

Type Systems for Programming Languages (15-814)

Lecture Notes, Fall 2006, Week 11

Jason Ganetsky (jganetsk@cs)

November 21, 28, and 30, 2006

1 Effects

$e ::= \mathbf{new} e \mid \mathbf{get} e \mid \mathbf{set}(e_1, e_2) \mid \ell \mid \dots$

$\overline{\ell \text{ value}}$

$e@_\mu$ ok if $\exists \Lambda$ such that

1. $\Lambda; \cdot \vdash e : \tau$
2. $\mu : \Lambda$

μ maps locations to values. Λ maps locations to types. $\Lambda(\ell) = \tau$ means location ℓ contains values of type τ .

Theorem (Progress). If $e@_\mu$ ok, then either e value or $e@_\mu \mapsto e'@_{\mu'}$.

Theorem (Preservation). If $e@_\mu$ ok and $e@_\mu \mapsto e'@_{\mu'}$, then $e'@_{\mu'}$ ok.

$$\frac{\Lambda(\ell) = \tau}{\Lambda; \Gamma \vdash \ell : \tau \text{ ref}} \quad \frac{\Lambda; \Gamma \vdash e : \tau}{\Lambda; \Gamma \vdash \mathbf{new} e : \tau \text{ ref}} \quad \frac{\Lambda; \Gamma \vdash e : \tau \text{ ref}}{\Lambda; \Gamma \vdash \mathbf{get} e : \tau} \quad \frac{\Lambda; \Gamma \vdash e_2 : \tau \quad \Lambda; \Gamma \vdash e_1 : \tau \text{ ref}}{\Lambda; \Gamma \vdash \mathbf{set}(e_1, e_2) : \tau}$$

$$\frac{e \text{ value} \quad \ell \# \mu}{\mathbf{new} e@_\mu \mapsto \ell@_\mu[\ell = e]} \quad \frac{\ell \in \text{dom}(\mu)}{\mathbf{get} \ell@_\mu \mapsto \mu(\ell)@_\mu} \quad \frac{e \text{ value} \quad \ell \in \text{dom}(\mu)}{\mathbf{set}(\ell, e) \mapsto e@_\mu[\ell \leftarrow e]}$$

Theorem (Preservation - revisited). If $\mu : \Lambda$ and $\Lambda; \cdot \vdash e : \tau$ and $e@_\mu \mapsto e'@_{\mu'}$, then $\exists \Lambda' \supseteq \Lambda$ such that $\mu' : \Lambda'$ and $\Lambda'; \cdot \vdash e' : \tau$.

Theorem (Canonical Forms). If e value and $\Lambda; \cdot \vdash e : \sigma \text{ ref}$ then $e = \ell \in \text{dom}(\Lambda)$, and $\Lambda(\ell) = \sigma$.

In the presence of ref, the meaning of $e : \tau$ has changed dramatically. We can now have unit \rightarrow unit, and unit \rightarrow int. Monads, discussed later, describe how types can be explicit about effects within.

1.1 Famous Pratfalls

1.1.1 SML

let val $r = \text{ref } (\text{fn } x \rightarrow x) \text{ in } \dots$

r has type $\forall \alpha. \alpha \rightarrow \alpha \text{ ref}$.

let val $_ = r := (\text{fn } x \rightarrow x + 1) \text{ in } \dots$

Ok. $\Lambda(r) = \forall \alpha. \alpha \rightarrow \alpha$. It works.

(!r) ("abc")

Typechecks, but at runtime we have a serious problem.

ML Static Semantics: **let val** $x = e \text{ in } e' \equiv [e/x]e'$... x gains a polymorphic type only if e *value*.

1.1.2 Java

Suppose $\text{int} <: \text{float}$. You may be tempted to say $\text{int ref} <: \text{float ref}$. This leads to some hairy issues.

Suppose $x : \text{int ref}$, and assume the above sub-typing relation. x can be coerced to float ref , so $!x$ can be used as a float. However, x cannot be coerced to float ref in this case: $x := 3.14$. Damn contravariance!

This problem appears in Java arrays. A Java array of size 1 is semantically equivalent to a reference. Java admits that if $\sigma <: \tau$, then $\sigma \text{ array} <: \tau \text{ array}$. But, the designers of Java did not account for contravariance in assignments, and you could then assign a float to an int.

2 Monads

Monads allow for the explicit serialization of execution, an absolute necessity for languages with side effects. The effect-free model we have previously studied employs rightist substitution as the mechanism of function application. This has the benefit of being simple and natural, but leads to willy-nilly replication of expressions. This is dangerous for effectful languages, as most side effects are not idempotent. Monadic expressions provide the programmer with exactly-once execution semantics, as well as allow for a strict ordering of events in the system. Ultimately, the result is represented as an effect-free value, which can be plugged in, using the rightist substitution model. The evaluation of effectful expressions is driven by leftist substitution, which provides progress and preservation while enforcing exactly-once semantics. For more information, see Haskell.

Courtesy of William Lovas:

$$\begin{aligned} \tau &::= \text{nat} \mid \tau_1 \rightarrow \tau_2 \mid \tau \text{ ref} \mid \tau \text{ comp} \\ e &::= x \mid \text{zero} \mid \text{succ } e \mid \text{natcase } e \text{ of } \{ \text{zero} \Rightarrow e_0, \text{succ } x \Rightarrow e_1 \} \mid \lambda x:\tau. e \mid e_1 e_2 \mid \ell \mid \text{val } m \\ m &::= \text{return } e \mid i; x:\tau. m \mid \text{let val } x : \tau = e \text{ in } m \\ i &::= \text{new } e \mid \text{get } e \mid \text{set } (e_1, e_2) \end{aligned}$$

$\Lambda; \Gamma \vdash_m e : \tau$

$$\frac{}{\Lambda; (\Gamma, x:\tau) \vdash_m x : \tau} \text{(VAR)}$$

$$\frac{}{(\Lambda, x:\tau); \Gamma \vdash_m \ell : \tau \text{ ref}} \text{(LOC)}$$

$$\frac{}{\Lambda; \Gamma \vdash_m \text{zero} : \tau} \text{(NAT-I-ZERO)}$$

$$\frac{\Lambda; \Gamma \vdash_m e : \text{nat}}{\Lambda; \Gamma \vdash_m \text{succ } e : \text{nat}} \text{(NAT-I-SUCC)}$$

$$\frac{\Lambda; \Gamma \vdash_m e : \text{nat} \quad \Lambda; \Gamma \vdash_m e_0 : \tau \quad \Lambda; \Gamma, x:\text{nat} \vdash_m e_1 : \tau}{\Lambda; \Gamma \vdash_m \text{natcase } e \text{ of } \{ \text{zero} \Rightarrow e_0, \text{succ } x \Rightarrow e_1 \} : \tau} \text{(NAT-E)}$$

$$\frac{\Lambda; \Gamma, x:\tau_1 \vdash_m e : \tau_2}{\Lambda; \Gamma \vdash_m \lambda x:\tau_1. e : \tau_1 \rightarrow \tau_2} (\rightarrow\text{-I})$$

$$\frac{\Lambda; \Gamma \vdash_m e_1 : \tau_2 \rightarrow \tau \quad \Lambda; \Gamma \vdash_m e_2 : \tau_2}{\Lambda; \Gamma \vdash_m e_1 e_2 : \tau} (\rightarrow\text{-E})$$

$$\frac{\Lambda; \Gamma \vdash_m m \sim \tau}{\Lambda; \Gamma \vdash_m \text{val } m : \tau \text{ comp}} \text{(COMP-I)}$$

 $\Lambda; \Gamma \vdash_m m \sim \tau$

$$\frac{\Lambda; \Gamma \vdash_m e : \tau}{\Lambda; \Gamma \vdash_m \text{return } e \sim \tau} \text{(RETURN)}$$

$$\frac{\Lambda; \Gamma \vdash_m i \sim \sigma \quad \Lambda; \Gamma, x:\sigma \vdash_m m \sim \tau}{\Lambda; \Gamma \vdash_m i; x:\sigma. m \sim \tau} \text{(SEQ)}$$

$$\frac{\Lambda; \Gamma \vdash_m e : \sigma \text{ comp} \quad \Lambda; \Gamma, x:\sigma \vdash_m m \sim \tau}{\Lambda; \Gamma \vdash_m \text{let val } x : \sigma = e \text{ in } m \sim \tau} \text{(COMP-E)}$$

 $\Lambda; \Gamma \vdash_m i \sim \tau$

$$\frac{\Lambda; \Gamma \vdash_m e : \tau}{\Lambda; \Gamma \vdash_m \text{new } e \sim \tau \text{ ref}} \text{(REF-I)}$$

$$\frac{\Lambda; \Gamma \vdash_m e : \tau \text{ ref}}{\Lambda; \Gamma \vdash_m \text{get } e \sim \tau} \text{(REF-E-GET)}$$

$$\frac{\Lambda; \Gamma \vdash_m e_1 : \tau \text{ ref} \quad \Lambda; \Gamma \vdash_m e_2 : \tau}{\Lambda; \Gamma \vdash_m \text{set}(e_1, e_2) \sim \tau} \text{(REF-E-SET)}$$

$$\begin{array}{l} m@_\mu \mapsto m'@_{\mu'} \\ m@_\mu \text{ final} \end{array}$$

$$\begin{array}{l} \mu \vdash e \mapsto e' \\ \mu \vdash e \text{ value} \end{array}$$

$$\begin{array}{l} \mu \vdash i \mapsto i' \\ \mu \vdash i \text{ ready} \end{array}$$

$$\frac{}{\mu \vdash \text{val } m \text{ value}}$$

$$\frac{\ell \in \text{dom}(\mu)}{\mu \vdash \ell \text{ value}}$$

$$\begin{aligned}
&\langle i; y:\tau. m_1/x \rangle m_2 = i; y:\tau. \langle m_1/x \rangle m_2 \\
&\langle \mathbf{let\ val\ } y : \tau = e \mathbf{ in\ } m_1/x \rangle m_2 = \mathbf{let\ val\ } y : \tau = e \mathbf{ in\ } \langle m_1/x \rangle m_2 \\
&\langle \mathbf{return\ } e/x \rangle m_2 = [e/x] m_2
\end{aligned}$$

$$m@_\mu \mapsto m'@_{\mu'}$$

$$\frac{\mu \vdash e \mapsto e'}{\mathbf{return\ } e@_\mu \mapsto \mathbf{return\ } e'@_\mu}$$

$$\frac{\mu \vdash i \mapsto i'}{i; x:\tau. m@_\mu \mapsto i'; x:\tau. m@_\mu}$$

$$\frac{e \text{ value} \quad \ell \notin \text{dom}(\mu)}{\mathbf{new\ } e; x:\tau. m@_\mu \mapsto [\ell/x] m@_\mu[\ell = e]}$$

$$\frac{\ell \in \text{dom}(\mu)}{\mathbf{get\ } \ell; x:\tau. m@_\mu \mapsto [\mu(\ell)/x] m@_\mu}$$

$$\frac{\ell \in \text{dom}(\mu) \quad e_2 \text{ value}}{\mathbf{set\ } (\ell, e_2); x:\tau. m@_\mu \mapsto [e_2/x] m@_\mu[\ell \leftarrow e_2]}$$

$$\frac{\mu \vdash e \mapsto e'}{\mathbf{let\ val\ } x : \tau = e \mathbf{ in\ } m@_\mu \mapsto \mathbf{let\ val\ } x : \tau = e' \mathbf{ in\ } m@_\mu}$$

$$\frac{}{\mathbf{let\ val\ } x : \tau = \text{val } m_1 \mathbf{ in\ } m_2@_\mu \mapsto \langle m_1/x \rangle m_2@_\mu}$$

$$m@_\mu \text{ final}$$

$$\frac{e \text{ value}}{\mathbf{return\ } e@_\mu \text{ final}}$$

$$\mu \vdash i \mapsto i'$$

$$\frac{\mu \vdash e \mapsto e'}{\mu \vdash \mathbf{new\ } e \mapsto \mathbf{new\ } e'}$$

$$\frac{\mu \vdash e \mapsto e'}{\mu \vdash \mathbf{get\ } e \mapsto \mathbf{get\ } e'}$$

$$\frac{\mu \vdash e_1 \mapsto e'_1}{\mu \vdash \mathbf{set\ } (e_1, e_2) \mapsto \mathbf{set\ } (e'_1, e_2)}$$

$$\frac{\mu \vdash e_1 \text{ value} \quad \mu \vdash e_2 \mapsto e'_2}{\mu \vdash \mathbf{set\ } (e_1, e_2) \mapsto \mathbf{set\ } (e_1, e'_2)}$$

$$\mu \vdash i \text{ ready}$$

$$\frac{\mu \vdash e \text{ value}}{\mu \vdash \mathbf{new\ } e \text{ ready}}$$

$$\frac{\mu \vdash e \text{ value}}{\mu \vdash \mathbf{get\ } e \text{ ready}}$$

$$\frac{\mu \vdash e_1 \text{ value} \quad \mu \vdash e_2 \text{ value}}{\mu \vdash \mathbf{set\ } (e_1, e_2) \text{ ready}}$$

Monads are double-edged swords. There are number of properties, each a pro and a con.

1. Monads segregate pure computations from impure ones, as well as classify the nature of side effects. This can be valuable to the programmer, as the typing information provides valuable clues as to what a piece of code does. However, when designing APIs, monadic type information overstates the implementation of a function. For example, $\cdot \vdash \text{lookup} : \text{dict} \times \text{key} \rightarrow \text{value}$ is the most natural signature of a lookup function. But, it rules out any possibility of a stateful implementation. The obvious choice for implementation is a hash table, which cannot be applied here.

2. When using a monad, we are always stuck inside of it. The monad type propagates upwards to the top-most loop of the code; there is no way to eliminate it. While this is good because it is accurate, it is terribly inconvenient. This inevitably leads to the need to combine different monad types, like I/O and Exception... and leads to the use of what they call the “sin bin”. This is a catch-all monad type with an implementation that is probably very ugly. So, as we go up to the top of the code, we lose any notion of granularity in our description of monadic effects.
3. The segregation of pure and impure code leads to the retention of equational properties. For example $(\text{map}f) \circ (\text{map}g) = \text{map}(f \circ g)$. However, this makes the code almost impossible to evolve. We cannot reasonably add logging, profiling, debugging, or instrumentation without dealing with a large amount of chaos and uncertainty.

3 Modularity

Types are the foundation of modularity.

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash [e/x] e' : \tau'} \qquad \frac{\Gamma \vdash \phi \text{ true} \quad \Gamma, \phi \text{ true} \vdash \psi \text{ true}}{\Gamma \vdash \psi \text{ true}}$$

Types mediate across the decomposition boundary

3.1 Revisit existentials as an abstraction

$\exists t. \tau$, for example...

```

∃d.{insert : e × d → d
    make : ... → d
    lookup : d × k → e}

```

pack τ **with** e **as** $\exists t. \tau$
open e **as** t **with** $x : \tau$ **in** $e' : \tau'$

Main unpleasantry is that opening a package always introduces a new type.

1. Eversion - most widely used abstractions must be given the widest scope
2. Consolidation - impedes separate compilation

Suppose $e = \text{if coinflip() then pack int with } \dots \text{ else pack float with } \dots$

Now, typing is based on effects, which violates the phase distinction.

Instead of $e : \exists t : \tau$, let us define the type as $\text{rpn}(e)$ and the operations as $\text{ops}(e)$.

An example: $\lambda x : \text{rpn}(\text{dict}). \text{ops}(\text{dict}). \text{insert}(\dots)$

We still violate the phase distinction. When is $\text{rpn}(e) = \text{rpn}(e')$? Only if $e = e'$. Why is $\text{rpn}(e)$ well-defined at all? It may not be well determined.

3.2 Dot Notation

```

τ ::= ... | sig(t.τ) | rpn(e)
e ::= ... | str(τ, e) | ops(e) | seal(e, τ)

```

1. Limit $\text{rpn}(e)$ to determinate expressions for e . Sealed expressions are indeterminate.
2. Open-scope abstraction: $e := \tau$