Type Inference Through Unification

(“how to systematically remove the guess-work from type-inference”)

Assaf Kfoury

November 16, 2005

This document is best read on a monitor screen, using a previewer (ghostscript for ps, acroread for pdf). If you print it out, which you can of course, it will not give you the effect of the overlayer pages.

Example 1: $M =$ `fn f => fn x => f (f x)`

This expression is well-formed in **mini-ML** and, therefore, in SML. This is confirmed by running the SML interpreter on the expression.

<u>Example 1</u>: $M =$ `fn f => fn x => f (f x)`

This expression is well-formed in **mini-ML** and, therefore, in SML. This is confirmed by running the SML interpreter on the expression.

A skeleton for a typing derivation for M, with unknown (mono) types inserted, can be built incrementally in top-down fashion.

1.	`f` $: \tau_1$	$\vdash$ `f` $: \tau_1$	VAR

Example 1: $M =$ `fn f => fn x => f (f x)`

This expression is well-formed in **mini-ML** and, therefore, in SML. This is confirmed by running the SML interpreter on the expression.

A skeleton for a typing derivation for M, with unknown (mono) types inserted, can be built incrementally in top-down fashion.

1.	`f` $: \tau_1$	$\vdash$ `f` $: \tau_1$	VAR
2.	`x` $: \tau_2$	$\vdash$ `x` $: \tau_2$	VAR

Example 1: $M = $ `fn f => fn x => f (f x)`

This expression is well-formed in **mini-ML** and, therefore, in SML. This is confirmed by running the SML interpreter on the expression.

A skeleton for a typing derivation for M, with unknown (mono) types inserted, can be built incrementally in top-down fashion.

1.	$\mathtt{f} : \tau_1, \mathtt{x} : \tau_2 \vdash \mathtt{f} : \tau_1$		VAR
2.	$\mathtt{f} : \tau_1, \mathtt{x} : \tau_2 \vdash \mathtt{x} : \tau_2$		VAR
3.	$\mathtt{f} : \tau_1, \mathtt{x} : \tau_2 \vdash \mathtt{f\ x} : \tau_3$	$\tau_1 = \tau_2 \texttt{ -> } \tau_3$	APP(1,2)

Example 1: $M =$ `fn f => fn x => f (f x)`

This expression is well-formed in **mini-ML** and, therefore, in SML. This is confirmed by running the SML interpreter on the expression.

A skeleton for a typing derivation for M, with unknown (mono) types inserted, can be built incrementally in top-down fashion.

1.	$\texttt{f} : \tau_1, \texttt{x} : \tau_2$	$\vdash \texttt{f} : \tau_1$		VAR
2.	$\texttt{f} : \tau_1, \texttt{x} : \tau_2$	$\vdash \texttt{x} : \tau_2$		VAR
3.	$\texttt{f} : \tau_1, \texttt{x} : \tau_2$	$\vdash \texttt{f x} : \tau_3$	$\tau_1 = \tau_2 \texttt{ -> } \tau_3$	APP(1,2)
4.	$\texttt{f} : \tau_1$	$\vdash \texttt{f} : \tau_1$		VAR

Example 1: $M =$ `fn f => fn x => f (f x)`

This expression is well-formed in **mini-ML** and, therefore, in SML. This is confirmed by running the SML interpreter on the expression.

A skeleton for a typing derivation for M, with unknown (mono) types inserted, can be built incrementally in top-down fashion.

1.	$\texttt{f} : \tau_1, \texttt{x} : \tau_2 \vdash \texttt{f} : \tau_1$		VAR
2.	$\texttt{f} : \tau_1, \texttt{x} : \tau_2 \vdash \texttt{x} : \tau_2$		VAR
3.	$\texttt{f} : \tau_1, \texttt{x} : \tau_2 \vdash \texttt{f x} : \tau_3$	$\tau_1 = \tau_2 \texttt{ -> } \tau_3$	APP(1,2)
4.	$\texttt{f} : \tau_1, \texttt{x} : \tau_2 \vdash \texttt{f} : \tau_1$		VAR
5.	$\texttt{f} : \tau_1, \texttt{x} : \tau_2 \vdash \texttt{f (f x)} : \tau_4$	$\tau_1 = \tau_3 \texttt{ -> } \tau_4$	APP(3,4)

Example 1: $M = $ `fn f => fn x => f (f x)`

This expression is well-formed in **mini-ML** and, therefore, in SML. This is confirmed by running the SML interpreter on the expression.

A skeleton for a typing derivation for M, with unknown (mono) types inserted, can be built incrementally in top-down fashion.

1.	$\texttt{f} : \tau_1, \texttt{x} : \tau_2 \vdash \texttt{f} : \tau_1$		VAR
2.	$\texttt{f} : \tau_1, \texttt{x} : \tau_2 \vdash \texttt{x} : \tau_2$		VAR
3.	$\texttt{f} : \tau_1, \texttt{x} : \tau_2 \vdash \texttt{f x} : \tau_3$	$\tau_1 = \tau_2$ `->` τ_3	APP(1,2)
4.	$\texttt{f} : \tau_1, \texttt{x} : \tau_2 \vdash \texttt{f} : \tau_1$		VAR
5.	$\texttt{f} : \tau_1, \texttt{x} : \tau_2 \vdash \texttt{f (f x)} : \tau_4$	$\tau_1 = \tau_3$ `->` τ_4	APP(3,4)
6.	$\texttt{f} : \tau_1 \vdash \texttt{fn x => f (f x)} : \tau_2$ `->` τ_4		ABS(5)

Example 1: M = `fn f => fn x => f (f x)`

This expression is well-formed in **mini-ML** and, therefore, in SML. This is confirmed by running the SML interpreter on the expression.

A skeleton for a typing derivation for M, with unknown (mono) types inserted, can be built incrementally in top-down fashion.

1.	`f` : τ_1, `x` : τ_2 $\vdash$ `f` : τ_1		VAR
2.	`f` : τ_1, `x` : τ_2 $\vdash$ `x` : τ_2		VAR
3.	`f` : τ_1, `x` : τ_2 $\vdash$ `f x` : τ_3	$\tau_1 = \tau_2$ `->` τ_3	APP(1,2)
4.	`f` : τ_1, `x` : τ_2 $\vdash$ `f` : τ_1		VAR
5.	`f` : τ_1, `x` : τ_2 $\vdash$ `f (f x)` : τ_4	$\tau_1 = \tau_3$ `->` τ_4	APP(3,4)
6.	`f` : τ_1 $\vdash$ `fn x => f (f x)` : τ_2 `->` τ_4		ABS(5)
7.	$\vdash$ `fn f => fn x => f (f x)` : τ_1 `->` τ_2 `->` τ_4		ABS(6)

This is just a skeleton, which becomes a valid typing derivation once the constraints are satisfied.

Solving Constraints Using Unification: We collect all the constraints in a sequence (in the order in which they are generated), which is then used as a stack.[1] We process the first constraint in the stack (the "top constraint"), which gives rise to one of three possible actions:

(A) Definition of a "small" substitution and elimination of the top constraint.

(B) Simplification of the top constraint into additional "simpler" constraints, which are then placed back on top of the stack.

(C) Contradiction – which blocks any further processing of the constraints.

After action (A) or (B), but not (C), we continue to process each of the remaining constraints in the stack. This procedure is bound to terminate, with one of two possible outcomes at the end:

- a contradiction, as in (C), indicating the constraints cannot be solved, or
- an empty stack of constraints, indicating the constraints can be solved.

We illustrate this procedure, called *Unification*, with the constraints generated in Example 1.

[1] A queue will work just as well here.

Example 1 (continued): The initial sequence of constraints is:

(1) $\tau_1 = \tau_2$ `->` $\tau_3, \quad \tau_1 = \tau_3$ `->` τ_4

Example 1 (continued): The initial sequence of constraints is:

(1) $\tau_1 = \tau_2$ `->` $\tau_3, \quad \tau_1 = \tau_3$ `->` τ_4

From the top constraint $\tau_1 = \tau_2$ `->` τ_3 in (1), we define the small substitution:

$\tau_1 := \tau_2$ `->` τ_3

and apply it to the remaining constraints, to obtain a new sequence of constraints:

(2) τ_2 `->` $\tau_3 = \tau_3$ `->` τ_4

Example 1 (continued): The initial sequence of constraints is:

(1) $\boxed{\tau_1 = \tau_2 \texttt{ -> } \tau_3, \quad \tau_1 = \tau_3 \texttt{ -> } \tau_4}$

From the top constraint $\tau_1 = \tau_2 \texttt{ -> } \tau_3$ in (1), we define the small substitution:

$$\tau_1 := \tau_2 \texttt{ -> } \tau_3$$

and apply it to the remaining constraints, to obtain a new sequence of constraints:

(2) $\boxed{\tau_2 \texttt{ -> } \tau_3 = \tau_3 \texttt{ -> } \tau_4}$

The top (and only) constraint in (2) gives rise to a simplication and a new sequence of constraints:

(3) $\boxed{\tau_2 = \tau_3, \quad \tau_3 = \tau_4}$

Example 1 (continued): The initial sequence of constraints is:

(1) $\boxed{\tau_1 = \tau_2 \texttt{ -> } \tau_3, \quad \tau_1 = \tau_3 \texttt{ -> } \tau_4}$

From the top constraint $\tau_1 = \tau_2 \texttt{ -> } \tau_3$ in (1), we define the small substitution:

$$\tau_1 := \tau_2 \texttt{ -> } \tau_3$$

and apply it to the remaining constraints, to obtain a new sequence of constraints:

(2) $\boxed{\tau_2 \texttt{ -> } \tau_3 = \tau_3 \texttt{ -> } \tau_4}$

The top (and only) constraint in (2) gives rise to a simplication and a new sequence of constraints:

(3) $\boxed{\tau_2 = \tau_3, \quad \tau_3 = \tau_4}$

From the top constraint in (3), we define the small substitution:

$$\tau_2 := \tau_3$$

and apply it to the remaining constraints, to obtain the sequence:

(4) $\boxed{\tau_3 = \tau_4}$

Example 1 (continued): The initial sequence of constraints is:

(1) $\boxed{\tau_1 = \tau_2 \texttt{->} \tau_3, \quad \tau_1 = \tau_3 \texttt{->} \tau_4}$

From the top constraint $\tau_1 = \tau_2 \texttt{->} \tau_3$ in (1), we define the small substitution:

$$\tau_1 := \tau_2 \texttt{->} \tau_3$$

and apply it to the remaining constraints, to obtain a new sequence of constraints:

(2) $\boxed{\tau_2 \texttt{->} \tau_3 = \tau_3 \texttt{->} \tau_4}$

The top (and only) constraint in (2) gives rise to a simplication and a new sequence of constraints:

(3) $\boxed{\tau_2 = \tau_3, \quad \tau_3 = \tau_4}$

From the top constraint in (3), we define the small substitution:

$$\tau_2 := \tau_3$$

and apply it to the remaining constraints, to obtain the sequence:

(4) $\boxed{\tau_3 = \tau_4}$

From the top (and only) constraint in (4), we define the small substitution:

$$\tau_3 := \tau_4$$

and apply it to obtain the empty sequence:

(5) $\boxed{\varnothing}$

Example 1 (continued): The initial sequence of constraints is:

(1) $\tau_1 = \tau_2 \texttt{->} \tau_3, \quad \tau_1 = \tau_3 \texttt{->} \tau_4$

From the top constraint $\tau_1 = \tau_2 \texttt{->} \tau_3$ in (1), we define the small substitution:

$$\tau_1 := \tau_2 \texttt{->} \tau_3$$

and apply it to the remaining constraints, to obtain a new sequence of constraints:

(2) $\tau_2 \texttt{->} \tau_3 = \tau_3 \texttt{->} \tau_4$

The top (and only) constraint in (2) gives rise to a simplication and a new sequence of constraints:

(3) $\tau_2 = \tau_3, \quad \tau_3 = \tau_4$

From the top constraint in (3), we define the small substitution:

$$\tau_2 := \tau_3$$

and apply it to the remaining constraints, to obtain the sequence:

(4) $\tau_3 = \tau_4$

From the top (and only) constraint in (4), we define the small substitution:

$$\tau_3 := \tau_4$$

and apply it to obtain the empty sequence:

(5) $\varnothing$

With no constraint left to process, we conclude there is a solution.

But we are not yet done, because we need to compose the small substitutions — in the order in which they are generated — to obtain a single large substitution.

Sequence of small substitutions	**Resulting large substitution**
$\{\tau_1 := \tau_2$ `->` $\tau_3\}$	$\{\tau_1 := \tau_2$ `->` $\tau_3\}$

But we are not yet done, because we need to compose the small substitutions — in the order in which they are generated — to obtain a single large substitution.

Sequence of small substitutions	**Resulting large substitution**
$\{\tau_1 := \tau_2$ `->` $\tau_3\}$	$\{\tau_1 := \tau_2$ `->` $\tau_3\}$
$\{\tau_1 := \tau_2$ `->` $\tau_3\}$ $\{\tau_2 := \tau_3\}$	$\{\tau_1 := \tau_3$ `->` $\tau_3, \quad \tau_2 := \tau_3\}$

But we are not yet done, because we need to compose the small substitutions — in the order in which they are generated — to obtain a single large substitution.

Sequence of small substitutions	**Resulting large substitution**
$\{\tau_1 := \tau_2$ `->` $\tau_3\}$	$\{\tau_1 := \tau_2$ `->` $\tau_3\}$
$\{\tau_1 := \tau_2$ `->` $\tau_3\}$ $\{\tau_2 := \tau_3\}$	$\{\tau_1 := \tau_3$ `->` $\tau_3, \quad \tau_2 := \tau_3\}$
$\{\tau_1 := \tau_2$ `->` $\tau_3\}$ $\{\tau_2 := \tau_3\}$ $\{\tau_3 := \tau_4\}$	$\{\tau_1 := \tau_4$ `->` $\tau_4, \quad \tau_2 := \tau_4, \quad \tau_3 := \tau_4\}$

But we are not yet done, because we need to compose the small substitutions — in the order in which they are generated — to obtain a single large substitution.

Sequence of small substitutions	**Resulting large substitution**
$\{\tau_1 := \tau_2$ `->` $\tau_3\}$	$\{\tau_1 := \tau_2$ `->` $\tau_3\}$
$\{\tau_1 := \tau_2$ `->` $\tau_3\}$ $\{\tau_2 := \tau_3\}$	$\{\tau_1 := \tau_3$ `->` $\tau_3, \quad \tau_2 := \tau_3\}$
$\{\tau_1 := \tau_2$ `->` $\tau_3\}$ $\{\tau_2 := \tau_3\}$ $\{\tau_3 := \tau_4\}$	$\{\tau_1 := \tau_4$ `->` $\tau_4, \quad \tau_2 := \tau_4, \quad \tau_3 := \tau_4\}$

In the final large substitution, the types τ_1, τ_2 and τ_3 are defined in terms of τ_4, while τ_4 is totally unrestricted. To make clear that τ_4 is unrestricted, let us substitute a fresh type variable α for τ_4. So, we can express the final large substitution, call it S, as follows:

$$S = \{\tau_1 := \alpha \texttt{ -> } \alpha, \quad \tau_2 := \alpha, \quad \tau_3 := \alpha, \quad \tau_4 := \alpha\}$$

The substitution S is a solution for the initial sequence of constraints. What is more, S is a most general solution — intuitively, this means we did the minumum work to produce a substitution satisfying all the constraints.

Now that we have the solution S, what do we do with it?

The substitution S is a solution for the initial sequence of constraints. What is more, S is a most general solution — intuitively, this means we did the minumum work to produce a substitution satisfying all the constraints.

Now that we have the solution S, what do we do with it?

We can apply S to the final skeleton for M to obtain a valid typing derivation for it:

1.	`f` : α `->` α, `x` : α	$\vdash$ `f` : α `->` α	VAR
2.	`f` : α `->` α, `x` : α	$\vdash$ `x` : α	VAR
3.	`f` : α `->` α, `x` : α	$\vdash$ `f x` : α	APP(1,2)
4.	`f` : α `->` α, `x` : α	$\vdash$ `f` : α `->` α	VAR
5.	`f` : α `->` α, `x` : α	$\vdash$ `f (f x)` : α	APP(3,4)
6.	`f` : α `->` α	$\vdash$ `fn x => f (f x)` : α `->` α	ABS(5)
7.		$\vdash$ `fn f => fn x => f (f x)` : (α `->` α) `->` α `->` α	ABS(6)

The substitution S is a solution for the initial sequence of constraints. What is more, S is a most general solution — intuitively, this means we did the minumum work to produce a substitution satisfying all the constraints.

Now that we have the solution S, what do we do with it?

We can apply S to the final skeleton for M to obtain a valid typing derivation for it:

1.	`f` $: \alpha$ `->` α, `x` $: \alpha$	$\vdash$ `f` $: \alpha$ `->` α	VAR
2.	`f` $: \alpha$ `->` α, `x` $: \alpha$	$\vdash$ `x` $: \alpha$	VAR
3.	`f` $: \alpha$ `->` α, `x` $: \alpha$	$\vdash$ `f x` $: \alpha$	APP(1,2)
4.	`f` $: \alpha$ `->` α, `x` $: \alpha$	$\vdash$ `f` $: \alpha$ `->` α	VAR
5.	`f` $: \alpha$ `->` α, `x` $: \alpha$	$\vdash$ `f (f x)` $: \alpha$	APP(3,4)
6.	`f` $: \alpha$ `->` α	$\vdash$ `fn x => f (f x)` $: \alpha$ `->` α	ABS(5)
7.		$\vdash$ `fn f => fn x => f (f x)` $: (\alpha$ `->` $\alpha)$ `->` α `->` α	ABS(6)

But we don't really need the full typing derivation for M — that was only to explain the theory of type-checking, type-inference and unification.

From the programmer's point of view, we are interested in the final type assigned to M, which can be obtained by simply applying S to the final type expression τ_1 `->` τ_2 `->` τ_4 in the skeleton for M:

$(\alpha$ `->` $\alpha)$ `->` α `->` α

which is precisely the type returned by the SML interpreter for M.

Example 2: In SML, the operation `(op +)` is predefined with type `int * int -> int`. We now apply the SML expression M of Example 1 to `(op +)`, call the resulting application N, i.e.,

$$N \;=\; (\texttt{fn f => fn x => f (f x)})\;\texttt{(op +)}$$

Example 2: In SML, the operation `(op +)` is predefined with type `int * int -> int`. We now apply the SML expression M of Example 1 to `(op +)`, call the resulting application N, i.e.,

$$N \;=\; \texttt{(fn f => fn x => f (f x)) (op +)}$$

We do not go through all the details of Example 1 here: We simply note that writing a skeleton for N will consist in extending the skeleton for M with two judgements and one constraint, namely:

8.	$\vdash$ `(op +)` : `int * int -> int`		OP
9.	$\vdash$ `(fn f => fn x => f (f x)) (op +)` : τ_5	τ_1 `->` τ_2 `->` τ_4 = (`int * int -> int`) `->` τ_5	APP(7,8)

Example 2: In SML, the operation `(op +)` is predefined with type `int * int -> int`. We now apply the SML expression M of Example 1 to `(op +)`, call the resulting application N, i.e.,

$$N = \texttt{(fn f => fn x => f (f x)) (op +)}$$

We do not go through all the details of Example 1 here: We simply note that writing a skeleton for N will consist in extending the skeleton for M with two judgements and one constraint, namely:

8. $\vdash$ `(op +)` : `int * int -> int` OP
9. $\vdash$ `(fn f => fn x => f (f x)) (op +)` : τ_5 $\boxed{\tau_1 \texttt{ -> } \tau_2 \texttt{ -> } \tau_4 = (\texttt{int * int -> int}) \texttt{ -> } \tau_5}$ APP(7,8)

The initial sequence of constraints for N is therefore:

$$\boxed{\tau_1 = \tau_2 \texttt{ -> } \tau_3, \quad \tau_1 = \tau_3 \texttt{ -> } \tau_4, \quad \tau_1 \texttt{ -> } \tau_2 \texttt{ -> } \tau_4 = (\texttt{int * int -> int}) \texttt{ -> } \tau_5}$$

The first two constraints are identical to those obtained for M, the third constraint is new.

Example 2 (continued): We use unification to process the constraints for N:

(1) $\tau_1 = \tau_2$ `->` τ_3, $\tau_1 = \tau_3$ `->` τ_4, τ_1 `->` τ_2 `->` $\tau_4 =$ (`int * int -> int`) `->` τ_5

Example 2 (continued): We use unification to process the constraints for N:

(1) $\tau_1 = \tau_2 \texttt{ -> } \tau_3, \quad \tau_1 = \tau_3 \texttt{ -> } \tau_4, \quad \tau_1 \texttt{ -> } \tau_2 \texttt{ -> } \tau_4 = (\texttt{int * int -> int}) \texttt{ -> } \tau_5$

Define small substitution $\tau_1 := \tau_2 \texttt{ -> } \tau_3$ and apply to remaining constraints.

(2) $\tau_2 \texttt{ -> } \tau_3 = \tau_3 \texttt{ -> } \tau_4, \quad (\tau_2 \texttt{ -> } \tau_3) \texttt{ -> } \tau_2 \texttt{ -> } \tau_4 = (\texttt{int * int -> int}) \texttt{ -> } \tau_5$

Example 2 (continued): We use unification to process the constraints for N:

(1) $\tau_1 = \tau_2$ `->` $\tau_3, \quad \tau_1 = \tau_3$ `->` $\tau_4, \quad \tau_1$ `->` τ_2 `->` $\tau_4 =$ `(int * int -> int) ->` τ_5

Define small substitution $\tau_1 := \tau_2$ `->` τ_3 and apply to remaining constraints.

(2) τ_2 `->` $\tau_3 = \tau_3$ `->` $\tau_4, \quad (\tau_2$ `->` $\tau_3)$ `->` τ_2 `->` $\tau_4 =$ `(int * int -> int) ->` τ_5

Simplify.

(3) $\tau_2 = \tau_3, \quad \tau_3 = \tau_4, \quad (\tau_2$ `->` $\tau_3)$ `->` τ_2 `->` $\tau_4 =$ `(int * int -> int) ->` τ_5

Example 2 (continued): We use unification to process the constraints for N:

(1) $\tau_1 = \tau_2$ `->` τ_3, $\tau_1 = \tau_3$ `->` τ_4, τ_1 `->` τ_2 `->` $\tau_4 =$ (`int * int -> int`) `->` τ_5

Define small substitution $\tau_1 := \tau_2$ `->` τ_3 and apply to remaining constraints.

(2) τ_2 `->` $\tau_3 = \tau_3$ `->` τ_4, (τ_2 `->` τ_3) `->` τ_2 `->` $\tau_4 =$ (`int * int -> int`) `->` τ_5

Simplify.

(3) $\tau_2 = \tau_3$, $\tau_3 = \tau_4$, (τ_2 `->` τ_3) `->` τ_2 `->` $\tau_4 =$ (`int * int -> int`) `->` τ_5

Define small substitution $\tau_2 := \tau_3$ and apply to remaining constraints.

(4) $\tau_3 = \tau_4$, (τ_3 `->` τ_3) `->` τ_3 `->` $\tau_4 =$ (`int * int -> int`) `->` τ_5

Example 2 (continued): We use unification to process the constraints for N:

(1) $\tau_1 = \tau_2 \texttt{ -> } \tau_3, \quad \tau_1 = \tau_3 \texttt{ -> } \tau_4, \quad \tau_1 \texttt{ -> } \tau_2 \texttt{ -> } \tau_4 = (\texttt{int * int -> int}) \texttt{ -> } \tau_5$

Define small substitution $\tau_1 := \tau_2 \texttt{ -> } \tau_3$ and apply to remaining constraints.

(2) $\tau_2 \texttt{ -> } \tau_3 = \tau_3 \texttt{ -> } \tau_4, \quad (\tau_2 \texttt{ -> } \tau_3) \texttt{ -> } \tau_2 \texttt{ -> } \tau_4 = (\texttt{int * int -> int}) \texttt{ -> } \tau_5$

Simplify.

(3) $\tau_2 = \tau_3, \quad \tau_3 = \tau_4, \quad (\tau_2 \texttt{ -> } \tau_3) \texttt{ -> } \tau_2 \texttt{ -> } \tau_4 = (\texttt{int * int -> int}) \texttt{ -> } \tau_5$

Define small substitution $\tau_2 := \tau_3$ and apply to remaining constraints.

(4) $\tau_3 = \tau_4, \quad (\tau_3 \texttt{ -> } \tau_3) \texttt{ -> } \tau_3 \texttt{ -> } \tau_4 = (\texttt{int * int -> int}) \texttt{ -> } \tau_5$

Define small substitution $\tau_3 := \tau_4$ and apply to remaining constraints.

(5) $(\tau_4 \texttt{ -> } \tau_4) \texttt{ -> } \tau_4 \texttt{ -> } \tau_4 = (\texttt{int * int -> int}) \texttt{ -> } \tau_5$

Example 2 (continued): We use unification to process the constraints for N:

(1) $\tau_1 = \tau_2$ `->` $\tau_3, \quad \tau_1 = \tau_3$ `->` $\tau_4, \quad \tau_1$ `->` τ_2 `->` $\tau_4 =$ (`int * int -> int`) `->` τ_5

Define small substitution $\tau_1 := \tau_2$ `->` τ_3 and apply to remaining constraints.

(2) τ_2 `->` $\tau_3 = \tau_3$ `->` $\tau_4, \quad (\tau_2$ `->` $\tau_3)$ `->` τ_2 `->` $\tau_4 =$ (`int * int -> int`) `->` τ_5

Simplify.

(3) $\tau_2 = \tau_3, \quad \tau_3 = \tau_4, \quad (\tau_2$ `->` $\tau_3)$ `->` τ_2 `->` $\tau_4 =$ (`int * int -> int`) `->` τ_5

Define small substitution $\tau_2 := \tau_3$ and apply to remaining constraints.

(4) $\tau_3 = \tau_4, \quad (\tau_3$ `->` $\tau_3)$ `->` τ_3 `->` $\tau_4 =$ (`int * int -> int`) `->` τ_5

Define small substitution $\tau_3 := \tau_4$ and apply to remaining constraints.

(5) $(\tau_4$ `->` $\tau_4)$ `->` τ_4 `->` $\tau_4 =$ (`int * int -> int`) `->` τ_5

Simplify.

(6) τ_4 `->` $\tau_4 =$ `int * int -> int`, $\quad \tau_4$ `->` $\tau_4 = \tau_5$

Example 2 (continued): We use unification to process the constraints for N:

(1) $\tau_1 = \tau_2$ `->` $\tau_3, \quad \tau_1 = \tau_3$ `->` $\tau_4, \quad \tau_1$ `->` τ_2 `->` $\tau_4 =$ (`int * int -> int`) `->` τ_5

Define small substitution $\tau_1 := \tau_2$ `->` τ_3 and apply to remaining constraints.

(2) τ_2 `->` $\tau_3 = \tau_3$ `->` $\tau_4, \quad (\tau_2$ `->` $\tau_3)$ `->` τ_2 `->` $\tau_4 =$ (`int * int -> int`) `->` τ_5

Simplify.

(3) $\tau_2 = \tau_3, \quad \tau_3 = \tau_4, \quad (\tau_2$ `->` $\tau_3)$ `->` τ_2 `->` $\tau_4 =$ (`int * int -> int`) `->` τ_5

Define small substitution $\tau_2 := \tau_3$ and apply to remaining constraints.

(4) $\tau_3 = \tau_4, \quad (\tau_3$ `->` $\tau_3)$ `->` τ_3 `->` $\tau_4 =$ (`int * int -> int`) `->` τ_5

Define small substitution $\tau_3 := \tau_4$ and apply to remaining constraints.

(5) $(\tau_4$ `->` $\tau_4)$ `->` τ_4 `->` $\tau_4 =$ (`int * int -> int`) `->` τ_5

Simplify.

(6) τ_4 `->` $\tau_4 =$ `int * int -> int`, $\quad \tau_4$ `->` $\tau_4 = \tau_5$

Simplify.

(7) $\tau_4 =$ `int * int`, $\quad \tau_4 =$ `int`, $\quad \tau_4$ `->` $\tau_4 = \tau_5$

<u>**Example 2 (continued)**</u>: We use unification to process the constraints for N:

(1) $\tau_1 = \tau_2$ `->` $\tau_3,\quad \tau_1 = \tau_3$ `->` $\tau_4,\quad \tau_1$ `->` τ_2 `->` $\tau_4 =$ (`int * int -> int`) `->` τ_5

Define small substitution $\tau_1 := \tau_2$ `->` τ_3 and apply to remaining constraints.

(2) τ_2 `->` $\tau_3 = \tau_3$ `->` $\tau_4,\quad (\tau_2$ `->` $\tau_3)$ `->` τ_2 `->` $\tau_4 =$ (`int * int -> int`) `->` τ_5

Simplify.

(3) $\tau_2 = \tau_3,\quad \tau_3 = \tau_4,\quad (\tau_2$ `->` $\tau_3)$ `->` τ_2 `->` $\tau_4 =$ (`int * int -> int`) `->` τ_5

Define small substitution $\tau_2 := \tau_3$ and apply to remaining constraints.

(4) $\tau_3 = \tau_4,\quad (\tau_3$ `->` $\tau_3)$ `->` τ_3 `->` $\tau_4 =$ (`int * int -> int`) `->` τ_5

Define small substitution $\tau_3 := \tau_4$ and apply to remaining constraints.

(5) $(\tau_4$ `->` $\tau_4)$ `->` τ_4 `->` $\tau_4 =$ (`int * int -> int`) `->` τ_5

Simplify.

(6) τ_4 `->` $\tau_4 =$ `int * int -> int`, $\quad \tau_4$ `->` $\tau_4 = \tau_5$

Simplify.

(7) $\tau_4 =$ `int * int`, $\quad \tau_4 =$ `int`, $\quad \tau_4$ `->` $\tau_4 = \tau_5$

Define small substitution $\tau_4 :=$ `int * int` and apply to remaining constraints.

(8) `int * int` $=$ `int`, $\quad$ `int * int -> int * int` $= \tau_5$

Example 2 (continued): We use unification to process the constraints for N:

(1) $\tau_1 = \tau_2$ `->` τ_3, $\tau_1 = \tau_3$ `->` τ_4, τ_1 `->` τ_2 `->` $\tau_4 =$ `(int * int -> int) ->` τ_5

Define small substitution $\tau_1 := \tau_2$ `->` τ_3 and apply to remaining constraints.

(2) τ_2 `->` $\tau_3 = \tau_3$ `->` τ_4, $(\tau_2$ `->` $\tau_3)$ `->` τ_2 `->` $\tau_4 =$ `(int * int -> int) ->` τ_5

Simplify.

(3) $\tau_2 = \tau_3$, $\tau_3 = \tau_4$, $(\tau_2$ `->` $\tau_3)$ `->` τ_2 `->` $\tau_4 =$ `(int * int -> int) ->` τ_5

Define small substitution $\tau_2 := \tau_3$ and apply to remaining constraints.

(4) $\tau_3 = \tau_4$, $(\tau_3$ `->` $\tau_3)$ `->` τ_3 `->` $\tau_4 =$ `(int * int -> int) ->` τ_5

Define small substitution $\tau_3 := \tau_4$ and apply to remaining constraints.

(5) $(\tau_4$ `->` $\tau_4)$ `->` τ_4 `->` $\tau_4 =$ `(int * int -> int) ->` τ_5

Simplify.

(6) τ_4 `->` $\tau_4 =$ `int * int -> int`, τ_4 `->` $\tau_4 = \tau_5$

Simplify.

(7) $\tau_4 =$ `int * int`, $\tau_4 =$ `int`, τ_4 `->` $\tau_4 = \tau_5$

Define small substitution $\tau_4 :=$ `int * int` and apply to remaining constraints.

(8) `int * int` $=$ `int`, `int * int -> int * int` $= \tau_5$

Contradiction: the top constraint `int * int` $=$ `int` cannot be satisfied.
There is no solution and N is not typable.