Programming Languages
Type Reconstruction II
Robert Harper

## Type Reconstruction

A typed term $e$ is a **reconstruction** of an untyped term $u$ iff the **type erasure** $e^-$ of $e$ is $u$.

$$
\begin{array}{rcl}
x^- & = & x \\
n^- & = & n \\
\texttt{true}^- & = & \texttt{true} \\
\texttt{false}^- & = & \texttt{false} \\
o(e_1, \ldots, e_n)^- & = & o(e_1^-, \ldots, e_n^-) \\
\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi}^- & = & \texttt{if}\, e^-\, \texttt{then}\, e_1^-\, \texttt{else}\, e_2^-\, \texttt{fi} \\
\texttt{fun}\, f\, (x{:}\tau_1){:}\tau_2\, \texttt{is}\, e\, \texttt{end}^- & = & \texttt{fun}\, f(x)\, \texttt{is}\, e^-\, \texttt{end}
\end{array}
$$

That is, $e$ is a possible interpretation of $u$ by inserting missing type information.



## Type Reconstruction

For example,

```
fun f(x:int):int is
  if x=0 then 1 else x * f (x-1) fi
end
```

is the unique reconstruction of

```
fun f(x) is
   if x=0 then 1 else x * f (x-1) fi
end
```



## Type Reconstruction

Similarly,

```
fun f(x) is fun g(y) is x+y end end
```

reconstructs uniquely as

```
fun f(x:int):int->int is
  fun g(y:int):int is x+y end
end
```

## Type Reconstruction

The untyped term

```
fun f(x) is f(f(x)) end
```

has **infinitely many** reconstructions of the form

fun f(x:$\tau$):$\tau$ is f(f(x)) end

But the untyped term



```
fun f(x) is
  if x=0 then true else x*f(x-1) fi
end
```

has **no** reconstructions.

## Valid Reconstructions

What are the **valid** reconstructions of $u$?

- Depends on the context $\Gamma$.
- Depends on an analysis of $u$.

Valid reconstructions: $\Gamma \vdash u \rightsquigarrow e : \tau$.

- Read: "$e$ is a valid reconstruction of $u$ with type $\tau$, relative to context $\Gamma$"
- Inductively defined by a set of rules.



## Valid Reconstructions

Variables:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \rightsquigarrow x : \tau}$$

Constants:

$$\frac{}{\Gamma \vdash n \rightsquigarrow n : \texttt{int}}$$

$$\frac{}{\Gamma \vdash \texttt{true} \rightsquigarrow \texttt{true} : \texttt{bool}}$$

$$\frac{}{\Gamma \vdash \texttt{false} \rightsquigarrow \texttt{false} : \texttt{bool}}$$

## Valid Reconstructions

Primitive operations:

$$\frac{\Gamma \vdash u_1 \leadsto e_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash u_n \leadsto e_n : \tau_n}{\Gamma \vdash o(u_1, \ldots, u_n) \leadsto o(e_1, \ldots, e_n) : \tau}$$

where $o$ has args of type $\tau_1, \ldots, \tau_n$ and result of type $\tau$.

Observe that $u_i$ must be reconstructible with type $\tau_i$.



## Valid Reconstructions

Conditionals:

$$\frac{\Gamma \vdash u \leadsto e : \texttt{bool} \quad \Gamma \vdash u_1 \leadsto e_1 : \tau \quad \Gamma \vdash u_2 \leadsto e_2 : \tau}{\Gamma \vdash \mathtt{if}\, u\, \mathtt{then}\, u_1\, \mathtt{else}\, u_2\, \mathtt{fi} \leadsto \mathtt{if}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2\, \mathtt{fi} : \tau}$$

Observe:

- $u$ must be reconstructible with type `bool`.
- $u_1$ and $u_2$ must be reconstructible with the same type (unspecified).



## Valid Reconstructions

Functions:

$$\frac{\Gamma[f{:}\tau_1{\to}\tau_2][x{:}\tau_1] \vdash u \leadsto e : \tau_2}{\Gamma \vdash \mathtt{fun}\, f\,(x)\, \mathtt{is}\, u\, \mathtt{end} \leadsto \mathtt{fun}\, f\ (x{:}\tau_1){:}\tau_2\, \mathtt{is}\, e\, \mathtt{end} : \tau_1{\to}\tau_2}$$

Observe:

- We "guess" the argument and result types.
- We ensure that $u$ is reconstructible with the "guessed" result type, given the "guessed" argument type.



## Valid Reconstructions

Applications:

$$\frac{\Gamma \vdash u_1 \leadsto e_1 : \tau_2{\to}\tau \qquad \Gamma \vdash u_2 \leadsto e_2 : \tau_2}{\Gamma \vdash \mathtt{apply}(u_1, u_2) \leadsto \mathtt{apply}(e_1, e_2) : \tau}$$

We "guess" the argument type, and reconstruct $u_1$ and $u_2$ consistently with it.

## Indeterminacy

The reconstruction relation is **not** a function of $\Gamma$ and $u$!

- There may be **no** valid reconstructions.
- There may be a **unique** reconstruction.
- There may be **infinitely many** reconstructions.

Sometimes such relations are said to be **non-deterministic** or **indeterminate** . . . but this is needlessly wordy.



## Indeterminacy

The metaphor of "guessing" is a reflection of the fact that reconstruction is a proper relation, instead of a function.

- $\Gamma \vdash u \leadsto e : \tau$ is true **whenever** $e$ is a valid reconstruction of $u$ relative to $\Gamma$.
- There may be **many** choices of $e$! "Guessing" just means "there exists a suitable choice".



## Properties of Reconstruction

**Theorem 1 (Soundness)**
*The elaboration relation is* **sound**: *if* $\Gamma \vdash u \leadsto e : \tau$, *then* $\Gamma \vdash e : \tau$ *and* $e^- = u$.

In words, the reconstructions given by the rules are indeed valid reconstructions. That is, **only** valid reconstructions are derivable.

Proof is by induction on the definition of the reconstruction relation.



## Soundness Proof

Suppose that

$$\Gamma \vdash \mathtt{fun}\, f(x)\, \mathtt{is}\, u\, \mathtt{end} \leadsto \mathtt{fun}\, f\ (x{:}\tau_1){:}\tau_2\, \mathtt{is}\, e\, \mathtt{end} : \tau_1 \to \tau_2,$$

where

$$\Gamma[f{:}\tau_1 \to \tau_2][x{:}\tau_1] \vdash u \leadsto e : \tau_2.$$

By induction we know that

$$\Gamma[f{:}\tau_1 \to \tau_2][x{:}\tau_1] \vdash e : \tau_2.$$

Therefore

$$\Gamma \vdash \mathtt{fun}\, f\ (x{:}\tau_1){:}\tau_2\, \mathtt{is}\, e\, \mathtt{end} : \tau_1 \to \tau_2,$$

as required.

## Completeness

**Theorem 2 (Completeness)**
*The elaboration relation is* **complete**: *if* $\Gamma \vdash e : \tau$ *and* $e^- = u$, *then* $\Gamma \vdash u \leadsto e : \tau$.

In words every valid reconstruction is derivable; nothing is omitted.

Proof is by induction on the MinML typing relation.



## Completeness Proof

Suppose that

$$\Gamma \vdash \mathbf{fun}\, f\, (x{:}\tau_1){:}\tau_2\, \mathbf{is}\, e_2\, \mathbf{end} : \tau_1{\rightarrow}\tau_2,$$

by

$$\Gamma[f{:}\tau_1{\rightarrow}\tau_2][x{:}\tau_1] \vdash e_2 : \tau_2,$$

and that

$$u = \mathbf{fun}\, f\, (x{:}\tau_1){:}\tau_2\, \mathbf{is}\, e_2\, \mathbf{end}^-.$$

By the definition of the erasure function,

$$u = \mathbf{fun}\, f\, (x)\, \mathbf{is}\, u_2\, \mathbf{end},$$

where $u_2 = e_2^-$.

It suffices to show that

$$\Gamma[f{:}\tau][x{:}\tau_1] \vdash u_2 \leadsto e_2 : \tau_2,$$

But this follows from the inductive hypothesis, since

$$\Gamma[f{:}\tau][x{:}\tau_1] \vdash e_2 : \tau_2.$$



## Constraint Generation

The reconstruction relation is simple and concise . . . but how do we implement it?

- How do we find $e$ and $\tau$ such that $\Gamma \vdash u \leadsto e : \tau$?
- How do we "determinize" the non-deterministic choices in the rules?

But keep in mind . . .



- The choices are made based on the context in which the expression occurs.
- The context constrains the possible choices by imposing equations between types.

## Constraint Generation

This suggests the following idea:

- We can **collect** the constraints imposed by the context in a one-pass analysis phase.

- We can then **solve** these constraints to resolve the choices; the solution determines the choices.
  - No solution: ill-formed. No choices will satisfy constraints.
  - One solution: all choices are forced.



  - Infinitely many solutions: some choices are underconstrained. There are many different reconstructions.

## Type Schemes, Term Schemes

**Type variable** $\alpha$, $\beta$, $\ldots$ = placeholder for a type to be determined later.

**Type scheme** $\tau^*$ = a type containing zero or more type variables.

- *e.g.*, $\alpha \to \alpha$, written `'a -> 'a` in ML.

- $\mathsf{TV}(\tau^*)$ = type variables in $\tau^*$.

**Term scheme** $e^*$ = a term containing type variables.



- *e.g.*, $\texttt{fun}\, f\ (x\,\texttt{:}\,\alpha)\,\texttt{:}\,\alpha\ \texttt{is}\ x\ \texttt{end}$.

- $\mathsf{TV}(e^*)$ = type variables in $e^*$.

**Context scheme** $\Gamma^*$ = context assigning type schemes to variables.

## Constraints

A **constraint** is a formula built up from equations between type schemes:

$$\overline{\top} \qquad \overline{\tau_1^* = \tau_2^*} \qquad \frac{\phi_1 \quad \phi_2}{\phi_1 \wedge \phi_2}$$

Intended reading "there exists $\alpha_1, \ldots, \alpha_n$ such that $\phi$."

**Constraint solving** = finding values for the variables such that the constraints are all true.



## Constraint Generation

Where do constraints arise? From the reconstruction rules!

For example,

$$\frac{\Gamma \vdash u_1 \rightsquigarrow e_1 : \tau_2 \rightarrow \tau \qquad \Gamma \vdash u_2 \rightsquigarrow e_2 : \tau_2}{\Gamma \vdash \texttt{apply}(u_1, u_2) \rightsquigarrow \texttt{apply}(e_1, e_2) : \tau}$$

Implicit requirement: equality between the type of the argument and the domain type of the function.

Other rules have similar implicit requirements.



## Constraint Generation

The **constraint generation** relation

$$\Gamma^* \vdash u \rightsquigarrow e^* : \tau^* \ [\phi]$$

is inductively defined by a set of rules.

- Constraint generation will be a **function** of $\Gamma^*$ and $u$, **even if** $u$ is ill-formed!
- The term scheme $e^*$ and type scheme $\tau^*$ will serve as a **pattern** for all possible reconstructions of $u$ relative to $\Gamma^*$.



## Constraint Generation

Variables and constants induce no constraints.

$$\overline{\Gamma^* \vdash x \rightsquigarrow x : \Gamma^*(x) \ [\top]}$$

$$\overline{\Gamma^* \vdash n \rightsquigarrow n : \texttt{int} \ [\top]}$$

$$\overline{\Gamma^* \vdash \texttt{true} \rightsquigarrow n : \texttt{bool} \ [\top]}$$

$$\overline{\Gamma^* \vdash \texttt{false} \rightsquigarrow n : \texttt{bool} \ [\top]}$$

## Constraint Generation

Primitive operations generate constraints corresponding to the typing requirements.

$$\frac{\Gamma^* \vdash u_1 \rightsquigarrow e_1^* : \tau_1^* \ [\phi_1] \quad \ldots \quad \Gamma^* \vdash u_n \rightsquigarrow e_n^* : \tau_n^* \ [\phi_n]}{\begin{array}{l}\Gamma^* \vdash o(u_1, \ldots, u_n) \rightsquigarrow o(e_1^*, \ldots, e_n^*) : \tau \\ \qquad\qquad [\phi_1 \wedge \cdots \wedge \phi_n \wedge \tau_1^* = \tau_1 \wedge \cdots \wedge \tau_n^* = \tau_n]\end{array}}$$

The constraints ensure that the type schemes assigned to the constituent expresssion are equivalent to those required by the operation.



## Constraint Generation

Conditionals are handled by generating constraints on the types of the test expression and the types of the "then" and "else" clauses. Note the use of a "fresh" type variable to serve as the result type.

$$\frac{\begin{array}{c}\Gamma^* \vdash u \rightsquigarrow e^* : \tau^* \ [\phi] \\ \Gamma^* \vdash u_1 \rightsquigarrow e_1^* : \tau_1^* \ [\phi_1] \quad \Gamma^* \vdash u_2 \rightsquigarrow e_2^* : \tau_2^* \ [\phi_2]\end{array}}{\begin{array}{l}\Gamma^* \vdash \texttt{if}\ u\ \texttt{then}\ u_1\ \texttt{else}\ u_2\ \texttt{fi} \rightsquigarrow \\ \qquad \texttt{if}\ e^*\ \texttt{then}\ e_1^*\ \texttt{else}\ e_2^*\ \texttt{fi} : \alpha \\ \qquad [\phi \wedge \phi_1 \wedge \phi_2 \wedge \tau^* {=} \texttt{bool} \wedge \alpha {=} \tau_1^* \wedge \alpha {=} \tau_2^*]\end{array}}$$



## Constraint Generation

Functions are handled by introducing "fresh" type variables for the domain and range of the function, and constraining the type of the body to match the result type.

$$\frac{\Gamma^*[f{:}\alpha_1 \to \alpha_2][x{:}\alpha_1] \vdash u \rightsquigarrow e^* : \tau^* \ [\phi]}{\begin{array}{l}\Gamma^* \vdash \texttt{fun}\ f\,(x)\ \texttt{is}\ u\ \texttt{end} \rightsquigarrow \\ \qquad \texttt{fun}\ f\ (x{:}\alpha_1){:}\alpha_2\ \texttt{is}\ e\ \texttt{end} : \alpha_1 \to \alpha_2 \\ \qquad [\phi \wedge \tau^* {=} \alpha_2]\end{array}}$$



## Constraint Generation

Applications are handled by introducing a "fresh" type variable for the result type.

$$\frac{\Gamma^* \vdash u_1 \rightsquigarrow e_1^* : \tau_1^* \ [\phi_1] \quad \Gamma^* \vdash u_2 \rightsquigarrow e_2^* : \tau_2^* \ [\phi_2]}{\begin{array}{l}\Gamma^* \vdash \texttt{apply(}u_1\texttt{,}u_2\texttt{)} \rightsquigarrow \\ \qquad \texttt{apply(}e_1^*\texttt{,}e_2^*\texttt{)} : \alpha \\ \qquad [\phi_1 \wedge \phi_2 \wedge \tau_1^* {=} \tau_2^* \to \alpha]\end{array}}$$

Programming Languages
Unification
Robert Harper
Spring, 1999

## Solving Constraints

A **solution** of a constraint $\phi$ is a **substitution** $S$ making all the equations in $\phi$ true.

A **substitution** is a function $S$ from type variables to type schemes that is **almost everywhere** the identity.

- That is, $S(\alpha) = \alpha$ for all but finitely many type variables, called the **domain** of $S$.
- [This is a technical convenience that makes some things simpler to handle.]



## Solving Constraints

Write $\widehat{S}(\tau^*)$ and $\widehat{S}(e^*)$ to apply the substitution $S$ to a type scheme or term scheme.

Formally,

$$\begin{array}{rcl} \widehat{S}(\alpha) & = & S(\alpha) \\ \widehat{S}(\mathtt{int}) & = & \mathtt{int} \\ \widehat{S}(\mathtt{bool}) & = & \mathtt{bool} \\ \widehat{S}(\tau_1^* \to \tau_2^*) & = & \widehat{S}(\tau_1^*) \to \widehat{S}(\tau_2^*) \end{array}$$

(and similarly for term schemes).



## Composition of Substitutions

We will need to compare substitutions in the sense that one is "less general" or "more specific" than another.

$T$ is less general than $S$, written $T \leq S$, iff it does what $S$ does, and then some.

Formally, $T \leq S$ iff $T = U \circ S$ for some substitution $U$.

Composition of substitutions is defined by

$$(U \circ S)(\alpha) = \widehat{U}(S(\alpha))$$

for every type variable $\alpha$.

That is, apply $S$, then apply $U$ to the result.

## Composition of Substitutions

Examples:

1. **Any** substitution is less general than the identity substitution: $S \leq I$ (because $S = S \circ I$).

2. Suppose $S(\alpha) = \texttt{int}$ and $S(\beta) = \gamma \rightarrow \gamma$. If $T(\alpha) = \texttt{int}$ and $T(\beta) = \texttt{int} \rightarrow \texttt{int}$, then $T \leq S$. If $T(\alpha) = \texttt{int}$ and $T(\beta) = \texttt{int} \rightarrow \texttt{bool}$, then $T \not\leq S$.



## Solving a Constraint

$S \models \phi$ means that $\phi$ is true, if we replace the variables in it by their bindings in $S$.

$$\begin{array}{rll} S \models \top & \textit{iff} & \textit{always} \\ S \models \tau_1^* = \tau_2^* & \textit{iff} & \widehat{S}(\tau_1^*) = \widehat{S}(\tau_2^*) \\ S \models \phi_1 \wedge \phi_2 & \textit{iff} & S \models \phi_1 \ \textit{and} \ S \models \phi_2 \end{array}$$



## Principal Solutions

A **principal**, or **most general**, solution $S$ of a constraint $\phi$ is a substitution $S$ such that

1. $S \models \phi$.

2. If $T \models \phi$, then $T \leq S$.

Thus $S$ is the **best** solution for $\phi$.



## Principal Solutions

1. The principal solution of $\alpha = \texttt{int} \wedge \beta = \alpha$ is any substitution $S$ such that $S(\alpha) = S(\beta) = \texttt{int}$ and $S(\gamma) = \gamma$ for any other type variable $\gamma$.

2. If $T(\alpha) = T(\beta) = \texttt{int}$ and $T(\gamma) = \texttt{bool}$ (say), then $T \leq S$, and is not the principal solution of $\phi$.

## Principal Reconstructions

Principal solutions give rise to principal reconstructions, the "best possible" or "most general" reconstruction of a term.

1. `fn (x:`$\alpha$`):`$\alpha$` in x end` is the principal reconstruction of `fn x in x end`.

2. `fn (x:int):int in x end` is a non-principal reconstruction of `fn x in x end`.



## Solving Constraints

**Unification Algorithm**: determine whether or not $\phi$ has a solution, yielding the principal solution if so.

It's an important result that if $\phi$ has a solution, it has a principal solution!

This need not be so! For example,

$$\alpha = \texttt{int} \vee \alpha = \texttt{bool}$$

has no principal solution. Fortunately, we don't need disjunctive constraints to do type reconstruction.



## Solving Constraints

To solve a constraint, we systematically simplify it by a process similar (but simpler than) to that used in linear algebra.

We maintain two things:

1. A **partial**, or **incomplete**, solution $S$, representing the bindings for variables we've found so far.

2. A constraint **remaining** to be solved.



## Solving Constraints

Initially, the incomplete solution is the identity substitution: no variables have been solved yet. The remaining constraint is the entire constraint to be solved.

A **complete solution** has the form $(S, \top)$ — all constraints have been solved, and the partial solution is complete.

## A Transition System

We can represent the unification process as a transition system, the **unification machine**.

States: $(S, \phi)$, where $S$ is the partial solution and $\phi$ is the remaining constraint.

Initial state: $(I, \phi)$.

Final state: $(S, \top)$.

A Transition $(S, \phi) \mapsto (S', \phi')$ represents a **simplification** of the constraint and **progress** towards a complete solution.



## Structural Equivalence

To make the transition system simpler, we impose the following equations on constraints:

$$\begin{array}{rcl} \top \wedge \phi & \equiv & \phi \\ \phi \wedge \top & \equiv & \phi \\ \phi_1 \wedge \phi_2 & \equiv & \phi_2 \wedge \phi_1 \\ \phi_1 \wedge (\phi_2 \wedge \phi_3) & \equiv & (\phi_1 \wedge \phi_2) \wedge \phi_3 \\ \tau_1^* {=} \tau_2^* & \equiv & \tau_2^* {=} \tau_1^* \end{array}$$

Informally, a constraint can be thought of as a multiset (set with repetitions) of unordered pairs of type schemes.



## Transition Rules

Equations between identical base types and identical type variables may be dropped.

$$\begin{array}{rcl} (S, \mathtt{int}{=}\mathtt{int} \wedge \phi) & \mapsto & (S, \phi) \\ (S, \mathtt{bool}{=}\mathtt{bool} \wedge \phi) & \mapsto & (S, \phi) \\ (S, \alpha{=}\alpha \wedge \phi) & \mapsto & (S, \phi) \end{array}$$

An equation between function types reduces to equations between their respective domains and ranges.

$$(S, \tau_1^* {\to} \tau_2^* {=} \sigma_1^* {\to} \sigma_2^* \wedge \phi) \mapsto (S, \tau_1^* {=} \sigma_1^* \wedge \tau_2^* {=} \sigma_2^* \wedge \phi)$$



## Transition Rules

An equation of the form $\alpha{=}\tau^*$ partially determines the solution for $\alpha$, provided that $\alpha$ does not occur in $\tau^*$:

$$(S, \alpha{=}\tau^* \wedge \phi) \mapsto ([\alpha{=}\tau^*] \circ S, [\tau^*/\alpha]\phi)$$

The requirement that $\alpha \notin \mathsf{TV}(\tau^*)$ is called the **occurs check**. It is of prime importance!

Things to notice:

1. The partial solution $S$ is extended with a binding for $\alpha$.

2. All occurrences of $\alpha$ in $\phi$ are replaced by $\tau^*$.

## Occurs Check

The equation $\alpha{=}\alpha{\to}\alpha$ is not solvable!

If $\tau$ solved this equation, then we would have to have

$$\begin{aligned}\tau &= \tau{\to}\tau \\ &= (\tau{\to}\tau){\to}(\tau{\to}\tau) \\ &= \ldots\end{aligned}$$

This is clearly impossible!

Contrast: over the reals, $x = x + 0$, $x = x \times 1$, $x = \sqrt{x}^2$ are all solvable!



## Irreducible States

A state $(S, \phi)$ is **irreducible** iff there is **no** state $(S', \phi')$ such that

$$(S, \phi) \mapsto (S', \phi')$$

That is, no further progress can be made.

Observe:

1. Every final state is irreducible.
2. Not every irreducible state is final. Non-final irreducible states are called **stuck states**.



## Stuck States

If $(S, \phi)$ is stuck, then every equation in $\phi$ has one of these forms:

1. `int`$=$`bool`.
2. $\tau_1^*{\to}\tau_2^*{=}\tau^*$, where $\tau^*$ is not a function type.
3. $\alpha{=}\tau^*$, where $\alpha \in \mathsf{TV}(\tau^*)$.

Proof: consider all possible equations $\tau_1^*{=}\tau_2^*$, and check that either progress can be made or the equation has one of these forms.

It is clearly decidable whether or not a state is stuck.



## Termination

An important property of the unification machine is that all computations terminate, no matter how the steps are chosen.

Fact: There is no infinite sequence of states

$$(S_1, \phi_1) \mapsto (S_2, \phi_2) \mapsto \cdots.$$

We can prove this by assigning a **size measure** to states and showing that every transition reduces the size of the state according to this measure.

## Proving Termination

Informally, what "goes down" on each transition?

1. In the first set, the size of the constraint is reduced because we're dropping an equation.

2. In the second set, the size of the constraint is reduced because we drop two occurrences of the $\rightarrow$ symbol.

3. In the third, the size of the constraint **increases** (in general) because we may replicate $\tau^*$ in $\phi$!

Hmmmm. . . .



## Proving Termination

Observation: in the third transition the number of variables goes down because we drop $\alpha{=}\tau^*$ and replace $\alpha$ by $\tau^*$ in $\phi$.

Observation: the other two transitions do not increase the number of variables.

Aha! **Either** the number of variables decreases, **or**, if it stays the same, then the size decreases.



## Lexicographic Orderings

Dictionary ordering for pairs of natural numbers: $(m,n) < (m',n')$ iff either $m < m'$ or $m = m'$ and $n < n'$. In the former case, the relative magnitudes of $n$ and $n'$ is irrelevant!

Analogy: Aqaba < Azerbaijan, Algebra < Zebra.

Fact: There are no infinite descending sequences of pairs of natural numbers ordered by the lexicographic ordering.



## Lexicographic Orderings

Spse $(m_1,n_1) > (m_2,n_2) > \ldots$.

If all $m_i$'s are equal, then we must have an infinite descending sequence of natural numbers $n_1, n_2, \ldots$, which is impossible.

Otherwise there is an embedded decreasing sequence $m_{i_1} > m_{i_2} > \ldots$ of natural numbers, which is also impossible.

## Size Measure

The **size** of a state $(S, \phi)$ is defined to be $(n, s)$, where

1. $n$ is the number of variables in $\phi$.

2. $s$ is the size of $\phi$, the sum of the sizes of the type schemes in $\phi$.

If $(S, \phi) \mapsto (S', \phi')$, then the size of $(S, \phi)$ is strictly larger than the size of $(S', \phi')$.



## Transfinite Ordinals

Cultural aside: we can use **transfinite ordinals** instead of pairs of numbers to measure the size of a state.

Define $(m, n) = m \times \omega + n$, where $\omega$ is the first transfinite ordinal (the one right after 0, 1, 2, ...).

Observe that if $m < m'$, then $m \times \omega + n < m' \times \omega + n'$, for any $n$ and $n'$, because $\omega$ is bigger than any finite number!

If $m = m'$, then $m \times \omega + n < m' \times \omega + n'$ iff $n < n'$. Just compare finite natural numbers.

Size of $(S, \phi) = n \times \omega + s$, where $n$ and $s$ are as above.

We are using **transfinite induction up to** $\omega^2$.



## Complete Solutions

A **complete solution** for a state $(S, \phi)$ is a substitution $T$ such that

1. $T \leq S$.

2. $T \models \phi$

That is, $T$ extends $S$ and solves $\phi$.



## Complete Solutions

1. Every final state $(S, \top)$ has a complete solution, namely $S$, which is also a principal solution.

2. No stuck state has a complete solution (the equations in it are unsolvable).

3. Suppose that $(S, \phi) \mapsto (S', \phi')$. Then $T$ is a complete solution for $(S, \phi)$ iff $T$ is a complete solution for $(S', \phi')$.

## Summary

1. By termination, $(I, \phi) \mapsto^* (S, \phi') \not\mapsto$ for some $S$ and $\phi'$.

2. If $(S, \phi)$ is a final state (*i.e.*, $\phi = \top$), then $S$ is the principal solution for $\phi$. If $(S, \phi)$ is stuck, then $\phi$ has no solution.

   If $(S, \phi)$ is final, then $S$ is a complete solution for $(I, \phi)$, and hence is a solution for $\phi$. If $T$ is any solution for $\phi$, then $T$ is a complete solution for $(S, \phi)$, which is to say that $T \leq S$.

   If $(S, \phi)$ is stuck, then it has no complete solution, and hence $(I, \phi)$ has no complete solution, hence $\phi$ has no solution.



3. If $\phi$ has a solution, then $\phi' = \top$. If $\phi$ has no solution, then $(S, \phi')$ is stuck.

   If $\phi$ has a solution $T$, then $T$ is a complete solution for $(S, \phi')$. But then $(S, \phi')$ cannot be stuck.

   If $\phi$ has no solution, then $(S, \phi')$ can have no solution, and hence must be stuck (since it is irreducible).

## Unification Algorithm

To find the principal solution (if any) of $\phi$, start the unification machine with $(I, \phi)$ and run it until termination.

If the result state is stuck, then $\phi$ is unsolvable.

If the result state is final, read off the principal solution.



## Type Reconstruction

To find the set of all possible reconstructions of $u$ relative to $\Gamma$:

1. Find $e^*$, $\tau^*$, and $\phi$ such that $\Gamma \vdash u \rightsquigarrow e^* : \tau^*\ [\phi]$.

2. Find the principal solution $S$ of $\phi$, if any.

3. If $\phi$ is unsolvable, then $u$ is not reconstructible. (Because every reconstruction determines a solution to $\phi$.)

4. If $\phi$ has a principal solution $S$, then $\widehat{S}(e^*)$ is a **schematic reconstruction** of $u$ with type $\widehat{S}(\tau^*)$. Every reconstruction is an instance of this schema.

   Every closed solution of $\phi$ determines a reconstruction of $u$. Since $S$ is the principal solution of $\phi$, any closed solution $T$ must be less general than $S$, *i.e.*, $T \leq S$. So every reconstruction is an instance of $\widehat{S}(e^*) : \widehat{S}(\tau^*)$.

## Examples

Here is the constraint $\phi$ from the `map` example:

$$\begin{array}{rcl} \beta & = & \beta'\,\texttt{list} \\ \beta & = & \beta''\,\texttt{list} \\ \beta & = & \beta'''\,\texttt{list} \\ \alpha & = & \alpha \\ \beta & = & \beta'''\,\texttt{list} \\ \alpha & = & \beta''{\rightarrow}\alpha' \\ \gamma & = & \gamma'\,\texttt{list} \\ \alpha' & = & \gamma' \\ \delta\,\texttt{list} & = & \gamma'\,\texttt{list} \\ \gamma & = & \delta\,\texttt{list} \end{array}$$



## Example

Dropping reflexive instances and solving for $\alpha$, we obtain

$$\begin{array}{rcl} \alpha & = & \beta''{\rightarrow}\alpha' \\ \hline \beta & = & \beta'\,\texttt{list} \\ \beta & = & \beta''\,\texttt{list} \\ \beta & = & \beta'''\,\texttt{list} \\ \beta & = & \beta'''\,\texttt{list} \\ \gamma & = & \gamma'\,\texttt{list} \\ \alpha' & = & \gamma' \\ \delta\,\texttt{list} & = & \gamma'\,\texttt{list} \\ \gamma & = & \delta\,\texttt{list} \end{array}$$



## Example

Solving for $\beta$ we obtain

$$\begin{array}{rcl} \beta & = & \beta'\,\texttt{list} \\ \alpha & = & \beta''{\rightarrow}\alpha' \\ \hline \beta'\,\texttt{list} & = & \beta''\,\texttt{list} \\ \beta'\,\texttt{list} & = & \beta'''\,\texttt{list} \\ \beta'\,\texttt{list} & = & \beta'''\,\texttt{list} \\ \gamma & = & \gamma'\,\texttt{list} \\ \alpha' & = & \gamma' \\ \delta\,\texttt{list} & = & \gamma'\,\texttt{list} \\ \gamma & = & \delta\,\texttt{list} \end{array}$$

## Example

Solving for $\gamma$, we obtain

$$\begin{array}{rcl} \gamma & = & \gamma'\,\texttt{list} \\ \beta & = & \beta'\,\texttt{list} \\ \alpha & = & \beta''\!\rightarrow\!\alpha' \\ \hline \beta'\,\texttt{list} & = & \beta''\,\texttt{list} \\ \beta'\,\texttt{list} & = & \beta'''\,\texttt{list} \\ \beta'\,\texttt{list} & = & \beta'''\,\texttt{list} \\ \alpha' & = & \gamma' \\ \delta\,\texttt{list} & = & \gamma'\,\texttt{list} \\ \gamma'\,\texttt{list} & = & \delta\,\texttt{list} \end{array}$$



## Example

Simplifying,

$$\begin{array}{rcl} \gamma & = & \gamma'\,\texttt{list} \\ \beta & = & \beta'\,\texttt{list} \\ \alpha & = & \beta''\!\rightarrow\!\alpha' \\ \hline \beta' & = & \beta'' \\ \beta' & = & \beta''' \\ \beta' & = & \beta''' \\ \alpha' & = & \gamma' \\ \delta & = & \gamma' \\ \gamma' & = & \delta \end{array}$$



## Example

Solving for $\beta'$ we obtain

$$\begin{array}{rcl} \beta' & = & \beta'' \\ \gamma & = & \gamma'\,\texttt{list} \\ \beta & = & \beta'\,\texttt{list} \\ \alpha & = & \beta''\!\rightarrow\!\alpha' \\ \hline \beta'' & = & \beta''' \\ \beta'' & = & \beta''' \\ \alpha' & = & \gamma' \\ \delta & = & \gamma' \\ \gamma' & = & \delta \end{array}$$



## Example

Solving for $\beta''$, we obtain

$$\begin{array}{rcl} \beta'' & = & \beta''' \\ \beta' & = & \beta'' \\ \gamma & = & \gamma'\,\texttt{list} \\ \beta & = & \beta'\,\texttt{list} \\ \alpha & = & \beta''\!\rightarrow\!\alpha' \\ \hline \beta''' & = & \beta''' \\ \alpha' & = & \gamma' \\ \delta & = & \gamma' \\ \gamma' & = & \delta \end{array}$$

## Example

Dropping reflexive instances and solving for $\gamma'$, we obtain

$$\begin{array}{rcl}
\gamma' & = & \alpha' \\
\beta'' & = & \beta''' \\
\beta' & = & \beta'' \\
\gamma & = & \gamma'\,\texttt{list} \\
\beta & = & \beta'\,\texttt{list} \\
\alpha & = & \beta''\to\alpha' \\ \hline
\delta & = & \alpha' \\
\alpha' & = & \delta
\end{array}$$



## Example

Solving for $\delta$ and dropping reflexive instances,

$$\begin{array}{rcl}
\delta & = & \alpha' \\
\gamma' & = & \alpha' \\
\beta'' & = & \beta''' \\
\beta' & = & \beta'' \\
\gamma & = & \gamma'\,\texttt{list} \\
\beta & = & \beta'\,\texttt{list} \\
\alpha & = & \beta''\to\alpha' \\ \hline
\end{array}$$



## Example

Compressing the substitution, we obtain

$$\begin{array}{rcl}
\delta & = & \alpha' \\
\gamma' & = & \alpha' \\
\beta'' & = & \beta''' \\
\beta' & = & \beta''' \\
\gamma & = & \alpha'\,\texttt{list} \\
\beta & = & \beta'''\,\texttt{list} \\
\alpha & = & \beta'''\to\alpha' \\ \hline
\end{array}$$



## Example

Applying this to the schematic reconstruction, we obtain

```
fun map (f:β'''→α',x:β''' list):α' list
is ... end
```

Renaming type variables for readability, this is

```
fun map (f:α→β,x:α list):β list
is ... end
```

Ta da!