OPLSS15, Notes for Amal Ahmed's lectures.

Lau
lask@cs.au.dk

July 6, 2015

The videos of the lectures of OPLSS 2015 can be found at `https://www.cs.uoregon.edu/research/summerschool/summer15/curriculum.html`

Lecture 1, Logical Relations

Simply Typed Lambda Calculus (STLC)

Types:

$$\tau ::= \mathit{bool} \mid \tau \to \tau$$

Terms:

$$e ::= x \mid \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \mid \lambda x : \tau.\ e \mid e\ e$$

Values:

$$v ::= \text{true} \mid \text{false} \mid \lambda x : \tau.\ e$$

Evaluation contexts:

$$E ::= [] \mid \text{if } E \text{ then } e \text{ else } e \mid E\ e \mid v\ E$$

Evaluations:

$$\text{if true then } e_1 \text{ else } e_2 \mapsto e_1$$
$$\text{if false then } e_1 \text{ else } e_2 \mapsto e_2$$
$$(\lambda x : \tau.\ e)\ v \mapsto e[v/x]$$
$$\frac{e \mapsto e'}{E[e] \mapsto E[e']}$$

Typing Contexts:

$$\Gamma ::= \bullet \mid \Gamma, x : \tau$$

Typing rules:

$$\frac{}{\Gamma \vdash \text{false} : \mathit{bool}}\ \text{T-False} \qquad \frac{}{\Gamma \vdash \text{true} : \mathit{bool}}\ \text{T-True}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \text{T-Var} \qquad \frac{\Gamma \vdash e : \mathit{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau}\ \text{T-If}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}\ \text{T-Abs}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}\ \text{T-App}$$

Logical Relations

Proof method to establish properties.

- Termination
- Type safety
- Equivalence of programs
  - Correctness of programs
  - Representation independence
  - Parametricity and free theorems, e.g.,

    $$f : \forall\alpha.\ \alpha \to \alpha$$

    Cannot inspect α as it has no idea which type it will be, therefore this function must be *id*.

    $$\forall.\ \mathit{int} \to \alpha$$

    A function with this type does not exist (the function would need to return something of type α, but it only has something of type *int* to work with, so it cannot possibly return a value of the proper type).
  - Security-Typed Languages (for Information Flow Control (IFC))
    Example: All types in the code snippet below are labeled with their security level. A type can be labeled with either L for *low* or H for *high*. We do not want any flow from variables with a *high* labeled type to a variable with a *low* labeled type. The following is an example of an *explicit flow* of information:

    ```
    x : int^L
    y : int^H
    x = y     //This assignment is illegal.
    ```

    Further, information may leak through a *side channel*. That is the value denotated by a variable with a *low* labeled type depends on the value of a variable with a *high* labeled type. If this is the case we may not have learned the secret value, but we may have learned some information about it. An example of a side channel:

$$
\begin{array}{l}
\mathtt{x} \ : \ \mathtt{int}^L \\
\mathtt{y} \ : \ \mathtt{int}^H \\
\mathtt{if} \ \ \mathtt{y} > 0 \ \ \mathtt{then} \ \ \mathtt{x} = 0 \ \ \mathtt{else} \ \ \mathtt{x} = 1
\end{array}
$$

How do we then argue a program is secure? Non-interference:

$$
\begin{array}{l}
\vdash P : int^L \times int^H \to int^L \\
P(v_L, v_{H1}) \approx_L P(v_L, v_{H2})
\end{array}
$$

If we run P with the same *low* value and with two different *high* values, then the low result of the two runs of the program should be equal. That is the *low* result does not depend on *high* values.

Categories of Logical Relations

Logical Predicates	Logical Relations
(Unary)	(Binary)
$P_\tau(e)$	$R\tau(e_1, e_2)$
- One property	- Program Equivalence
- Strong normalization	
- Type safety	

Strong Normalization of STLC

A first try on normalization of STLC

We start with a couple of abbreviations:

$$
\begin{array}{l}
e \Downarrow v \stackrel{\text{def}}{=} e \mapsto^* v \\
e \Downarrow \stackrel{\text{def}}{=} \exists v.\ e \Downarrow v
\end{array}
$$

What we want to prove is:

Theorem (Strong Normalization)**.** *If* $\bullet \vdash e : \tau$ *then* $e \Downarrow$

Proof. ¡ This proof gets stuck and is not complete. It is included to motivate the use of a logical relation. !
Induction on the structure of the typing derivation.
Case $\bullet \vdash \text{true} : bool$, this term has already terminated.
Case $\bullet \vdash \text{false} : bool$, same as for true.
Case $\bullet \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau$, simple, but requires the use of canonical forms of bool.
Case $\bullet \vdash \lambda x : \tau.\ e : \tau_1 \to \tau_2$, it is a value, it has terminated.

$$
\textbf{Case} \ \ \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \ e_2 : \tau} \ \text{T-App} \ ,
$$

By the induction hypothesis we get $e_1 \Downarrow v_1$ and $e_2 \Downarrow v_2$. By the type of e_1 we conclude $e_1 \Downarrow \lambda x : \tau_2.\ e'$. What we need to show is $e_1 \ e_2 \Downarrow$.

$$
\begin{array}{ll}
e_1 \ e_2 \mapsto^* & (\lambda x : \tau_2.\ e') \ e_2 \\
\mapsto^* & (\lambda x : \tau_2.\ e') \ v_2 \\
\mapsto^* & e'[v_2/x]
\end{array}
$$

Here we run into an issue as we do not know anything about e'. Our induction hypothesis is not strong enough.[1] □

A logical predicate for strongly normalizing expressions

We want to define a logical predicate, $\mathrm{SN}_\tau(e)$. We want $\mathrm{SN}_\tau(e)$ to be the set of expressions of type τ that are strongly normalizing, but let us first consider what properties we in general want a logical relation to have.

In general for a logical predicate, $P_\tau(e)$, we want an expression, e, in accepted by this predicate to satisfy the following properties[2]:

1. $\bullet \vdash e : \tau$
2. The property we wish e to have. In this case it would be: e is strongly normalizing.
3. The condition is preserved by eliminating forms.

With the above in mind we define the strongly normalizing predicate as follows:

$$\mathrm{SN}_{bool}(e) \Leftrightarrow \bullet \vdash e : bool \ \wedge \ e \Downarrow$$
$$\mathrm{SN}_{\tau_1 \to \tau_2}(e) \Leftrightarrow \bullet \vdash e : \tau_1 \to \tau_2 \ \wedge \ e \Downarrow \ \wedge (\forall e'.\ \mathrm{SN}_{\tau_1}(e') \implies \mathrm{SN}_{\tau_2}(e\ e'))$$

It is here important to consider whether the logical predicate is well-founded. $\mathrm{SN}_\tau(e)$ is defined over the structure of τ, so it is indeed well-founded.

Strongly normalizing using a logical relation

We are now ready to show strong normalization using $\mathrm{SN}_\tau(e)$. The proof is done in two steps:

(a) $\bullet \vdash e : \tau \implies \mathrm{SN}_\tau(e)$

(b) $\mathrm{SN}_\tau(e) \implies e \Downarrow$

The structure of this proof is common to proofs that use logical relations. We first prove that well-typed terms are in the relation. Then we prove that terms in the relation actually have the property we want to show (in this case strong normalization).

The proof of (b) is by induction on τ.[3]

We could try to prove (a) by induction over $\bullet \vdash e : \tau$, but the case

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}\ \text{T-Abs}$$

gives issues. What we instead do is to prove a generalization of (a)

Theorem ((a) Generalized)**.** *If* $\Gamma \vdash e : \tau$ *and* $\gamma \models \Gamma$ *then* $\mathrm{SN}_\tau(\gamma(e))$

[1] :(

[2] Note: when we later want to prove type safety we do not want well-typedness to be a property of the predicate.

[3] This should not be difficult, as we baked the property we want into the relation. That was the second property we in general wanted a logical relation to satisfy.

Here γ is a substitution, $\gamma = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$.

In English the theorem reads: If e is well-typed wrt some type τ and we have some substitution that satisfy the typing environment, then if we close of e with γ, then this closed expression is in SN_τ.

$\gamma \models \Gamma$ is read "the substitution, γ, satisfies the type environment, Γ." It is defined as follows:

$$\gamma \models \Gamma \stackrel{\text{def}}{=} \mathrm{dom}(\gamma) = \mathrm{dom}(\Gamma) \ \wedge \ \forall x \in \mathrm{dom}(\Gamma).\ \mathrm{SN}_{\Gamma(x)}(\gamma(x))$$

To prove the generalized theorem we need further two lemmas

Lemma (Substitution Lemma)**.** *If* $\Gamma \vdash e : \tau$ *and* $\gamma \models \Gamma$ *then* $\bullet \vdash \gamma(e) : \tau$

Lemma (SN preserved by forward/backward reduction)**.** *Suppose* $\bullet \vdash e : \tau$ *and* $e \mapsto e'$

1. *if* $\mathrm{SN}_\tau(e')$ *then* $\mathrm{SN}_\tau(e)$
2. *if* $\mathrm{SN}_\tau(e)$ *then* $\mathrm{SN}_\tau(e')$

Proof. Left as an exercise. □

Proof. (Substitution Lemma). Probably also left as an exercise (not proved during the lecture). □

Proof. (ⓐ Generalized). Proof by induction on $\Gamma \vdash e : \tau$.
Case $\Gamma \vdash \mathrm{true} : \mathit{bool}$,
We have:

$$\gamma \models \Gamma$$

We need to show:

$$\mathrm{SN}_{\mathit{bool}}(\gamma(\mathrm{true}))$$

If we do the substitution we just need to show $\mathrm{SN}_{\mathit{bool}}(\mathrm{true})$ which is true by definition of $\mathrm{SN}_{\mathit{bool}}(\mathrm{true})$.
Case $\Gamma \vdash \mathrm{false} : \mathit{bool}$, similar to the true case.
Case $\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textsc{T-Var}$,
We have:

$$\gamma \models \Gamma$$

We need to show:

$$\mathrm{SN}_\tau(\gamma(x))$$

This case follows from the definition of $\Gamma \models \gamma$. We know that x is well-typed, so it is in the domain of Γ. From the definition of $\Gamma \models \gamma$ we then get $\mathrm{SN}_{\Gamma(x)}(\gamma(x))$. From well-typedness of x we have $\Gamma(x) = \tau$ which then gives us what we needed to show.
Case $\Gamma \vdash \mathrm{if}\ e\ \mathrm{then}\ e_1\ \mathrm{else}\ e_2 : \tau$, left as an exercise.
Case $\dfrac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}\ \textsc{T-App}$,
We have:

$$\gamma \models \Gamma$$

We need to show:

$$\mathrm{SN}_\tau(\gamma(e_1\ e_2)) \equiv \mathrm{SN}_\tau(\gamma(e_1)\ \gamma(e_2))$$

By induction hypothesis we get

$$\mathrm{SN}_{\tau_2 \to \tau}(\gamma(e_1)) \tag{1}$$
$$\mathrm{SN}_{\tau_2}(\gamma(e_2)) \tag{2}$$

If we use the 3rd property of (1), $\forall e'.\ \mathrm{SN}_{\tau_1}(e') \implies \mathrm{SN}_{\tau_2}(\gamma(e)\ e')$, instantiated with (2), then we get $\mathrm{SN}_{\tau_2}(\gamma(e_1)\ \gamma(e_2))$ which is what we wanted.

Case $\dfrac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}$ T-Abs,
We have:

$$\gamma \models \Gamma$$

We need to show:

$$\mathrm{SN}_{\tau_1 \to \tau_2}(\gamma(\lambda x : \tau_1.\ e)) \equiv \mathrm{SN}_{\tau_1 \to \tau_2}(\lambda x : \tau_1.\ \gamma(e))$$

Our induction hypothesis in this case reads:

$$\Gamma, x : \tau_1 \vdash e : \tau_2\ \wedge\ \gamma' \models \Gamma, x : \tau_1 \quad \implies \quad \mathrm{SN}_{\tau_2}(\gamma'(e))$$

It suffices to show the following three things:

1. $\bullet \vdash \lambda x : \tau_1.\ \gamma(e) : \tau_1 \to \tau_2$
2. $\lambda x : \tau_1.\ \gamma(e) \Downarrow$
3. $\forall e'.\, \mathrm{SN}_{\tau_1}(e') \implies \mathrm{SN}_{\tau_2}((\lambda x : \tau_1.\ \gamma(e))\ e')$

If we use the substitution lemma, and then push the γ in under the λ-abstraction, then we get 1[4]. 2 is okay as the lambda-abstraction is a value.

It only remains to show 3. To do this we want to somehow apply the induction hypothesis. To do that we need a γ' such that $\gamma' \models \Gamma, x : \tau_1$. We already have γ and $\gamma \models \Gamma$, so our γ' should probably have have the form $\gamma' = \gamma[x \mapsto v_?]$ for some $v_?$.

Let e' be given and assume $\mathrm{SN}_{\tau_1}(e')$. We then need to show $\mathrm{SN}_{\tau_2}((\lambda x : \tau_1.\ \gamma(e))\ e')$. From $\mathrm{SN}_{\tau_1}(e')$ it follows that $e' \Downarrow v'$. v' is a good candidate for $v_?$ so let $v_? = v'$. From the forward part of the preservation lemma we can further conclude $\mathrm{SN}_{\tau_1}(v')$. We use this to conclude $\gamma[x \mapsto v'] \models \Gamma, x : \tau_1$ which we use with the assumption $\Gamma, x : \tau_1 \vdash e : \tau_2$ to instantiate the induction hypothesis and get $\mathrm{SN}_{\tau_2}(\gamma[x \mapsto v'](e))$.

Now consider the following evaluation:

$$\begin{aligned}(\lambda x : \tau_1.\ \gamma(e))\ e' &\mapsto^* (\lambda x : \tau_1.\ \gamma(e))\ v' \\ &\mapsto \gamma(e)[v'/x] \equiv \gamma[x \mapsto v'](e)\end{aligned}$$

We already concluded that $e' \mapsto^* v'$ which corresponds to the first series of steps. We can then do a β-reduction to take the next step and finally we get something that is equivalent to $\gamma[x \mapsto v'](e)$. That is we have the evaluation

$$(\lambda x : \tau_1.\ \gamma(e))\ e' \mapsto^* \gamma[x \mapsto v'](e)$$

[4]Substitution has not been formally defined here, but one can find a sound definition in Pierce's Types and Programming Languages.

If we use this with $\mathrm{SN}_{\tau_2}(\gamma[x \mapsto v'](e))$, the fact that $(\lambda x : \tau_1.\ \gamma(e))\ e'$ is well-typed[5], and the backwards part of the SN preservation lemma, then we get $\mathrm{SN}_{\tau_2}((\lambda x : \tau_1.\ \gamma(e))\ e')$ which is exactly what we needed to show.

□

Exercises

1. Prove SN preserved by forward/backward reduction
2. Prove the substitution lemma
3. Prove the if-case of "ⓐ Generalized"
4. Extend the language with pairs and do the proofs

[5]Follows from $\mathrm{SN}_{\tau_1}(e')$, the assumption $\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2$, $\gamma \models \Gamma$, the substitution lemma and the typing rule for lambda-abstractions

Lecture 2, Type Safety

The goal of this lecture was to proof type safety for STLC using logical relations. First we need to consider what type safety is. The classical mantra for type safety is "Well-typed programs do not *go wrong*." It depends on our language and type system what *go wrong* means, but in our case a program has *gone wrong*, if it is stuck[6].

Type Safety for STLC

Theorem (Type Safety for STLC)**.** *If* $\bullet \vdash e : \tau$ *and* $e \mapsto^* e'$ *then* $\mathrm{Val}(e)$ *or* $\exists e'.\ e \mapsto e'$.

Traditionally type safety is proven by two lemmas, progress and preservation.

Lemma (Progress)**.** *If* $\bullet \vdash e : \tau$ *then* $\mathrm{Val}(e)$ *or* $\exists e'.\ e \mapsto e'$.

Progress is normally proved by induction on the typing derivation.

Lemma (Preservation)**.** *If* $\bullet \vdash e : \tau$ *and* $e \mapsto e'$ *then* $\bullet \vdash e' : \tau$.

Preservation is normally proved by induction on the evaluation. Preservation is also known as *subject reduction*. Progress and preservation talks about one step, so to prove type safety we have to do induction on the evaluation. Here we do not want to prove type safety the traditional way, we want to prove it using a logical predicate. We use a logical predicate rather than a logical relation, because type safety is a unary property.

The notation will here further be changed compared to the one from lecture 1. We define the logical predicate in two parts: a value predicate and an expression predicate. We define the value predicate as:

$$\begin{aligned} \mathcal{V}[\![\mathit{bool}]\!] &= \{\mathrm{true}, \mathrm{false}\} \\ \mathcal{V}[\![\tau_1 \to \tau_2]\!] &= \{\lambda x : \tau_1.\ e \mid \forall v \in \mathcal{V}[\![\tau_1]\!].\ e[v/x] \in \mathcal{E}[\![\tau_2]\!]\} \end{aligned}$$

We define the expression predicate as:

$$\mathcal{E}[\![\tau]\!] = \{e \mid \forall e'.\ e \mapsto^* e' \wedge \mathrm{irred}(e') \implies e' \in \mathcal{V}[\![\tau]\!]\}$$

[7]The predicate irred is defined as:

$$\mathrm{irred}(e) \stackrel{\mathrm{def}}{=} \nexists e'.\ e \mapsto e'$$

The sets are defined on the structure of the types. $\mathcal{V}[\![\tau_1 \to \tau_2]\!]$ contains $\mathcal{E}[\![\tau_2]\!]$, but $\mathcal{E}[\![\tau_2]\!]$ uses τ_2 directly, so the definition is structurally well-founded.

To prove safety we first define a new predicate, safe:

$$\mathrm{safe}(e) \stackrel{\mathrm{def}}{=} \forall e'.\ e \mapsto^* e' \implies \mathrm{Val}(e') \vee \exists e''.\ e' \mapsto^* e''$$

We are now ready to prove type safety. Just like we did for strong normalization, we prove type safety in two steps:

ⓐ $\bullet \vdash e : \tau \implies e \in \mathcal{E}[\![\tau]\!]$

[6]If we consider language-based security for information flow control the notion of *go wrong* would then be that there is an undesired flow of information

[7]Notice that neither $\mathcal{V}[\![\tau]\!]$ nor $\mathcal{E}[\![\tau]\!]$ requires well-typedness. Normally this would be a part of the predicate, but as the goal is to prove type safety we do not want it as a part of the predicate. In fact, if we did include a well-typedness requirement, then we would end up having to prove preservation for some of the proofs to go through.

(b) $e \in \mathcal{E}[\![\tau]\!] \implies \text{safe}(e)$

Rather than proving (a) directly we prove a more general theorem and get (a) as a corollary. But we are not yet in a position to state the theorem. First we need to define define the interpretation of type environments:

$$\begin{aligned}\mathcal{G}[\![\bullet]\!] &= \{\emptyset\} \\ \mathcal{G}[\![\Gamma, x : \tau]\!] &= \{\gamma[x \mapsto v] \mid \gamma \in \mathcal{G}[\![\Gamma]\!] \wedge v \in \mathcal{V}[\![\tau]\!]\}\end{aligned}$$

Further we need to define semantic type safety:

$$\Gamma \models e : \tau \stackrel{\text{def}}{=} \forall \gamma \in \mathcal{G}[\![\Gamma]\!].\ \gamma(e) \in \mathcal{E}[\![\tau]\!]$$

We can now define our generalized version of (a).

Theorem (Fundamental Property)**.** *If* $\Gamma \vdash e : \tau$ *then* $\Gamma \models e : \tau$

A theorem like this would typically be the first you prove after a logical relation has been defined. The theorem says that every syntactic type safety implies semantic type safety.

We will start by proving the easy part, namely (b).

Proof. *((b)).* Suppose $e \mapsto^* e'$ for some e', then we need to show $\text{Val}(e')$ or $\exists e''.\ e' \mapsto e''$. Now either $\neg\,\text{irred}(e')$ or $\text{irred}(e')$.
Case $\neg\,\text{irred}(e')$, this case follows directly from the definition of irred. $\text{irred}(e')$ is defined as $\nexists e''.\ e' \mapsto e''$ and as the assumption is $\neg\,\text{irred}(e')$ we get $\exists e''.\ e' \mapsto e''$.
Case *irred*(e'), By assumption we have $\bullet \models e : \tau$. As the typing context is empty we choose the empty substitution and get $e \in \mathcal{E}[\![\tau]\!]$. We now use the definition of $e \in \mathcal{E}[\![\tau]\!]$ with what we supposed, $e \mapsto^* e'$, and the case assumption, $\text{irred}(e')$, to conclude $e' \in \mathcal{V}[\![\tau]\!]$. As e' is in the value interpretation of τ we can conclude $\text{Val}(e')$. □

Proof. *(Fundamental Property).* Proof by induction on the typing judgment.
Case $\dfrac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}$ T-Abs,
We need to show $\Gamma \models \lambda x : \tau_1.\ e : \tau_1 \to \tau_2$. First suppose $\gamma \in \mathcal{G}[\![\Gamma]\!]$. Then it suffices to show

$$\gamma(\lambda x : \tau_1.\ e) \in \mathcal{E}[\![\tau_1 \to \tau_2]\!] \equiv (\lambda x : \tau_1.\ \gamma(e)) \in \mathcal{E}[\![\tau_1 \to \tau_2]\!]$$

Now suppose that $\lambda x : \tau_1.\ \gamma(e) \mapsto^* e'$ and $\text{irred}(e')$. We then need to show $e' \in \mathcal{V}[\![\tau_1 \to \tau_2]\!]$. Since $\lambda x : \tau_1.\ \gamma(e)$ is a value it is irreducible, and we can conclude it took no steps. In other words $e' = \lambda x : \tau_1.\ \gamma(e)$. So we need to show $\lambda x : \tau_1.\ \gamma(e) \in \mathcal{V}[\![\tau_1 \to \tau_2]\!]$. Now suppose $v \in \mathcal{V}[\![\tau_1]\!]$ then we need to show $\gamma(e)[v/x] \in \mathcal{E}[\![\tau_2]\!]$.

Now let us keep the above proof goal in mind and consider the induction hypothesis. From the induction hypothesis we have:

$$\Gamma, x : \tau_1 \models e : \tau_2$$

Instantiate this with $\gamma[x \mapsto v]$. We have $\gamma[x \mapsto v] \in \mathcal{G}[\![\Gamma, x : \tau_1]\!]$ because we started by supposing $\gamma \in \mathcal{G}[\![\Gamma]\!]$ and we also had $v \in \mathcal{V}[\![\tau_2]\!]$. The instantiation gives us $\gamma[x \mapsto v]e \in \mathcal{E}[\![\tau_2]\!] \equiv \gamma(e)[v/x] \in \mathcal{E}[\![\tau_2]\!]$. Now recall our proof goal, we now have what we wanted to show.
Case $\dfrac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}$ T-App, show this case as an exercise.

The remaining cases were not proved during the lecture. □

Now consider what happens if we add pairs to the language. We need to add a clause to the value interpretation:

$$\mathcal{V}[\![\tau_1 \times \tau_2]\!] = \{< v_1, v_2 > \mid v_1 \in \mathcal{V}[\![\tau_1]\!] \wedge v_2 \in \mathcal{V}[\![\tau_2]\!]\}$$

There is nothing surprising in this addition to the value relation, and it should not be a challenge to show the pair case of the proofs.

If we extend our language with sum types.

$$e ::= \ldots \mid \text{inl } v \mid \text{inr } v \mid \text{case } e \text{ of inl } x => e_1 \quad \text{inr } x => e_2$$

Then we need to add the following clause to the value interpretation:

$$\mathcal{V}[\![\tau_1 + \tau_2]\!] = \{\text{inl } v \mid v \in \mathcal{V}[\![\tau_1]\!]\} \cup \{\text{inr } v \mid v \in \mathcal{V}[\![\tau_2]\!]\}$$

It turns out this clause is sufficient. One might think that is is necessary to require the body to be in the expression relation, that is a requirements that looks something like $\forall e_1 \in \mathcal{E}[\![\tau]\!]$. This requirement will, however, give well-foundedness problems, as τ is not a structurally smaller type than $\tau_1 + \tau_2$. It may come as a surprise that we do not need to relate the expressions, as the slogan for logical relations is "Related inputs to related outputs."

Recursive Types

In this subsection we will motivate and introduce recursive types. This will set the scene for lecture 3.

First consider the following program in the untyped lambda calculus:

$$\Omega = (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$$

The interested reader can now try to evaluate the above expression. After a β-reduction and a substitution we end up with Ω again, so the evaluation of this expression diverges. Moreover, it is not possible to assign a type to Ω (again the interested reader may try to verify this by attempting to assign a type). It can hardly come as a surprise that it cannot be assigned a type, as we previously proved that the simply typed lambda calculus is strongly normalising, so if we could assign Ω a type it would not diverge.

We would like to have recursive types. For one they allow us to type things such as lists, trees, and streams. They do, however, also give us non-termination. In an ML like language a declaration of a tree type would look like this:

```
type tree = Leaf
          | Node of int * tree * tree
```

In Java we could define a tree class with an int field and fields for the subtrees:

```
class Tree {
  int value;
  Tree left, right;
}
```

So we can define trees in our programming languages, but we cannot define them in the lambda calculus. So let us first consider what it is we want to define. we want a type that can either be a node or a leaf. A leaf can be represented by the unit (as it here does not carry any information),

and a node is the product of an int and two nodes. We put the two together with the sum type, as it can be either:

$$tree = 1 + (int * tree * tree)$$

This is what we want, but we cannot specify this. We try to define *tree*, but *tree* appears on the right hand side, it is self-referential. Now let us examine what we want. Instead of writing *tree* we use a type variable α:

$$\begin{aligned}\alpha &= 1 + (int \times \alpha \times \alpha) \\ &= 1 + (int \times (int \times \alpha \times \alpha) \times (int \times \alpha \times \alpha)) \\ &\vdots\end{aligned}$$

All the sides of the above equations are equal, they are all trees. We could keep going and get an infinite system of equations. If we keep substituting the definition of α for α we keep getting bigger and bigger types. All of the types are trees, and all of them are finite. If we take the limit of this process, then we end up with an infinite tree, and that tree is the tree we conceptually have in our minds. So what we need is the fixedpoint of the above equation.

Let us define a recursive function who's fixedpoint we want to find:

$$F = \lambda :: type.1 + (int \times \alpha \times \alpha)$$

We want the fixedpoint, that is by definition t such that

$$t = F(t)$$

So we want

$$tree = F(tree)$$

The fixed point of this function is written:

$$\mu\alpha.\ F(\alpha)$$

Here μ is a fixedpoint type constructor. As the above is the fixedpoint, then by definition it should be equal to F applied to it:

$$\mu\alpha.\ F(\alpha) = F(\mu\alpha.\ F(\alpha))$$

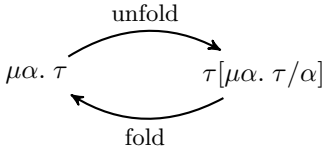
Now let us make this look a bit more like types by substituting $F(\alpha)$ for τ.

$$\mu\alpha.\ \tau = F(\mu\alpha.\ \tau)$$

The right hand side is really just τ with $\mu\alpha.\ \tau$ substituted with τ:

$$F(\mu\alpha.\ \tau) = \tau[\mu\alpha.\ \tau/\alpha]$$

We are going to introduce the recursive type $\mu\alpha.\ \tau$ to our language. When we have a recursive type we can shift our view to an expanded version $\tau[\mu\alpha.\ \tau/\alpha]$ and contract back to the original type. Expanding the type is called fold and contracting is is called unfold.



With recursive types in hand we can now define our tree type:

$$tree \stackrel{\text{def}}{=} \mu\alpha.\ 1 + (int \times \alpha \times \alpha)$$

When we want to work with this we want to be able to get under the μ. Say we have $e : tree$ that is an expression e with type $tree$, then we want to be able to say whether it is a leaf or a node. To do this we unfold the type to get the type where α has been substituted with the definition of $tree$ and the outer $\mu\alpha.$ has been removed. With the outer $\mu\alpha.$ gone we can match on the sum type to find out whether it is a leaf or a node. We can fold the type back to the original tree type, when we are done working with it.

$$tree = \mu\alpha.\ 1 + (int \times \alpha \times \alpha) \quad \underset{\text{fold}}{\overset{\text{unfold}}{\rightleftarrows}} \quad 1 + (int \times (\mu\alpha.\ 1 + (int \times \alpha \times \alpha)) \times (\mu\alpha.\ 1 + (int \times \alpha \times \alpha)))$$

This kind of recursive types is called iso-recursive types, because there is an isomorphism between a $\mu\alpha.\ \tau$ and its unfolding $\tau[\mu\alpha.\ \tau/\alpha]$.

STLC extended with recursive types is defined as follows:

$$\begin{array}{rl}
\tau ::= & \ldots \mid \mu\alpha.\ \tau \\
e ::= & \ldots \mid \text{fold } e \mid \text{unfold } e \\
v ::= & \ldots \mid \text{fold } v \\
E ::= & \ldots \mid \text{fold } E \mid \text{unfold } E
\end{array}$$

$$\text{unfold (fold } v) \mapsto v$$

$$\frac{\Gamma \vdash e : \tau[\mu\alpha.\ \tau/\alpha]}{\Gamma \vdash \text{fold } e : \mu\alpha.\ \tau}\ \text{T-Fold} \qquad \frac{\Gamma \vdash e : \mu\alpha.\ \tau}{\Gamma \vdash \text{unfold } e : \tau[\mu\alpha.\ \tau/\alpha]}\ \text{T-Unfold}$$

With this we could define the type of an integer list as:

$$int\ list \stackrel{\text{def}}{=} \mu\alpha.\ 1 + (int \times \alpha)$$

Exercises

1. Prove the TApp case of the Fundamental Property

Lecture 3, Step-Indexing

This lecture was on extending the unary logical relation for type safety of simply typed lambda calculus to recursive types, introducing universal types, and finally a short introduction to contextual equivalence.

Simply typed lambda calculus extended with μ

In a naive first attempt to make the value interpretation we could write something like

$$\mathcal{V}[\![\mu\alpha.\ \tau]\!] = \{\text{fold}\ \ v \mid \text{unfold (fold}\ \ v) \in \mathcal{E}[\![\tau[^{\mu\alpha.\ \tau}/_{\alpha}]]\!]\}$$

We can simplify this slightly; first we use the fact that unfold (fold v) reduces to v. Next, we use the fact that v must be a value and the fact that we want v to be in the expression interpretation of $\tau[\mu\alpha.\ \tau/\alpha]$. By unfolding the definition of the expression interpretation we conclude that it suffices to require v to be in the value interpretation of the same type. We then end up with the following definition:

$$\mathcal{V}[\![\mu\alpha.\ \tau]\!] = \{\text{fold}\ \ v \mid v \in \mathcal{V}[\![\tau[^{\mu\alpha.\ \tau}/_{\alpha}]]\!]\}$$

This gives us a well-foundness issue. The value interpretation is defined by induction on the type, $\tau[^{\mu\alpha.\ \tau}/_{\alpha}]$ is not a structurally smaller type than $\mu\alpha.\ \tau$.

To solve this issue we index the interpretation by a natural number which we write as follows:

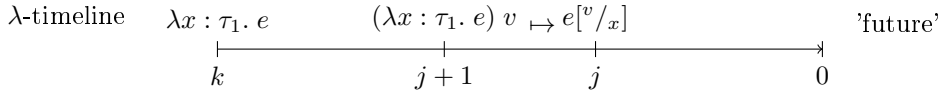
$$\mathcal{V}_k[\![\tau]\!] = \{v \mid \ldots\}$$

If v belongs to the interpretation, then this is read as "v belongs to the interpretation of τ for k steps." We interpret this in the following way; given a value that we run run for k or fewer steps as in the value is used in some program context for k or fewer steps, then we will never notice that it does not have type τ. If we use the same value in a program context that wants to run for more than k steps, then we might notice that it does not have type τ which means that we might get stuck. This gives us an approximate guarantee.

We use this as an inductive metric to make our definition well-founded, so we define the interpretation on induction on the step-index followed by inner induction on the type structure. Let us start by adding the step-index to our existing value interpretation:

$$\begin{aligned}\mathcal{V}_k[\![\mathit{bool}]\!] &= \{\text{true}, \text{false}\}\\ \mathcal{V}_k[\![\tau_1 \to \tau_2]\!] &= \{\lambda x : \tau_1.\ e \mid \forall j \leq k.\ \forall v \in \mathcal{V}_j[\![\tau_1]\!].\ e[^{v}/_{x}] \in \mathcal{E}_j[\![\tau_2]\!]\}\end{aligned}$$

true and false are in the value interpretation of *bool* for any k, so true and false will for any k look like it has type *bool*. To illustrate how to understand the value interpretation of $\tau_1 \to \tau_2$ please consider the following timeline:



Here we start at index k and as we run the program we use up steps until we at some point reach 0 and rund out of steps. At step k we are looking at a lambda. A lambda is used by applying it, but it is not certain that the application will happen right away, as the we only do β-reduction

when we try to apply a lambda to a value, but we might be looking at a context where we want to apply the lambda to an expressions, i.e. $(\lambda x : \tau_1.\ e)\ e_2$. We might use a bunch of steps to reduce e_2 down to a value, but we cannot say how many. So say that sometime in the futurewe have fully evaluated e_2 to v and say that we have $j+1$ steps left at this time, then we can do the β reduction which gives us $e[^v/_x]$ at step j.

We can now define the value interpretation of $\mu\alpha.\ \tau$:

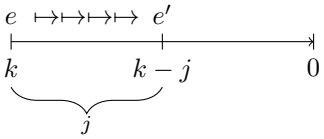
$$\mathcal{V}_k[\![\mu\alpha.\ \tau]\!] = \{\text{fold}\ v \mid \forall j < k.\ v \in \mathcal{V}_j[\![\tau\mu\alpha.\ \tau[^\alpha/_{\_}]]\!]\}$$

This definition is like the one we previously proposed, but with a step-index. This definition is well-founded because j is required to be *strictly* less than k and as we define the interpretation on induction over the step-index this is indeed well founded. We do not define a value interpretation for type variables α, as we have no polymorphism yet. The only place we have a type variable at the moment is in $\mu\alpha.\ \tau$, but in the interpretation we close of the τ under the μ, so we will never encounter a free type variable.

Finally, we define the expression interpretation:

$$\mathcal{E}_k[\![\tau]\!] = \{e \mid \forall j < k.\ \forall e'.\ e \mapsto^j e'\ \wedge\ \text{irred}(e') \implies e' \in \mathcal{V}_{k-j}[\![\tau]\!]\}$$

To illustrate what is going on here consider the following timeline:



We start with an expression e, then we take j steps and get to expression e'. At this point if e' is irreducable, then we want it to belong to the value interpretation of τ for $j-k$ steps. The reason we use a strict inequality is that we do not want to hit 0 steps. If we hit 0 steps, then all bets are off.

We also need to lift the interpretation of type environments to step-indexing:

$$\begin{aligned}\mathcal{G}_k[\![\bullet]\!] &= \{\emptyset\}\\ \mathcal{G}_k[\![\Gamma, x:\tau]\!] &= \{\gamma[x \mapsto v] \mid \gamma \in \mathcal{G}_k[\![\Gamma]\!]\ \wedge\ v \in \mathcal{V}_k[\![\tau]\!]\}\end{aligned}$$

$$\Gamma \models e : \tau \stackrel{\text{def}}{=} \forall k \geq 0.\ \forall \gamma \in \mathcal{G}_k[\![\Gamma]\!] \implies \gamma(e) \in \mathcal{E}_k[\![\tau]\!]$$

Theorem (Fundemental property)**.** *If* $\Gamma \vdash e : \tau$ *then* $\Gamma \models e : \tau$.

(b) $\bullet \models e : \tau \implies \text{safe}(e)$

Lemma. *If* $v \in \mathcal{V}_k[\![\tau]\!]$ *and* $j \leq k$ *then* $v \in \mathcal{V}_j[\![\tau]\!]$.

Proof. We prove this by case on τ.
Case $\tau = bool$, assume $v \in \mathcal{V}_k[\![bool]\!]$ and $j \leq k$, we then need to show $v \in \mathcal{V}_j[\![bool]\!]$. As $v \in \mathcal{V}_k[\![bool]\!]$ we know that either $v = \text{true}$ or $v = \text{false}$. If we assume $v = \text{true}$, then we immidiately get what we want to show, as true is in $\mathcal{V}_j[\![bool]\!]$ for any j. Likewise for the case

$v = \text{false}$.
Case $\tau = \tau_1 \to \tau_2$, assume $v \in \mathcal{V}_k[\![\tau_1 \to \tau_2]\!]$ and $j \leq k$, we then need to show $v \in \mathcal{V}_j[\![\tau_1 \to \tau_2]\!]$. As v is a member of $\mathcal{V}_k[\![\tau_1 \to \tau_2]\!]$ we can conclude that $v = \lambda x : \tau_1.\ e$ for some e. By definition of $v \in \mathcal{V}_j[\![\tau_1 \to \tau_2]\!]$ we need to show $\forall i \leq j. \forall v' \in \mathcal{V}_i[\![\tau_1]\!].\ v' \in \mathcal{E}_i[\![\tau_2]\!]$. Suppose $i \leq j$ and $v' \in \mathcal{V}_i[\![\tau_1]\!]$, we then need to show $v' \in \mathcal{E}_i[\![\tau_2]\!]$.

By assumption we have $v \in \mathcal{V}_k[\![\tau_1 \to \tau_2]\!]$ which gives us $\forall n \leq k. \forall v' \in \mathcal{V}_n[\![\tau_1]\!].\ v' \in \mathcal{E}_n[\![\tau_2]\!]$. From $j \leq k$ and $i \leq j$ we get $i \leq k$ by transitivity. We use this with $v' \in \mathcal{V}_i[\![\tau_1]\!]$ to get $v' \in \mathcal{E}_i[\![\tau_2]\!]$ which is what we needed to show.
Case $\tau = \mu\alpha.\ x$, assume $v \in \mathcal{V}_k[\![\mu\alpha.\ \tau]\!]$ and $j \leq k$, we then need to show $v \in \mathcal{V}_j[\![\mu\alpha.\ \tau]\!]$. From v's assumed membership of the value interpretation of τ for k steps we conclude that there must exist a v' sucht that $v = \text{fold}\ v'$. If we suppose $i < j$, then we need to show $v' \in \mathcal{V}_i[\![\tau[\mu\alpha.\ \tau/\alpha]]\!]$. From $i < j$ and $j \leq k$ we can conclude $i < k$ which we use with $\forall n < k.\ v' \in \mathcal{V}_n[\![\tau[\mu\alpha.\ \tau/\alpha]]\!]$, which we get from $v \in \mathcal{V}_k[\![\mu\alpha.\ \tau]\!]$, to get $v' \in \mathcal{V}_i[\![\tau[\mu\alpha.\ \tau/\alpha]]\!]$. □

Proof (Fundemental Property). □

$$\mathcal{V}_k[\![\tau_1 + \tau_2]\!] = \{\text{inl}\ v_1 \mid v_1 \in \mathcal{V}_k[\![\tau_1]\!]\} \cup \{\text{inr}\ v_2 \mid v_2 \in \mathcal{V}_k[\![\tau_2]\!]\}$$

Universal Types

$$sort\ :\ \forall\alpha.\ (list\ \alpha) \times (\alpha \to bool) \to list\ \alpha$$

$$sort[int](3, 5, 7) <$$

$$\Lambda\alpha.\ e$$

System F (Simply Typed Lambda Calculus With Universal Types)

$$\begin{aligned}
\tau &::= \ldots \mid \forall\alpha.\ \tau \\
e &::= \ldots \mid \Lambda\alpha.\ e \mid e[\tau] \\
v &::= \ldots \mid \Lambda\alpha.\ e \\
E &::= \ldots \mid \text{fold}\ E[\tau]
\end{aligned}$$

$$(\Lambda\alpha.\ e[\tau]) \mapsto e[\tau/\alpha]$$

Type environment

$$\Delta ::= \bullet \mid \Delta,\ \alpha$$

Type judegement form:

$$\Delta, \Gamma \vdash e : \tau$$

Well formed type:

$$\Delta \vdash \tau \stackrel{\text{def}}{=} \text{FTV}(\tau) \subseteq \Delta$$

where $\text{FTV}(\tau)$ is the set of free type variables in τ.

Well-formed environment

$$\Delta \vdash \Gamma \stackrel{\text{def}}{=} \forall x \in \text{dom}(\Gamma).\ \Delta \vdash \Gamma(x)$$

For any type judgement $\Delta, \Gamma \vdash e : \tau$ we have as an invariant that $\Delta \vdash \Gamma$. Typing rules:

$$\frac{\Delta \vdash \Gamma}{\Gamma \vdash \text{false} : \mathit{bool}}\ \text{T-False} \qquad \frac{\Delta \vdash \Gamma}{\Gamma \vdash \text{true} : \mathit{bool}}\ \text{T-True}$$

$$\frac{\Gamma(x) = \tau \qquad \Delta \vdash \Gamma}{\Gamma \vdash x : \tau}\ \text{T-Var} \qquad \frac{\Gamma \vdash e : \mathit{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \Delta \vdash \Gamma}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau}\ \text{T-If}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \qquad \Delta \vdash \Gamma}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}\ \text{T-Abs} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2 \qquad \Delta \vdash \Gamma}{\Gamma \vdash e_1\ e_2 : \tau}\ \text{T-App}$$

Theorem. *If* $\bullet; \bullet \vdash e : \forall\alpha.\ \alpha \to \alpha$,
$\bullet \vdash \tau$, *and*
$\bullet; \bullet \vdash v : \tau$
then $e[\tau]\ v \mapsto^* v$

Exercises

1. Do the lambda and application case of the *Fundemental Property* theorem.
2. Try to prove the monotonicity lemma where the definition of the value interpretation has been adjusted with:

   $$\mathcal{V}_k[\![\tau_1 \to \tau_2]\!] = \{\lambda x : \tau_1.\ e \mid\ \forall v \in \mathcal{V}_k[\![\tau_1]\!].\ e[^v/_x] \in \mathcal{E}_k[\![\tau_2]\!]\}$$

   This will fail, but it is instructive to see how it fails.