Proof of Unique Typing for MinML Programs

Venkatesh Choppella

February 17, 2002

Abstract

This note states and proves the Unique Typing Lemma for MinML Programs.

1 Introduction

The Unique Typing lemma asserts that if a MinML program is typable in a given type environment, then that type is unique for that program and environment. In class, there was a lot of discussion about the statement of this property. However, the proof was done quite informally with a lot of handwaving. The proof is worked out in a more formal way with a level of detail that makes it resemble an assembly language program. Such a detailed style is seldom attempted by hand, but sometimes it helps to make all the steps explicit.

In class, the proof was presented using induction on derivations, and as several of you pointed out, it is easier to use induction on programs for this proof. The proof presented here uses induction on programs. It relies on the Inversion Lemma given in the MinML document. Please refer to Amr's notes for the definition of the Inversion Lemma, the syntax of programs and the type rules of MinML.

We use the "obvious" and ubiquitous rule of matching known more formally as the Principle of Extensionality for inductively defined terms (essentially trees). Extensionality says that two trees are identical if their roots are identical and the corresponding subtrees are identical. This principle holds for programs because they can be viewed as trees.

So, for instance, if $p = o(p_1, p_2)$ and $p' = o'(p'_1, p'_2)$ where $p, p', p_1, p'_1, p_2, p'_2$ are programs, and $o, o \in \{+, -, /, *, <, =\}$, then by appealing to extensionality, we can claim that $p = p'$ if and only if $o = o'$, $p_1 = p'_1$ and $p_2 = p'_2$.

2 Unique Typing Lemma

Notation: Each case of a proof is written out in a tabular fashion. Each row of the proof has a number, a proof step, and a justification as to how that step was derived. The justification is of the form $Rule(x, y...)$, where $Rule$ is a rule in logic, a Lemma that we refer to, or a type system rule. $x, y...$ denote steps of the proof already derived that are needed for $Rule$ to be applicable.

The TRANS rule denotes transitivity of equality. We also use the Substitution rule when we want to substitute e_1 for e_2 (or vice versa) in an entity e, given that $e_1 = e$ is proven. This is abbreviated $Subst(x, y)$. Here x is the label of the step deriving $e_1 = e_2$, and y is the step containing e.

Lemma 2.1 (Unique typing for MinML Programs)
If $\Gamma \vdash p : t$ *and* $\Gamma \vdash p : t'$*, then* $t = t'$.

Proof By induction on p.

We pick arbitrary Γ, p, t, t' and assume the following:

$$\begin{array}{ll} A. & \Gamma \vdash p : t \\ B. & \Gamma \vdash p : t' \end{array}$$

The rest of the proof is by cases on p. In all the cases the proof is by applying Substitution, then invoking the Inversion Lemma, and then either the Transitivity rule TRANS, or the induction hypothesis IH (for the **if** case).

1. $p = n$ for some integer n:

1.	$p = n$	Assumption for this case
2.	$\Gamma \vdash n : t$	Subst(1, A)
3.	$\Gamma \vdash n : t'$	Subst(1, B)
4.	$t =$ `int`	Inversion(2)
5.	$t' =$ `int`	Inversion(3)
6.	$t = t'$	TRANS(4,5)

2. $p =$ true: This case is obtained from the previous case by replacing n and `int` with true and `bool`, respectively.

1.	$p =$ true	Assumption for this case
2.	$\Gamma \vdash$ true $: t$	Subst(1, A)
3.	$\Gamma \vdash$ true $: t'$	Subst(1, B)
4.	$t =$ `bool`	Inversion(2)
5.	$t' =$ `bool`	Inversion(3)
6.	$t = t'$	TRANS(4,5)

3. $e =$ false: This case is obtained by replacing true with false in the previous case.

1.	$p =$ false	Assumption for this case
2.	$\Gamma \vdash$ false $: t$	Subst(1, A)
3.	$\Gamma \vdash$ false $: t'$	Subst(1, B)
4.	$t =$ `bool`	Inversion(2)
5.	$t' =$ `bool`	Inversion(3)
6.	$t = t'$	TRANS(4,5)

4. $p = x$ for some variable x:

1.	$p = x$	Assumption for this case
2.	$\Gamma \vdash x : t$	Subst(1, A)
3.	$\Gamma \vdash x : t'$	Subst(1, B)
4.	$\Gamma(x) = t$	Inversion(2)
5.	$\Gamma(x) = t'$	Inversion(3)
6.	$t = t'$	TRANS(4,5)

5. $p = o(p_1, p_2)$ for $o \in \{+, -, *, /\}$ and for some programs p_1, p_2

1.	$p = o(p_1, p_2)$	Assumption for this case
2.	$\Gamma \vdash o(p_1, p_2) : t$	Subst(1, A)
3.	$\Gamma \vdash o(p_1, p_2) : t'$	Subst(1, B)
4.	$t =$ `int`	Inversion(2)
5.	$t' =$ `int`	Inversion(3)
6.	$t = t'$	TRANS(4,5)

6. $p = o(p_1, p_2)$ for $o \in \{<, =\}$ and for some programs p_1, p_2. This case is obtained by replacing `int` with `bool` in the previous case.

1.	$p = o(p_1, p_2)$	Assumption for this case
2.	$\Gamma \vdash o(p_1, p_2) : t$	Subst(1, A)
3.	$\Gamma \vdash o(p_1, p_2) : t'$	Subst(1, B)
4.	$t = \texttt{bool}$	Inversion(2)
5.	$t' = \texttt{bool}$	Inversion(3)
6.	$t = t'$	TRANS(4,5)

7. $p =$ **if** p_1 **then** p_2 **else** p_3 for some programs p_1, p_2 and p_3:

1.	$p =$ **if** p_1 **then** p_2 **else** p_3	Assumption for this case
2.	$\Gamma \vdash$ **if** p_1 **then** p_2 **else** $p_3 : t$	Subst(1, A)
3.	$\Gamma \vdash$ **if** p_1 **then** p_2 **else** $p_3 : t'$	Subst(1, B)
4.	$\Gamma \vdash p_2 : t$	Inversion(2)
5.	$\Gamma \vdash p_2 : t'$	Inversion(3)
6.	$t = t'$	IH(4,5)

In the last step, we can appeal to the inductive hypothesis, because p_2 is a subterm of p.

8. p is equal to (**fun** t_r f $(t_x$ $x)$ $\{b\}$), where t_r, t_x are types, x and f are variables, and b is a program.

1.	$p =$ (**fun** t_r f $(t_x$ $x)$ $\{b\}$)	Assumption for this case
2.	$\Gamma \vdash$ (**fun** t_r f $(t_x$ $x)$ $\{b\}$) $: t$	Subst(1, A)
3.	$\Gamma \vdash$ (**fun** t_r f $(t_x$ $x)$ $\{b\}$) $: t'$	Subst(1, B)
4.	$t = t_x \to t_r$	Inversion(2)
5.	$t' = t_x \to t_r$	Inversion(3)
6.	$t = t'$	TRANS(4,5)

$\dashv$