CS411 Notes 12 Type Soundness

A. Demers

13 Mar 2001

Here we argue for the soundness of the typing rules of Notes 10-11. Before proceeding, be sure to have read "Translating LS to SS". Based on the discussion there, we shall *assume* the existence of small step evaluation rules for the language of Notes 10-11. Since these rules are not explicitly written down, we can't explicitly write down a type soundness proof based on them. However, we can say quite a bit about the structure of such a proof.

A natural way to state formally that a set of typing rules is sound might go something like

> If the typing rules assign type τ to a program expression e, and the evaluation rules reduce e to v, then the typing rules assign τ to v.

Roughly, "the value of an expression is in its type."

It is easy to convince yourself that such a condition is necessary – to assert that expression e has type τ when the "value" of e is not in τ would clearly be unacceptable. But would the proposed theorem constitute "soundness" of the type system? Possibly not. There are at least a couple of things that might go wrong:

Inconsistent type rules. One potential problem is that the typing rules themselves might be inconsistent – in worst case, for every expression e it might be possible to deduce

$$\{\} \vdash e : \tau$$

for *every* type τ, making the proposed theorem statement identically true and, consequently, not very compelling as a statement of soundness of the type system.

This particular problem does not arise for us. Our typing rules have a *unique typing* property:

$$(\pi \vdash (e : \tau) \ \wedge \ \pi \vdash (e : \tau')) \quad \Rightarrow \quad (\tau = \tau')$$

That is, for any expression and any type environment, there is at most one type that can be deduced for the expression.

Incomplete evaluation A second potential problem is incomplete evaluation. Our programming language includes potentially nonterminating constructs. Thus, we could not hope to strengthen the proposed theorem to anything like

> If the typing rules assign type τ to a program expression e, then there exists a value v such that the evaluation rules reduce e to v, and the typing rules assign τ to v.

since many expressions e legitimately fail to reduce to values. However, it is not obvious that failure to reduce an expression *always* reflects legitimate nontermination. Instead, a reduction sequence might simply "get stuck," reaching a configuration to which no evaluation rule applies. Our own evaluation rules exhibit just this behavior if evaluation ever attempts to read from an uninitialized location in the store.

In principle, it might be possible to reach a configuration from which the "obvious" next step would intuitively be type-erroneous. If in this case the evaluation rules simply "get stuck" – fail to take the fatal next step – the proposed type soundness theorem still holds, albeit vacuously.

It is nontrivial to capture formally the notion of "getting stuck" for large step rules; so for the remainder of this discussion we will assume we have a set of small step evaluation rules. Refer to the "Translating LS to SS" handout for some justification for this.

Small step rules produce a final answer (if evaluation terminates) and all intermediate configurations. In principle we can make statements about all cofigurations visited during an evaluation, and we can insist that the evaluation either completes or visits infinitely many configurations. Thus, our type soundness theorem should include a subject reduction (or "type preservation") condition like

$$(\{\} \vdash e : \tau) \wedge (\langle e, \phi, \sigma \rangle \rightarrow_1^* \langle e', \sigma' \rangle) \quad \Rightarrow \quad \{\} \vdash e' : \tau$$

and a "progress" condition like

$$\begin{aligned}&(\{\} \vdash e : \tau) \wedge (\langle e, \phi, \sigma\rangle \rightarrow_1^* \langle e', \sigma'\rangle)\\ &\qquad\qquad \Rightarrow (e' \in V) \quad \vee \quad (\exists e'', \sigma'')(\langle e', \phi, \sigma'\rangle \rightarrow_1 \langle e'', \sigma''\rangle)\end{aligned}$$

Together, these conditions say an evaluation either terminates correctly or goes through an infinite sequence of well-typed configurations. This is a much stronger notion of type soundness than our original proposed theorem, and is the one we'll pursue.

To make a completely formal statement of this theorem, and to facilitate a proof by induction on derivations, we need to characterize the "well-typed" configurations.

The interesting objects in our configurations are expressions e, type assignments π, environments ϕ, and stores σ. A well-typed configuration is one in which all these components are consistent with one another, as we'll describe now.

The type assignment π is

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

The environment ϕ is

$$\phi = \{\ldots\ x_i \sim b_i\ \ldots\}$$

where b_i is a bindable (closed) value as in Notes 10.

We say ϕ is *consistent* with π if

$$\begin{aligned}&(\mathbf{dom}(\phi) = \mathbf{dom}(\pi))\wedge\\ &\qquad\qquad (((x \sim \tau) \in \pi) \wedge ((x \sim b) \in \phi)) \Rightarrow (\{\} \vdash b : \tau)\end{aligned}$$

that is, π and ϕ define the same set of names in a type-consistent way.

The store σ is

$$\sigma = \{\ldots\ a_i^\tau \sim v_i\ \ldots\}$$

where v_i is a storable value as in Notes 10.

We say σ is *proper* if

$$((a^\tau \sim v) \in \sigma) \ \Rightarrow\ (\{\} \vdash v : \tau)$$

that is, the types of values in the store agree with the types of their locations.

We say σ *supports* expression e if

$$AC(e) \;\subseteq\; \mathbf{dom}(\sigma)$$

where $AC(e)$ is the set of all address constants that occur anywhere in e. $AC(e)$ is easily defined by induction on expressions.

With all this machinery, we are finally able to state the soundness theorem for our rules:

Theorem (soundness of type rules): Suppose e, π, ϕ and σ satisfy

$\pi \vdash e : \tau$
ϕ is consistent with π
σ is proper
σ supports e

and suppose

$$\langle e, \phi, \sigma \rangle \;\rightarrow_1^* \; \langle e', \sigma' \rangle$$

Then (subject reduction):

$\pi \vdash e' : \tau$
σ' is proper
σ' supports e'

and (progress):

$$(e' \in V) \vee (\exists e'', \sigma'')(\langle e', \phi, \sigma' \rangle \rightarrow_1 \langle e'', \sigma'' \rangle)$$

□

A proof would proceed by induction on derivations in the small step semantics.

Of course the above statement is the "induction hypothesis" version. For programs, it is applied with

e is closed $\qquad \pi = \emptyset \qquad \phi = \emptyset \qquad \sigma = \emptyset$

which can easily be seen to satisfy the consistency conditions of the theorem. The only remaining requirement is that

$$\sigma\ (=\emptyset)\ \text{ supports } e \qquad \text{i.e.} \qquad AC(e) = \emptyset$$

This requirement is slightly questionable. It illustrates that address constants are an artifact of evaluation, usable during evaluation to construct closed expressions, but not allowed in initial program expressions. An alternative would be to replace the emtpy store by one that has been initialized to an appropriate default value at each location in $AC(e)$, but we won't pursue that here.