1 Introduction

The programming language TFPL_0 is to serve as an object of study on typed functional programming languages. TFPL_0 is a pure call-by-value typed function programming language with base types \texttt{int} and \texttt{bool}, product types and function types. It is essentially a variant of PCF, a language for Programming Computable Functions (Scott 1993). TFPL_0 is a start point. We will gradually introduce additional language features into TFPL_0 for study.

2 Syntax

We formally present the syntax for the programming language TFPL_0.

2.1 Types

TFPL_0 is a typed programming language. The types in TFPL_0 are defined as follows.

\[
\begin{align*}
types & \quad \tau ::= \texttt{bool} \mid \texttt{int} \mid 1 \mid \tau \ast \tau \mid \tau \rightarrow \tau \\
contexts & \quad \Gamma ::= \cdot \mid \Gamma, x : \tau
\end{align*}
\]

\texttt{bool} and \texttt{int} are base types. Given types \(\tau_1\) and \(\tau_2\); \(\tau_1 \ast \tau_2\) is the product type for a pair whose first and second components are of type \(\tau_1\) and \(\tau_2\), respectively; \(\tau_1 \rightarrow \tau_2\) is the function type for a function that takes an argument of type \(\tau_1\) and returns a result of type \(\tau_2\). For instance, we may think that the addition function on integers has the type \(\texttt{int} \ast \texttt{int} \rightarrow \texttt{int}\), that is, it takes a pair of integers and returns an integer. For each context of form \(\Gamma, x : \tau\), we require that \(x \notin \text{dom}(\Gamma)\). In other words, every variable can be declared at most once in a given context.

2.2 Expressions

The syntax for language expression in TFPL_0 is given below.

\[
\begin{align*}
\text{expressions} & \quad e ::= x \mid f \mid b \mid i \mid op(e_1, \ldots, e_n) \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \text{ fi} \mid \\
& \quad \langle \rangle \mid \langle e_1, e_2 \rangle \mid \text{fst}(e) \mid \text{snd}(e) \mid \\
& \quad \text{fun } f(x : \tau_1) : \tau_2 \text{ is } e \mid \text{app}(e_1, e_2) \mid \text{let } x = e_1 \text{ in } e_2 \text{ end}
\end{align*}
\]

\texttt{fun} \(f(x : \tau_1) : \tau_2\) is \(e\) defines a function, where \(\tau_1\) and \(\tau_2\) are the types of the argument and the result of the function, respectively.

\textbf{Example 2.1} The following defines the factorial function on natural numbers.

\[
\text{fun fact(x : int) : int is if x = 0 then 1 else x * app(fact, x - 1) fi}
\]
Example 2.2 The following defines a higher order function that takes two functions and returns their composition. Note that we use $\tau$ for int $\rightarrow$ int.

\[
\text{fun} \circ (f : \tau) : \tau \rightarrow \tau \text{ is fun } \circ (g : \tau) : \tau \text{ is fun } \circ f \circ g \ (x : \text{int}) : \text{int is app}(f, \text{app}(g, x))
\]

Example 2.3 Let $eo$ be the following function.

\[
\text{fun } f(x : \text{int}) : \text{bool} \times \text{bool is } \\
\text{if } x = 0 \text{ then } (\text{true, false}) \text{ else let } y = \text{app}(f, x - 1) \text{ in } (\text{snd}(y), \text{fst}(y)) \text{ end fi}
\]

Then $\text{fun even}(x : \text{int}) : \text{bool is } \text{fst(\text{app}(eo, x))}$ and $\text{fun odd}(x : \text{int}) : \text{bool is } \text{snd(\text{app}(eo, x))}$ are the even and odd functions on natural numbers, respectively.

Given $\text{fun } f(x : \tau_1) : \tau_2$ is $e$, we write $\text{lam } x : \tau_1. e$ for $\text{fun } f(x : \tau_1) : \tau_2$ is $e$ if $f \notin \text{FV}(e)$. For instance, the function in Example 2.2 can also be written as

\[
\text{lam } f : \text{int} \rightarrow \text{int. lam } g : \text{int} \rightarrow \text{int. lam } x : \text{int. app}(f, \text{app}(g, x)).
\]

2.3 Type Erasure

We define a function $| \cdot |$ that maps an expression $e$ in TFPL₀ into one in UFPL by erasing all the types in $e$.

\[
| x | = x \quad | b | = b \quad | i | = i \\
| \text{op}(e_1, \ldots, e_n) | = \text{op}(|e_1|, \ldots, |e_n|) \\
| \text{if } e \text{ then } e_1 \text{ else } e_2 \text{ fi} | = \text{if } |e| \text{ then } |e_1| \text{ else } |e_2| \text{ fi} \\
| () | = () \quad | (e_1, e_2) | = (|e_1|, |e_2|) \\
| \text{fst}(e) | = \text{fst}(|e|) \quad | \text{snd}(e) | = \text{snd}(|e|) \\
| \text{fun } f(x : \tau_1) : \tau_2 \text{ is } e | = \text{fun } f(x) \text{ is } |e| \\
| \text{app}(e_1, e_2) | = \text{app}(|e_1|, |e_2|) \\
| \text{let } x = e_1 \text{ in } e_2 \text{ end} | = \text{let } x = |e_1| \text{ in } |e_2| \text{ end}
\]

We call $|e|$ the type erasure of $e$ for every $e$ in TFPL₀.

3 Dynamic Semantics

3.1 Evaluation Semantics

The evaluation rules for TFPL₀ are essentially the same as those for UFPL. The difference, which is reflected in the following rules (eval-fun) and (eval-app) is that we now need to carry types around when evaluating an expression in TFPL₀.

\[
\frac{\text{fun } f(x : \tau_1) : \tau_2 \text{ is } e \downarrow \text{fun } f(x : \tau_1) : \tau_2 \text{ is } e}{e_1 \downarrow \text{fun } f(x : \tau_1) : \tau_2 \text{ is } e \quad e_2 \downarrow v_2 \quad e[f \mapsto \text{fun } f(x) \text{ is } e, x \mapsto v_2] \downarrow v \quad \text{app}(e_1, e_2) \downarrow v} \quad \text{(eval-app)}
\]

The following theorem states that types are indifferent to evaluation.

Theorem 3.1 Let $e$ and $v$ be an expression and a value in TFPL₀, respectively. Then $e \downarrow v$ is derivable if and only if $|e| \downarrow |v|$ is.

Proof This is straightforward.
\[
\Gamma \vdash b : \text{bool} \quad \text{(type-bool)}
\]

\[
\Gamma \vdash i : \text{int} \quad \text{(type-int)}
\]

\[
\Gamma \vdash x : \tau \quad \text{(type-var)}
\]

\[
\Sigma(op) = \tau_1 \ast \cdots \ast \tau_n \rightarrow \tau \quad \Gamma \vdash e_1 : \tau_1 \cdots \Gamma \vdash e_n : \tau_n \quad \text{(type-op)}
\]

\[
\Gamma \vdash op(e_1, \ldots, e_n) : \tau \quad \text{(type-op)}
\]

\[
\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \text{(type-if)}
\]

\[
\Gamma \vdash \{\} : 1 \quad \text{(type-unit)}
\]

\[
\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \ast \tau_2 \quad \text{(type-tup)}
\]

\[
\Gamma \vdash e : \tau_1 \ast \tau_2 \quad \Gamma \vdash \text{fst}(e) : \tau_1 \quad \text{(type-fst)}
\]

\[
\Gamma \vdash e : \tau_1 \ast \tau_2 \quad \Gamma \vdash \text{snd}(e) : \tau_2 \quad \text{(type-snd)}
\]

\[
\Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e : \tau_2 \quad \text{(type-fun)}
\]

\[
\Gamma \vdash \text{fun } f(x : \tau_1) : \tau_2 \text{ is } e : \tau_1 \rightarrow \tau_2 \quad \text{(type-app)}
\]

\[
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \text{ end} : \tau_2 \quad \text{(type-let)}
\]

Figure 1: Typing rules for TFPL_0

3.2 Reduction Semantics

The reduction semantics of TFPL_0 is exactly like that of UFPL except that we now treat an expression of form \text{app}(\text{fun } f(x : \tau_1) : \tau_2 \text{ is } e, v) as a redex and \(e \mapsto f \mapsto \text{fun } f(x : \tau_1) : \tau_2 \text{ is } e_1, x \mapsto \tau_2\) as its reduction. The following theorem states that types are indifferent to reduction.

**Theorem 3.2** Let \(e_1\) and \(e_2\) be expressions in TFPL_0. Then \(e_1 \rightarrow e_2\) holds if and only if \(|e_1| \rightarrow |e_2|\) does.

**Proof** This is straightforward.

4 Static Semantics

4.1 Type system

TFPL_0 is a typed programming language and we present the typing rules for TFPL_0 in Figure 1. For each context \(\Gamma\), we use \(\text{dom}(\Gamma)\) for the domain of \(\Gamma\), that is, the set of variables declared in \(\Gamma\). We use a judgment \(\Gamma \vdash \theta : \Gamma_2\) to mean that \(\text{dom}(\theta) = \text{dom}(\Gamma_2)\) and \(\Gamma_1 \vdash \theta(x) : \Gamma_2(x)\) for each \(x \in \text{dom}(\theta)\).
Lemma 4.1 If $\Gamma_1 \vdash e : \tau$ and $\Gamma_2 \vdash \theta : \Gamma_1$ are derivable, then $\Gamma_2 \vdash e[\theta] : \tau$ is also derivable.

Proof This follows from structural induction on the derivation $\mathcal{D}$ of $\Gamma_1 \vdash e : \tau$.

$$
\begin{array}{c}
\mathcal{D} = \Gamma_1 \vdash e_1 : \tau_1 \quad \Gamma_1 \vdash e_2 : \tau_2 \\
\implies \Gamma_1 \vdash \langle e_1, e_2 \rangle : \tau_1 \ast \tau_2 \\
\end{array}
$$

By induction hypothesis, $\Gamma_2 \vdash e_1[\theta]$ and $\Gamma_2 \vdash e_2[\theta]$ are derivable.

This leads to the following.

$$
\begin{array}{c}
\Gamma_2 \vdash e_1[\theta] : \tau_1 \quad \Gamma_2 \vdash e_2[\theta] : \tau_2 \\
\implies \Gamma_2 \vdash \langle e_1[\theta], e_2[\theta] \rangle : \tau_1 \ast \tau_2 \\
\end{array}
$$

(type-tup)

Note that $\langle e_1, e_2 \rangle[\theta] = \langle e_1[\theta], e_2[\theta] \rangle$. Hence, the case concludes.

$$
\begin{array}{c}
\mathcal{D} = \Gamma_1, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e : \tau_2 \\
\implies \Gamma_1 \vdash \text{fun } f(x) \text{ is } e : \tau_1 \rightarrow \tau_2 \\
\end{array}
$$

Since $f$ and $x$ are bound variables in $\text{fun } f(x)$ is $e$, we can require that neither of $f$ and $x$ is declared in $\Gamma_2$. Note that we can derive the following since $\Gamma_2 \vdash \theta : \Gamma_1$.

$$
\begin{array}{c}
\Gamma_2, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash \theta[f \mapsto f][x \mapsto x] : \Gamma_1, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \\
\end{array}
$$

By induction hypothesis, $\Gamma_2, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e[\theta[f \mapsto f][x \mapsto x]] : \tau_2$ is derivable, and this yields the following.

$$
\begin{array}{c}
\Gamma_2, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e[\theta[f \mapsto f][x \mapsto x]] : \tau_2 \\
\implies \Gamma_2 \vdash \text{fun } f(x : \tau_1) : \tau_2 \text{ is } e[\theta[f \mapsto f][x \mapsto x]] : \tau_1 \rightarrow \tau_2 \\
\end{array}
$$

(type-fun)

Note that $\text{fun } f(x) = \text{fun } f(x : \tau_1) : \tau_2$ is $e[\theta[f \mapsto f][x \mapsto x]]$. Hence, the case concludes.

The rest of cases can be dealt with similarly.

Proposition 4.2 Let $r$ be a redex and $e$ be its reduction. If $\Gamma \vdash \tau : \tau$ is derivable, then $\Gamma \vdash e : \tau$ is also derivable.

Proof This follows from an inspection on the form of $r$. We present one interesting case in which $r = \text{let } x = v \text{ in } e \text{ end}$. Since $\Gamma \vdash r : \tau$ is derivable, the derivation must be of the following form.

$$
\begin{array}{c}
\Gamma, x : \tau_1 \vdash e : \tau \\
\Gamma \vdash v : \tau_1 \\
\Gamma \vdash \text{let } x = v \text{ in } e \text{ end} : \tau \\
\end{array}
$$

(type-let)

By Lemma 4.1, $\Gamma \vdash e[x \mapsto v] : \tau$ is derivable. Since the reduction of $r$ is $e[x \mapsto v]$, we are done in this case. The other cases can also be readily handled.

Theorem 4.3 (Subject Reduction) Assume $\Gamma \vdash e_1 : \tau$ is derivable. If $e_1 \rightarrow e_2$ holds then $\Gamma \vdash e_2 : \tau$ is also derivable.

Proof Clearly, we have $e_1 = E[r]$ for some evaluation context $E$ and redex $r$ and $e_2 = E[e]$, where $e$ is the reduction of $r$. The theorem is proven by structural induction on $E$. We present some interesting cases.

- $E = \emptyset$. This case follows from Proposition 4.2.
• $E = \langle v, E_1 \rangle$. Then we have a derivation of the following form.

$$
\begin{align*}
\Gamma \vdash v : \tau_1 & \quad \Gamma \vdash E_1[r] : \tau_2 \\
\Gamma \vdash E[r] = \langle v, E_1[r] \rangle : (\tau_1, \tau_2) = \tau
\end{align*}
$$

\(\text{type-tup}\)

By induction hypothesis, $\Gamma \vdash E_1[e] : \tau_2$ is derivable and this leads to the following.

$$
\begin{align*}
\Gamma \vdash v : \tau_1 & \quad \Gamma \vdash E_1[e] : \tau_2 \\
\Gamma \vdash E[e] = \langle v, E_1[e] \rangle : (\tau_1, \tau_2) = \tau
\end{align*}
$$

\(\text{type-tup}\)

The rest of the cases can be treated similarly.

**Theorem 4.4 (Type Preservation)** Suppose that $\cdot \vdash e : \tau$ is derivable. If $e \Downarrow v$ is derivable then $\cdot \vdash v : \tau$ is also derivable.

**Proof** Since $e \Downarrow v$ is derivable, we have $e \rightarrow^* v$. By Theorem 4.3, $\cdot \vdash v : \tau$ is derivable since $\cdot \vdash e : \tau$ is derivable. This theorem can also be proven with structural induction on the derivation of $\cdot \vdash e : \tau$.

**Proposition 4.5** Let $v$ be a value such that $\cdot \vdash v : \tau$ is derivable. We have the following.

• If $\tau = \tau_1 \ast \tau_2$ for some $\tau_1$ and $\tau_2$, then $v$ is of form $\langle v_1, v_2 \rangle$.

• If $\tau = \tau_1 \rightarrow \tau_2$, then $v$ is of form $\text{fun} \ f(x : \tau_1) : \tau_2$ is $e$.

**Proof** This follows an inspection of the typing rules.

**Lemma 4.6** Suppose that $\cdot \vdash e : \tau$ is derivable. If $e$ is not a value, then $e = E[r]$ for some evaluation context $E$ and redex $r$.

**Proof** The lemma follows from structural induction on the derivation $D$ of $e$. We present some interesting cases.

$$
D = \begin{array}{c}
\vdash e_1 : \tau_1 \\
\vdash e_2 : \tau_2 \\
\vdash \langle e_1, e_2 \rangle : \tau_1 \ast \tau_2
\end{array}
$$

Then we have two cases.

• $e_1$ is not a value. By induction hypothesis, $e_1 = E_1[r]$ for some $E_1$ and $r$. Thus, $e = E[r]$ for $E = \langle E_1, e_2 \rangle$.

• $e_1$ is some value $v_1$. Since $e$ is not a value, $e_2$ cannot be a value. By induction hypothesis, $e_2 = E_2[r]$ for some $E_2$ and $r$. Thus, $e = E[r]$ for $E = \langle v_1, E_2 \rangle$.

$$
D = \begin{array}{c}
\vdash e_1 : \tau_1 \ast \tau_2 \\
\vdash \text{fst}(e_1) : \tau_1
\end{array}
$$

Then we have two cases.

• $e_1$ is not a value. By induction hypothesis, $e_1 = E_1[r]$ for some $E_1$ and $r$. Thus, $e = E[r]$ for $E = \text{fst}(E_1)$.

• $e_1$ is some value. By Proposition 4.5, $e_1$ is of form $\langle v_1, v_2 \rangle$. Then $e = \text{fst}(e_1)$ is a redex. Hence, $e = E[r]$ for $E = []$ and $r = e$.

The rest of cases can be treated similarly.

**Theorem 4.7 (Progress)** Suppose that $\cdot \vdash e : \tau$ is derivable. Then $e$ is either a value or $e \rightarrow e'$ holds for some $e'$.

**Proof** The theorem follows from Lemma 4.6.
4.2 Typability

Given an expression $e$ in UFPL, we can form a judgment $\Gamma \vdash e : \tau$ to mean that $e$ can be assigned type $\tau$ under context $\Gamma$. The typing rules for derivating such a judgment are precisely those in TFPL except that we replace the rule \textit{(type-fun)} in TFPL with the following one.

\[
\Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e : \tau_2
\]

\[\Gamma \vdash \text{fun } f(x) \text{ is } e : \tau_1 \rightarrow \tau_2 \text{ (type-fun)}\]

\textbf{Proposition 4.8} Given an expression $e_1$ in UFPL, $e_1$ is typable if and only if $e_1$ is the type erasure of some expression $e_2$ in TFPL such that $\Gamma \vdash e_2 : \tau$ is derivable for some $\Gamma$ and $\tau$.

\textit{Proof} This is obvious.

5 Type Inference

It can be burdensome for the programmer to write types when programming. Therefore, there is an immediate question as to whether we can find a satisfactory approach that allows the programmer to omit writing types and then infers the omitted types from the structure of a program. We present a positive answer to this question in this section.

5.1 External Language

We present an external language for TFPL$_0$. In this language, the programmer is allowed to omit writing types when defining a function \texttt{fun} $f(x)$ is $e$. On the other hand, the programmer is also allowed to write ($e : \tau$) to indicate that the expression $e$ must have type $\tau$.

expressions $e ::= x \mid f \mid b \mid i \mid \text{op}(e_1, \ldots, e_n) \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \text{ fi} \mid \langle \rangle \mid \langle e_1, e_2 \rangle \mid \text{fst}(e) \mid \text{snd}(e) \mid \text{fun } f(x) \text{ is } e \mid \text{fun } f(x : \tau_1) : \tau_2 \text{ is } e \mid \text{app}(e_1, e_2) \mid \text{let } x = e_1 \text{ in } e_2 \text{ end} \mid (e : \tau)$

Note that we define the type erasure of ($e : \tau$) as $|e|$, that is, $|(e : \tau)| = |e|$.

5.2 Type Inference Algorithm

We need the following syntax for presenting a type inference algorithm for TFPL$_0$.

\begin{align*}
\text{Existential Type Variables} & \quad X \\
\text{Types} & \quad T ::= X \mid \text{bool} \mid \text{int} \mid 1 \mid T \times T \mid T \rightarrow T \\
\text{Contexts} & \quad G ::= \cdot \mid x : T, G \\
\text{Type Constraints} & \quad \Phi ::= \top \mid T_1 =_T T_2 \mid \Phi_1 \land \Phi_2 \\
\text{Substitution} & \quad \Theta ::= [] \mid \Theta[X \mapsto \tau]
\end{align*}

Given a substitution $\Theta$, the satisfiability of $\Phi$ under $\Theta$ is defined as follows.

- $\Phi$ is satisfied under $\Theta$ if $\Phi$ is $\top$;
- $\Phi$ is satisfied under $\Theta$ if $\Phi$ is $T_1 =_T T_2$ and $T_1[\Theta] = T_2[\Theta]$,
- $\Phi$ is satisfied under $\Theta$ if $\Phi$ is $\Phi_1 \land \Phi_2$ and both $\Phi_1$ and $\Phi_2$ are satisfied under $\Theta$
We say that a type constraint $\Phi$ is satisfiable if $\Phi$ is satisfiable under some substitution $\Theta$.

We first introduce two kinds of judgments; a judgment $G \vdash e \uparrow T \Rightarrow \Phi$ basically means that we can generate $T$ and $\Phi$ for given $G$ and $e$ such that type constraint $\Phi$ needs to be solved for typing $e$ with $T$ under context $G$; a judgment $G \vdash e \downarrow T \Rightarrow \Phi$ basically states that we can generate $\Phi$ for given $G$, $e$ and $T$ such that type constraint $\Phi$ needs to be solved for typing $e$ with type $T$ under context $G$. The rules for a bidirectional type inference algorithm are presented in Figure 2 and Figure 3. The following theorem establishes the soundness of these rules.

**Theorem 5.1 (Soundness)** We have the following.

1. Assume that $G \vdash e \uparrow T \Rightarrow \Phi$ is derivable. If $\Phi$ is satisfiable under $\Theta$ and $\text{dom}(\Theta)$ contains all existential variables in the judgment $G \vdash e \uparrow T \Rightarrow \Phi$, then $G[\Theta] \vdash \ |	ext{e}|[\Theta] : T[\Theta]$ is derivable.

2. Assume that $G \vdash e \downarrow T \Rightarrow \Phi$ is derivable. If $\Phi$ is satisfiable under $\Theta$ and $\text{dom}(\Theta)$ contains all existential variables in the judgment $G \vdash e \downarrow T \Rightarrow \Phi$, then $G[\Theta] \vdash \ |	ext{e}|[\Theta] : T[\Theta]$ is derivable.

**Proof** (1) and (2) are proven simultaneously with structural induction on the derivations $D$ of $G \vdash e \uparrow T \Rightarrow \Phi$ and $G \vdash e \downarrow T \Rightarrow \Phi$.

**Lemma 5.2** Let $\Theta$ be a substitution on existential variables. We have the following.

1. If $G \vdash e \uparrow T \Rightarrow \Phi$ is derivable, then $G[\Theta] \vdash e \uparrow T[\Theta] \Rightarrow \Phi[\Theta]$ is also derivable.

2. If $G \vdash e \downarrow T \Rightarrow \Phi$ is derivable, then $G[\Theta] \vdash e \downarrow T[\Theta] \Rightarrow \Phi[\Theta]$ is also derivable.

**Proof** (1) and (2) are proven simultaneously with structural induction on the derivations of $G \vdash e \uparrow T \Rightarrow \Phi$ and $G \vdash e \downarrow T \Rightarrow \Phi$.

The following theorem establishes the completeness of the presented type inference rules.

**Theorem 5.3 (Completeness)** Assume that $\Gamma \vdash e : \tau$ is derivable. Then we have the following.

1. $\Gamma \vdash \ |	ext{e}| \uparrow T \Rightarrow \Phi$ is derivable for some $T$ and $\Phi$ and there exists $\Theta$ such that $T[\Theta] = \tau$ and $\Phi$ is satisfiable under $\Theta$.

2. $\Gamma \vdash \ |	ext{e}| \downarrow \tau \Rightarrow \Phi$ is derivable for some $\Phi$ and $\Phi$ is satisfiable.

**Proof** (1) and (2) are proven simultaneously with structural induction on the derivation of $\Gamma \vdash e : \tau$.

**References**

\[ G \vdash b \uparrow \text{bool} \Rightarrow \top \] (ti-bool-up)
\[ G \vdash b \downarrow T \Rightarrow T = \text{bool} \] (ti-bool-dn)
\[ G \vdash i \uparrow \text{int} \Rightarrow \top \] (ti-int-up)
\[ G \vdash i \downarrow T \Rightarrow T = \text{int} \] (ti-int-dn)
\[ G \vdash x \uparrow G(x) \Rightarrow \top \] (ti-var-up)
\[ G \vdash x \downarrow T \Rightarrow T = G(x) \] (ti-var-dn)
\[ \Sigma(op) = \tau_1 \times \cdots \times \tau_n \rightarrow \tau \quad G \vdash e \downarrow \tau_1 \Rightarrow \Phi_1 \quad \ldots \quad G \vdash e_n \downarrow \tau_n \Rightarrow \Phi_n \] (ti-op-up)
\[ G \vdash op(e_1, \ldots, e_n) \downarrow T \Rightarrow \Phi_1 \land \cdots \land \Phi_n \land T = \tau \] (ti-op-dn)
\[ G \vdash e \downarrow \text{bool} \Rightarrow \Phi \quad G \vdash e_1 \downarrow T \Rightarrow \Phi_1 \quad G \vdash e_2 \downarrow T \Rightarrow \Phi_2 \] (ti-if-up)
\[ G \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \uparrow T \Rightarrow \Phi \land \Phi_1 \land \Phi_2 \] (ti-if-dn)
\[ G \vdash \langle e_1, e_2 \rangle \uparrow T_1 \times T_2 \Rightarrow \Phi \land \Phi_1 \land \Phi_2 \] (ti-tup-up)
\[ G \vdash \langle e_1, e_2 \rangle \downarrow T_1 \times T_2 \Rightarrow \Phi \land \Phi_1 \land \Phi_2 \] (ti-tup-dn)
\[ G \vdash \langle e_1, e_2 \rangle \downarrow X \Rightarrow \Phi_1 \land \Phi_2 \land X = T_1 \times T_2 \] (ti-tup-dn-atom)

Figure 2: Type inference rules generating types constraints (I)
\[ G \vdash e \uparrow T_1 * T_2 \Rightarrow \Phi \quad \text{(ti-fst-up)} \]
\[ G \vdash \text{fst}(e) \uparrow T_1 \Rightarrow \Phi \]  
(\text{ti-fst-up-atom})
\[ G \vdash \text{fst}(e) \uparrow X_1 \Rightarrow \Phi \land X_1 = X_1 * X_2 \]  
(\text{ti-fst-atom})
\[ G \vdash e \uparrow T * X \Rightarrow \Phi \]  
(\text{ti-fst-dn})
\[ G \vdash \text{snd}(e) \uparrow T_2 \Rightarrow \Phi \]  
(\text{ti-snd-up})
\[ G \vdash \text{snd}(e) \uparrow X_2 \Rightarrow \Phi \land X_1 = X_1 * X_2 \]  
(\text{ti-snd-atom})
\[ G \vdash e \uparrow X * T \Rightarrow \Phi \]  
(\text{ti-snd-dn})
\[ G, f : X_1 \rightarrow X_2, x : X_1 \vdash e \downarrow X_2 \Rightarrow \Phi \]  
(\text{ti-fun})
\[ G \vdash \text{fun } f(x) \text{ is } e \downarrow X_1 \rightarrow X_2 \Rightarrow \Phi \]  
(\text{ti-fun-dn})
\[ G \vdash \text{fun } f(x) \text{ is } e \uparrow T \Rightarrow \Phi \]  
(\text{ti-fun-dn-atom})
\[ G, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e \downarrow \tau_2 \Rightarrow \Phi \]  
(\text{ti-fun-anno-up})
\[ G \vdash \text{fun } f(x : \tau_1) : \tau_2 \text{ is } e \uparrow \tau_1 \rightarrow \tau_2 \Rightarrow \Phi \]  
(\text{ti-fun-anno-dn})
\[ G \vdash \text{fun } f(x : \tau_1) : \tau_2 \text{ is } e \downarrow T \Rightarrow \Phi \land \Phi \equiv \tau_1 \rightarrow \tau_2 \]  
(\text{ti-app})
\[ G \vdash \text{app}(e_1, e_2) \downarrow T_2 \Rightarrow \Phi \land \Phi_1 \land \Phi_2 \]  
(\text{ti-app-atom})
\[ G \vdash \text{app}(e_1, e_2) \uparrow T_2 \Rightarrow \Phi \]  
(\text{ti-app-dn})
\[ G \vdash \text{let } x = e_1 \text{ in } e_2 \text{ end } \downarrow T_2 \Rightarrow \Phi_1 \land \Phi_2 \]  
(\text{ti-let})
\[ G \vdash \text{let } x = e_1 \text{ in } e_2 \text{ end } \uparrow T_2 \Rightarrow \Phi_1 \land \Phi_2 \]  
(\text{ti-let-dn})
\[ G \vdash (e : \tau) \uparrow \tau \Rightarrow \Phi \]  
(\text{ti-anno-up})
\[ G \vdash (e : \tau) \downarrow T \Rightarrow \Phi \land \tau \equiv T \]  
(\text{ti-anno-dn})