Programming Languages

MinML: A MINiMaL Functional Language
Static Semantics

David Walker

Static Semantics

The **static semantics**, or **type system**, imposes context-sensitive restrictions on the formation of expressions.

- Distinguishes **well-typed** from **ill-typed** expressions.

- Type constraints eliminate **prima facie** nonsensical programs.

The static semantics is inductively defined by a set of **typing rules**.

Typing Judgements

A **typing judgement**, or **typing assertion**, is a triple

$$\Gamma \vdash e : \tau$$

with three parts

1. A **type assignment**, or **type context**, Γ that assigns types to some finite set of variables. Think of Γ as a "symbol table".

2. An **expression** e whose free variables are given types by Γ.

3. A **type** τ for the expression e.

Type Assignments

Formally, a type assignment is a **finite function**

$$\Gamma : \mathit{Variables} \rightharpoonup \mathit{Types}$$

That is, Γ is a function whose domain dom(Γ) is a finite set of variables.

We write $\Gamma, x{:}\tau$, or $\Gamma[x{:}\tau]$, for the function Γ' defined as follows:

$$\Gamma'(y) = \begin{cases} \tau & \mathit{if}\ \ x = y \\ \Gamma(y) & \mathit{if}\ \ x \neq y \end{cases}$$

Typing Rules

A variable has whatever type Γ assigns to it:

$$\frac{}{\Gamma \vdash x : \Gamma(x)}$$

The constants have the evident types:

$$\frac{}{\Gamma \vdash n : \mathtt{int}}$$

$$\frac{}{\Gamma \vdash \mathtt{true} : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{bool}}$$

Typing Rules

The primitive operations have the expected typing rules:

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{+}(e_1, e_2) : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{=}(e_1, e_2) : \texttt{bool}}$$

(and similarly for the others).

Typing Rules

Both "branches" of a conditional must have the **same** type!

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} : \tau}$$

Intuitively, we cannot predict the outcome of the test (in general) so we must insist that both results have the same type. Otherwise we could not assign a unique type to the conditional.

Typing Rules

Functions may only be applied to arguments in their domain:

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{apply}(e_1, e_2) : \tau}$$

The result type is the co-domain (range) of the function.

Typing Rules

Type checking recursive functions:

$$\frac{\Gamma[f{:}\tau_1{\to}\tau_2][x{:}\tau_1] \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\, f\ (x{:}\tau_1){:}\tau_2 = e : \tau_1{\to}\tau_2}$$

We tacitly **assume** that $\{f, x\} \cap \mathrm{dom}(\Gamma) = \emptyset$. This is always possible by our conventions on binding operators.

Typing Rules

Type checking a recursive function is tricky! We **assume** that

1. the function has the specified domain and range types, and
2. the argument has the specified domain type.

We then **check** that the body has the range type under these assumptions.

If the assumptions are **consistent**, the function is type correct, otherwise not.

Well-Typed and Ill-Typed Expressions

An expression e is **well-typed**, or **typable**, in a context Γ iff there exists a type τ such that $\Gamma \vdash e : \tau$.

If there is no τ such that $\Gamma \vdash e : \tau$, then e is **ill-typed**, or **untypable**, in context Γ.

Typing Example

Consider the following expression f:

```
fun f(n:int):int is
  if n=0 then 1 else n * f(n-1) end
```

Proposition 1
The expression f has type $\texttt{int}\rightarrow\texttt{int}$.

To prove this, we must show that $\emptyset \vdash f : \texttt{int}\rightarrow\texttt{int}$ is a valid typing judgement according to the rules above.

Typing Example

$\emptyset \vdash f : \texttt{int}{\rightarrow}\texttt{int}$ because

> $\texttt{f}{:}\texttt{int}{\rightarrow}\texttt{int}, \texttt{n}{:}\texttt{int} \vdash \texttt{if n=0 then 1 else n*f(n-1)} : \texttt{int}$ because
>
> > $\texttt{f}{:}\texttt{int}{\rightarrow}\texttt{int}, \texttt{n}{:}\texttt{int} \vdash \texttt{n=0} : \texttt{bool}$
> >
> > $\texttt{f}{:}\texttt{int}{\rightarrow}\texttt{int}, \texttt{n}{:}\texttt{int} \vdash \texttt{1} : \texttt{int}$
> >
> > $\texttt{f}{:}\texttt{int}{\rightarrow}\texttt{int}, \texttt{n}{:}\texttt{int} \vdash \texttt{n*f(n-1)} : \texttt{int}$

Typing Example

$\mathtt{f{:}int{\rightarrow}int, n{:}int} \vdash \mathtt{n=0} : \mathtt{bool}$ because

$$\mathtt{f{:}int{\rightarrow}int, n{:}int} \vdash \mathtt{n} : \mathtt{int}$$

$$\mathtt{f{:}int{\rightarrow}int, n{:}int} \vdash 0 : \mathtt{int}$$

$\mathtt{f{:}int{\rightarrow}int, n{:}int} \vdash 1 : \mathtt{int}$ is immediate.

Typing Example

$\texttt{f:int}\to\texttt{int}, \texttt{n:int} \vdash \texttt{n*f(n-1)} : \texttt{int}$ because

$$\texttt{f:int}\to\texttt{int}, \texttt{n:int} \vdash \texttt{n} : \texttt{int}$$

$$\texttt{f:int}\to\texttt{int}, \texttt{n:int} \vdash \texttt{f(n-1)} : \texttt{int}$$

The first case is immediate, the second requires a bit more work.

Typing Example

$\mathtt{f}{:}\mathtt{int}{\rightarrow}\mathtt{int}, \mathtt{n}{:}\mathtt{int} \vdash \mathtt{f(n-1)} : \mathtt{int}$ because

$\mathtt{f}{:}\mathtt{int}{\rightarrow}\mathtt{int}, \mathtt{n}{:}\mathtt{int} \vdash \mathtt{f} : \mathtt{int}{\rightarrow}\mathtt{int}$

$\mathtt{f}{:}\mathtt{int}{\rightarrow}\mathtt{int}, \mathtt{n}{:}\mathtt{int} \vdash \mathtt{n-1} : \mathtt{int}$ because

$\mathtt{f}{:}\mathtt{int}{\rightarrow}\mathtt{int}, \mathtt{n}{:}\mathtt{int} \vdash \mathtt{n} : \mathtt{int}$

$\mathtt{f}{:}\mathtt{int}{\rightarrow}\mathtt{int}, \mathtt{n}{:}\mathtt{int} \vdash \mathtt{1} : \mathtt{int}$

Typing Example

This completes the proof! It's rather tedious to do by hand, but what's nice is that there are **precise rules** to fall back on if you get stuck.

In practice we use computers to find typing proofs. This is the job of a **type checker**:

> Given Γ, e, and τ, is there a derivation of $\Gamma \vdash e : \tau$ according to the typing rules?

Type Checking

How does the type checker find typing proofs?

Important fact: the typing rules are **syntax-directed** — there is **one** rule per expression form.

Therefore the checker can **invert** the typing rules and work backwards towards the proof, just as we did above.

For example, if the expression is a function, the only possible proof is one that applies the function typing rule. So we work backwards from there.

Type Checking

We can say something even stronger for MinML: every expression has **at most one** type.

Therefore to determine whether or not $\Gamma \vdash e : \tau$, we may

1. Compute the unique type τ_e (if any) of e in Γ.

2. Compare τ_e with τ.

This is called **type synthesis** because we **synthesize** the unique (if it exists) type of e relative to Γ.

Type Checking

Formally, we prove that the three-place relation $\Gamma \vdash e : \tau$ is a **partial function** of Γ and e.

That is, if $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$, then $\tau_1 = \tau_2$.

This is proved by induction on the structure of e (exercise).

For homework you will implement this style of type checker.

Properties of Typing

Theorem 2 (Inversion)
The typing rules are necessary, as well as sufficient.

1. *If* $\Gamma \vdash x : \tau$, *then* $\Gamma(x) = \tau$.

2. *If* $\Gamma \vdash n : \tau$, *then* $\tau = \texttt{int}$.

3. *If* $\Gamma \vdash \texttt{true} : \tau$, *then* $\tau = \texttt{bool}$, *and similarly for* $\texttt{false}$.

4. *If* $\Gamma \vdash \texttt{+}(e_1, e_2) : \tau$, *then* $\tau = \texttt{int}$ *and* $\Gamma \vdash e_1 : \texttt{int}$ *and* $\Gamma \vdash e_2 : \texttt{int}$.

5. *If* $\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} : \tau$, *then* $\Gamma \vdash e : \texttt{bool}$, $\Gamma \vdash e_1 : \tau$ *and* $\Gamma \vdash e_2 : \tau$.

6. *If* $\Gamma \vdash \texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2 = e : \tau$, *then* $\tau = \tau_1 {\rightarrow} \tau_2$ *and then* $\Gamma[f{:}\tau_1{\rightarrow}\tau_2][x{:}\tau_1] \vdash e : \tau_2$.

7. *If* $\Gamma \vdash \texttt{apply(}e_1, e_2\texttt{)} : \tau$, *then there exists* τ_2 *such that* $\Gamma \vdash e_1 : \tau_2 {\rightarrow} \tau$ *and* $\Gamma \vdash e_2 : \tau_2$.

Proof by **rule induction** on the typing rules.

Induction on Typing

To show that some property $P(\Gamma, e, \tau)$ holds whenever $\Gamma \vdash e : \tau$, it is enough to show

- $P(\Gamma, x, \Gamma(x))$
- $P(\Gamma, n, \mathtt{int})$
- $P(\Gamma, \mathtt{true}, \mathtt{bool})$
- $P(\Gamma, \mathtt{false}, \mathtt{bool})$
- if $P(\Gamma, e_1, \mathtt{int})$ and $P(\Gamma, e_2, \mathtt{int})$, then $P(\Gamma, \mathtt{+}(e_1, e_2), \mathtt{int})$ (and similarly for the other primitive operators)
- if $P(\Gamma, e, \mathtt{bool})$, $P(\Gamma, e_1, \tau)$, and $P(\Gamma, e_2, \tau)$, then $P(\Gamma, \mathtt{if}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2\, \mathtt{fi}, \tau)$
- if $P(\Gamma, e_1, \tau_2 \rightarrow \tau)$ and $P(\Gamma, e_2, \tau_2)$, then $P(\Gamma, \mathtt{apply(}e_1, e_2\mathtt{)}, \tau)$.
- if $P(\Gamma[f{:}\tau_1 \rightarrow \tau_2][x{:}\tau_1], e, \tau_2)$, then $P(\Gamma, \mathtt{fun}\, f\ \mathtt{(}x\mathtt{:}\tau_1\mathtt{):}\tau_2 = e, \tau_1 \rightarrow \tau_2)$.

Properties of Typing

Theorem 3 (Weakening)
If $\Gamma \vdash e : \tau$ and $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash e : \tau$.

Intuitively, “junk” in the symbol table doesn’t matter. We may always α-convert expressions to “steer around” the junk.

The proof is by induction on typing.

Properties of Typing

Theorem 4 (Substitution)
If $\Gamma[x{:}\tau] \vdash e' : \tau'$ *and* $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash \{e/x\}e' : \tau'$.

Intuitively, we may "click in" the second derivation wherever the type of x is required in the first derivation.

Formally, we prove this by rule induction on the **first** typing judgement.

- Consider each rule in turn.
- Show in each case that substitution preserves type.

Summary

1. The static semantics of MinML is specified by an inductive definition of the **typing judgement** $\Gamma \vdash e : \tau$.

2. Properties of the type system may be proved by **induction on typing derivations**.