9.2.1.

From type inference to type verification in logic programming
A SIMPLE DOMAIN OF TYPES FOR LOGIC PROGRAMS

[Coddish & Lagoon, TCS, 2000]

- Concrete terms are abstracted to type terms

- Concrete terms are made out of:
  - Numeric constants
  - Variables (capital letters)
  - Lists: [] , [t1, t2]
  - Trees: void, tree (t1, t2, t3)
  (Trees are representative of generic recursive type constructors)

- Type terms are (associative, commutative, idempotent) terms built using
  - A binary set constructor +
  - A collection of monomorphic and polymorphic description symbols
    - Monomorphic symbols: num, nil, void
    - Polymorphic symbols: list(-), tree(-)

- The "abstraction function" \( T: \) concrete term -> type term
  (to be extended to the real world \( \tilde{T}: \) concrete term -> type term)

\[
T(t) = \begin{cases} 
  x & \text{if } t \text{ is the variable } x \\
  \text{num} & \text{if } t \text{ is a number} \\
  \text{nil} & \text{if } t = [] \\
  \text{list}(T(t_1), T(t_2)) & \text{if } t = [t_1, t_2] \\
  \text{void} & \text{if } t = \text{void} \\
  \text{tree}(T(t_1), T(t_2), T(t_3)) & \text{if } t = \text{tree}(t_1, t_2, t_3)
\end{cases}
\]

- Partial order is essentially set inclusion
Examples of type abstractions

\[ \Upsilon([-3,0,7]) = \text{list}(\Upsilon(-3)) + \Upsilon([0,7]) = \text{list(num)} + \text{list}(\Upsilon(0)) + \Upsilon([7]) = \text{list(num)} + \text{list(num)} + \text{nil} = \text{list(num)} + \text{nil} \]

\[ \Upsilon([-x,y]) = \text{list}(x) + \text{list}(y) + \text{nil} \]

\[ \Upsilon(\text{tree}(2, \text{void}, \text{void})) = \text{tree(num)} + \text{void} \]
The abstract semantic evaluation function for types

\[ \text{Type}^e_{\mu} (\text{Type}) = \lambda p(x). \]

\[ \left\{ \begin{array}{l}
p(\text{Type}) \mu \\
p(\bar{t}) \in p_1(x_1), \ldots, p_n(x_n) \in p \\
T_i \in I(p_i(x_i)) \\
\mu \in \text{CUACI}((\text{Type}^e_{\mu}), \ldots, \text{Type}^e_{\mu+1}), (T_1, \ldots, T_n)) \end{array} \right\} \]

\text{CUACI} \ is \ the \ ACI-unification \ procedure.

Given a specification \( S_p \), we prove correctness criterion is, for each clause \( c \),

\[ \text{Type}^e_{\mu} (S_p) = S_p \]
The specification:

\[ S_{\text{fib}} = \text{fib}(x, y) \mapsto \{ \text{fib}(\text{num}, \text{num}) \} \]

\[ T_{(x, y)}^\wedge (S_{\text{fib}}) = \{ \text{fib}(\text{num}, \text{num}) \} \]

\[ T_{x \in \text{fib}}^\wedge (S_{\text{fib}}) = \{ \text{fib}(\text{num}, \text{num}) \} \]

\[ T_{y \in \text{fib}}^\wedge (S_{\text{fib}}) = \{ \text{fib}(\text{num}, \text{num}), \text{num} \} \]

If we update \( N \) to \( N' \) we succeed in proving the recursion condition.

A "concrete semantics" error often shows up as a type error even within "first-order" types.
Example 2

1. \text{append} ([], Xs, Xs).
2. \text{append} ([X1|Xs], Ys, [X1|Ys]): \text{append} (Xs, Ys, Zs).

The specification:

\[ S_n = \text{append}(X_1, Z_1) \Rightarrow \]
\[ \{ \text{append}(_{X1}, _{me+last(T)}, _{me+last(T)}) \}, \]
\[ \text{append}([_{me+last(T)}, _{me+last(T)}, _{me+last(T)}]) \} \]

\[ T_n \approx (S_n) \not\approx S_n \]

- The variable Xs is not necessarily a list.

\[ T_n \not\approx (S_n) = \bigvee_{p(c, Xs)} \left\{ p(c, \text{last}(T)) \right\} \]

\[ \downarrow \]

\[ \text{append}(_{me}, _{Xs}, _{Xs}) \]

- No clear body.
- No use of the specification.
I/O Correctness on Types

A specification is a pair of "type interpretations"

\[
\left( S^I, S^O \right)
\]

input \uparrow \hspace{1cm} output

- If we expand the "most adequate" fixpoint semantics using the abstraction on types we get the following partial correctness condition.
1. "Unify" the clause head with the input specification

\[ \Theta = \{ \mu \mid A \in S_1^\Theta (p(\vec{x})), \mu \in \text{cvac} (A, p(c(\vec{f}))) \} \]

2. Abstract semantics of the procedure calls

\[ t_i = \begin{cases} S_0^\Theta (p_j(\vec{x}_j)) & \text{if } p_i(\vec{x}_i) \Theta \equiv S_1^\Theta (p_j(\vec{x}_j)) \\ T & \text{otherwise} \end{cases} \]

3. Expanded condition

\[ \{ p(c(\vec{f})) \mu \mid A_j \in T_j, A \in p(\vec{x}) \Theta, \mu \in \text{cvac} (A, A_1, ..., A_n, p(c(\vec{f})), p_j(c(\vec{f})), ..., p_m(c(\vec{f}))) \} \]

\[ \subseteq S_{90}^\Theta (p(c(\vec{x}))) \]

Clause

\[ p(c) \leftarrow p_1(c_{i_1}), ..., p_m(c_{i_m}) \]
1. "unify" the clause head with the input specification
   \[ \Theta = \{ \mu | A \in S^T (p(x)) , \mu \in \text{CUACI} (A, p(T(f))) \} \]

2. abstract semantics of the procedure calls
   (only for those procedure calls which can satisfy the input spec)
   \[ T_j = \begin{cases} 
   S^0_j (p_j(x_j)) & \text{if } p_j(x_j) \Theta \leq S^T_j (p_j(x_j)) \\
   T & \text{otherwise} 
   \end{cases} \]

3. expanded condition
   \[ \{ p(T(f)) \mu | A_j \in T_j , A \in p(x) \Theta, \mu \in \text{CUACI} (A, A_1, ..., A_n), (p(T(f)), p_k(T(f_i)), ...) \} \leq S^0_j (p(x)) \]

The first clause of append can now be proved I/O correct w.r.t. \( \Theta \) on input specification
\[ \{ \text{append} (\text{nil} + \text{list}(x), \text{nil} + \text{list}(y), u) , \text{append} (\text{nil}, \text{nil} + \text{list}(x), u) \} \]
\[ \mu = \{ x \leftarrow \text{list} (\text{nil} + \text{list}(x)) \} \]

clause
\[ p(f) \leftarrow p_a(f_a), ..., p_n (f_n) \]
Specifications are still parts of type preconditions and postconditions.

- The choice of the axiomatic guarantees that preconditions are always satisfied if we prove the sufficient condition.
1. "Unity" The change need with the input spec

$$
\Theta = \{ \mu | A \in S_p^\Xi (p(\bar{x})), \mu \in c\text{VHC } (A, p(YH)) \}
$$

2. "Output" correctness

$$
\{ p (T(\bar{c})) \mu | A_j \in S_p^\varnothing (p_j (\bar{x}_j)), A \in p(\bar{x}) \Theta,
\mu \in c\text{VHC } ( (A, A_2, \ldots, A_m), \)
(pCT(\bar{c})), p_k (T(\bar{c}_k)), \ldots, p_m (T(\bar{c}_m)) \}
\leq S_{p\tilde{\sigma}}^+ (p(\bar{x}) )
$$

3. "Cell" correctness

$$
\{ p_j (T(\bar{c}_j)) \mu | A_k \in S_p^\varnothing (p_k (\bar{x}_k)), A \in p(\bar{x}) \Theta,
\mu \in c\text{VHC } ( (A, A_2, \ldots, A_j-2),
(pCT(\bar{c})), p_k (T(\bar{c}_k)), \ldots, p_{j-2} (T(\bar{c}_{j-2}))) \}
\leq S_{p\tilde{\sigma}}^+ (p_j (\bar{x}_j))
$$

All the procedural cells preceding $p_j (\bar{c}_j)$ are newly the output spec.

Conversion $p (\bar{c}) \rightarrow p_k (\bar{c}_k), \ldots, p_m (\bar{c}_m)$