Higher-order unification.

Examples of higher-order programs: map, function lists, lazy lists, etc.

Encoding structures with binders.
Higher-order unification

- The *higher-order unification problem* is the problem of finding a substitution $\theta$, given $\lambda$ terms $s$ and $t$, such that $s\theta =_{\beta\eta} t\theta$. It is [undecidable](https://en.wikipedia.org/wiki/Unification_problem).
- M.g.u doesn’t always exist even when the unification problem is solvable.
- Example: Let $f$ and $M$ be a function symbol and a variable of type $\alpha \rightarrow \alpha$. Consider the following unification problem:

$$\lambda x. M \,(f\,x) = ? \lambda x. f \,(M\,x).$$

[$f^k x$ means $k$ applications of $f$ to $x$.]

Then any substitution of the form

$$\theta_k = [M \mapsto \lambda x. f^k x], \quad k \geq 0,$$

is a unifier, and it is not an instance of any other unifier.
Complete set of unifiers

Let $S$ be a unification problem. Then a complete set of unifiers (CSU) for $S$ is a set of substitutions $\mathcal{U}$ such that every $\sigma \in \mathcal{U}$ is a unifier for $S$ and such that for every unifier $\theta$ of $S$, there exists a unifier $\theta' \in \mathcal{U}$ and a substitution $\rho$ such that

$$\theta = \theta' \circ \rho.$$

For example, for the unification problem in the previous slide:

$$\lambda x. M (f \, x) =? \lambda x. f (M \, x).$$

the set

$$\mathcal{U} = \{\theta_k \mid k \geq 0\},$$

where $\theta_k = [M \mapsto \lambda x. f^k x]$, is a CSU for the above unification problem.
Some terminology

- Let \( t \) be the \( \lambda \)-normal term:

\[
\lambda x_1 \cdots \lambda x_m . u \ t_1 \cdots t_n
\]

where \( u \) is a variable or a constant.

The term \( u \) is called the **head of** \( t \); \( t \) is a **rigid** term if \( u \) is a constant or in \( \{x_1, \ldots, x_m\} \), otherwise it is a **flexible** term.

- A unification problem is \( s = ? t \) is a **rigid-rigid** problem if both \( s \) and \( t \) are rigid terms; it is a **flex-rigid** if one of \( s \) and \( t \) is a flexible term and the other is a rigid term; it is a **flex-flex** problem if both terms are flexible.
Huet’s unification algorithm

It is a semi-decision procedure for generating complete set of unifiers. It consists of iterations of two simplication steps:

- **The SIMPL procedure**: it simplifies the rigid-rigid problems. Given a unification problem $S$, it either returns a new unification problem containing only flex-rigid/flex-flex pairs, or terminates with failure.

- **The MATCH procedure**: it computes a set of substitutions that may form “initial segments” of unifiers. It consists of an *imitation step* and a *projection step* for simplifying flex-rigid problems. This step generates a tree of possible simplifications.

Huet’s algorithm does not deal with flex-flex case, so it is really a *pre-unification* algorithm (but one could get lucky and solves the problem before reaching flex-flex cases).
The SIMPL procedure

Let $S$ be the input unification problem. SIMPL($S$) =

- If $S$ is empty then return *Success*.
- If there is $(s = ? t) \in S$ such that
  \[
  s = \lambda x_1 \cdots \lambda x_n. u \ s_1 \cdots s_k \quad t = \lambda x_1 \cdots \lambda x_n. v \ t_1 \cdots t_l
  \]
  and $u \neq v$ then return *Fail*.
- Otherwise, let $S'$ be the union of $S$ and
  \[
  \{ \lambda x_1 \cdots \lambda x_n.s_1 = ? \lambda x_1 \cdots \lambda x_n.t_1, \ldots, \lambda x_1 \cdots \lambda x_n.s_1 = ? \lambda x_1 \cdots \lambda x_n.t_1 \}
  \]
  Return SIMPL($S'$).
- If there are no rigid-rigid pairs left, reorient any remaining flex-rigid pairs:
  Return
  \[
  \{ t = ? s \mid (s = ? t) \in S \text{ rigid-flex} \}
  \cup \{ s = ? t \mid (s = ? t) \in S \text{ flex-rigid or flex-flex} \}.
  \]
Let $F$ be a variable of type $\sigma_1 \to \cdots \to \sigma_k \to \tau$, where $\tau$ is a base type. Suppose we are given a flex-rigid pair $s = ? t$, where

$$s = (\lambda x_1 \cdots \lambda x_m. F \ s_1 \cdots s_k) \text{ and } t = (\lambda x_1 \cdots \lambda x_m. c \ t_1 \cdots t_n),$$

and a set of variables $V$ (containing the free variables of $s$ and $t$):

If $c \in \{x_1, \ldots, x_m\}$ then $IMIT(s, t, V) = \emptyset$. Otherwise,

$$IMIT(s, t, V) = \{ [F \mapsto \lambda y_1 \cdots \lambda y_k. (c \ H_1 \ y_1 \cdots y_k) \cdots (H_n \ y_1 \cdots y_k)) ] \}$$

where $H_1, \ldots, H_n$ are new variables distinct from $V, x_1, \ldots, x_m$ and $y_1, \ldots, y_k$, of the appropriate types.
The MATCH procedure: Projection

Let $F$ be a variable of type $\sigma_1 \rightarrow \cdots \rightarrow \sigma_k \rightarrow \tau$, where $\tau$ is a base type. Suppose we are given a flex-rigid pair $s = ? t$, where

$$s = (\lambda x_1 \cdots \lambda x_m. F \ s_1 \cdots s_k) \text{ and } t = (\lambda x_1 \cdots \lambda x_m. c \ t_1 \cdots t_n),$$

and a set of variables $V$ (containing the free variables of $s$ and $t$):

If $\sigma_i$, where $1 \leq i \leq k$, is of the form $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau$, then

$$\text{PROJ}_i(s, t, V) = \{[F \mapsto \lambda y_1 \cdots \lambda y_k.(y_i \ (H_1 y_1 \cdots y_k) \cdots (H_n y_1 \cdots y_k))]\}$$

were $H_1, \ldots, H_n$ are new variables. Otherwise, $\text{PROJ}_i(s, t, V) = \emptyset$. Let

$$\text{PROJ}(s, t, V) = \bigcup \{\text{PROJ}_i(s, t, V) \mid 1 \leq i \leq k\}.$$

Then define:

$$\text{MATCH}(s, t, V) = \text{IMIT}(s, t, V) \cup \text{PROJ}(s, t, V)$$
Constructing Match trees

The main unification algorithm is just an exhaustive search, represented as a search tree below: Nodes are either a Success node, a Failure node, or a unification problem $S$.

Successors of a node $S$:

- A Success or Failure node has no successor.
- If $S$ has a rigid-rigid or rigid-flex pair, then $SIMPL(S)$ is the successor of $S$.
- Otherwise, if $S$ has a flex-rigid pair, say $s \equiv^? t$, then $S\theta$ is a successor of $S$, for any $\theta \in MATCH(s, t, FV(s, t))$. 
Example: higher-order unification

Use Huet’s algorithm to solve the following problems:

- $S_1 = \{\lambda x. \lambda y. \, g \, x \, (\lambda z. y) =^? \lambda x. \lambda y. \, g \, y \, (\lambda z. y)\}.$
- $S_2 = \{\lambda x. M =^? \lambda x. x\}.$
- $S_3 = \{F \, a =^? a\}.$
- $S_4 = \{\lambda x. M \, (f \, x) =^? \lambda x. f \, (M \, x)\}.$
- $S_5 = \{\lambda f \, \lambda x. M \, (f \, x) =^? \lambda f \, \lambda x. f \, (M \, x)\}.$
A decidable fragment

- A λ normal term \( t \) is a higher-order pattern term if for every free variable \( M \), every occurrence of \( M \) in \( t \) is applied to distinct bound variables in \( t \).

- Some examples:
  
  \[
  (\lambda x \lambda y. M \times y) \quad f (\lambda x. N \times)(\lambda y.g \ y) \quad \lambda x \lambda f.f \ (f \ x)
  \]

  and some non-examples:

  \[
  (\lambda x. M \times x) \quad (\lambda x \lambda y. M \ (\lambda x.x) N)
  \]

- A unification problem \( S \) is a higher-order pattern unification problem if for every \( s \ =^? \ t \in S \), \( s \) and \( t \) are higher-order pattern terms.

- Higher-order pattern unification is decidable [Miller 91] and an m.g.u. exists if the unification problem is solvable.
Higher-order pattern unification

- Many theorem proving problems in higher-order logic involve only higher-order pattern unification.
- Higher-order pattern unification can be solved by simply applying the SIMPL and MATCH procedures repeatedly.
- In the MATCH procedure, only one of imitation and projection steps needs to be applied, e.g., in the flex-rigid problem
  \[(\lambda x \lambda y. M \times y) =? (\lambda x \lambda y. f \times (g \ y))\]
  only the imitation step will lead to a solution.
- Similarly, in
  \[(\lambda x \lambda f. M \times f) =? (\lambda x \lambda f. f \times (f \ x))\]
  only the projection step needs to be applied (i.e., projecting \( M \) on its second argument).
Higher-order features of λProlog

- The most recent implementation, the Teyjus system, implements only the higher-order pattern unification, not the full unification.
- Non-higher-order-pattern unification are handled partially, with the remaining flex-rigid and flex-flex pairs returned, e.g.,

\[ ?- (x \backslash f (M (f x))) = (x \backslash f x). \]

The answer substitution:
M = M

The remaining disagreement pairs list:
<M (f #1), #1>

- “Flexible goals” are not handled, e.g., a goal like \( \exists X. X \) will generate runtime error, although syntactically it is allowed.
Example: predicate quantification

- Given a list $L$, a list $R$ and a binary predicate $R$, check that the $i$-th element of $L$ is related by $R$ to the $i$-th element of $R$, for every $i$:
  
  type \text{mappred} \ (A \rightarrow B \rightarrow o) \rightarrow \text{(list } A) \rightarrow \text{(list } B) \rightarrow o.

  m\text{appred } P \ \text{nil \ nil}.
  m\text{appred } P \ (X :: L) \ (Y :: K) :- P \ X \ Y, m\text{appred } P \ L \ K.

- Given a list $L$ and a unary predicate $P$, check that $P$ holds for every element of $L$:
  
  type \text{forevery} \ (A \rightarrow o) \rightarrow \text{list } A \rightarrow o.

  \text{forevery } P \ \text{nil}.
  \text{forevery } P \ (X :: L) :- P \ X, \text{forevery } P \ L.
Example: function quantification

Given a list $L$ and a one-argument function $F$, apply $F$ to every element of $L$:

type mapfun (A -> B) -> list A -> list B -> o.

mapfun F nil nil.
mapfun F (X :: L) ((F X) :: K) :- mapfun F L K.

Note that since $F$ is just a simply typed term, applying $F$ to $X$ is just $\beta$-reduction. More complicated function evaluation mechanisms have to be encoded explicitly.

?- mapfun (x\ x + 1) [1,2,3] L.

The answer substitution:
$L = 1 + 1 :: 2 + 1 :: 3 + 1 :: \text{nil}$

?- mapfun (x\ x + 1) [1,2,3] L, mappred (x\ y\ y is x) L R.

The answer substitution:
$R = 2 :: 3 :: 4 :: \text{nil}$
$L = 1 + 1 :: 2 + 1 :: 3 + 1 :: \text{nil}$
Example: function-lists

- This is similar to difference-list, but instead of attaching a logic variable to the “tail” of a list, we use \( \lambda \)-abstraction.
- The list \([a_1, \ldots, a_n]\) is represented as the function
  \[
  \lambda x.(a_1 :: \cdot :: a_n :: x).
  \]
  Thus a function list is of type: \((\text{list } A \rightarrow \text{list } A)\).
- The empty list is represented as the identity function \(\lambda x.x\).
- Appending function-lists via \(\beta\)-reduction:
  \[
  \text{type } fappend \ (\text{list } A \rightarrow \text{list } A) \rightarrow \ (	ext{list } A \rightarrow \text{list } A) \rightarrow (\text{list } A \rightarrow \text{list } A) \rightarrow o.
  \]
  \[
  fappend \ L \ R \ (z\L (R \ z)).
  \]
Example: recursion over function lists

- Check whether a function list is well-formed:
  \[
  \text{isflist} \ (x \ x).
  \text{isflist} \ (x \ A :: \ (L \ x)) :- \text{isflist} \ L.
  \]

- Reversing a function list:
  \[
  \text{frev} \ (z \ z) \ (z \ z).
  \text{frev} \ (z \ A :: \ (L \ z)) \ (z \ RL \ (A :: \ z)) :- \text{frev} \ L \ RL.
  \]

- Converting a list to a function list:
  \[
  \text{list2flist} \ \text{nil} \ (z \ z).
  \text{list2flist} \ (A :: \ L) \ (z \ A :: \ (FL \ z)) :- \text{list2flist} \ L \ FL.
  \]
Example: encoding streams

- In lazy functional languages, one can represent a (possibly infinite) list such that the elements of the list are constructed only when needed.

- For example, the sequence of natural numbers can be generated in a typical lazy functional language via the function:
  \[ f(n) = (n :: f(n+1)) \]
  That is, as the list generated by executing \( f(0) \). The function call in the tail of the list \( f(n+1) \) is suspended until the tail is queried.

- In an eager language, calling \( f(0) \) will lead infinite loop.

- We shall see a \( \lambda \)Prolog version of lazy lists; encoded via suspended higher-order predicates.
Example: data structures for streams

- Cells: containing either a value (a **forced cell**) or a suspended computation (a **delayed cell**).

  \[
  \text{kind cell type} \rightarrow \text{type.} \\
  \text{type fcell } A \rightarrow (\text{cell } A). \quad \% \text{ forced cells} \\
  \text{type dcell } (A \rightarrow o) \rightarrow (\text{cell } A). \quad \% \text{ delayed cells}
  \]

  The delayed cell takes a higher-order unary predicate.

- Streams:

  \[
  \text{kind strm type} \rightarrow \text{type.} \\
  \text{type empty } (\text{strm } A). \\
  \text{type stream } A \rightarrow (\text{cell } (\text{strm } A)) \rightarrow (\text{strm } A).
  \]

  The constructor **stream** constructs a stream from an element and a cell containing a stream.
Example: operations on streams

- Taking the head and the tail of a stream:
  \[
  \text{head} (\text{stream } V S) V.
  \text{tail} (\text{stream } V S) T :\text{-} \text{getcell} S T.
  \]

  \[
  \text{getcell} (\text{fcell } V) V.
  \text{getcell} (\text{dcell } P) V :\text{-} P V.
  \]

  \text{getcell} “executes” the suspended predicate in the delayed cell, and passes its output to \( V \).

- The ‘list-like’ interface to lazy streams:
  \[
  \text{snil} \text{ empty}.
  \text{scons} X P (\text{stream } X (\text{dcell } P)).
  \]
Example: some simple streams

- **Natural numbers, even and odd numbers:**

  ```prolog
  % A sequence with I increment.
  inc X I T :- Y is X + I, scons X (S \ inc Y I S) T.
  nat S :- inc 0 1 S.
  even S :- inc 0 2 S.
  odd S :- inc 1 2 S.
  ```

- **Fibonacci sequence:**

  ```prolog
  fb X Y S :-
      Z is X + Y, scons Z (T \ fb Y Z T) S.
  fib X Y (stream X (fcell (stream Y (fcell T)))) :-
      fb X Y T.
  ```
Example: querying streams

- **Merging two streams:**
  
  
  ```prolog
  merge empty S S.
  merge S empty S.
  merge (stream A S) (stream B T) (stream A (fcell W)) :-
    tail (stream A S) S', tail (stream B T) T',
    scons B (Z\ merge S' T' Z) W.
  ```

- **Querying a finite segment of a stream:**
  
  ```prolog
  take 0 S nil.
  take N S (V :: L) :-
    N > 0, M is N - 1, head S V, tail S T, take M T L.
  ```
Encoding structures with binders

Recall the language of untyped $\lambda$-terms:

$$s, t ::= x \mid \lambda x.t \mid (s \ t)$$

A direct encoding would represent variables explicitly, e.g., as a string. For example, consider the following constructors:

$$\text{var} : \text{string} \rightarrow \text{tm} \quad \text{app} : \text{tm} \rightarrow \text{tm} \rightarrow \text{tm} \quad \text{abs} : \text{string} \rightarrow \text{tm} \rightarrow \text{tm}.$$  

Thus, the term $\lambda x\lambda f.(f \ x)$ would be encoded as:

$$\text{abs} "x" (\text{abs} "f" (\text{app} (\text{var} "f")) (\text{var} "x")).$$

But with this encoding, $\alpha$-equivalence needs to be defined explicitly as well, since $\lambda x.x$ and $\lambda y.y$ have different encodings:

$$\text{abs} "x" (\text{var} "x") \text{ and } \text{abs} "y" (\text{var} "y").$$
A different style of encoding uses $\lambda$-tree syntax.

Variables are not explicitly encoded via constructors. Abstraction in the object terms is encoded as meta-level abstraction: $\lambda$-calculus).

\[
app : tm \to tm \to tm \quad abs : (tm \to tm) \to tm.
\]

We get two things for free:

- $\alpha$-equivalence in the object syntax co-incides with $\alpha$-equivalence in the meta-level. For example, the encodings of $\lambda x.x$ and $\lambda y.y$ are

\[
(abs (\lambda x.x)) \quad \text{and} \quad (abs (\lambda y.y))
\]

which are $\alpha$-equivalent terms at the meta level.

- Capture-avoiding substitution in the object level is captured by $\beta$-reduction at the meta level. For example, if $t = (\lambda x.s)$ then to substitute $x$ with $u$, we simply use

\[
(\lambda x.s) \ u.
\]
Example: encoding first-order formulae

Consider encoding formulae of classical first-order logic:

kind form type. % Syntactic category for formulae.
kind tm type. % Syntactic category for (untyped) terms.

type const string -> tm. % constants

type atm string -> (list tm) -> form. % atomic formulae
type and form -> form -> form. % conjunction
type or form -> form -> form. % disjunction
type imp form -> form -> form. % implication
type all (tm -> form) -> form. % universal quantifier
type some (tm -> form) -> form. % existential quantifier

% Some operators can be written with infix notation
infixr and 120.
infixr or 120.
infixr imp 120.
Example: prenex normal form

Every first-order classical formula can be transformed into an equivalent formula in prenex normal form (pnf), i.e., formulae of the form

$$Q_1x_1 \cdots Q_nx_n F$$

where each $Q_i$ is either a $\forall$ or an $\exists$ and $F$ is a quantifier free formula.

The transformation uses the following logical equivalence (where $Q$ can be $\forall$ or $\exists$):

$$(Qx.G) \lor F \equiv F \lor (Qx.G) \equiv Qx.(F \lor G)$$

$$(Qx.G) \land F \equiv F \land (Qx.G) \equiv Qx.(F \land G)$$

$$F \Rightarrow (\forall x.G) \equiv \forall x.(F \Rightarrow G)$$

$$(\forall y.F) \Rightarrow G \equiv \exists y.(F \Rightarrow G)$$

$$F \Rightarrow (\exists x.G) \equiv \exists x.(F \Rightarrow G)$$

$$(\exists y.F) \Rightarrow G \equiv \forall y.(F \Rightarrow G)$$

In all of the above: $x$ is not free in $F$ and $y$ is not free in $G$. 
Example: prenex normal form (2)

\[ \text{pnf} \ (\text{atm} \ x \ L) \ (\text{atm} \ x \ L) . \]
\[ \text{pnf} \ (A \ \text{and} \ B) \ C \ :- \]
\[ \quad \text{pnf} \ A \ F , \ \text{pnf} \ B \ G , \ \text{merge} \ (F \ \text{and} \ G) \ C . \]
\[ \text{pnf} \ (A \ \text{or} \ B) \ C \ :- \]
\[ \quad \text{pnf} \ A \ F , \ \text{pnf} \ B \ G , \ \text{merge} \ (F \ \text{or} \ G) \ C . \]
\[ \ldots \]

\[ \text{merge} \ ((\text{all} \ x \ \text{\backslash} \ B \ x) \ \text{or} \ C) \ (\text{all} \ D) \ :- \]
\[ \quad \text{pi} \ x \ \text{\backslash} \ \text{merge} \ ((B \ x) \ \text{or} \ C) \ (D \ x) . \]
\[ \text{merge} \ ((\text{some} \ x \ \text{\backslash} \ B \ x) \ \text{or} \ C) \ (\text{some} \ D) \ :- \]
\[ \quad \text{pi} \ x \ \text{\backslash} \ \text{merge} \ ((B \ x) \ \text{or} \ C) \ (D \ x) . \]
\[ \ldots \]

The side condition that “\(x\) is not free in \(C\)” is satisfied trivially: \(C\) is not in the scope of \(x\) and substitution is capture-avoiding.