Type Soundness

- A static type system is *sound* if well-typed programms cannot cause type errors at runtime.
- Definition of a *type error* depends on the language.
- A language with a sound static type system is *strongly typed.*

An "interpretation" function *eval* takes programs a produces Answers or type errors. *eval* is a partial function – it is not defined when the program does not terminate.

$$eval : \ Program \rightarrow Answers \cup \{\mathrm{WRONG}\}$$

Two notions of soundness:

- **Weak:** If $\triangleright\, e \ : \ \tau$ then $eval(e) \neq \mathrm{WRONG}$.
- **Strong:** If $\triangleright\, e \ : \ \tau$ and $eval(e) = v$ then $v \in V^{\tau}$

Type Soundness for ML

Type soundness proofs exist for:

- The functional subset of ML.
- ML with mutable refs.
- Other extensions, e.g., first-class continuations, subtyping, etc.

The proofs differ in their *eval* functions, which may be based on:

1. denotational semantics
2. operational semantics

The idea of being able to prove soundness was relatively new. It changed the style of language design work.

Type Soundness

In the Wright Felliesen paper, the soundness proof is based on operational semantics, under which a program may:

- reduce infinitely often ($e \Uparrow$)
- or may reach a final "irreducible" state. $e_1 \mapsto\!\!\!\rightarrow e_k$ and $e_k \not\mapsto\!\!\!\rightarrow e_{k+1}$

To prove type soundness, show that in the latter case, if e_k is WRONG, e_1 is not well typed.

Observation 1: Reductions preserve type.

$$\text{if } \ \triangleright\, e_1 \ : \ \tau \text{ and } e_1 \mapsto e_2 \text{ then } \triangleright\, e_e \ : \ \tau$$

Observation 2: Sound type systems do not assign a type to expressions that contain type errors.

$$\text{if } e \text{ is faulty, there is not } \tau \text{ such that } \triangleright\, e \ : \ \tau$$

Type Soundness

Three steps in proving type soundness:

- characterize answers and faulty expressions
- showing that reduction preserves types (subject reduction)
- showing that the faulty expressions are untypable

Type Soundness for Functional ML

The ML kernel language constructs:

$$e \in Expressions$$
$$v \in Values$$
$$c \in Constant$$
$$x \in Identifier$$

$$e ::= v \mid e_1\ e_2 \mid \texttt{let}\ x \texttt{be}\ e_1 \texttt{in}\ e_2$$
$$v ::= c \mid x \mid Y \mid \lambda x.e$$

SOS for Functional ML

The operational semantics for ML is similar to that of PCF or FL.

$$c\ v \to \delta(c, v) \qquad (\delta)$$
$$\text{if } \delta(c, v) \text{ is defined}$$

$$(\lambda x\ .\ e)v \to [v/x]e \qquad (\beta_v)$$

$$\texttt{let}\ x\ \texttt{be}\ v\ \texttt{in}\ e\ \to\ [v/x]e \quad (let)$$

$$Yv \to v(\lambda x\ .\ (Y\ v)\ x) \qquad (Y)$$

where δ is an evaluation function on a fixed set of built-in functions.

A convention on variable reuse prevents capture.

The abstraction in the recursion rule is used to make the argument to v a value.

Expression Contexts

Recall that a *context* is an expression with a hole in it.

$$\begin{aligned} C ::= & \; [] \mid C \; e \mid e \; C \mid \texttt{let} \; x \; \texttt{be} \; C \; \in e \\ & \mid \texttt{let} \; x \; \texttt{be} \; e \; \in C \mid \lambda x \, . \, C \end{aligned}$$

A special kind of *evaluation context* avoids holes where variable capture could occur.

$$E ::= \; [] \mid E \; e \mid v \; E \mid \texttt{let} \; x \; \texttt{be} \; E \; \in e$$

They also force a left-most outermost evaluation strategy for the associated rewriting relation.

$$E[e] \mapsto E[e'] \text{ iff } e \to e'$$

The left-most outermost reduction is clearly unique, so the evaluation function defined as the reflexive transitive closer of $\mapsto$ is a function.

$$eval(e) = v \text{ iff } e \not\mapsto v$$

Types in Functional ML

Types (τ) and type schemes (σ) in ML are given by:

$$\tau ::= \iota_1 \mid \ldots \mid \iota_n \mid \alpha \mid \tau_1 \rightarrow \tau_2 \quad (Types)$$

$$\sigma ::= \forall\ \alpha^*\ .\ \tau \qquad (TypeSchemes)$$

Where ιs are ground types like *bool* and *int*.

As in the TGR book, polymorphic functions are represented by type expressions containing universally quantified type variables.

There is a partial ordering $\succ$ on type schemes defined by the the "generality" of the scheme.

$$\forall \sigma_1 \ldots \sigma_n\ .\ \tau' \succ \tau \text{ iff } S\tau' = \tau \text{ and } \mathrm{Dom}(S) = \{\sigma_1 \ldots \sigma_n\} \text{ for some substitution } S.$$

Typing Rules for Functional ML

$$\Gamma \triangleright x : \tau \text{ if } \Gamma(x) \succ \tau \qquad (var)$$

$$\Gamma\ c : \tau \text{ if } TypeOf(c) \succ \tau \qquad (const)$$

$$\Gamma \triangleright Y : ((\tau_1 \rightarrow \tau_2) \rightarrow \tau_1 \rightarrow \tau_2) \rightarrow \tau_1 \rightarrow \tau_2 \qquad (Y)$$

$$\frac{\Gamma[\tau_1/x] \triangleright e : \tau_2}{\Gamma \triangleright \lambda\, x\, .\, e : \tau_1 \rightarrow \tau_2} \qquad (abs)$$

$$\frac{\Gamma \triangleright e_1 : \tau_1 \rightarrow \tau_2 \;\&\; \Gamma \triangleright e_2 : \tau_2}{\Gamma \triangleright e_1\, e_2 : \tau_2} \qquad (app)$$

$$\frac{\Gamma \triangleright e_1 : \tau_1 \;\&\; \Gamma[Close(\tau_1, \Gamma)/x] \triangleright e_2 | \tau_2}{\Gamma \triangleright \texttt{let}\ x\ \textbf{be}\ e_1\ \texttt{in}\ e_2 : \tau_2} \qquad (let)$$

$$Close(\tau, \Gamma) = \forall \alpha_1 \ldots \alpha_n\, .\, \tau$$

$$\text{where } \{\alpha_1 \ldots \alpha_n\} = FTV(\tau)/FTV(\Gamma)$$

Decidability of Types

An important property of any static type system is that it be decidable. In practice, we would like something stronger, namely an efficient algorithm for checking the type system.

Milner gave an algorithm W for checking the types in this system, based on the idea of unification. He proved that is was sound a complete (for the type checking problem) and terminating.

This is not the same as soundness of the type system, which is a relationship between the dynamic semantics and the type rules.

Type Soundness

As noted earlier, one of the key steps in proving soundness of the type system is subject reduction.

Main Lemma: If $\Gamma \ \triangleright \ e_1 \ : \ \tau$ and $e_1 \rightarrow e_2$ then $\Gamma \ \triangleright \ e_2 : \tau$.

Proof: By case analysis on the reduction $e_1 \rightarrow e_2$

- $(\lambda x \ . \ e)v \rightarrow [v/x]e$. From $\Gamma \triangleright (\lambda x \ . \ e)v \ : \ \tau$ we have

  1. $\Gamma \ \triangleright \ v \ : \ \tau'$
  2. $\Gamma \ \triangleright \ (\lambda x \ . \ e \ : \ \tau' \rightarrow \tau$ from the application rule.

  From (2), $\Gamma[\tau'/x] \triangleright e \ : \ \tau$ follows from the abstraction rule. Therefore, $\Gamma \triangleright \ e[v/x] : \tau$ by a substitution lemma.

- $\texttt{let} \ x \ \texttt{be} \ v \ \texttt{in} \ e \rightarrow e[v/x]$. From $\Gamma \triangleright \texttt{let} \ x \ \texttt{be} \ v \ \texttt{in} \quad : \ \tau$ we have

  1. $\Gamma \triangleright \ v \ : \ \tau'$
  2. and $\Gamma[Close(\tau', \Gamma)/x] \triangleright \ e \ : \ \tau$ by the *let* rule.

  Since $Close(\tau', \Gamma) = \forall \alpha_1 ... \alpha_n \ . \ \tau'$, we have $\Gamma \triangleright e[v/x] : \tau$ again by a subsitution lemma.

Type Soundness (con't)

- $c\ v \;\rightarrow\; \delta(c, v)$. Then

  1. $\Gamma \triangleright c \;:\; \tau' \rightarrow \tau$
  2. and $\Gamma \triangleright v \;:\; \tau'$

  follow from $\Gamma \triangleright\ c\ v \;:\; \tau$ by the application rule. So $\Gamma \triangleright \delta(c, v) : \tau$ by a constraint on proper δ.

- case for Y omitted.

Additional Lemmas

Substitution Lemma: If $\Gamma[\forall \alpha_1 \ldots \alpha_n / x \,.\, \tau] \triangleright e \; : \; \tau'$ and $x \notin Dom(\Gamma)$ and $\Gamma \; \triangleright \; v \; : \; \tau$ and $\{\alpha_1, ..., \alpha_n\} \cup FTV(\Gamma) = \emptyset$ then $\Gamma \triangleright e[v/x] \; : \; \tau'$.

Proof: By induction on length of proof of $\Gamma[\forall \alpha_1 \ldots \alpha_n / x \,.\, \tau] \; \triangleright \; e \; : \; \tau'$ using case analysis on the last step.

- $e = c$. Then $TypeOf(c) \succ \tau'$ by the constant rule. Thus, $\Gamma \; \triangleright \; c \; : \; \tau'$. Then $\Gamma \; \triangleright \; c[v/x] \; : \; \tau'$ since $c[v/x] = c$.
- ...