u_2}$$

$$\quad [lambda = _] \Rightarrow \overline{u_2}$$

$$\}$$

The foundation of modularity is that each and every component is designed to work independent of every other. There is no way for one function to communicate to another what it is doing internally it can only communicate what it returns as a result. If a language has only a single type then functions are greatly limited in what information they can pass. Whenever a component receives an input it must do a case analysis to determine which input was received. This is wasteful and time consuming.

3 Subtyping

A subtype relation is a pre-order on types that validates the subsumption principle.

Principle:

$$\frac{\Gamma \vdash e : \tau \quad \tau <: \tau'}{\Gamma \vdash e : \tau'} \text{ (SUBSUMPTION)}$$

$\tau <: \tau'$

1. Principle of attenuation

Passage to a super type causes a *loss of information*.

2. $\tau <: \tau'$ means that wherever a τ' is required it is sufficient to provide a τ .

3.1 Record Subtyping

When is it reasonable to consider one record type to be a subtype of another?

$$\{\ell_1 : \tau_1 \dots \ell_m : \tau_m\} <: \{\ell'_1 : \tau'_1 \dots \ell'_n : \tau'_n\}$$

The subtypes in this relation can be considered subtypes if when $\{\ell'_1 : \tau'_1 \dots \ell'_n : \tau'_n\}$ is required $\{\ell_1 : \tau_1 \dots \ell_m : \tau_m\}$ can be given. Thus any elimination form that takes $\{\ell'_1 : \tau'_1 \dots \ell'_n : \tau'_n\}$ should also be an elimination form for $\{\ell_1 : \tau_1 \dots \ell_m : \tau_m\}$.

Records have only one elimination form which is $e.\ell$. This elimination form takes a record e and returns the element labeled with ℓ . Subtyping for records must obey the following two rules.

1. Labels on the left must be a superset of the labels on the right.
2. Ordering is irrelevant
 $e.\ell$ depends only on the existence of the ℓ field not i t's order in the record.

$$\{\ell_1 : \tau_1 \dots \ell_n : \tau_n\} <: \{\ell'_1 : \tau'_1 \dots \ell'_n : \tau'_n\}$$

$$\frac{m \geq n}{\{\ell_1 : \tau_1 \dots \ell_m : \tau_m\} <: \{\ell_1 : \tau_1 \dots \ell_n : \tau_n\}} \text{ (RECORD WIDTH)}$$

$$\frac{m \leq n}{[\ell_1 : \tau_1 \dots \ell_m : \tau_m] <: [\ell_1 : \tau_1 \dots \ell_n : \tau_n]} \text{ (VARIANT WIDTH SUBTYPING)}$$

The following are examples of subtyping for variants

$$\begin{aligned} [a, b] &<: [a, b, c] \\ [a] &<: [a, b] <: [a, b, c] \text{ and } [a] <: [a, b, c] \end{aligned}$$

3.2 Variance Principles

How do type constructors interact with subtyping?

$$\frac{?}{\sigma \times \tau <: \sigma' \times \tau'} \qquad \frac{?}{\sigma + \tau <: \sigma' + \tau'} \qquad \frac{?}{\sigma \rightarrow \tau <: \sigma' \rightarrow \tau'}$$

Three possibilities for each argument

Covariant Preserves subtyping

Contravariant Reverses subtyping

Invariant Neither preserves nor reverses subtyping

The product type is covariant in both positions.

$$\frac{\sigma <: \sigma' \quad \tau <: \tau'}{\sigma \times \tau <: \sigma' \times \tau'}$$

The $+$ type is also covariant.

$$\frac{\sigma <: \sigma' \quad \tau <: \tau'}{\sigma + \tau <: \sigma' + \tau'}$$

The \rightarrow type is contravariant in the domain while being covariant in the range.

$$\frac{\sigma' <: \sigma \quad \tau <: \tau'}{\sigma \rightarrow \tau <: \sigma' \rightarrow \tau'}$$

Example

$$\begin{aligned} \{a, b\} \rightarrow \{u, v, w\} &<: \{a, b, c\} \rightarrow \{u, w\} \\ [a, b, c] \rightarrow [u, v] &<: [a, b] \rightarrow [u, v, w] \end{aligned}$$

This example shows that it is ok to give a $[a, b, c] \rightarrow [u, v]$ when a $[a, b] \rightarrow [u, v, w]$ is expected. This is because the context is expecting something that can handle both an a and a b case. If the function can also handle a c case that will not effect anything. Similarly whatever called this function is expecting to have to case analyze a u, v or w . If it never receives a w then it will simply never execute the w branch.

For Variance the subtyping relation looks like:

$$[a, b] \rightarrow [u] <: [a] \rightarrow [u, v]$$

For Records the subtyping relation looks like:

$$\{a\} \rightarrow \{u, v\} <: \{a, b\} \rightarrow \{u\}$$

3.3 Subtyping for recursive types

To begin with we want to find out under what premise $\mu X.\tau <: \mu X.\tau'$ is true.

$$\frac{?}{\mu X.\tau <: \mu X.\tau'}$$

Since here X is the fixed point X is isomorphic to its unrolling. So X stands for the type itself where X can be unrolled.

To start with lets define two natural types of natural number. The first one, N , is our normal definition of a natural number where N can be zero or a $\text{succ}(N)$. The second is a slightly modified version of the natural number, N' , where N' can be zero, one or $\text{succ}(N')$.

$$\begin{aligned} N &= \mu X.1 + X \\ N' &= \mu X.1 + 1 + X \end{aligned}$$

So now we wonder how can we show that one N is a subtype of the other. The problem we instantly notice is that while $N' <: N$ the typing within N' includes N' s not N s.

$$\begin{aligned} N &\cong 1 + N \\ N' &\cong 1 + 1 + N' \end{aligned}$$

Or written more precisely:

$$\begin{aligned} N &\cong [\text{zero} : 1, S : N] \\ N' &\cong [\text{zero} : 1, \text{one} : 1, S : N'] \end{aligned}$$

The problem is that while both N and N' have successor cases one returns a N and the other returns a N' .

If we were to guess a solution to this we might think that the following might work:

$$\frac{\Delta, X \vdash \tau <: \tau'}{\Delta \vdash \mu X. \tau <: \mu X. \tau'}$$

If we try this using N and N' we can see how this will break down. The problem is that we are using a form of α conversion to force the X on both sides of the $<:$ to be the same variable. In short we are saying that:

$$\frac{\Delta, X = X' \vdash \tau <: \tau'}{\Delta \vdash \mu X. \tau <: \mu X'. \tau'}$$

If we typecheck under the assumption that $X = X'$ we will produce type unsafe programs. By assuming that the two types are the same we are saying that not only can we provide an X when a X' is needed but we can provide a X' when an X is needed. However, the elimination form for the X may not have branches for every possible term in X' . In the previous example if we were to use a N' in a *case* statement that expects a N the case statement would not be able to handle N' being of type *one* and would be in an unsafe state. The correct way to handle this situation is to say that $X <: X'$.

$$\frac{\Delta, X <: X' \vdash \tau <: \tau'}{\Delta \vdash \mu X. \tau <: \mu X'. \tau'}$$

Example:

$$\begin{aligned} \tau &= \mu X. [f : X \rightarrow X] \\ \tau' &= \mu X'. [f : X' \rightarrow X', a : X'] \end{aligned}$$

If we assume that $X <: X'$ then we just need to prove that $\tau <: \tau'$. So we just need to prove that $X <: X' \vdash X \rightarrow X <: X' \rightarrow X'$

STS:

1. $X <: X' \vdash X <: X'$ which works
2. $X <: X' \vdash X' <: X$ which is contravariant.

We can't validate the subtyping so τ' is not a subtype of τ .

4 Safety in Subtyping

4.1 Dynamic Semantics for Records

$$\{\ell_0 = e_0 \dots \ell_i = e_i \dots \ell_n = e_n\}. \ell_i \mapsto e_i$$

Each ℓ is a label and each e is a value.

4.2 Static Semantics for Records and Variants

4.2.1 Introduction Forms

$$\frac{(\Gamma \vdash e_i : \tau_i)_{i=1}^n}{\Gamma \vdash \{\ell_i = e_i\}_{i=1}^n : \{\ell_i : \tau_i\}_{i=1}^n} (\{\}-I)$$

$$\frac{\Gamma \vdash e_i : \tau_i}{\Gamma \vdash [\ell_j = e_i]_{[\ell_i = e_i]_{i=1}^n} : [\ell_i : \tau_i]_{i=1}^n} ([]-I)$$

4.2.2 Elimination Forms

$$\frac{\Gamma \vdash e : \{\ell : \tau\}}{\Gamma \vdash e.l : \tau} \text{ (}\{\}\text{-E)}$$

4.2.3 Subsumption

$$\frac{\Gamma \vdash e : \tau \quad \tau <: \tau'}{\Gamma \vdash e : \tau'} \text{ (SUBSUMPTION)}$$

4.2.4 Subtyping

$$\frac{\rho <: \sigma \quad \sigma <: \tau}{\rho <: \tau} \text{ (S-TRANSITIVITY)} \quad \frac{}{\tau <: \tau} \text{ (S-REFLEXIVITY)} \quad \frac{\sigma' <: \sigma \quad \tau <: \tau'}{\sigma \rightarrow \tau <: \sigma' \rightarrow \tau'} \text{ (S-}\rightarrow\text{)}$$

$$\frac{}{\Sigma \vdash \{\ell_i : \sigma_i\}_{i=1}^{n+k} <: \Sigma \vdash \{\ell_i : \sigma_i\}_{i=1}^n} \text{ (}\{\}\text{-WIDTH)} \quad \frac{}{\Sigma \vdash [\ell_i : \sigma_i]_{i=1}^n <: \Sigma \vdash [\ell_i : \sigma_i]_{i=1}^{n+k}} \text{ ([]-WIDTH)}$$

$$\frac{(\Sigma \vdash \sigma_i <: \tau_i)_{i=1}^n}{\Sigma \vdash \{\ell_i : \sigma_i\}_{i=1}^n <: \{\ell_i : \tau_i\}_{i=1}^n} \text{ (S-}\{\}\text{-DEPTH)} \quad \frac{(\Sigma \vdash \sigma_i <: \tau_i)_{i=1}^n}{\Sigma \vdash [\ell_i : \sigma_i]_{i=1}^n <: [\ell_i : \tau_i]_{i=1}^n} \text{ (S-[]-DEPTH)}$$

5 Preservation of Typing

The preservation theorem states that if $e : \tau$ and $e \mapsto e'$ then $e' : \tau$. To prove this we will use induction on transition. There are many cases that need to be proven to show preservation but in these notes we will be looking at the following one.

$$e = \{\dots \ell = e' \dots\}.l \mapsto e'$$

To begin with we suppose that $e : \tau$ then we will need to show that $e' : \tau$ by using inversion. In order to show this we need to use an inversion lemma that will let us get information about e' . We define and prove this lemma below. We will see that since $\{\dots \ell = e' \dots\} : \{\ell : \tau\}$ then by the inversion lemma $e' : \tau$.

5.1 Inversion Lemma

Definition 5.1 (Inversion Lemma Part 1). If $e.l : \tau$ then $e : \{\dots \ell : \tau \dots\}$

This can be proved by using induction on typing. The problem is that because of subtyping there are now two rules that could result in $e.l$. Either e was a record type or subsumption could have been used and $e.l : \tau$ was previously $e.l : \tau'$. By using the depth subtyping rule we can say that:

$$\frac{e.l : \tau' \quad \tau' <: \tau}{e.l : \tau}$$

So if $e : \{\ell : \tau'\}$ as long as $\tau' <: \tau$ we can use $e : \{\ell : \tau\}$. This works out.

Definition 5.2 (Inversion Lemma Part 2). If $\{\dots \ell = e \dots\} : \{\ell : \tau\}$ then $e : \tau$

We can prove this by induction on typing. The first case is that this could have been done by the intro rule.

$$\frac{(\Gamma \vdash e_i : \tau_i)_{i=1}^n}{\Gamma \vdash \{\ell_i = e_i\}_{i=1}^n : \{\ell_i : \tau_i\}_{i=1}^n} \text{ (}\{\}\text{-I)}$$

If that was the case then we know that $e : \tau$ by $\{\} - I$.
 Alternatively this could have been done by the subsumption rule.

$$\frac{\{\dots \ell = e\} : \tau' \quad \tau' <: \{\ell : \tau\}}{\{\dots \ell = e \dots\} : \{\ell : \tau\}} \text{ (SUBSUMPTION)}$$

The problem is that now the premise has a τ' in it and we don't know anything about τ' . In order to say something we have to use the Subtyping Lemma which tells us that τ' has to be of the form of a record.

$$\tau' = \{\dots \ell : \tau'' \dots\} \quad \text{ST: } \tau'' <: \tau$$

We needed to show that $e : \tau$ and we have that $e : \tau''$ and $\tau'' <: \tau$ so by subsumption $e : \tau$.

5.2 Subtyping Inversion Lemma

Definition 5.3 (Subtyping Inversion Lemma). If $\tau <: \{\ell : \tau\}$ then $\tau = \{\dots \ell : \tau' \dots\}$ where $\tau' <: \tau$.

This says that if anything is a subtype of a singleton record type then τ itself has to be a record type containing at least a label which is a subtype of the label in the singleton record.

We do this by an induction on the rules that involve subtyping. Proving this is trivial for all the cases except for transitivity because of the third type it introduces. To analyze this subtyping relationship we have to show that transitivity is admissible. The problem is that we know nothing about the third type and so we have to show $e <: \sigma$ from nothing and that $\sigma <: \tau$

$$\frac{\{\ell_i : \tau_i\}_{i=1}^n <: \{\ell_i : \tau_i\}_{i=1}^m \quad m \leq n \quad \{\ell_i : \tau_i\}_{i=1}^m <: \{\ell_i : \tau_i\}_{i=1}^k \quad k \leq m}{\{\ell_i : \tau_i\}_{i=1}^n <: \{\ell_i : \tau_i\}_{i=1}^k}$$

\Rightarrow (Using Record Width)

$$\frac{k \leq n}{\{\ell_i : \tau_i\}_{i=1}^n <: \{\ell_i : \tau_i\}_{i=1}^k}$$

If you give $D_1 = e <: \sigma$ and $D_2 = \sigma <: \tau$ then we can look at all possible combinations of derivation of D1 and D2 and show that in every case there is some other way to derive this than transitivity. The trick is that by doing this we end up pushing the transitivity off the end of the derivation.

5.3 Progress Theorem

If $e : \tau$ then e value or $e \mapsto e'$.

In order to prove this we need to do an induction on typing not structure. Because of subsumption doing an induction on structure will lead no where because e dose not decrease.

Progress can be proved by doing an induction on typing but in order to do so we need the canonical forms lemma for subtyping.

Definition 5.4 (Canonical Forms Lemma). If e value and $e : \{\ell : \tau\}$ then $e = \{\dots \ell = e' \dots\}$

The canonical forms lemma tells us that unless we got e from an intro rule we have no way to know the size of the record e . The type τ tells us what we can find in e but sometime in the past e may have been a super type of τ and therefore has more records.