Contents

1 The Lambda Calculus

1.1 Foundations

- **Application** is left-associative and **binds more tightly than abstraction**: $\lambda x.M_1M_2M_3 \equiv \lambda x.(M_1M_2M_3) \equiv \lambda x.(M_1M_2)M_3$
  - Thus, abstraction is right associative
- **De-Bruijn notation** is a compact way of expressing λ-calculus expressions: Each instance of a lambda (together with the variable it binds) creates a `[` and each use of a variable is replaced by a number showing how many `[` there are to the left of it up to and including the one that bound it **e.g.** $\lambda x.x(\lambda y.y(xx))$ **becomes [1[1 (2 2)]]**

- **Capture avoidance: Free variables cannot become bound, other than if they have been abstracted away** — scope
- **Freshness: Variables that have been bound cannot be reused** — immutability
- $\mathrm{FV}(M)$ = set of free variables in the term M
- **α-reduction:** $\lambda x.M \underset{\alpha}{\to} \lambda y.M[x \leftarrow\!\!\prec y]$ **if** $y \notin \mathbf{FV}(M)$ — find and replace all
- α-equivalence: Expressions are α-equivalent iff one can be obtained from the other by a series of α-reductions

1.2 Contractions

- **β-reduction (contraction):** $(\lambda x.M)N \underset{(\beta)}{\to} M[x \leftarrow N]$ — function application
  - The abstraction which the application is happening to is called the redex
  - A series of β-reductions is called a reduction sequence — this is the process by which actual computation will happen
- **η-reduction (eta-reduction):** $\lambda x.Mx \underset{\eta}{\to} M$ — if all an abstraction does is function application, we can discard it as we already have function application as an implicit part of our language
  - Arises from β-reduction: $(\lambda x.Mx)N \to MN$

1.3 Evaluation

- $A \Rightarrow B$ (A evaluates to B) iff there exists a reduction sequence that takes A to B and B is a normal form
  - There may be multiple reduction sequences from a given term, only some of which may converge
  - Normal forms are unique: All convergent reduction sequences from one term converge upon the same normal form
- $\nexists B : A \Rightarrow B$ iff A diverges (throws an error or causes an infinite loop)
- **Normal reduction order = always reduce the leftmost redex**
- **Applicative reduction order = always reduce the rightmost redex**
- Normal order is guaranteed to find a normal form if one exists (whereas applicative is not)

1.4 Normal forms

- Let $E, E_1, ..., E_n$ be expressions in the relevant normal form
- Let $e, e_1, ..., e_n$ be expressions (may or may not be in the normal form)
- **(Full) Normal form** (NF) $= \lambda x.E \mid xE_1...E_n \mid x$
  - There really is no way to simplify it: **Cannot be β-reduced**
- **Head normal form** (HNF) $= \lambda x.E \mid xe_1...e_n \mid x$
  - **Arguments of applications are not required to be β-irreducible but bodies of abstractions are**
  - *The lazy normal form of theoreticians*

- **Weak head normal form** (WHNF) = $\lambda x.e \mid xe_1...e_n \mid x$
  - **Outermost term is not an application where the LHS is an abstraction**
  - *The lazy normal form of actual programming languages*
- **Weak normal form** (WNF) = $\lambda x.e \mid xE_1...E_n \mid x$
  - **No applications anywhere that could be evaluated unless they are within an abstraction**
  - *Seemingly only included for completeness*
- If in NF, then in HNF and in WNF
- If in HNF, then in WHNF
- If in WNF, then in WHNF

1.5 Combinators

- Combinators abbreviate commonly used closed (no free variables) λ-calculus expressions
- The definitions of any specific combinators needed will be included in the exam
- Any closed form λ-calculus expression can be written using only the S $(\lambda x.y.z.xz(yz))$ and K $(\lambda x.y.x)$ combinators
  - *As the λ-calculus is Turing complete and an expression can be fully evaluated only if it is closed, any computable function can be written only using the S and K combinators*

2 Explicit Typing (STLC and PCF)

2.1 Terminology for proof rules

- An environment = an association of values to free variables
  - Type environment = list of pairs of variables and their types
- A judgment = an evaluation of an expression given an environment
  - Typing judgment = evaluation of the type of a term in a given type environment
- A derivation = a proof of a judgement using steps of axiomatic smaller judgements

2.2 Syntax of the simply typed lambda calculus (STLC)

- Simply typed means the only base type is a generic value (o) and all other types are functions on types
- The syntax of the STLC has explicit type annotations on the lambda bindings of variables (and only here)

2.3 Typed evaluation rules (dynamic semantics) for STLC

- $$\frac{\Sigma \vdash M \Rightarrow \lambda x{:}t.\, M' \qquad \Sigma \vdash M'[x \leftarrow P] \Rightarrow N}{\Sigma \vdash MP \Rightarrow N}\ \text{appl}$$

- $$\frac{}{\Sigma, x = v, \ldots \vdash x \Rightarrow v}\ \text{var}$$

- $$\frac{\Sigma \vdash M \Rightarrow N}{\Sigma \vdash \lambda x{:}t.\, M \Rightarrow \lambda x{:}t.\, N}\ \text{abs}$$

2.4 Typing rules (static semantics) for STLC

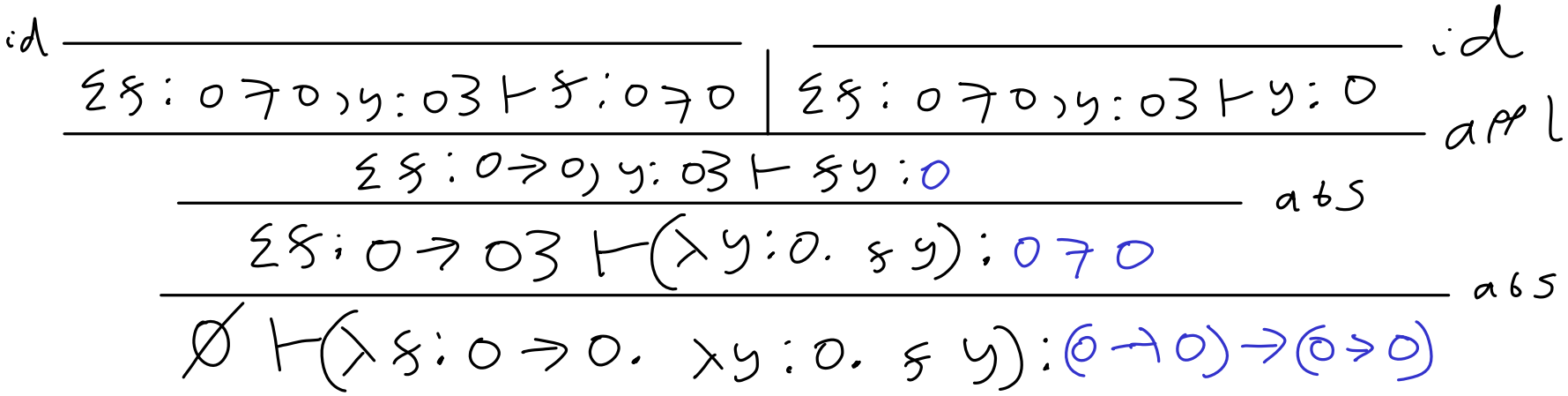
$$\frac{\Gamma \vdash M : S \to t \qquad \Gamma \vdash N : S}{\Gamma \vdash M\,N : t}\ \text{app}$$

$$\frac{}{\{x : t\} \vdash x : t}\ \text{id}$$

$$\frac{\{x = S, \ldots\} \vdash M : t}{\Gamma \vdash (\lambda x : S.\ M) : S \to t}\ \text{abs}$$

Subject reduction theorem: Reduction to a normal form preserves the type of an expression

An example of static semantics:



Proof trees are built bottom to top but verified (and types are filled in) top-to-bottom

2.5 PCF (Programming with Computable Functions)

- PCF extends the STLC by introducing distinct base types for different classes of value and allowing let-declarations (of constants not variables)
  - *As λ-calculus (even the S and K combinators alone) was Turing-complete, PCF is not literally more powerful but it is far more convenient to have syntactic sugar than to have to encode numerals in unary with function compositions etc. Also, a more expressive type system allows stronger correctness guarantees*
- Base types are num (n) and bool (b) — num refers to Church numerals (naturals (hence the operations we define for them)) not a more general class of numbers
- $t := \mathrm{n} \mid \mathrm{b} \mid t * t \mid t \mapsto t$

- $M := 0 \mid 1 \mid 2 \mid ... \mid \text{true} \mid \text{false} \mid \text{succ}(M) \mid \text{pred}(M) \mid \text{zero?}(M) \mid M + M$
  $\mid M \times M \mid \langle M, M \rangle \mid \text{fst}(M) \mid \text{snd}(M) \mid \text{if } M \text{ then } M \text{ else } M \mid x \mid$
  $\lambda x : t.M \mid MM \mid \text{let } d \text{ in } M$
  $d := (x = M) \mid d \text{ and } d$

2.6 PCF Dynamic Semantics

$$\bullet\ \frac{E \vdash M_1 \Rightarrow \text{true} \qquad E \vdash M_2 \Rightarrow N}{E \vdash \text{if } M_1 \text{ then } M_2 \text{ else } M_3 \Rightarrow N}\ \text{if true}$$

$$\bullet\ \frac{E \vdash M_1 \Rightarrow \text{false} \qquad E \vdash M_3 \Rightarrow N}{E \vdash \text{if } M_1 \text{ then } M_2 \text{ else } M_3 \Rightarrow N}\ \text{if false}$$

$$\bullet\ \frac{E \vdash d \Rightarrow E' \qquad E E' \vdash M \Rightarrow V}{E \vdash \text{let } d \text{ in } M \Rightarrow V}\ \text{let}$$

$$\bullet\ \frac{E \vdash M \Rightarrow V}{E \vdash x = M \Rightarrow \{(x, V)\}}\ \text{decl}$$

$$\bullet\ \frac{E \vdash d_1 \Rightarrow E_1 \qquad E \vdash d_2 \Rightarrow E_2}{E \vdash d_1 \text{ and } d_2 \Rightarrow E_1 E_2}$$

- Note that **declarations "evaluate" into additional environments** rather than expressions
- **Applications are evaluated using lazy evaluation** (substitute then evaluate whole expression) **whereas declarations use eager evaluation** (evaluate before substituting (then do the new remaining evaluation))
- When two environments are placed next to each other (e.g. EE') they are implicitly unioned and this is done in a way that respects the scope of bindings (e.g. let $x = 1$ in let $x = 2$ in $x \Rightarrow 2$)

2.7 PCF Static Semantics

$$\bullet\ \frac{\mathcal{E} \vdash M_1 : b \quad \mathcal{E} \vdash M_2 : t \qquad \mathcal{E} \vdash M_3 : t}{\mathcal{E} \vdash \text{if } M_1 \text{ then } M_2 \text{ else } M_3 : t}\ \text{if}$$

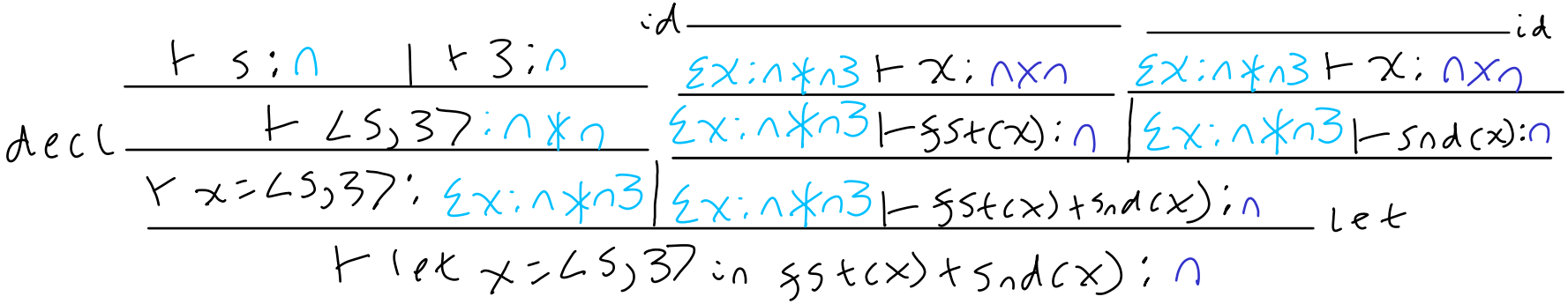
$$\bullet\ \frac{\mathcal{E} \vdash e_1 : t_1 \qquad \mathcal{E} \vdash e_2 : t_2}{\mathcal{E} \vdash \langle e_1, e_2 \rangle : t_1 * t_2}$$

$$\bullet\ \frac{\mathcal{E} \vdash d : \mathcal{E}' \qquad \mathcal{E}\mathcal{E}' \vdash e : t}{\mathcal{E} \vdash (\text{let } d \text{ in } e) : t}\ \text{let}$$

$$\frac{\mathcal{E} \vdash e : t}{\mathcal{E} \vdash (x = e) : \{(x : t)\}}\ \text{decl}$$

Note that each branch of an if must be assigned the same type

An example:



Note that the rules for +, `fst` etc are common sense

3 Miscellaneous Content

3.1 Recursion

- A function in the λ-calculus cannot directly refer to itself in its own definition, but the Y combinator applies a function infinitely many times: $Yf \to f(Yf) \to f(f(Yf)) \to ...$
- $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
- Under strict evaluation the Y-combinator is useless as everything diverges under it, but under lazy evaluation recursive functions are evaluated down to their base case then return
- $Y(\lambda f.M)$ **is abbreviated to** $\mu f.M$

$$\frac{M[\mu f{:}t.M] \Rightarrow N}{\mu f{:}t.M \Rightarrow N}$$

- *The dynamic semantics of functions on numbers are recursive (Peano arithmetic) — the fact the halting problem exists by sometimes divergent terms such as the Y combinator is the price we pay for expressiveness*

3.2 Structural Semantics

- So far we have mainly considered natural semantics ($\Rightarrow$), how an expression becomes a normal form.
- Structural semantics ($\rightarrow$) describe the step-by-step process of evaluation
- By convention we evaluate left-to-right

- Call-by-name is lazy (as have been our natural semantics):

$$\frac{e_0 \Rightarrow e'}{e_0 \ \mathrm{op}\ e_1 \Rightarrow e' \ \mathrm{op}\ e_1}$$

$$\frac{e \Rightarrow e'}{\mathrm{n}\ \mathrm{op}\ e \Rightarrow \mathrm{n}\ \mathrm{op}\ e'}$$

$$\frac{e_0 \Rightarrow e'}{e_0\, e_1 \Rightarrow e'\, e_1}$$

$$\frac{}{(\lambda x{:}t.\mathrm{n})\, e \Rightarrow \mathrm{n}[x \leftarrow e]}$$

$$\frac{}{(\mu x{:}t.e) \Rightarrow e[x \leftarrow (\mu x{:}t.e)]}$$

$$\frac{e \Rightarrow e'}{\mathrm{if}\ e\ \mathrm{then}\ e_0\ \mathrm{else}\ e_1 \Rightarrow \mathrm{if}\ e'\ \mathrm{then}\ e_0\ \mathrm{else}\ e_1}$$

$$\frac{}{\mathrm{if}\ \mathrm{true}\ \mathrm{then}\ e_0\ \mathrm{else}\ e_1 \Rightarrow e_0}$$

- Call-by-value is eager, only requires some changes to call-by-name:
  - $$\frac{e \;\rightarrow\; e'}{\wedge e \;\rightarrow\; \wedge e'}$$
  - $$\frac{}{(\lambda x{:}t.\wedge_1)\wedge_2 \;\rightarrow\; \wedge[x \leftarrow \wedge_2]}$$
  - $$\frac{}{(\nu x{:}t.e)\wedge \rightarrow (e[x \leftarrow (\nu x{:}t.e)])\wedge}$$

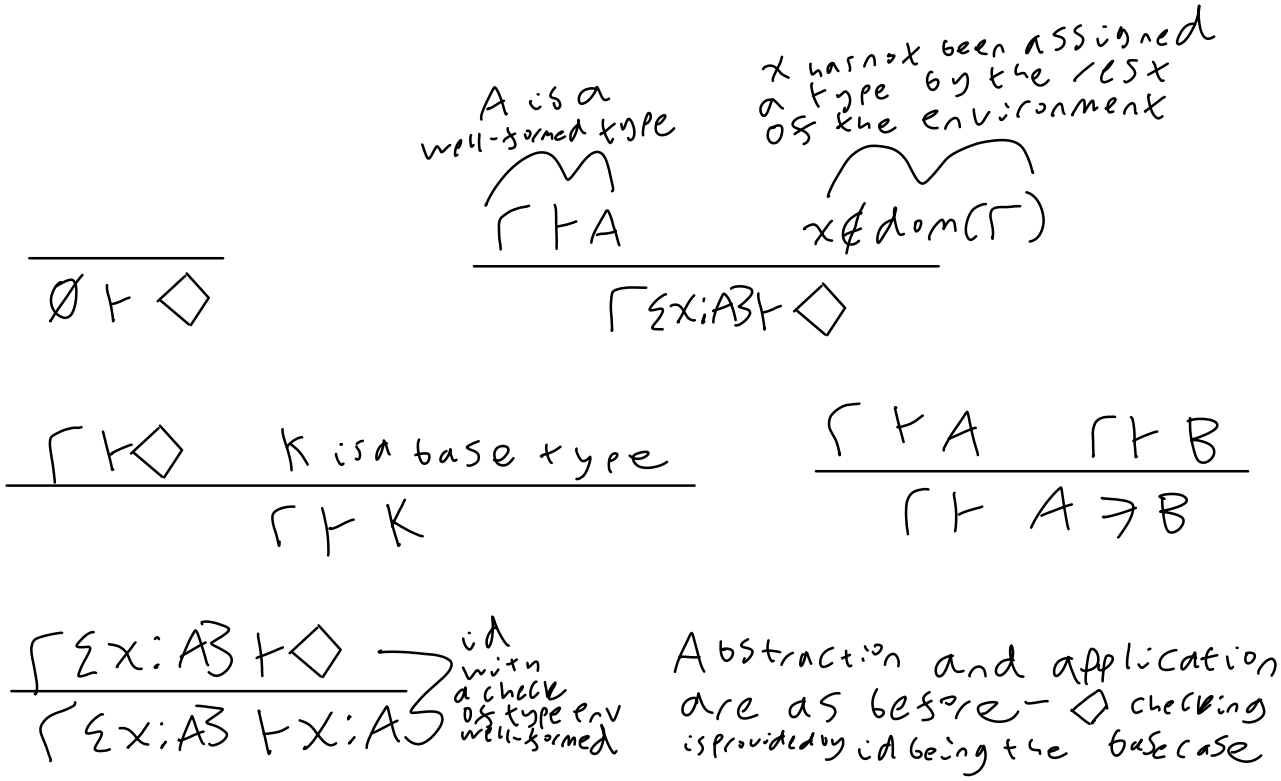
3.3 Type theory

- **Curry-Howard Isomorphism: Types are propositions and vice versa. Moreover, the type of every closed expression is a tautology.**
  - $\mapsto$ is isomorphic to logical implication (including being right associative like it), $*$ is isomorphic to logical conjunction etc
  - *Functions can curried because* $(A \wedge B) \to C \equiv A \to (B \to C)$

3.4 Type judgements

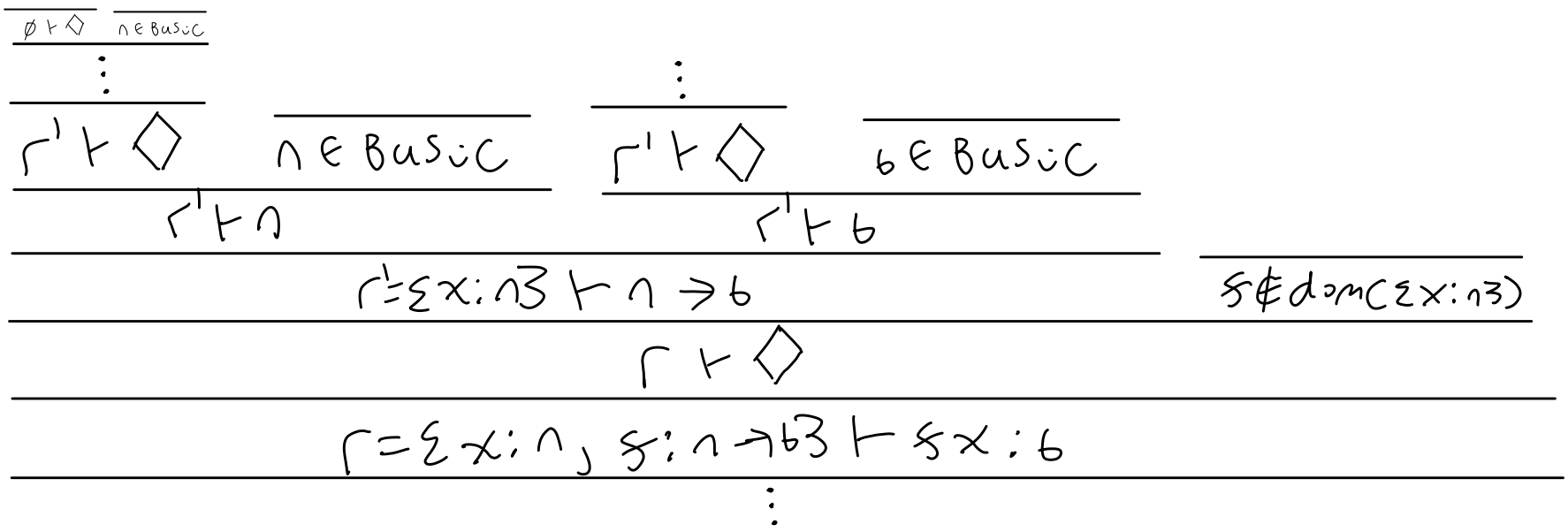
- Types allowed us to check that terms were well-formed but Cardelli's type judgements will allow us to check that types themselves are well-formed
- $\Gamma \vdash \diamond$ **means the environment** Γ **is well formed**
- $\Gamma \vdash T$ **means the the type** T **is well formed in** Γ
- $\Gamma \vdash M : T$ **means the the term** M **has type** T **(and so is well formed) in** Γ

Cardelli rules:



Additional rules for well-formed types will arise as our type system becomes richer (e.g. polymorphic types) but these follow the same pattern as the function type rule shown above.

Type trees begin much as before but now continue past the id term rule to only bottom out at empty environment is well formed. An example:



Note that at each check of the environment admitting a type, that type is removed from the environment

4 Implicit Typing (MiniML)

4.1 MiniML

- MiniML uses `fn` instead of λ but otherwise the only notable difference to PCF is the use of type inference instead of explicit typing. (If we carried on using λ, then MiniML expressions could be confused for untyped lambda calculus expressions)

4.2 Polymorphism

- MiniML (sort of) has universally quantified types, which allows it to have lists
  - `nil`: $\forall\alpha.\alpha$ `list`
  - `::`: $\forall\alpha.\alpha * \alpha$ `list` $\mapsto \alpha$ `list`
  - `hd`: $\forall\alpha.\alpha$ `list` $\mapsto \alpha$

- $\forall$s are not actually allowed in MiniML types, instead MiniML has so called type schemes which do allow $\forall$
- Type schemes allow sub-terms to be universally quantified as an intermediate step in deriving a true type for a fully formed term (the context will cause the type scheme to specialise into a particular type)
- Closure of the type scheme of a type expression σ in a typing environment $\mathrm{T} = \mathrm{CL}_{\mathrm{T}}(\sigma) = \forall\alpha_1...\alpha_n.\sigma$ where $\alpha_1...\alpha_n$ are the free variables in σ (do not occur in T)
- As well as there being polymorphic built-ins (lists), programmers can create polymorphic functions of a sort. This is achieved by using a let ... in ... to declare the generic function then immediately use it in a particular context

- **The type environment produced by a declaration is (in generality) the closed type scheme arising from the type expression of the definition**, we just didn't realise before because previously we had no free variables so it was implicitly closed

4.3 Type inference

Hindley-Damas-Milner type inference algorithm:

1. **Build tree of type variables using structural induction over terms — create a fresh type variable for each instance of a universal quantifier**

2. **Build a set of constraints based on the rules used to build the tree (using $\sim$ to denote an equivalence relation on types). A constraint that a closure C_i instantiates into a type variable t_j of a particular form is written as $C_i \succ t_j \sim ...$**

3. **Combine constraints to find the most general unifier at each level and so eventually at the root — consume by substituting into each other until only the type at the root remains. Even if the type of the root is found early, all constraints must be consumed to check that nothing has been assigned an additional incompatible type**

- *PCF has roughly the most powerful type system for which type checking (and so certainly type inference) is not in general undecidable*

Constraints:

- $$\frac{T \oplus [x : t_2] \vdash e : t_3}{T \vdash (\lambda x.\, e) : t_1}\ \text{abs} \qquad t_1 \sim t_2 \to t_3$$

- $$\frac{T \vdash e_1 : t_2 \qquad T \vdash e_2 : t_3}{T \vdash (e_1\ e_2) : t_1}\ \text{app} \qquad t_2 \sim t_3 \to t_1$$

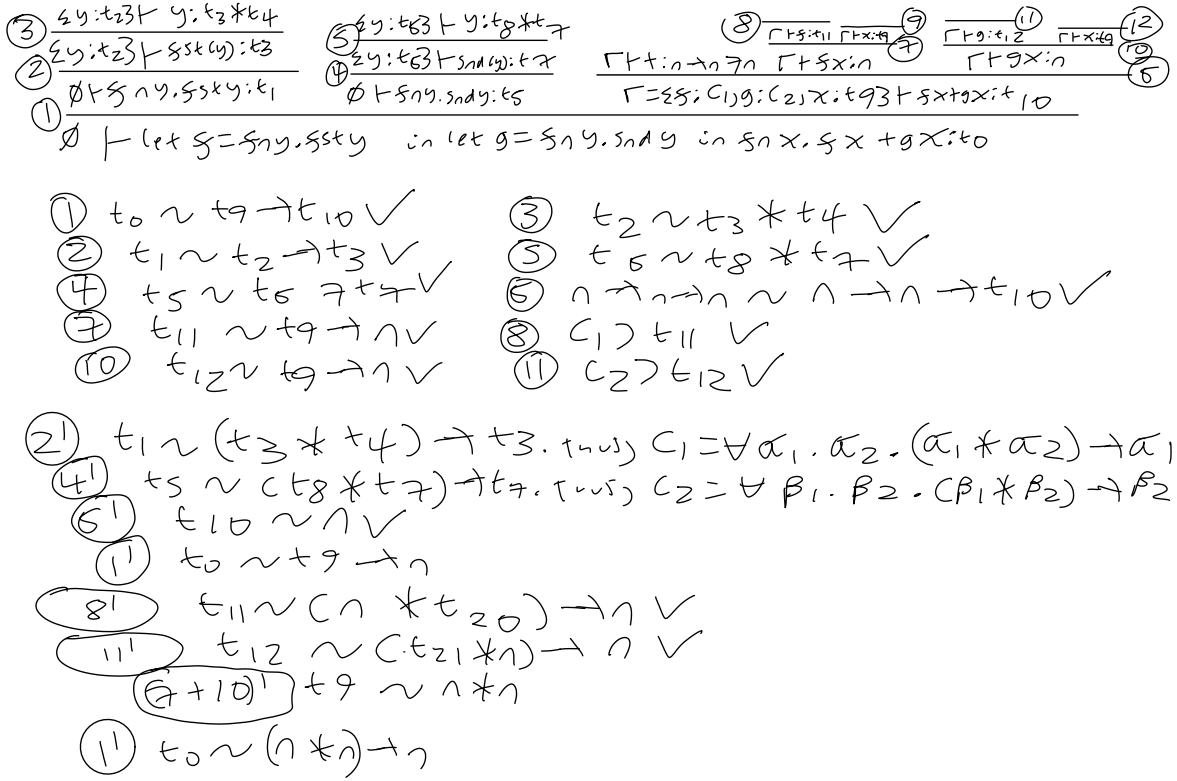
  - Note that the application rule is written in terms of abstraction

Types unify if (this makes it sound complicated but is it actually very intuitive):

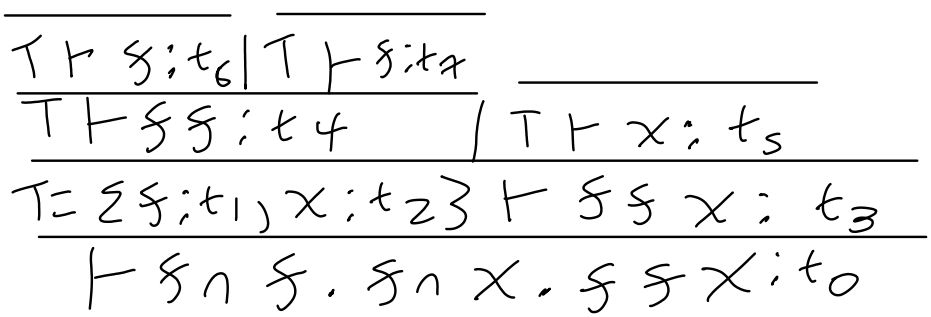
- They are the same base type (e.g. both are int)
- One is a type variable and the other is not a type expression in which that type variable occurs
- They are the same type constructor and the sub expressions unify (e.g. are list τ_1 and list τ_2 and $\tau_1 \sim \tau_2$)

If types cannot be unified, the expression is illegal so a type error is raised

Example of legal expression:



Example of un-typeable expression:



① $t_0 \sim t_1 \to t_2 \to t_3$

② $t_4 \sim t_5 \to t_3$ ✓

③ $t_6 \sim t_7 \to t_4$ ✓

④ $t_5 \sim t_2$ ✓

⑤ $t_6 \sim t_1$ ✓

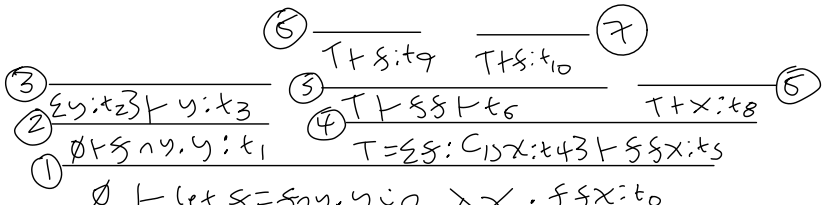
⑥ $t_7 \sim t_1$ ✓

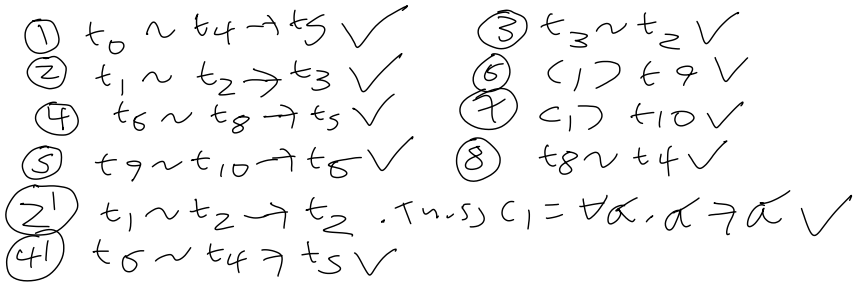
②′ $t_4 \sim t_2 \to t_3$

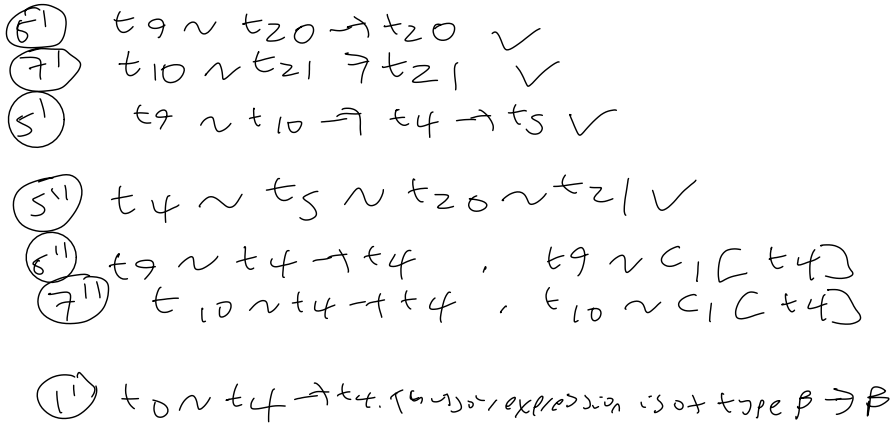
③′ $t_1 \sim t_1 \to t_4$ ✗ Illegal type

This expression is actually well-formed but can only be assigned a recursive type (which this system isn't powerful enough to do)

However we have enough polymorphism to be able to handle this if we already know the general definition of f:







Note that each invocation of a polymorphic function can instantiate it with fresh type variables

5 Quantified Types (PLC and SLS)

5.1 PLC

- Although MiniML implicitly had polymorphism, polymorphic λ-calculus (PLC) has polymorphic types as a part of the language itself, however this means that we revert to explicit typing
  - *As type checking now requires checking for a tautology in first-order logic instead of propositional logic, it is now undecidable*
- **A capital lambda (Λ) denotes a type-level function as opposed to a value-level function (λ)** — these generic expressions genuinely have a universally quantified type
- A generic expression e must be instantiated with a particular type expression t ($e[t]$) before it can be used

- $$\frac{\Gamma \vdash e : \forall V.\, t_2}{\Gamma \vdash e[t_1] : t_2[V \ll t_1]}$$

- $$\frac{\Gamma, V \vdash e : t}{\Gamma \vdash \Lambda V.\, e : \forall V.\, t}$$

Note that the first rule resembles the one for (value-level) application and the second rule resembles the one for (value-level) abstraction. Note that we are now allowing lone type variables to be members of our type environments. Example: bool $= \forall\alpha.\alpha \mapsto (\alpha \mapsto \alpha)$ and False $= \Lambda\alpha.\lambda x_1 : \alpha.\lambda x_2 : \alpha.x_2$. Show that False: bool.

$$\dfrac{\dfrac{\dfrac{}{\alpha, x_1 : \alpha, x_2 : \alpha \vdash x_2 : \alpha}}{\alpha \vdash (\lambda x_1 : \alpha.\, x_2 : \alpha.\, x_2) : \alpha \to \alpha \to \alpha}}{\vdash (\Lambda\alpha.\, \lambda x_1 : \alpha.\, \lambda x_2 : \alpha.\, x_2) : \forall\alpha.\alpha \to \alpha \to \alpha \sim \text{bool}}$$

5.2 Subtyping

- $S <: T = S$ is a subtype of T
  - This is actually reflexive but this is the standard notation
- T occurs co-variantly in $F[T]$ iff $(F[T_1] <: F[T_2]$ iff $T_1 <: T_2)$ — *producers (i.e. writing data i.e. function return values) can produce more than is needed*
- T occurs contra-variantly in $F[T]$ iff $(F[T_1] <: F[T_2]$ iff $T_1 >: T_2)$ — *consumers (i.e. reading data i.e. function parameters) can consume less than they are given*
- T occurs in-variantly in $F[T]$ iff $(F[T_1] <: F[T_2]$ iff $T_1 = T_2)$ — *mutable data can be written and read so must be co- and contra-variant so as the types must obey the ordering relation in both directions they must be equal*
- *PECS: Producer Extends, Consumer Super*

- $\dfrac{E \vdash e : S \qquad E \vdash S <: T}{E \vdash e : T}$ subsumption

- $\dfrac{E \vdash T_1 <: S_1 \qquad E \vdash S_2 <: T_2}{E \vdash S_1 \to S_2 <: T_1 \to T_2}$

- $\dfrac{E \vdash e_1 : T_1 \to T_2 \qquad E \vdash e_2 : U \qquad E \vdash U <: T_1}{E \vdash e_1\, e_2 : T_2}$

- The first rule encodes the Liskov substitution principle (behavioural sub-typing): If S is a subtype of T, then an expression of type T may be replaced with an expression of type S without altering the semantic properties of the program

- The second rule encodes the fact that function arguments are contravariant and function return values are covariant
- The third rule is a corollary of the first two but corresponds to the fact that it is advisable to consider sub-typing lazily (most rules are guaranteed to make progress as they break down the syntax of terms whereas subsumption can in principle be applied infinitely many times)

5.3 Records

- A record associates expressions with labelled fields
- A particular field of a record is accessed using @

$$\bullet\ \frac{E \vdash e_1 : T_1, \ldots, e_n : T_n}{E \vdash \{\ell_1 = e_1, \ldots, \ell_n = e_n\} : \{\ell_1 : T_1, \ldots, \ell_n : T_n\}}$$

$$\bullet\ \frac{E \vdash e : \{\ell_i : T_1, \ldots, \ell_n : T_n\}}{E \vdash e @ \ell_i : T_i}$$

- Note that the type of a record is itself a record of sorts
- **A record is a subtype of another iff its labels are a superset of the superrecord and those labels have types which are subtypes of those of the corresponding labels in the superrecord**

5.4 SLS

- The Simple Language with Subtypes (SLS) extends PLC with bounded polymorphism and existential quantification
- Previously we only had unbounded polymorphism but we can use the sub-typing relation $<:$ as a part of expressions in order to place upper or lower bounds

5.5 Abstract data types (ADTs)

- Existentially quantified types allow us to type interfaces in terms of a hidden internal representation
  - top (the super-type of every well-formed type) = $\exists a.a$ — any type can be represented as a for some a
  - Only pairs obey $\exists a.\exists b.a * b$

- **An ADT is declared using a let declaration and the pack keyword, and instantiated using the new keyword**
- An ADT only allows the underlying data to be manipulated through the functions declared by the ADT instead of those of the wrapped type (as the wrapped type is not exposed)

$$\frac{\Gamma \vdash M : t_2[\alpha \leftarrow t_1]}{\Gamma \vdash (p = \text{pack}[\alpha = t_1 \text{ in } t_2] \text{ with } M) : \{p : \exists \alpha . t_2\}}$$

$$\frac{\Gamma \vdash p : \exists \alpha . t_2}{\Gamma \vdash (x, pt = \text{new } p) : \Gamma[x : t_2[\alpha \leftarrow pt]]}$$

- **The pack[...] provides signatures of methods and the with ... provides implementations of methods**
- **The ... = ... at the beginning of the pack provides a type for which the only value occupying is "self"/"this" for the use of the in ...**
- **Note that even if the same ADT is constructed twice with the same internal representation, the returned "objects" will have different types** $(\tau_1 \neq \tau_2)$ — this is the only way in SLS of enforcing the fact that the "methods" of each instance are a collection of functions to be applied only to that instance

Example:

```
let point = [pack pt <: {x: num, y: num, active: bool} =
{x:num, y:num, active: bool} in
{create : num -> num -> pt, move : pt -> num -> num -> pt,
switch : pt -> pt] with
{create = \a:num. \b:num. {x=a, y=b, active=true},
switch = \p.pt. if p@active=true then {x=p@x, y=p@x, false} else
  {x=p@x, y=p@x, true}
};
p, T = new point;
p1 = p@create 4 5;
p2 = p@switch;
```

Note, as we do not have mutability, every updating operation returns a new "object" (of the same "self" type). **The use of (bounded) sub-typing with the existential quantification is only necessary to provide "getter"s**.

5.6 Sub-typing relationships of bounded quantified types

$$\frac{\Gamma \vdash \alpha_1 <: \alpha_2 \qquad \Gamma, t <: \alpha_3 \vdash \beta_1 <: \beta_2}{\Gamma \vdash (\exists t <: \alpha_1 . \beta_1) <: (\exists t <: \alpha_2 . \beta_2)}$$

$$\frac{\Gamma \vdash \alpha_2 <: \alpha_1 \qquad \Gamma, t <: \alpha_3 \vdash \beta_1 <: \beta_2}{\Gamma \vdash (\forall t <: \alpha_1 . \beta_1) <: (\forall t <: \alpha_2 . \beta_2)}$$

That is that **existential quantification is co-variant in its type parameters whereas universal quantification is contra-variant in its type parameters**. It is intuitive that an ADT which exposes a superset of getters will be a safe substitution. The contra-variance is less intuitive a priori but consider the fact that all meaningful (ok yes, this is hand-wavy) universally quantified types are functions which take parameters of types of the type parameters and that functions are contravariant in their arguments.