Chapter 11

Newton Ni

September 21, 2020

1 11.4.1

Show how to formulate ascription as a derived form. Prove that the "official" typing and evaluation rules given here correspond to your definition in a suitable sense.

We can formulate ascription using the following transformation `Lower`:

$$\begin{aligned}\texttt{Lower(t as T)} &= (\lambda \texttt{x} : \texttt{T}.\ \texttt{x})\ \texttt{t} \\ \texttt{Lower(t)} &= \texttt{t}\end{aligned}$$

We wish to prove the following:

$$\Gamma \vdash \texttt{t} : \texttt{T} \iff \Gamma \vdash \texttt{Lower(t)} : \texttt{T} \tag{1}$$

$$\texttt{t} \longrightarrow \texttt{t}' \iff \texttt{Lower(t)} \longrightarrow \texttt{Lower(t')} \tag{2}$$

We induct on terms `t`. For both (1) and (2), all cases except $\texttt{t = t}_1\texttt{ as T}$ follow from straightforward induction. For the forward case of (1), we know the following:

1.F.1 $\Gamma \vdash \texttt{t}_1 : \texttt{T}$

1.F.2 $\texttt{Lower(t}_1\texttt{ as T)} = (\lambda \texttt{x} : \texttt{T}.\ \texttt{x})\ \texttt{t}_1$

The conclusion follows from applying T-App to (1.F.2), after type-checking the identity function with T-Abs, T-Var, and (1.F.1) (proof tree omitted).

For the backward case of (1), we know:

1.B.1 $\Gamma \vdash \texttt{Lower(t}_1\texttt{ as T)} : \texttt{T}$

1.B.2 $\Gamma \vdash (\lambda \texttt{x} : \texttt{T}.\ \texttt{x})\ \texttt{t}_1 : \texttt{T}$

Applying T-Abs, T-Var in reverse (probably need lemma to prove this is okay, as there is only one typing judgment each for these syntactic forms) yields (1.B.3): $\texttt{t}_1 : \texttt{T}$, which is enough to finish with T-Ascribe.

For the forward case of (2), there are two sub-cases: when $\mathtt{t_1}$ is a value or not. When it is a value $\mathtt{v_1}$, we apply E-Ascribe to the left-hand side and have $\mathtt{v_1\ as\ T} \longrightarrow \mathtt{v_1}$. On the right-hand side, we want to show that $(\lambda\mathtt{x} : \mathtt{T}.\,\mathtt{x})\ \mathtt{v_1} \longrightarrow \mathtt{v_1}$. This follows directly from E-AppAbs.

When $\mathtt{t_1}$ is not a value, we apply E-Ascribe1 to the left-hand side and get $\mathtt{t_1\ as\ T} \longrightarrow \mathtt{t_1}'$. On the right-hand side, since the identity function is a value, the conclusion follows from applying E-App2 and the determinancy of small-step evaluation.

For the backward case of (2), we again distinguish when $\mathtt{t_1}$ is a value or not, and the logic is similar to above.

Suppose that, instead of the pair of evaluation rules E-Ascribe *and* E-Ascribe1*, we had given an "eager" rule that throws away an ascription as soon as it is reached. Can ascription still be considered as a derived form?*

It depends on the evaluation strategy for the base lambda calculus. For the evaluation rules in the textbook, it would no longer be a derived form, as E-AscribeEager would diverge from the behavior of E-App2 when $\mathtt{t_1}$ is not a value. But if we changed the evaluation strategy to apply functions to their argument eagerly as well, then this would be a derived form.

2 11.5.2

Not sure how to define "good idea", but this seems to tangle the evaluation and typing judgments (requiring substitions in order to type-check), and require more computation. I suppose errors would also be harder to track to their source spans, if the error arises within substituted code?

3 11.8.2

We can add a simple form of pattern matching to an untyped lambda calculus with records by adding a new syntactic category of patterns . . . *Give typing rules for the new constructs (making any changes to the syntax you feel are necessary in the process).*

$$\frac{\mathit{match}(p, \mathtt{v_1}) = \sigma \qquad \Gamma \vdash \sigma\ \mathtt{t_2} : \mathtt{T}}{\Gamma \vdash \mathit{match}(\mathtt{p}, \mathtt{v_1})\ \mathtt{t_2} : \mathtt{T}} \ \textsc{T-Match}$$

Sketch a proof of type preservation and progress for the whole calculus.

q

4 11.11.1

We can define factorial as:

$$\begin{aligned}
&\texttt{fix}\\
&\lambda\texttt{factorial} : \texttt{int} \rightarrow \texttt{int}.\\
&\lambda\texttt{x} : \texttt{int}.\\
&\texttt{if x } = 0 \texttt{ then } 1 \texttt{ else factorial } (\texttt{x} - 1) * \texttt{x}
\end{aligned}$$

5 11.11.2

We can rewrite factorial as:

$$\begin{aligned}
&\texttt{letrec factorial: } \texttt{int} \rightarrow \texttt{int} =\\
&\lambda\texttt{x} : \texttt{int}.\\
&\texttt{if x } = 0 \texttt{ then } 1 \texttt{ else factorial } (\texttt{x} - 1) * \texttt{x}
\end{aligned}$$

6 11.12.1

Verify that the progress and preservation theorems hold for the simply typed lambda-calculus with booleans and lists.

7 11.12.2

The presentation of lists here includes many type annotations that are not really needed, in the sense that the typing rules can easily derive the annotations from context. Can all *the type annotations be deleted?*

The type annotation for `nil` cannot be deleted, because there's no argument to derive the type from. But all of the other syntactic forms for lists (`cons`, `isnil`, `head`, `tail`) can check the type of their argument, so their type annotations are unnecessary.