1 A core language

a, b, A, B	::=	x	variable
		$\lambda x.a$	abstraction
		$a\ b$	application
		$(x:A) \to B$	function type
		$\mathbf{Type}_i$	type
		$(x : A)$	annotation
Γ	::=	$\cdot$	empty
		$\Gamma, x : A$	variable assumption

1.1 Initial Specification

This system *specifies* when expressions are well typed, but is not syntax directed. We can't read an algorithm for type checking from these rules.

$$\frac{(x : A) \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash x : A} \quad \text{T\_Var}$$

$$\frac{\Gamma, x : A \vdash a : B \qquad \Gamma \vdash A : \mathbf{Type}_i}{\Gamma \vdash \lambda x.a : (x:A) \to B} \quad \text{T\_ILam}$$

$$\frac{\Gamma \vdash b : (x:A) \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash b\ a : [a/x]B} \quad \text{T\_App}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathbf{Type}_i : \mathbf{Type}_{(1+i)}} \quad \text{T\_Star}$$

$$\frac{\Gamma \vdash A : \mathbf{Type}_i \qquad \Gamma, x : A \vdash B : \mathbf{Type}_j}{\Gamma \vdash (x:A) \to B : \mathbf{Type}_{(i \sqcap j)}} \quad \text{T\_Arrow}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash A = B \qquad \Gamma \vdash B : \mathbf{Type}_i}{\Gamma \vdash a : B} \quad \text{T\_Conv}$$

(Rules for equality and context checking omitted.)

1.2 Syntax-directed system

One way to make the system syntax directed is to remove the conversion rule (which could apply at any time) and modify the rules T_ILam (annotating the type of the variable) and T_App (inlining a use of conversion).

We also drop context checking judgment. We'll do it incrementally. We'll start with the empty context and make sure that we check everything as we add it to the context.

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A} \quad \text{I\_Var}$$

$$\frac{\Gamma, x : A \vdash a \Rightarrow B \qquad \Gamma \vdash A \Rightarrow A' \qquad \Gamma \vdash A' = \mathbf{Type}_i}{\Gamma \vdash \lambda x : A.a \Rightarrow (x:A) \to B} \quad \text{I\_Lam}$$

$$\frac{\begin{array}{l}\Gamma \vdash b \Rightarrow B \\ \Gamma \vdash B = (x : A_1) \to B' \\ \Gamma \vdash a \Rightarrow A_2 \\ \Gamma \vdash A_1 = A_2\end{array}}{\Gamma \vdash b\ a \Rightarrow [a/x]B'}\ \text{I\_AppSD}$$

$$\frac{}{\Gamma \vdash \mathbf{Type}_i \Rightarrow \mathbf{Type}_{(1+i)}}\ \text{I\_Star}$$

$$\frac{\begin{array}{l}\Gamma \vdash A \Rightarrow A' \\ \Gamma \vdash A' = \mathbf{Type}_i \\ \Gamma, x : A \vdash B : B' \\ \Gamma \vdash B' = \mathbf{Type}_j\end{array}}{\Gamma \vdash (x : A) \to B \Rightarrow \mathbf{Type}_{(i \sqcap j)}}\ \text{I\_Arrow}$$

1.3 Bidirectional system

Alternatively, we could make the system bidirectional, where the type system is composed of two different modes: type inference and type checking. We assume that the type provided to the checking judgment is a well-formed type.

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

$$\frac{\begin{array}{l}\Gamma \vdash b \Rightarrow (x : A) \to B \\ \Gamma \vdash a \Leftarrow A\end{array}}{\Gamma \vdash b\ a \Rightarrow [a/x]B}\ \text{I\_App}$$

$$\frac{}{\Gamma \vdash \mathbf{Type}_i \Rightarrow \mathbf{Type}_{(1+i)}}\ \text{I\_Star}$$

$$\frac{\begin{array}{l}\Gamma \vdash A \Rightarrow A' \\ \Gamma \vdash A' = \mathbf{Type}_i \\ \Gamma, x : A \vdash B : B' \\ \Gamma \vdash B' = \mathbf{Type}_j\end{array}}{\Gamma \vdash (x : A) \to B \Rightarrow \mathbf{Type}_{(i \sqcap j)}}\ \text{I\_Arrow}$$

$$\frac{\Gamma \vdash a \Leftarrow A}{\Gamma \vdash (a : A) \Rightarrow A}\ \text{I\_Ann}$$

$$\frac{\Gamma, x : A \vdash a \Leftarrow B}{\Gamma \vdash \lambda x.a \Leftarrow (x : A) \to B}\ \text{C\_Lam}$$

$$\frac{\begin{array}{l}\Gamma \vdash a \Rightarrow A \\ \Gamma \vdash A = B\end{array}}{\Gamma \vdash a \Leftarrow B}\ \text{C\_Conv}$$

1.4 Discussion

- Type annotations $(a : A)$ are never necessary in the syntax directed system.
- The bidirectional system reduces the number of necessary type annotations. Particularly if the types of top-level functions are already known. (Even more so with the addition of new language features.)
- The bidirectional system is **not** closed under (normal) substitution. Variables are always checked in inference mode, that means that they cannot be simply substituted by pure lambda expressions (or other terms that must be validated in checking mode.)

- The bidirectional system requires type annotations to switch from inference mode to checking mode. These annotations cannot be eliminated. Yet at the same time, they may interfere with equality.
- We can add extra rules to the bidirectional system for flexibility. In this little language, we might add back the inference mode rule for annotated lambda expressions.

$$\frac{\begin{array}{l}\Gamma, x : A \vdash a \Rightarrow B \\ \Gamma \vdash A \Rightarrow A' \\ \Gamma \vdash A' = \mathbf{Type}_i\end{array}}{\Gamma \vdash \lambda x : A.a \Rightarrow (x{:}A) \to B} \quad \text{I\_Lam}$$

The upshot is that bidirectionality makes sense for a surface language, but perhaps we should elaborate it to an *annotated* internal language where everything can be checked in inference mode.

2 Equivalence

What does it mean for two terms to be equivalent. I wrote it suggestively as $\Gamma \vdash A = B$, but some systems judge equivalence without using the context.

In this version, we will use the context to retrieve variable definitions only. In fact, we will be able to show that: $\Gamma \vdash x = A$ when $x = A$ is in the context.

3 Additional forms for homework 1

We can add a number of standard forms to this language. These are the non-syntax directed rules. You'll need to figure out how to turn them into an algorithm.

$$\frac{}{\Gamma \vdash \mathbf{Unit} : \mathbf{Type}_0} \quad \text{T\_TyUnit}$$

$$\frac{}{\Gamma \vdash \mathbf{tt} : \mathbf{Unit}} \quad \text{T\_LitUnit}$$

$$\frac{}{\Gamma \vdash \mathbf{Bool} : \mathbf{Type}_0} \quad \text{T\_Bool}$$

$$\frac{}{\Gamma \vdash \mathbf{true} : \mathbf{Bool}} \quad \text{T\_True}$$

$$\frac{}{\Gamma \vdash \mathbf{false} : \mathbf{Bool}} \quad \text{T\_False}$$

$$\frac{\begin{array}{l}\Gamma \vdash a : \mathbf{Bool} \\ \Gamma \vdash b : A \\ \Gamma \vdash c : A\end{array}}{\Gamma \vdash \mathbf{if}\ a\ \mathbf{then}\ b\ \mathbf{else}\ c : A} \quad \text{T\_If}$$

$$\frac{\begin{array}{l}\Gamma \vdash a : A \\ \Gamma, x : A, x = a \vdash b : B \\ \Gamma \vdash B : \mathbf{Type}_i\end{array}}{\Gamma \vdash \mathbf{let}\ x = a\ \mathbf{in}\ b : B} \quad \text{T\_Let}$$

Notes:

1. Let expressions add a definition of the variable to the context as well as the type. The result type of the let expression is not allowed to mention the bound variable.

2. When the scrutinee of an if-expression is a variable, we can do refinement: i.e. add an equation to the context that witnesses the pattern match.

$$\frac{\begin{array}{l}\Gamma \vdash x : \mathbf{Bool} \\ \Gamma, x = \mathbf{true} \vdash b : A \\ \Gamma, x = \mathbf{false} \vdash c : A\end{array}}{\Gamma \vdash \mathbf{if}\ x\ \mathbf{then}\ b\ \mathbf{else}\ c : A}\ \textsc{T_IfVar}$$