1 Veritas (source language)

1.1 Syntax

$$c ::= n \mid \mathsf{true} \mid \mathsf{false}$$

Base values

$$\begin{aligned} e ::= & \, c \mid x \mid e_L \ \mathsf{op} \ e_R \mid f(\overrightarrow{e}) \mid \\ & x[e_i] \mid \mathsf{if} \ e_{\mathsf{cond}} \ B_2 \ \mathsf{else} \ B_2' \mid \\ & [e_{\mathsf{val}}; e_{\mathsf{size}}] \end{aligned}$$

Expressions

$$\begin{aligned} P ::= & \, e < e \mid e <= e \mid e == e \mid \\ & e > e \mid e >= e \mid \\ & P \ \&\& \ P \mid P \mid\mid P \mid !P \end{aligned}$$

Propositions

$$\begin{aligned} S ::= & \, \mathsf{let} \ x : T = e; \mid \\ & x = e; \mid \\ & x[e_i] = e; \mid e; \mid \\ & \mathsf{for} \ x \ \mathsf{in} \ N_{\mathsf{start}}..N_{\mathsf{end}}\{B_1\} \mid \\ & \mathsf{return} \ e; \end{aligned}$$

Statements

$$\begin{aligned} D ::= & \, \mathsf{fn} \ f(x_1 : T_1, ...) \to T_{\mathsf{ret}} \\ & \{B_2\} \mid \\ & \mathsf{fn} \ f(x_1 : T_1, ...) \to T_{\mathsf{ret}} \\ & \mathsf{requires}(P) \\ & \{B_2\} \mid \end{aligned}$$

Function definition

$$\begin{aligned} B_1 ::= & \, \cdot \mid B \, S \\ B_2 ::= & \, B_1 \mid B_1 \, e \end{aligned}$$

Blocks

$$\begin{aligned} T ::= & \, \mathsf{int} \mid \mathsf{bool} \mid [T; N] \mid \\ & \&T \mid \&\mathsf{mut} \ T \mid \mathsf{int}(N) \mid \\ & \{x : \mathsf{int} \mid P\} \end{aligned}$$

Types

1.2 Bidirectional type-checking

Hindley-Milner cannot deal with higher-ranked parametric polymorphism, subtyping and ad-hoc poymorphism. It can only support rank one polymorphism. It also does not scale well for more complex type systems such as dependent type systems due to decidability issues due to the intractibility of type inference via unification.

Bidirectional type-checking uses two types of judgements: *checking* and *synthesis*:

- Type checking takes a context Γ, program e and type τ as input and produces a true/false output reflecting whether e is well typed.
- Type synthesis (i.e. inference) takes a context Γ and program e as input and produces a type τ as output for e.

The core design principles and motivations behind bidirectional type-checking relates to improving on the aforementioned weaknesses of Hindley-Milner. Unlike Hindley-Milner, this type-checking method is not solely based on type inference or type checking. Checking is used in instances where full type inference would be undecidable and synthesis helps to relieve some annotation burden from the programmer such that the language does not have to be fully explicity typed.

The typing rules for a language that wants to use bidirectional type-checking has to specify if checking or synthesis is being performed. The Pfenning recipe provides an approach for designing a bidirectional type system. In general, introduction rules use the checking judgement and elimination rules use a synthesising judgement.

There are certain special cases for rules that are neither introduction nor elimination. For variables, we have an assumption that we can use to deduce its type. The conclusion is closest to what could be considered as a principal judgement. This results in the synthesis rule:

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A}$$

For explicit type annotations $e : A$, we have an assumption as the premise, which we can apply checking to. In the conclusion, we can use this checked knowledge and synthesise the type A.

$$\frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash (e : A) \Rightarrow A}$$

For rules like the variable rule which does not have checking, we need a rule that verifies that a fact is consistent with a requirement. For example, for a function $f : X \to Y$ applied to an argument of variable x, the requirement is $x : X$:

$$\frac{\Gamma \vdash e \Rightarrow A}{\Gamma \vdash e \Leftarrow A}$$

This rule synthesises a type for the variable and then in the conclusion we can check that this meets the requirement that it is of the type A. An alternative form of this rule is (as used in the Pfenning recipe is):

$$\frac{\Gamma \vdash e \Rightarrow A \quad A = B}{\Gamma \vdash e \Leftarrow B}$$

This form more closely resembles the implementation, as it enables the first premise to be output free and the constraint is imposed by the equality $A = B$. This rule can then also be used for subtyping by replacing $=$ with $<:$:

$$\frac{\Gamma \vdash e \Rightarrow A \quad A <: B}{\Gamma \vdash e \leftarrow B}$$

However, this rule does not address all the concerns of subtyping.

1.3 Typing rules

Global contexts

Σ_F is a global read-only context mapping function names to their types.

Local contexts

ϕ is a context containing propositions and constraints.
Δ is a mutable context, mapping mutable variables to their types.
Γ is an immutable context, mapping immutable variables to their types.

Judgements

Checking: $\phi; \Delta; \Gamma \vdash e \leftarrow T : (\phi'; \Delta')$
Synthesis: $\phi; \Delta; \Gamma \vdash e \Rightarrow (\phi'; \Delta'; \tau)$
Subtyping: $\phi \vdash \tau_1 <: \tau_2$
Provability: $\phi \vdash P$

Expression typing rules

$$\frac{}{\phi; \Delta; \Gamma \vdash n \Rightarrow (\phi; \Delta; \text{int}(n))} \quad \text{(T-Int)}$$

$$\frac{}{\phi; \Delta; \Gamma \vdash \text{true} \Rightarrow (\phi; \Delta; \text{bool})} \quad \text{(T-True)}$$

$$\frac{}{\phi; \Delta; \Gamma \vdash \text{false} \Rightarrow (\phi; \Delta; \text{bool})} \quad \text{(T-False)}$$

$$\frac{(x : \tau) \in \Gamma}{\phi; \Delta; \Gamma \vdash x \Rightarrow (\phi; \Delta; \tau)} \quad \text{(T-Var-Imm)}$$

$$\frac{(x : \tau) \in \Delta}{\phi; \Delta; \Gamma \vdash x \Rightarrow (\phi; \Delta; \tau)} \quad \text{(T-Var-Mut)}$$

$$\frac{\phi; \Delta; \Gamma \vdash e \Rightarrow (\phi'; \Delta'; \tau_e) \quad \phi' \vdash \tau_e <: T}{\phi; \Delta; \Gamma \vdash e \leftarrow T : (\phi'; \Delta')} \quad \text{(T-Sub)}$$

$$\frac{\begin{array}{c}\phi_0; \Delta_0; \Gamma \vdash e_L \Rightarrow (\phi_1; \Delta_1; \tau_L) \quad \phi_1 \vdash \tau_L <: \text{int} \\ \phi_1; \Delta_1; \Gamma \vdash e_R \Rightarrow (\phi_2; \Delta_2; \tau_R) \quad \phi_2 \vdash \tau_R <: \text{int} \\ \tau_{res} = \text{join-op}(+, \tau_L, \tau_R)\end{array}}{\phi_0; \Delta_0; \Gamma \vdash e_L \ \text{op} e_R \Rightarrow (\phi_2; \Delta_2; \tau_{res})} \quad \text{(T-Op)}$$

The 'join-op' operator attempts to resolve types τ_L and τ_R e.g. join-op$(+, \text{int}(1), \text{int}(2)) = \text{int}(3)$.

$$\frac{\begin{array}{c}\phi_0; \Delta_0; \Gamma \vdash e_{val} \Rightarrow (\phi_1; \Delta_1; \tau_{val}) \\ \phi_1; \Delta_1; \Gamma \vdash e_{size} \Rightarrow (\phi_2; \Delta_2; \tau_{size}) \\ \phi_2 \vdash \tau_{size} <: \text{int} \quad N = \text{val}(\tau_{size})\end{array}}{\phi_0; \Delta_0; \Gamma \vdash [e_{val}; e_{size}] \Rightarrow (\phi_2; \Delta_2; [\tau_{val}; N])} \quad \text{(T-Array-Create)}$$

$$\frac{\begin{array}{c}(x : [T; N]) \in (\Gamma \cup \Delta_0) \\ \phi_0; \Delta_0; \Gamma \vdash e_i \Rightarrow (\phi_1; \Delta_1; \tau_i) \quad \phi_1 \vdash \tau_i <: \text{int} \\ \phi_1 \wedge \text{prop}(\tau_i) \vdash 0 \leq \text{val}(\tau_i) < N \quad \text{(Proof Obligation)}\end{array}}{\phi_0; \Delta_0; \Gamma \vdash x[e_i] \Rightarrow (\phi_1; \Delta_1; T)} \quad \text{(T-Array-Idx)}$$

$$\frac{\begin{array}{c}\phi_0; \Delta_0; \Gamma \vdash e_{cond} \Rightarrow (\phi_1; \Delta_1; \tau_c) \quad \phi_1 \vdash \tau_c <: \text{bool} \\ \phi_1 \wedge \text{prop}(\tau_c); \Delta_1; \Gamma \vdash B_3 \Rightarrow (\phi_2; \Delta_2; \tau_2) \\ \phi_1 \wedge \neg\text{prop}(\tau_c); \Delta_1; \Gamma \vdash B_3' \Rightarrow (\phi_3; \Delta_3; \tau_3) \\ (\phi_{join}, \Delta_{join}, \tau_{join}) = \text{join}(\phi_2, \Delta_2, \tau_2, \phi_3, \Delta_3, \tau_3)\end{array}}{\phi_0; \Delta_0; \Gamma \vdash (\text{if } e_{cond} B_3 \text{ else } B_3') \Rightarrow (\phi_{join}; \Delta_{join}; \tau_{join})} \quad \text{(T-If-Expr)}$$

The 'join' operator attempts to join contexts, or join types and define a refinement type.

$$\frac{\begin{array}{c}\Sigma_F(f) = ((x_i : T_i)_{i=1}^n \to T_{ret} \text{ requires } P_{req}) \\ \phi_0; \Delta_0; \Gamma \vdash \vec{e} \Rightarrow (\phi_n; \Delta_n; \vec{\tau}) \quad \text{(Type args sequentially, threading state)} \\ \forall i \in 1..n, \quad \phi_n \vdash \tau_i <: T_i \quad \text{(Arguments match types)} \\ \phi_n \wedge \text{prop}(\vec{\tau}) \vdash P_{req}[\text{val}(\vec{\tau})/\vec{x}] \quad \text{(Proof Obligation)}\end{array}}{\phi_0; \Delta_0; \Gamma \vdash f(\vec{e} : \vec{T}) \Rightarrow (\phi_n; \Delta_n; T_{ret})} \quad \text{(T-Call)}$$

Statement typing rules

$$\frac{}{\phi; \Delta; \Gamma \vdash \cdot : (\phi; \Delta; \Gamma)} \quad \text{(T-Block-Empty)}$$

$$\frac{\phi_0; \Delta_0; \Gamma_0 \vdash S : (\phi_1; \Delta_1; \Gamma_1) \quad \phi_1; \Delta_1; \Gamma_1 \vdash B : (\phi_2; \Delta_2; \Gamma_2)}{\phi_0; \Delta_0; \Gamma_0 \vdash (S\ B) : (\phi_2; \Delta_2; \Gamma_2)} \quad \text{(T-Block-Seq)}$$

$$\frac{\phi_0; \Delta_0; \Gamma_0 \vdash e \leftarrow T : (\phi_1; \Delta_1)}{\phi_0; \Delta_0; \Gamma_0 \vdash (\text{let } x : T = e;) : (\phi_1; \Delta_1; \Gamma_0, x : T)} \quad \text{(T-Let)}$$

$$\frac{\phi_0; \Delta_0; \Gamma \vdash e \leftarrow T : (\phi_1; \Delta_1)}{\phi_0; \Delta_0; \Gamma \vdash (\text{let mut } x : T = e;) : (\phi_1; (\Delta_1, x : T); \Gamma)} \quad \text{(T-Let-Mut)}$$

$$\frac{\begin{array}{c}(x : T_{master}) \in \Delta_0 \quad \text{(Get master type of } x\text{)} \\ \phi_0; \Delta_0; \Gamma \vdash e \Rightarrow (\phi_1; \Delta_1; \tau_e) \\ \phi_1 \vdash \tau_e <: T_{master} \quad \text{(New value must conform to master type)}\end{array}}{\phi_0; \Delta_0; \Gamma \vdash (x = e;) : (\phi_1; \Delta_1[x \mapsto \tau_e]; \Gamma)} \quad \text{(T-Assign)}$$

$$\frac{\begin{array}{c}(x : [T; N]) \in \Delta_0 \\ \phi_0; \Delta_0; \Gamma \vdash e_i \Rightarrow (\phi_1; \Delta_1; \tau_i) \quad \phi_1 \vdash \tau_i <: \text{int} \\ \phi_1; \Delta_1; \Gamma \vdash e \Rightarrow (\phi_2; \Delta_2; \tau_v) \quad \phi_2 \vdash \tau_v <: T \\ \phi_2 \wedge \text{prop}(\tau_i) \vdash 0 \leq \text{val}(\tau_i) < N \quad \text{(Proof Obligation)}\end{array}}{\phi_0; \Delta_0; \Gamma \vdash (x[e_i] = e;) : (\phi_2; \Delta_2; \Gamma)} \quad \text{(T-Array-Assign)}$$

$$\frac{\begin{array}{c}\text{Loop invariant is } (\phi_I, \Delta_I) \\ \phi_0; \Delta_0 \vdash \exists \phi_I.\Delta_I \quad \text{(Invariant holds on entry)} \\ \phi_I \wedge N_{start} \leq i < N_{end}; \Delta_I; (\Gamma, i : \text{int}) \vdash B_1 : (\phi_1; \Delta_1; \Gamma) \\ \phi_1; \Delta_1 \vdash \exists \phi_I.\Delta_I \quad \text{(Invariant preserved by body)}\end{array}}{\phi_0; \Delta_0; \Gamma \vdash (\text{for } i \text{ in } N_{start}..N_{end}\{B_1\})_{\text{inv}(\phi_I, \Delta_I)} : (\phi_I; \Delta_I; \Gamma)} \quad \text{(T-For)}$$

Subtyping rules

$$\frac{}{\phi \vdash T <: T} \quad \text{(Sub-Refl)}$$

$$\frac{}{\phi \vdash T <: \text{top}} \quad \text{(Sub-Top)}$$

$$\frac{}{\phi \vdash \text{int}(N) <: \text{int}} \quad \text{(Sub-Singleton-Int)}$$

$$\frac{}{\phi \vdash \{x : \text{int} | P\} <: \text{int}} \quad \text{(Sub-Refine-Int)}$$

$$\frac{\phi \wedge P_1 \vdash P_2}{\phi \vdash \{x : \text{int} | P_1\} <: \{x : \text{int} | P_2\}} \quad \text{(Sub-Refine-Refine)}$$

$$\frac{\phi \vdash P[N/x]}{\phi \vdash \text{int}(N) <: \{x : \text{int} | P\}} \quad \text{(Sub-Singleton-Refine)}$$