OPLSS15, Notes for Amal Ahmed's lectures.

Lau Skorstengaard
lask@cs.au.dk

July 30, 2015

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

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

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

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-foundedness 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:

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

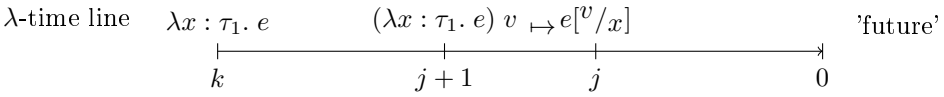
If v belongs to the interpretation, then this is read as "v belongs to the interpretation of τ for k steps." We interpret this in the following way; given a value that we run run for k or fewer steps as in the value is used in some program context for k or fewer steps, then we will never notice that 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 time line:



Here we start at index k and as we run the program we use up steps until we at some point reach 0 and run 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 future 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 $\mu\alpha.\ \tau$:

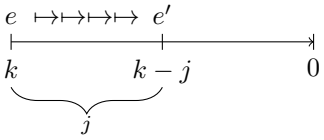
$$\mathcal{V}_k[\![\mu\alpha.\ \tau]\!] = \{\text{fold}\ \ v \mid \forall j < k.\ v \in \mathcal{V}_j[\![\tau[^{\mu\alpha.\ \tau}/_\alpha]]\!]\}$$

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 time line:



We start with an expression e, then we take j steps and get to expression e'. At this point if e' is irreducible, 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 (Fundamental property)**.**
If $\Gamma \vdash e : \tau$ *then* $\Gamma \models e : \tau$.

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

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 $\tau = \mathit{bool}$, assume $v \in \mathcal{V}_k[\![\mathit{bool}]\!]$ and $j \leq k$, we then need to show $v \in \mathcal{V}_j[\![\mathit{bool}]\!]$. As $v \in \mathcal{V}_k[\![\mathit{bool}]\!]$ we know that either $v = \text{true}$ or $v = \text{false}$. If we assume $v = \text{true}$, then we immediately get what we want to show, as true is in $\mathcal{V}_j[\![\mathit{bool}]\!]$ 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' such 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). Proof by induction over the typing derivation. □

The example with lists from the previous lecture used the sum type. Sums are a straight forward extension of the language. The extension of the value interpretation would be:

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

We can use k directly or k decremented by one. It depends on whether we want casing to take up a step. Either way the definition is well-founded.

Universal Types

Now we shift focus from type safety and termination to program equivalence. To motivate the need for arguing about program equivalence we first introduce universal types.

Say we want to implement a sort function. Maybe we even have a function that sorts integer lists:

$$\mathit{sortint}\ :\ \mathit{list\ int} \to \mathit{list\ int}$$

sortint takes a list of integers and returns a sorted version of that list. Say we now want a function that sorts lists of strings, then instead of implementing a separate function we could factor the code and have just one function. The type signature of such a generic sort function is:

$$\mathit{sort}\ :\ \forall\alpha.\, (\mathit{list}\ \alpha) \times (\alpha \times \alpha \to \mathit{bool}) \to \mathit{list}\ \alpha$$

sort takes a type, a list of elements of this type, and a comparison that compares to elements of this type and returns a list sorted according to the comparison function. And example of an application of this function could be

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

Whereas sort instantiated with the *string* type, but given an integer list would not be a well typed instantiation.

$$sort\ [string]\ \cancel{(3,7,5)}\ (\text{"}a\text{"},\text{"}c\text{"},\text{"}b\text{"})\ string <$$

Here the application with the list $(3, 7, 5)$ is not well typed, but if we instead use a list of strings, then it is well typed.

We want to extend the simply typed lambda calculus with functions that abstract over types in the same way lambda abstractions, $\lambda x : \tau.\ e$, abstract over terms. We do that by introducing a type abstraction:

$$\Lambda\alpha.\ e$$

This function abstracts over the type α which allows e to depend on α.

System F (Simply Typed Lambda Calculus With Universal Types)

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

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

Type environment[1]

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

With the addition of type environments of type variables we our typing judgments now have the following form:

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

We now need a notion of well-formed types. If τ is well formed with respect to Δ, then we write:

$$\Delta \vdash \tau$$

We do not include the formal rules here, but they amount to $\mathrm{FTV}(\tau) \subseteq \Delta$, where $\mathrm{FTV}(\tau)$ is the set of free type variables in τ.

We further introduce a notion of well formed environments. An environment is well formed if all the types that appear in the range of Γ are well formed.

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

[1] We do not annotate α with a kind, as we only have one kind in this language.

For any type judgment $\Delta, \Gamma \vdash e : \tau$ we have as an invariant that *tau* is well-formed in Δ and Γ is well formed in Δ. The old typing system extended with the new form of the typing judgment looks like this:

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

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

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

Notice that the only thing that has changed is that Δ has been added to the environment in the judgments. We further extend the typing rules with the following two rules to account for our new language constructs:

$$\frac{\Delta; \Gamma \vdash e : \forall \alpha.\tau \qquad \Delta \vdash \tau'}{\Delta; \Gamma \vdash e[\tau'] : \tau[^{\tau'}/_{\alpha}]}\ \text{T-TApp} \qquad \frac{\Delta, \alpha; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\alpha.\ e : \forall \alpha.\tau}\ \text{T-TAbs}$$

Properties of System-F

In System-F certain types reveal the behaviour of the functions with that type. Let us consider terms with the type $\forall \alpha.\alpha \to \alpha$. Recall from the motivation in lecture 1 that this had to be the identity function. We can now phrase this as a theorem:

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

This is a free theorem in this language. Another free theorem that was mentioned in the motivation of lecture 1 was about expressions with type $\forall \alpha.\alpha \to \mathit{bool}$. Here all expressions with this type had to be constant functions. We can also phrase this as a theorem

Theorem. *If* $\bullet \vdash \tau$, $\bullet \vdash v_1 : \tau$, *and* $\bullet \vdash v_1 : \tau$, *then* $e[\tau]\ v_1 \approx^{ctx} e[\tau]\ v_2$.

Or in a slightly more general fashion where we allow different types:

Theorem. *If* $\bullet \vdash \tau$, $\bullet \vdash \tau'$, $\bullet \vdash v_1 : \tau$, *and* $\bullet \vdash v_1 : \tau'$, *then* $e[\tau]\ v_1 \approx^{ctx} e[\tau']\ v_2$.

[2]The reason we get these free theorems is that the function has no way of inspecting the argument as it does not know what type it is. As the function has to treat its argument as an unknown 'blob' it has no choice but to return the same value every time.

The question now is: "how do we prove these free theorems?" The two last theorems both talk about program equivalence which we prove using logical relations. The first theorem did not mention equivalence, but the proof technique of choice is still logical relations.

[2]We have not yet defined $\approx^{ctx}$ so for now just treat it as the two programs are equivalent without thinking too much about what equivalence means.

Contextual Equivalence

To define a contextual equivalence we first define the notion of a program context. A program context is a complete program with exactly one hole in it. It is defined as follows:

$$
\begin{aligned}
C ::= {} & [\cdot] \\
& \mid \text{if } C \text{ then } e \text{ else } e \\
& \mid \text{if } e \text{ then } C \text{ else } e \\
& \mid \text{if } e \text{ then } e \text{ else } C \\
& \mid \lambda x : \tau.\ C \\
& \mid C\ e \\
& \mid e\ C \\
& \mid \Lambda\alpha.\ C \\
& \mid C[\tau]
\end{aligned}
$$

We need a notion of context typing. For simplicity we just introduce it for simply typed lambda calculus. The context typing is written as:

$$C :\ (\Gamma \vdash \tau) \implies (\Gamma' \vdash \tau') \implies \Gamma \vdash e : \tau \implies \Gamma' \vdash C[e] : \tau'$$

This means that for any expression e of type τ under Γ. If we embed it into C, then the type of the embedding is τ' under Γ'.

Informally we want contextual equivalence to say that no matter what program context we embed either of the two expressions in, it gives the same result. This is also called as observational equivalence as the program context is unable to observe any difference no matter what expression we embed in it. We can of course not plug anything into the hole, so we annotate the equivalence with the type of the hole which means that the two contextual equivalent expressions have to have that type.

$$\Delta; \Gamma \vdash e_1 \approx^{ctx} e_2 : \tau \stackrel{\text{def}}{=} \forall C.\ (\Delta; \Gamma \vdash \tau) \implies (\bullet; \bullet \vdash \tau') \implies (C[e_1] \Downarrow v \iff C[e_2] \Downarrow v)$$

This definition assumes that e_1 and e_2 has type τ under the contexts specified.

Contextual equivalence is handy, because we want to be able to reason about the equivalence of two implementations. Say we have two implementations of a stack, one is implemented using an array the other using a list. If we can show that the two implementations are contextual equivalent, then we can use the more efficient one over the less efficient one and know that the complete program will behave the same.

In the next lecture we will introduce a logical relation such that

$$\Delta; \Gamma \vdash e_1 \approx^{LR} e_2 : \tau \implies \text{contextual equivalence } \approx^{ctx}$$

That is we want to show that the logical relation is sound with respect to contextual equivalence.

If we can prove the above soundness, then we can state our free theorems with $\approx^{LR}$ rather than $\approx^{ctx}$ and get the same result, if we can prove the logical equivalence. We would like to do this as it is difficult to directly prove two things are contextual equivalent. A direct proof has to talk about all possible program contexts which we could do using induction, but the lambda-abstraction case turns out to be difficult. This motivates the use of other proof methods where logical relations is one of them.

Exercises

1. Do the lambda and application case of the *Fundamental 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.