3.1 Type Soundness

타입 규칙이 갖추어야 할 조건중 첫째 것은, 검증된 프로그램은 실행중에 잘못되지 않는다는 것이다. 이것을 타입규칙의 안전성이라고 한다. 우리가 정의한 타입시스템이 안전하다는 것을 증명하자.

안전성 증명을 용이하게 하기 위해서 프로그램이 실행되는 매 스텝을 다음의 관계로 정의한다:

$$e \longmapsto e'$$

위의 관계는 "프로그램 e가 한 스텝 후에 프로그램 e'로 변한다"로 읽는다. 예를 들어

$$\texttt{1 + 2 * 3} \longmapsto \texttt{1 + 6} \longmapsto \texttt{7}$$

과 같이 식이 변해가면서 최종 값을 만들어내는 과정을 표현하는 것이다. 프로그램 식이 변하는 까닭은 프로그램의 부분식들이 계산되는 순서에 따라 값으로 변하기 때문이다.

프로그램의 부분식들이 계산되는 순서를 결정하는 방법은 *실행의 맥*(evaluation context)이라는 기발한 방법으로 가능하다:

$$
\begin{array}{lcll}
\multicolumn{3}{l}{\text{실행의 맥 evaluation context}} & \\
C & ::= & [] & \text{빈칸} \\
 & | & C\ e & \\
 & | & v\ C & \\
 & | & \texttt{if}\ C\ \texttt{then}\ e\ \texttt{else}\ e & \\
 & | & \texttt{let}\ x\ \texttt{=}\ C\ \texttt{in}\ e & \\
 & | & \texttt{let rec}\ x\ \texttt{=}\ C\ \texttt{in}\ e & \\
 & | & C\ op\ e & \\
 & | & v\ op\ C & \\
 & | & \texttt{write}\ C & \\
 & | & \texttt{(}C\texttt{)} &
\end{array}
$$

$C[]$는 빈칸이 있는 프로그램 식을 표현한다. 위의 C의 정의에 의하면 $C[]$에서 빈칸은 오직 하나뿐이다. $C[e]$ 는 그 빈칸에 식 e가 든 것을 표현한다. 위에서 v는 값을 프로그램 식으로 표현한 것이다:

$$
\begin{array}{lcll}
v & ::= & const & \text{constant value} \\
 & | & \texttt{fn}\ x\ \texttt{=>}\ e & \text{function value} \\
 & | & f\ \texttt{fn}\ x\ \texttt{=>}\ e & \text{recursive function value}
\end{array}
$$

위에서 재귀함수 값 "f `fn` x `=>` e"는 다음의 식을 간단하게 표현한 것이다:

$$\texttt{let rec}\ f\ \texttt{=}\ \texttt{fn}\ x\ \texttt{=>}\ e\ \texttt{in f end}.$$

어느정도 눈치챘겠지만, C에서 빈칸이 있을 수 있는 자리가 바로 다음에 수행해야 할 식이 위치한 자리가 된다. 값이 계산된 장소(v가 있는 장소)에는 빈칸([])이 있을 수 없고, 계산순서가 v와 C의 적절한 위치를 통해서 표현되고 있다. 예를 들어, "$C\ e$"는 함수식 파트가 우선 계산되야 한다는 것을 표현하고 있고, "$v\ C$"에서는 인자 파트가 계산되는 경우는 항상 함수 파트가 값 v로 계산된 다음이다 라는 것을 표현한다.

위에서 정의한 프로그램 실행의 한 스텝 $e \longmapsto e'$ 은 다음의 규칙에 의해서 정의된다:

$$\frac{r \to c}{C[r] \longmapsto C[c]}$$

즉, e에서 바로 다음에 계산해야 할 식이 r이고 ($e = C[r]$) 그 r이 c가 될 때 ($r \to c$) e는 e' ($e' = C[c]$) 으로 한 스텝 진행하는 것이다. 규칙 $r \to c$는 다음과 같다:

$$\begin{array}{rcll}
(\texttt{fn}\ x\ \texttt{=>}\ e)\, v & \to & e\,/\,\{v/x\} & \\
(f\ \texttt{fn}\ x\ \texttt{=>}\ e)\, v & \to & e\,/\,\{v/x, (f\ \texttt{fn}\ x\ \texttt{=>}\ e)/f\} & \\
\texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\ e & \to & e\,/\,\{v/x\} & \\
\texttt{let rec}\ x\ \texttt{=}\ v\ \texttt{in}\ e & \to & e\,/\,\{x\, v/x\} & \\
\texttt{if true then}\ e_1\ \texttt{else}\ e_2 & \to & e_1 & \\
\texttt{if false then}\ e_1\ \texttt{else}\ e_2 & \to & e_2 & \\
v_1\ \mathit{op}\ v_2 & \to & v & \text{where } v = \mathit{op}(v_1, v_2) \\
\texttt{write}\ v & \to & v & \\
(v) & \to & v &
\end{array}$$

Definition 3 (Program's Semantics) *A program e computes v iff $e \overset{*}{\longmapsto} v$.*

Fact 1 $\emptyset \vdash e \Rightarrow v$ *iff* $e \overset{*}{\longmapsto} v$.

Definition 4 (Program Goes Wrong) *If program e stucks at e' because no value v exists such that $e' \overset{*}{\longmapsto} v$, then the program e goes wrong, writing $e \overset{*}{\longmapsto}$ wrong.*

Theorem 1 (Type's Weak Soundness) *For any program e of tcoML, if $\emptyset \vdash e : \tau$ then not $e \overset{*}{\longmapsto}$ wrong.*

Theorem 2 (Type's Strong Soundness) *For any program e of tcoML, if $\emptyset \vdash e : \tau$ and $e \overset{*}{\longmapsto} \bullet$ then $\emptyset \vdash \bullet : \tau$.*

Proof. We prove that if $\emptyset \vdash e : \tau$ and $e \longmapsto e'$ then $\emptyset \vdash e' : \tau$. Then by induction on the number of evaluation steps, the theorem holds. The proof is by induction on the evaluation context C of $C[r] = e$ and by using Lemma 2.

Base case: $C[] = []$. True by Lemma 2.

Inductive case: $C[] = C_1[]\, e'$. By the assumption $\emptyset \vdash C_1[r]\, e' : \tau$ and by rule [App],

$$\begin{array}{rr} \emptyset \vdash C_1[r] : \tau' \to \tau & (0) \\ \emptyset \vdash e' : \tau' & (1) \end{array}$$

By induction hypothesis for (0),

$$\emptyset \vdash C_1[c] : \tau' \to \tau.$$

Hence by (1) and rule [App]

$$\emptyset \vdash C_1[c]\, e' : \tau.$$

Other cases are similarly done. $\Box$

Note that the strong soundness implies the weak soundness because the *wrong* has no type.

Lemma 1 *If* $\Gamma[x : \tau_1] \vdash e : \tau$ *and* $\Gamma \vdash v_1 : \tau_1$ *then* $\Gamma \vdash e \,/\, \{v_1/x\} : \tau$.

Proof. By induction on e. This Lemma is necessary to prove Lemma 2. $\Box$

Lemma 2 *If* $\emptyset \vdash r : \tau$ *and* $r \to c$ *then* $\emptyset \vdash c : \tau$.

Proof. By case analysis on r.

Case $r = (f\ \texttt{fn}\ x\ \texttt{=>}\ e)\, v$. Because $\emptyset \vdash r : \tau$,

$$\begin{array}{rr} \emptyset \vdash f\ \texttt{fn}\ x\ \texttt{=>}\ e : \tau' \to \tau & (0) \\ \emptyset \vdash v : \tau'. & (1) \end{array}$$

By (0) and rule [RecLet],

$$\{f : \tau' \to \tau, x : \tau'\} \vdash e : \tau. \qquad (2)$$

By (1), (2), and Lemma 1

$$f : \tau' \to \tau \vdash e/\{v/x\} : \tau$$

and by (0) and Lemma 1

$$\emptyset \vdash e/\{v/x, f\ \texttt{fn}\ x\ \texttt{=>}\ e/f\} : \tau.$$

Other cases are similarly done. $\Box$

4 Type Checking

After we made sure that our type system was sound, we have to implement the type system. That is, we need a program `typecheck` such that

$$\texttt{typecheck}(e) = \tau \quad \text{iff} \quad \emptyset \vdash e : \tau.$$

Determining for given a program e whether $\emptyset \vdash e : \tau$ or not is reduced to the provability in a simple logical system called *unification logic*.

The unification logic formulas for type terms t are:

$$\begin{array}{lcll} \multicolumn{4}{l}{\mathit{Formula}} \\ \phi & ::= & t \doteq t & \\ & | & \phi \wedge \phi & \\ & | & \exists \alpha . \phi & \\ \\ \multicolumn{4}{l}{\mathit{TyTerm}} \\ t & ::= & i & \text{int type} \\ & | & b & \text{bool type} \\ & | & s & \text{string type} \\ & | & t \rightarrow t & \text{function type} \\ & | & \alpha & \text{type variable} \end{array}$$

The proof rules for the unification logic formulas are:

$$\vdash t \doteq t \qquad \frac{\vdash \phi\,/\{t/\alpha\}}{\vdash \exists \alpha . \phi} \qquad \frac{\vdash \phi_1 \quad \vdash \phi_2}{\vdash \phi_1 \wedge \phi_2}$$

Therefore, a solution of a set of unification logic formulas is a substitution of type terms for type variables that will satisfy all the $\doteq$. The most general such subsitution is achieved by Robinson's most general unifier (mgu).

We define an algorithm T that collects unification formulas:

$$T : \mathit{Expr} \times \mathit{TyEnv} \times \mathit{TyTerm} \rightarrow \mathit{Formula}$$

such that

$$\vdash T(e, \emptyset, \tau) \quad \text{iff} \quad \emptyset \vdash e : \tau.$$

$$
\begin{array}{rcl}
T(n, \Gamma, t) & = & t \doteq i \\
T(s, \Gamma, t) & = & t \doteq s \\
T(b, \Gamma, t) & = & t \doteq b \\
T(x, \Gamma, t) & = & t \doteq \Gamma(x) \\
T(\texttt{fn}\ x\ \texttt{=>}\ e, \Gamma, t) & = & \exists\alpha.\exists\beta.\, t \doteq \alpha \to \beta \ \wedge\ T(e, \Gamma[x : \alpha], \beta) \\
T(e_1\, e_2, \Gamma, t) & = & T(e_1, \Gamma, \alpha \to t) \ \wedge\ T(e_2, \Gamma, \alpha) \\
T(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2, \Gamma, t) & = & \exists\alpha.\, T(e_1, \Gamma, \alpha) \ \wedge\ T(e_2, \Gamma[x : \alpha], t) \\
T(\texttt{let rec}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2, \Gamma, t) & = & \exists\alpha.\, T(e_1, \Gamma[x : \alpha], \alpha) \ \wedge\ T(e_2, \Gamma[x : \alpha], t) \\
T(e_1 \texttt{+} e_2, \Gamma, t) & = & T(e_1, \Gamma, i) \ \wedge\ T(e_2, \Gamma, i)
\end{array}
$$

Theorem 3 $\vdash T(e, \Gamma, \tau)$ *iff* $\Gamma \vdash e : \tau$.

Proof. By induction on e.

Case $e = \texttt{fn}\ x\ \texttt{=>}\ e'$:

$$
\begin{array}{lll}
 & \vdash T(\texttt{fn}\ x\ \texttt{=>}\ e', \Gamma, \tau) & \\
\text{iff} & \tau = \tau_1 \to \tau_2 & \text{and} \\
 & \vdash T(e', \Gamma[x : \tau_1], \tau_2) & \\
\text{iff} & \Gamma[x : \tau_1] \vdash e' : \tau_2 & \text{by I.H.} \\
\text{iff} & \Gamma \vdash \texttt{fn}\ x\ \texttt{=>}\ e' : \tau_1 \to \tau_2 & \text{by rule [Fun]}
\end{array}
$$

□

Our type checking algorithm `typecheck` can be implemented as follows

$$\texttt{typecheck}(e) = (mgu \circ T)(e, \emptyset, \alpha)(\alpha)$$

If the unification fails the program is ill-typed.

One drawback of the above implementation inside a compiler is that it is hard to report the region of the input program that causes, if any, a type error.

A rather more direct-style implementation is the following algorithm:

$$W : \mathit{Expr} \times \mathit{TyEnv} \to \mathit{TyTerm} \times \mathit{Subst}$$

$$
\begin{array}{rcl}
W(n,\Gamma) & = & (i,\{\}) \\
W(x,\Gamma) & = & (\Gamma(x),\{\}) \\
W(\texttt{fn}\ x\ \texttt{=>}\ e,\Gamma) & = & \text{let } (t,S) = W(e,\Gamma[x:\alpha]) \quad \alpha \text{ new} \\
 & & \text{in } (\alpha/S \to t, S) \\
W(e_1\, e_2,\Gamma) & = & \text{let } (t_1,S_1) = W(e_1,\Gamma) \\
 & & \quad (t_2,S_2) = W(e_2,\Gamma/S_1) \\
 & & \quad S_3 = mgu(t_1/S_2, t_2 \to \alpha) \quad \alpha \text{ new} \\
 & & \text{in } (\alpha/S_3, S_3S_2S_1) \\
W(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2,\Gamma) & = & \text{let } (t_1,S_1) = W(e_1,\Gamma) \\
 & & \quad (t_2,S_2) = W(e_2,\Gamma/S_1) \\
 & & \text{in } (t_2, S_2S_1) \\
W(\texttt{let rec}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2,\Gamma) & = & \text{let } (t_1,S_1) = W(e_1,\Gamma[x:\alpha]) \quad \alpha \text{ new} \\
 & & \quad (t_2,S_2) = W(e_2,\Gamma[x:\alpha]/S_1) \\
 & & \text{in } (t_2, S_2S_1) \\
W(e_1\texttt{+}e_2,\Gamma) & = & \text{let } (t_1,S_1) = W(e_1,\Gamma) \\
 & & \quad (t_2,S_2) = W(e_2,\Gamma/S_1) \\
 & & \quad S_3 = mgu(t_1/S_2, t_2, i) \\
 & & \text{in } (i, S_3S_2S_1)
\end{array}
$$

Now it becomes clear where the type-error occurs: where the unfication mgu fails.

Theorem 4 (W's Soundness) *If* $W(e,\Gamma) = (\tau, S)$ *then* $\Gamma/S \vdash e : \tau/S$.

Proof. By induction on e. □

Theorem 5 (W's Completeness) *If* $\Gamma/P \vdash e : \tau/P$ *then* $W(e,\Gamma) = (\tau, S)$ *and* $P = TS$ *for some* T.

Proof. By induction on e. □