Secure Compilation
Lecture 2
Closure Conversion

Renate Robin Eilers  Cristina Matache  Baber Rehman

June 24, 2019

This is the second talk presented by Amal Ahmed in OPLSS 2019, University of Oregon, USA.

1 Source Language

Types  We just have integers and function in source language.

$$\sigma ::= \text{int} \mid \sigma_1 \rightarrow \sigma_2$$

Terms

$$v ::= x \mid n \mid \lambda x : \sigma. e$$

$$e ::= v \mid \text{if} 0 v e_1 e_2 \mid v_1 v_2 \mid \text{let} \; x = e_1 \text{ in } e_2$$

So $e_1 \; e_2$ is a shorthand for $\text{let} \; x = e_1 \text{ in } \text{let} \; y = e_2 \text{ in } x \; y$.

Evaluation contexts:

$$E ::= [\cdot] \mid \text{let} \; x = E \text{ in } e_2$$

The language has a typing judgement $\Gamma \vdash e : \sigma$ and a small-step call-by-value operational semantics $e \rightarrow e'$.

2 Target Language

Types and terms

$$\tau ::= \text{int} \mid (\tau_1, \ldots, \tau_n) \rightarrow \tau' \mid \{\tau_1, \ldots, \tau_n\} \mid \alpha \mid \exists \alpha. \tau$$

$$v ::= x \mid n \mid \lambda (\tau_1, \ldots, \tau_n). e \mid \langle v_1, \ldots, v_n \rangle \mid \text{pack}(\tau, v) \text{ as } \exists \alpha. \tau$$

$$e ::= v \mid \text{if} 0 v e_1 e_2 \mid v_1 (\overrightarrow{v}) \mid \pi_i v \mid \text{unpack}(\alpha, x) = v \text{ in } e_1 \mid \text{let} \; x = e_1 \text{ in } e_2$$
Typing contexts:

\[ \Delta ::= \cdot \mid \Delta, \alpha \]
\[ \Gamma ::= \cdot \mid \Gamma, x : \tau \]

Typing judgements: \[ \Delta, \Gamma \vdash e : \tau \]
To do closure conversion, we want functions to have a closed body:

\[ \Delta \vdash x : \tau \]
\[ \Delta ; \Gamma \vdash \lambda (x : \tau). e : (\tau' \to \tau) \]
\[ \Delta ; \Gamma \vdash v : \tau[\tau'/\alpha] \]
\[ \Delta ; \Gamma \vdash \text{pack}(\tau', v) \text{ as } \exists \alpha. \tau : \exists \alpha. \tau \]

Example 1. A term of type \( \exists \alpha. \alpha \times (\alpha \to \text{int}) \) is:

\[ w = \text{pack}(\text{bool}, (\text{true}, \lambda x : \text{bool}.5)) \text{ as } \exists \alpha. \alpha \times (\alpha \to \text{int}) \]

\[ \Delta; \Gamma \vdash v : \exists \alpha. \tau \]
\[ \Delta, \alpha; \Gamma, x : \tau \vdash e_2 : \tau_2 \]
\[ \Delta \vdash \tau_2 \]
\[ \Delta; \Gamma \vdash \text{unpack}(\alpha, x) = v \text{ in } e_2 : \tau_2 \]

In the rule above \( \alpha \) is not allowed to appear in \( \tau_2 \).

Example 2. A well-typed term is:

\[ \text{unpack}(\alpha, x) = w \text{ in } (\pi_2 x) (\pi_1 x) \]

where \( w \) is defined as in the previous example.

3 Translation

Translation of types: \( \sigma^+ \)

\[ \text{int}_S^+ = \text{int}_T \]
\[ (\sigma_1 \to \sigma_2)^+ = \exists \alpha_{\text{env}}. ((\alpha_{\text{env}}, \sigma_1^+ \to \sigma_2^+), \alpha_{\text{env}}) \]

Typing context translation: \( \Gamma_S^+ \)

\[ ()^+ = \cdot \]
\[ (\Gamma_S, x_S : \sigma)^+ = \Gamma_S^+, x_T : \sigma^+ \]
Term translation: \( \Gamma_S \vdash e_S : \sigma \rightsquigarrow e_T \) where \( \cdot \vdash e_T : \sigma^+ \)

\[
\begin{align*}
\Gamma_S(x_S) & = \sigma \\
\Gamma_S \vdash x_S : \sigma \rightsquigarrow x_T \\
\Gamma_S \vdash n_S : \text{int}_S \rightsquigarrow n_T \\
y_{S_1}, \ldots, y_{S_n} & = \text{free variables}(\lambda x_S : \sigma. e_S) \\
v_{\text{code}} & = \lambda(x_T : (\sigma^+_1, \ldots, \sigma^+_n)) . e_T[(\pi_i z)/y_T_i] \\
\Gamma_S \vdash \lambda x_S : \sigma. e_S : \sigma \rightarrow \sigma' \rightsquigarrow \text{pack}(\langle \sigma^+_1, \ldots, \sigma^+_n \rangle, \langle v_{\text{code}}, (y_{T_1}, \ldots, y_{T_n}) \rangle) \text{ as } (\sigma \rightarrow \sigma')^+ \\
\end{align*}
\]

where \( e_T[(\pi_i z)/y_T_i] \) is a shorthand for

\[
\begin{align*}
\text{let } y_{T_1} = \pi_1 z \text{ in } \ldots \text{let } y_{T_n} = \pi_n z \text{ in } e_T \\
\Gamma_S \vdash v_{S_1} : \sigma_2 \rightsquigarrow v_{T_2} \\
\Gamma_S \vdash v_{S_1} : \sigma \rightsquigarrow v_{T_1} \\
\Gamma_S \vdash v_{S_1}, v_{S_2} : \sigma \rightsquigarrow \text{unpack}(\alpha, p) = v_{T_1} \text{ in } (\pi_1 p, (\pi_2 p, v_{T_2}))
\end{align*}
\]

The rules for \textbf{if} and \textbf{let} are defined according to the structure of the terms.

4 Preservation Theorem

**Theorem 4.1 (Type Preservation).** If \( \Gamma \vdash e_S : \sigma \) and \( \Gamma \vdash e_S : \alpha \rightsquigarrow e_T \) then \( \Gamma^+_S \vdash e_T : \sigma^+ \)

For correctness, we want to show \( e_S \approx e_T \). This is not contextual equivalence because source language and target language are two different languages. There are many ways to prove compiler correction. We want to say that:

\[
\text{when } e_S \approx e_T \text{ then } \sigma \approx \sigma^+
\]

5 Logical Relations

In logical relations we map related input to related outputs. Same source value and target value are related.

**Values** \( V[\sigma] = \{(v_S, v_T) | \cdot \vdash v_S : \sigma \wedge \cdot \vdash v_T : \sigma^+ \ldots \} \)

**Integers** \( V[\text{ints}] = \{(n_S, n_T)\} \)

**Function** \( V[\sigma_1 \rightarrow \sigma_2] = \{(\lambda x : \sigma_1 . e_S \text{ pack } (\tau_{\text{env}}, (\lambda (Z : T . x_T : \sigma^+_1) . e_T, V_{\text{env}})) \mid \forall (v_S, v_T) E V[\sigma_1] . (e_S[v_S/x_S], e_T [v_{\text{env}}/Z, v_T/x_T]) \in E[\sigma_2]\} \)

3
Typing Judgement \[ E[\sigma] = \{(e_S, e_T) : \vdash e_S : \sigma \land \vdash e_T : \sigma^+ \land \forall v_S \cdot e_S \rightarrow^* v_S \Rightarrow \exists v_T \cdot e_T \rightarrow^* v_T \} \]

Target language behaviour is shown in source language.

Definition 5.0.1. \[ \Gamma_S \vdash e_S \approx e_T : \sigma = \Gamma_S \vdash e_S : \sigma \land \cdot \vdash e_T : \sigma^+ \land \forall (\gamma_S, \gamma_T) \in G[\sigma] \cdot (\gamma_S(e_S), \gamma_T(e_T)) \in E[\sigma]. \]

\( S \) and \( T \) are like holes in expression. They are not complete program. This is like substitution and linking. It will be combined with other code.

\[ \gamma_S = \{ x_S \mapsto v_S \} \]
\[ \gamma_T = \{ x_T \mapsto v_T \} \]

\[ G[\cdot] = \{ \phi \cdot \phi \} \]
\[ G[x_S : \sigma] = \{ (\gamma_S[x_S \mapsto v_S], \gamma_T[x_T \mapsto v_T]) \mid (\gamma_S, \gamma_T) \in G[x_S : \sigma] \land (v_S, v_T) \in V[\sigma] \} \]

Theorem 5.1 (Compiler Correctness). If \( \Gamma_S \vdash e_S : \sigma \Rightarrow e_T \) then \( \Gamma \vdash e_S \approx e_T : \sigma. \)

Proof. This theorem can be proved by induction on typing derivation of source language. It can be proved just by unfolding the definitions. \( \square \)

Lemma 5.2 (Fundamental Property). \( \Gamma \vdash e_S : \sigma \Rightarrow \Gamma \vdash e_S \approx e_T : \sigma \)