CS411 Notes 8 – Types I

A. Demers

21 Feb 2001

1 Fundamentals

1.1 A Tiny Language Fragment

When we first introduced abstraction, we used syntactic sets for typechecking. In particular, the distinction between program variable names and function names was enforced by using separate syntactic sets **Loc** and **Fname**, which were treated separately by the formation rules.

We are about to start doing much more sophisticated typechecking, for which we'll use proof rules analogous to our evaluation rules. This will eliminate the need for separate syntactic sets to enforce type-related rules. Thus, we will replace **Loc** and **Fname** by a single infinite syntactic set **Id** of *identifiers*

$$x \in \mathbf{Id}$$

and all the names declared in a program will be taken from this set.

We shall also introduce a set **Type** of *type expressions*. Initially we define

$$\tau \in \mathbf{Type} \qquad \mathbf{Type} = \{\text{int}, \text{bool}, \text{string}\}$$

but this set will get much richer.

We introduce syntactic sets providing constants for each of our three initial base types:

$$n \in \mathbf{N} \qquad \mathbf{N} = \{0, \pm 1, \pm 2, \ldots\}$$

$$t \in \mathbf{B} \qquad \mathbf{B} = \{\mathbf{true}, \ \mathbf{false}\}$$

$$s \in \mathbf{S} \qquad \mathbf{S} = \text{ character strings}$$

With these definitions, our initial simple language fragment becomes

$$e ::= n \mid t \mid s \mid \ldots \mid e_0 \theta e_1 \mid e_0; e_1$$
$$\mid \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \mid \ldots$$

Note this fragment does not (yet) contain **let** blocks or assignment.

1.2 Type Rules

We can now give proof rules that define the well-typed programs. A judgement in the system (the conclusion of one of the rules) takes the form of a *type annotation*, that is

$$e : \tau$$

and should be interpreted to mean e is a valid syntactic object of type τ.

The proof rules for the initial language fragment are completely straightforward. For the constants of each type we have the axioms:

$$\frac{}{n : \text{ int}} \qquad \frac{}{t : \text{ Bool}} \qquad \frac{}{s : \text{ string}}$$

The types of compound expressions are built up from the types of their subexpressions:

$$\frac{e_0 : \text{int} \qquad e_1 : \text{int}}{(e_0 + e_1) : \text{int}}$$

There are similar rules for other arithmetic operators, and for Boolean and string operations. Some operators combine a fixed set of types, for example the rule

$$\frac{e_0 : \text{string}}{\textbf{len } e_0 : \text{int}}$$

describes a string length operator that returns a number. Other operators and syntactic constructs are "overloaded" or "generic" (we will discuss these concepts in some detail later on) resulting in rules with metavariables ranging over **Type**, for example

$$\frac{e_0 : \tau \qquad e_1 : \tau}{(e_0 = e_1) : \text{bool}}$$

$$\frac{e_0 : \text{bool} \qquad e_1 : \tau \qquad e_2 : \tau}{(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2) : \tau}$$

$$\frac{e_0 : \tau_0 \qquad e_1 : \tau}{(e_0; e_1) : \tau}$$

Instantiating such a rule involves choosing an arbitrary element $\tau \in \mathbf{Type}$. Note in particular for the sequential composition rule, the type τ_0 chosen for e_0 does not occur in the conclusion of the rule; but it is necessary that *some* τ_0 exist to show that e_0 is a well-formed expression.

Together these rules satisfy the property

Unique Typing: For any expression e there is at most one type τ for which the type attribution $e : \tau$ is derivable. Moreover, there is at most one derivation of $e : \tau$.

1.3 Evaluation

For the tiny language fragment, which does not include assignable program variables or **let** blocks, the evaluation relation is rather trivial. We need neither an environment nor a store. Judgements take the form

$$e \to v$$

where v is a "value," which for the tiny language is just exactly an irreducible expression, an element of $N \cup B \cup S$.

There are rules for applying operators and essentially nothing else. For example

$$\frac{e_0 : \ \mathrm{bool} \to \ \mathbf{true} \qquad e_1 : \tau \to v}{(\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) : \tau \to v}$$

is one of the most complicated rules; we leave the others as an exercise.

An important point to note: the valid programs are just the well-typed ones. Thus, in the rule above we include type annotations for all expressions that appear. Technically we can assume an evaluation rule is instantiated with a derivation of the type annotation for each expression *and subexpression* that appears in it. In the example above, the subexpressions e_1 and e_2 could be typed $e_i : \tau$. By the unique typing property introduced above this is well-defined; that is, for any expression or subexpression there can be at most one derivation of a typing.

All evaluations terminate deterministically, as is easily shown by WF induction.

2 Adding Abstractions

Next we extend the language by adding abstractions.

Then we add two new formation rules, one for abstraction definition and another for invocation:

$$e \ ::= \ldots \mid \textbf{let} \ \ldots \ x_i \sim e_i \ \ldots \ \textbf{in} \ e \mid x$$

2.1 Type Rules

Now what should the form of a judgement be? We can get some intuition by considering the proof rule for **let**. The rule will need a hypothesis that says something like "if x has the type of e_0 then e_1 has type τ." We don't yet have enough mechanism to express the "if x has the type ..." part. So we need to develop it.

Define a *type assignment* π to be a finite set of type annotations

$$\pi \quad = \quad \{\ldots, x_i : \tau_i, \ldots\}$$

We actually require π to be a finite partial function; that is,

$$((x : \tau_0 \in \pi) \wedge (x : \tau_1 \in \pi)) \quad \Rightarrow \quad (\tau_0 \equiv \tau_1)$$

To manipulate type assignments as functions, we give a few auxiliary definitions: The domain of π when viewed as a partial function:

$$\textbf{dom}(\pi) \quad = \quad \{x | \exists \tau \ x : \tau \in \pi\}$$

The restriction of π to a subset of the identifiers:

$$\pi|_S \quad = \quad \{x : \tau \in \pi \mid x \in S\} \qquad \text{where} S \subset \textbf{Id}$$

The update of π by π':

$$\pi \oplus \pi' \quad = \quad (\pi|_{\textbf{Id}-\textbf{dom}(\pi')}) \cup \pi'$$

In the last definition, bindings from $\textbf{dom}(\pi')$ are deleted from π before the union is taken.

Judgements will take the form

$$\pi \ \vdash \ e : \tau$$

which you should interpret to mean that if the free variables in e have the types specified in π, then e will have type τ.

It will turn out that such a judgement will be derivable only if every free variable in e has a typing in π. This will be provable by induction on derivations. Intuitively, such a judgement makes sense only for well-formed environments. For example, if $x : \tau \in \pi$ then τ should be a well-formed type expression. In our current language fragment this condition is trivial, as there are only three types; but later it will become more interesting.

The rules for the language fragment without **let** blocks can be adapted to this new language extension by inserting a common type assignment everywhere:

$$\frac{}{\pi \ \vdash \ n : \ \text{int}} \qquad \frac{}{\pi \ \vdash \ t : \ \text{bool}} \qquad \frac{}{\pi \ \vdash \ s : \ \text{string}}$$

$$\frac{\pi \ \vdash \ e_0 : \text{int} \qquad \pi \ \vdash \ e_1 : \text{int}}{\pi \ \vdash \ (e_0 + e_1) : \text{int}}$$

$$\frac{\pi \ \vdash \ e_0 : \text{string}}{\pi \ \vdash \ \textbf{len} \ e_0 : \text{int}}$$

$$\frac{\pi \ \vdash \ e_0 : \tau \qquad \pi \ \vdash \ e_1 : \tau}{\pi \ \vdash \ (e_0 = e_1) : \text{Bool}}$$

$$\frac{\pi \ \vdash \ e_0 : \text{bool} \qquad \pi \ \vdash \ e_1 : \tau \qquad \pi \ \vdash \ e_2 : \tau}{\pi \ \vdash \ (\textbf{if} \ e_0 \ \textbf{then} \ e_1 \ \textbf{else} \ e_2) : \tau}$$

The interesting rules are those for for abstraction and invocation:

Let binding

$$\frac{\ldots \ \pi \ \vdash \ e_i : \tau_i \qquad \pi' \ \vdash \ e : \tau}{\pi \ \vdash \ (\textbf{let} \ \ldots \ x_i \sim e_i \ \ldots \ \textbf{in} \ e) : \tau} \qquad \text{where } \pi' \ = \ \pi \oplus \{\ldots \ x_i : \tau_i \ \ldots\}$$

The **let** block has type τ using a given type assignment π if π can be used to derive typings $e_i : \tau_i$ for the abstracted expressions, and then π augmented by the derived typings can be used to derive the typing $e : \tau$ for the block body.

Identifier reference

$$\frac{}{\pi \ \vdash \ x : \tau} \qquad \text{if } \ x : \tau \in \pi$$

The type of an identifier reference is taken from the type assignment.

A unique typing theorem is a bit more subtle in the presence of type assignments. The type of an expression cannot be unconditionally unique, since it clearly can depend on the type assignment used to derive it. We present unique typing in two parts:

Support: Let π and π' be type assignments where

$$\pi'|_{FV(e)} \quad = \quad \pi|_{FV(e)}$$

Then

$$(\pi \vdash e : \tau) \implies (\pi' \vdash e : \tau)$$

That is, the derivation of a type annotation $e : \tau$ using π depends only on the types assigned to *free variables* of e by π.

Unique Typing: For any π and e, there is at most one derivation of $\pi \vdash e : \tau$.

A result that is conditional on π is arguably the best we can do. For example, the expression

$$\textbf{if } x \textbf{ then } y \textbf{ else } z$$

may be invalid (if $x : \textbf{bool} \notin \pi$) or may have any type τ (if $y : \tau \in \pi$ and $z : \tau \in \pi$). The presence or absence of $w : \tau$ in π has no effect on the type, since w does not occur free in the expression.

2.2 Evaluation

This section is not yet written $\cdots$.