Monomorphic Type Inference Through Unification

"how to systematically remove the guess-work from type-inference"

Assaf Kfoury

3 November 2016

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: $M = \backslash$ `f -> \ x -> f (f x)`

This expression is well-formed in Haskell. This is confirmed by running a Haskell interpreter on the expression.

Example 1: $M = \backslash$ `f -> \ x -> f (f x)`

This expression is well-formed in Haskell. This is confirmed by running a Haskell interpreter on the expression.

A skeleton for a typing derivation for M, with unknown (monomorphic) types inserted, can be built incrementally in top-down fashion, based on a parsing of the expression. We say "skeleton" because we enter names of unknown types $t_1, t_2, t_3, \ldots$ into the derivation, together with constraints (in boxes) between these unknown types.

1. $\texttt{f} : t_1 \qquad \vdash \texttt{f} : t_1 \qquad\qquad\qquad\qquad\qquad\qquad$ VAR

Example 1: $M = \backslash \texttt{f} \texttt{->} \backslash \texttt{x} \texttt{->} \texttt{f}\,(\texttt{f}\,\texttt{x})$

This expression is well-formed in Haskell. This is confirmed by running a Haskell interpreter on the expression.

A skeleton for a typing derivation for M, with unknown (monomorphic) types inserted, can be built incrementally in top-down fashion, based on a parsing of the expression. We say "skeleton" because we enter names of unknown types $t_1, t_2, t_3, \ldots$ into the derivation, together with constraints (in boxes) between these unknown types.

1. $\texttt{f} : t_1 \qquad \vdash \texttt{f} : t_1 \qquad\qquad\qquad\qquad$ VAR
2. $\texttt{x} : t_2 \qquad \vdash \texttt{x} : t_2 \qquad\qquad\qquad\qquad$ VAR

Example 1: $M = \backslash \texttt{f} \texttt{->} \backslash \texttt{x} \texttt{->} \texttt{f}\,(\texttt{f}\,\texttt{x})$

This expression is well-formed in Haskell. This is confirmed by running a Haskell interpreter on the expression.

A skeleton for a typing derivation for M, with unknown (monomorphic) types inserted, can be built incrementally in top-down fashion, based on a parsing of the expression. We say "skeleton" because we enter names of unknown types $t_1, t_2, t_3, \ldots$ into the derivation, together with constraints (in boxes) between these unknown types.

1.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{f} : t_1$		VAR
2.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{x} : t_2$		VAR
3.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{f}\,\texttt{x} : t_3$	$\boxed{t_1 = t_2 \texttt{->} t_3}$	APP (1,2)

Example 1: $M = \backslash \texttt{f} \texttt{->} \backslash \texttt{x} \texttt{->} \texttt{f}\,(\texttt{f}\,\texttt{x})$

This expression is well-formed in Haskell. This is confirmed by running a Haskell interpreter on the expression.

A skeleton for a typing derivation for M, with unknown (monomorphic) types inserted, can be built incrementally in top-down fashion, based on a parsing of the expression. We say "skeleton" because we enter names of unknown types $t_1, t_2, t_3, \ldots$ into the derivation, together with constraints (in boxes) between these unknown types.

1.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{f} : t_1$		VAR
2.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{x} : t_2$		VAR
3.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{f}\,\texttt{x} : t_3$	$\boxed{t_1 = t_2 \texttt{->} t_3}$	APP (1,2)
4.	$\texttt{f} : t_1 \vdash \texttt{f} : t_1$		VAR

Example 1: $M = \backslash \texttt{f} \texttt{ -> } \backslash \texttt{x} \texttt{ -> f (f x)}$

This expression is well-formed in Haskell. This is confirmed by running a Haskell interpreter on the expression.

A skeleton for a typing derivation for M, with unknown (monomorphic) types inserted, can be built incrementally in top-down fashion, based on a parsing of the expression. We say "skeleton" because we enter names of unknown types $t_1, t_2, t_3, \ldots$ into the derivation, together with constraints (in boxes) between these unknown types.

1.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{f} : t_1$		VAR
2.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{x} : t_2$		VAR
3.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{f x} : t_3$	$\boxed{t_1 = t_2 \texttt{ -> } t_3}$	APP (1,2)
4.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{f} : t_1$		VAR
5.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{f (f x)} : t_4$	$\boxed{t_1 = t_3 \texttt{ -> } t_4}$	APP (3,4)

Example 1: $M = \backslash \texttt{f} \texttt{ -> } \backslash \texttt{x} \texttt{ -> } \texttt{f (f x)}$

This expression is well-formed in Haskell. This is confirmed by running a Haskell interpreter on the expression.

A skeleton for a typing derivation for M, with unknown (monomorphic) types inserted, can be built incrementally in top-down fashion, based on a parsing of the expression. We say "skeleton" because we enter names of unknown types $t_1, t_2, t_3, \ldots$ into the derivation, together with constraints (in boxes) between these unknown types.

1.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{f} : t_1$			VAR
2.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{x} : t_2$			VAR
3.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{f x} : t_3$	$\boxed{t_1 = t_2 \texttt{ -> } t_3}$		APP (1,2)
4.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{f} : t_1$			VAR
5.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{f (f x)} : t_4$	$\boxed{t_1 = t_3 \texttt{ -> } t_4}$		APP (3,4)
6.	$\texttt{f} : t_1 \quad \vdash \backslash \texttt{x} \texttt{ -> } \texttt{f (f x)} : t_2 \texttt{ -> } t_4$			ABS (5)

Example 1: $M = \backslash \texttt{f} \texttt{->} \backslash \texttt{x} \texttt{->} \texttt{f}\,(\texttt{f}\,\texttt{x})$

This expression is well-formed in Haskell. This is confirmed by running a Haskell interpreter on the expression.

A skeleton for a typing derivation for M, with unknown (monomorphic) types inserted, can be built incrementally in top-down fashion, based on a parsing of the expression. We say "skeleton" because we enter names of unknown types $t_1, t_2, t_3, \ldots$ into the derivation, together with constraints (in boxes) between these unknown types.

1.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{f} : t_1$		VAR
2.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{x} : t_2$		VAR
3.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{f}\,\texttt{x} : t_3$	$\boxed{t_1 = t_2 \texttt{->} t_3}$	APP (1,2)
4.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{f} : t_1$		VAR
5.	$\texttt{f} : t_1, \texttt{x} : t_2 \vdash \texttt{f}\,(\texttt{f}\,\texttt{x}) : t_4$	$\boxed{t_1 = t_3 \texttt{->} t_4}$	APP (3,4)
6.	$\texttt{f} : t_1 \vdash \backslash \texttt{x} \texttt{->} \texttt{f}\,(\texttt{f}\,\texttt{x}) : t_2 \texttt{->} t_4$		ABS (5)
7.	$\vdash \backslash \texttt{f} \texttt{->} \backslash \texttt{x} \texttt{->} \texttt{f}\,(\texttt{f}\,\texttt{x}) : t_1 \texttt{->} t_2 \texttt{->} t_4$		ABS-2 (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) $\boxed{t_1 = t_2 \texttt{ -> } t_3, \quad t_1 = t_3 \texttt{ -> } t_4}$

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

(1) $\boxed{t_1 = t_2 \texttt{ -> } t_3, \quad t_1 = t_3 \texttt{ -> } t_4}$

From the top constraint $t_1 = t_2 \texttt{ -> } t_3$ in (1), we define the small substitution:

$$t_1 := t_2 \texttt{ -> } t_3$$

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

(2) $\boxed{t_2 \texttt{ -> } t_3 = t_3 \texttt{ -> } t_4}$

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

(1) $\boxed{t_1 = t_2 \texttt{ -> } t_3, \quad t_1 = t_3 \texttt{ -> } t_4}$

From the top constraint $t_1 = t_2 \texttt{ -> } t_3$ in (1), we define the small substitution:

$$t_1 := t_2 \texttt{ -> } t_3$$

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

(2) $\boxed{t_2 \texttt{ -> } t_3 = t_3 \texttt{ -> } t_4}$

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

(3) $\boxed{t_2 = t_3, \quad t_3 = t_4}$

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

(1) $\boxed{t_1 = t_2 \texttt{ -> } t_3, \quad t_1 = t_3 \texttt{ -> } t_4}$

From the top constraint $t_1 = t_2 \texttt{ -> } t_3$ in (1), we define the small substitution:

$$t_1 := t_2 \texttt{ -> } t_3$$

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

(2) $\boxed{t_2 \texttt{ -> } t_3 = t_3 \texttt{ -> } t_4}$

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

(3) $\boxed{t_2 = t_3, \quad t_3 = t_4}$

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

$$t_2 := t_3$$

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

(4) $\boxed{t_3 = t_4}$

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

(1) $\boxed{t_1 = t_2 \texttt{ -> } t_3, \quad t_1 = t_3 \texttt{ -> } t_4}$

From the top constraint $t_1 = t_2 \texttt{ -> } t_3$ in (1), we define the small substitution:

$$t_1 := t_2 \texttt{ -> } t_3$$

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

(2) $\boxed{t_2 \texttt{ -> } t_3 = t_3 \texttt{ -> } t_4}$

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

(3) $\boxed{t_2 = t_3, \quad t_3 = t_4}$

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

$$t_2 := t_3$$

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

(4) $\boxed{t_3 = t_4}$

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

$$t_3 := t_4$$

and apply it to obtain the empty sequence:

(5) $\boxed{\varnothing}$

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

(1) $\boxed{t_1 = t_2 \texttt{ -> } t_3, \quad t_1 = t_3 \texttt{ -> } t_4}$

From the top constraint $t_1 = t_2 \texttt{ -> } t_3$ in (1), we define the small substitution:

$$t_1 := t_2 \texttt{ -> } t_3$$

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

(2) $\boxed{t_2 \texttt{ -> } t_3 = t_3 \texttt{ -> } t_4}$

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

(3) $\boxed{t_2 = t_3, \quad t_3 = t_4}$

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

$$t_2 := t_3$$

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

(4) $\boxed{t_3 = t_4}$

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

$$t_3 := t_4$$

and apply it to obtain the empty sequence:

(5) $\boxed{\varnothing}$

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**
$\{t_1 := t_2$ -> $t_3\}$	$\{t_1 := t_2$ -> $t_3\}$

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**
$\{t_1 := t_2 \texttt{ -> } t_3\}$	$\{t_1 := t_2 \texttt{ -> } t_3\}$
$\{t_1 := t_2 \texttt{ -> } t_3\}$ $\{t_2 := t_3\}$	$\{t_1 := t_3 \texttt{ -> } t_3, \quad t_2 := t_3\}$

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**
$\{t_1 := t_2 \texttt{ -> } t_3\}$	$\{t_1 := t_2 \texttt{ -> } t_3\}$
$\{t_1 := t_2 \texttt{ -> } t_3\}$ $\{t_2 := t_3\}$	$\{t_1 := t_3 \texttt{ -> } t_3, \quad t_2 := t_3\}$
$\{t_1 := t_2 \texttt{ -> } t_3\}$ $\{t_2 := t_3\}$ $\{t_3 := t_4\}$	$\{t_1 := t_4 \texttt{ -> } t_4, \quad t_2 := t_4, \quad t_3 := t_4\}$

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**
$\{t_1 := t_2$ `->` $t_3\}$	$\{t_1 := t_2$ `->` $t_3\}$
$\{t_1 := t_2$ `->` $t_3\}$ $\{t_2 := t_3\}$	$\{t_1 := t_3$ `->` $t_3, \quad t_2 := t_3\}$
$\{t_1 := t_2$ `->` $t_3\}$ $\{t_2 := t_3\}$ $\{t_3 := t_4\}$	$\{t_1 := t_4$ `->` $t_4, \quad t_2 := t_4, \quad t_3 := t_4\}$

In the final large substitution, the types t_1, t_2 and t_3 are defined in terms of t_4, while t_4 is totally unrestricted. To make clear that t_4 is unrestricted, let us substitute a fresh type variable `a` for t_4. So, we can express the final large substitution, call it S, as follows:

$$S = \{t_1 := \texttt{a} \texttt{ -> } \texttt{a}, \quad t_2 := \texttt{a}, \quad t_3 := \texttt{a}, \quad t_4 := \texttt{a}\}$$

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 : a -> a, x : a`	$\vdash$ `f : a -> a`	VAR
2.	`f : a -> a, x : a`	$\vdash$ `x : a`	VAR
3.	`f : a -> a, x : a`	$\vdash$ `f x : a`	APP (1,2)
4.	`f : a -> a, x : a`	$\vdash$ `f : a -> a`	VAR
5.	`f : a -> a, x : a`	$\vdash$ `f (f x) : a`	APP (3,4)
6.	`f : a -> a`	$\vdash$ `\ x -> f (f x) : a -> a`	ABS (5)
7.		$\vdash$ `\ f -> \ x -> f (f x) : (a -> a) -> a -> a`	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 : a -> a, x : a` $\vdash$ `f : a -> a`	VAR
2.	`f : a -> a, x : a` $\vdash$ `x : a`	VAR
3.	`f : a -> a, x : a` $\vdash$ `f x : a`	APP (1,2)
4.	`f : a -> a, x : a` $\vdash$ `f : a -> a`	VAR
5.	`f : a -> a, x : a` $\vdash$ `f (f x) : a`	APP (3,4)
6.	`f : a -> a` $\vdash$ `\ x -> f (f x) : a -> a`	ABS (5)
7.	$\vdash$ `\ f -> \ x -> f (f x) : (a -> a) -> a -> a`	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 t_1 `->` t_2 `->` t_4 in the skeleton for M:

`(a -> a) -> a -> a`

which is precisely the type returned by a Haskell interpreter for M.

Example 2: In Haskell we can program `plus` as the function `plus = (\ x -> \ y -> x + y)` with type `Int -> Int -> Int`. We omit below how this type is derived.

We now apply the expression M of Example 1 to `plus` and call the resulting application N, i.e.,

$$N \;=\; (\backslash\, \texttt{f -> } \backslash\, \texttt{x -> f (f x))}\, \texttt{plus}$$

Example 2: In Haskell we can program `plus` as the function `plus = (\ x -> \ y -> x + y)` with type `Int -> Int -> Int`. We omit below how this type is derived.

We now apply the expression M of Example 1 to `plus` and call the resulting application N, i.e.,

$$N \;=\; (\backslash\,\texttt{f -> } \backslash\,\texttt{x -> f (f x))}\,\texttt{plus}$$

We do not go through all the details of Example 1 here: We simply extend the skeleton for M with two judgements and one constraint, namely:

8. $\vdash$ `plus : Int -> Int -> Int`
9. $\vdash$ $(\backslash\,\texttt{f -> } \backslash\,\texttt{x -> f (f x))}\,\texttt{plus} : t_5$ $\boxed{t_1 \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$ APP (7,8)

Example 2: In Haskell we can program `plus` as the function `plus = (\ x -> \ y -> x + y)` with type `Int -> Int -> Int`. We omit below how this type is derived.

We now apply the expression M of Example 1 to `plus` and call the resulting application N, i.e.,

$$N \;=\; (\backslash\, \texttt{f} \texttt{ -> } \backslash\, \texttt{x} \texttt{ -> } \texttt{f}\,(\texttt{f}\,\texttt{x}))\,\texttt{plus}$$

We do not go through all the details of Example 1 here: We simply extend the skeleton for M with two judgements and one constraint, namely:

8. $\vdash$ `plus : Int -> Int -> Int`
9. $\vdash$ $(\backslash\, \texttt{f} \texttt{ -> } \backslash\, \texttt{x} \texttt{ -> } \texttt{f}\,(\texttt{f}\,\texttt{x}))\,\texttt{plus} : t_5$ $\boxed{t_1 \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$ APP (7,8)

Note that t_1 `->` t_2 `->` t_4, on the left-hand side of the new constraint, is the type derived for M when we built a skeleton for it, at the end of Example 1.

The initial sequence of constraints for N is therefore:

$$\boxed{t_1 = t_2 \texttt{ -> } t_3, \quad t_1 = t_3 \texttt{ -> } t_4, \quad t_1 \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$$

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) $\boxed{t_1 = t_2 \texttt{ -> } t_3, \quad t_1 = t_3 \texttt{ -> } t_4, \quad t_1 \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

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

(1) $\boxed{t_1 = t_2 \texttt{ -> } t_3, \quad t_1 = t_3 \texttt{ -> } t_4, \quad t_1 \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Define small substitution $t_1 := t_2 \texttt{ -> } t_3$ and apply to remaining constraints:

(2) $\boxed{t_2 \texttt{ -> } t_3 = t_3 \texttt{ -> } t_4, \quad (t_2 \texttt{ -> } t_3) \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

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

(1) $\boxed{t_1 = t_2 \texttt{ -> } t_3, \quad t_1 = t_3 \texttt{ -> } t_4, \quad t_1 \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Define small substitution $t_1 := t_2 \texttt{ -> } t_3$ and apply to remaining constraints:

(2) $\boxed{t_2 \texttt{ -> } t_3 = t_3 \texttt{ -> } t_4, \quad (t_2 \texttt{ -> } t_3) \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Simplify:

(3) $\boxed{t_2 = t_3, \quad t_3 = t_4, \quad (t_2 \texttt{ -> } t_3) \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

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

(1) $\boxed{t_1 = t_2 \texttt{ -> } t_3, \quad t_1 = t_3 \texttt{ -> } t_4, \quad t_1 \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Define small substitution $t_1 := t_2 \texttt{ -> } t_3$ and apply to remaining constraints:

(2) $\boxed{t_2 \texttt{ -> } t_3 = t_3 \texttt{ -> } t_4, \quad (t_2 \texttt{ -> } t_3) \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Simplify:

(3) $\boxed{t_2 = t_3, \quad t_3 = t_4, \quad (t_2 \texttt{ -> } t_3) \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Define small substitution $t_2 := t_3$ and apply to remaining constraints:

(4) $\boxed{t_3 = t_4, \quad (t_3 \texttt{ -> } t_3) \texttt{ -> } t_3 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

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

(1) $\boxed{t_1 = t_2 \texttt{ -> } t_3, \quad t_1 = t_3 \texttt{ -> } t_4, \quad t_1 \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Define small substitution $t_1 := t_2 \texttt{ -> } t_3$ and apply to remaining constraints:

(2) $\boxed{t_2 \texttt{ -> } t_3 = t_3 \texttt{ -> } t_4, \quad (t_2 \texttt{ -> } t_3) \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Simplify:

(3) $\boxed{t_2 = t_3, \quad t_3 = t_4, \quad (t_2 \texttt{ -> } t_3) \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Define small substitution $t_2 := t_3$ and apply to remaining constraints:

(4) $\boxed{t_3 = t_4, \quad (t_3 \texttt{ -> } t_3) \texttt{ -> } t_3 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Define small substitution $t_3 := t_4$ and apply to remaining constraints:

(5) $\boxed{(t_4 \texttt{ -> } t_4) \texttt{ -> } t_4 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

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

(1) $\boxed{t_1 = t_2 \texttt{ -> } t_3, \quad t_1 = t_3 \texttt{ -> } t_4, \quad t_1 \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Define small substitution $t_1 := t_2 \texttt{ -> } t_3$ and apply to remaining constraints:

(2) $\boxed{t_2 \texttt{ -> } t_3 = t_3 \texttt{ -> } t_4, \quad (t_2 \texttt{ -> } t_3) \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Simplify:

(3) $\boxed{t_2 = t_3, \quad t_3 = t_4, \quad (t_2 \texttt{ -> } t_3) \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Define small substitution $t_2 := t_3$ and apply to remaining constraints:

(4) $\boxed{t_3 = t_4, \quad (t_3 \texttt{ -> } t_3) \texttt{ -> } t_3 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Define small substitution $t_3 := t_4$ and apply to remaining constraints:

(5) $\boxed{(t_4 \texttt{ -> } t_4) \texttt{ -> } t_4 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Simplify:

(6) $\boxed{t_4 \texttt{ -> } t_4 = \texttt{Int -> Int -> Int}, \quad t_4 \texttt{ -> } t_4 = t_5}$

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

(1) $\boxed{t_1 = t_2 \texttt{ -> } t_3, \quad t_1 = t_3 \texttt{ -> } t_4, \quad t_1 \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Define small substitution $t_1 := t_2 \texttt{ -> } t_3$ and apply to remaining constraints:

(2) $\boxed{t_2 \texttt{ -> } t_3 = t_3 \texttt{ -> } t_4, \quad (t_2 \texttt{ -> } t_3) \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Simplify:

(3) $\boxed{t_2 = t_3, \quad t_3 = t_4, \quad (t_2 \texttt{ -> } t_3) \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Define small substitution $t_2 := t_3$ and apply to remaining constraints:

(4) $\boxed{t_3 = t_4, \quad (t_3 \texttt{ -> } t_3) \texttt{ -> } t_3 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Define small substitution $t_3 := t_4$ and apply to remaining constraints:

(5) $\boxed{(t_4 \texttt{ -> } t_4) \texttt{ -> } t_4 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Simplify:

(6) $\boxed{t_4 \texttt{ -> } t_4 = \texttt{Int -> Int -> Int}, \quad t_4 \texttt{ -> } t_4 = t_5}$

Simplify:

(7) $\boxed{t_4 = \texttt{Int -> Int}, \quad t_4 = \texttt{Int}, \quad t_4 \texttt{ -> } t_4 = t_5}$

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

(1) $\boxed{t_1 = t_2 \texttt{ -> } t_3, \quad t_1 = t_3 \texttt{ -> } t_4, \quad t_1 \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Define small substitution $t_1 := t_2 \texttt{ -> } t_3$ and apply to remaining constraints:

(2) $\boxed{t_2 \texttt{ -> } t_3 = t_3 \texttt{ -> } t_4, \quad (t_2 \texttt{ -> } t_3) \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Simplify:

(3) $\boxed{t_2 = t_3, \quad t_3 = t_4, \quad (t_2 \texttt{ -> } t_3) \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Define small substitution $t_2 := t_3$ and apply to remaining constraints:

(4) $\boxed{t_3 = t_4, \quad (t_3 \texttt{ -> } t_3) \texttt{ -> } t_3 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Define small substitution $t_3 := t_4$ and apply to remaining constraints:

(5) $\boxed{(t_4 \texttt{ -> } t_4) \texttt{ -> } t_4 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Simplify:

(6) $\boxed{t_4 \texttt{ -> } t_4 = \texttt{Int -> Int -> Int}, \quad t_4 \texttt{ -> } t_4 = t_5}$

Simplify:

(7) $\boxed{t_4 = \texttt{Int -> Int}, \quad t_4 = \texttt{Int}, \quad t_4 \texttt{ -> } t_4 = t_5}$

Define small substitution $t_4 := \texttt{Int -> Int}$ and apply to remaining constraints:

(8) $\boxed{\texttt{Int -> Int} = \texttt{Int}, \quad (\texttt{Int -> Int}) \texttt{ -> Int -> Int} = t_5}$

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

(1) $\boxed{t_1 = t_2 \texttt{ -> } t_3, \quad t_1 = t_3 \texttt{ -> } t_4, \quad t_1 \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Define small substitution $t_1 := t_2 \texttt{ -> } t_3$ and apply to remaining constraints:

(2) $\boxed{t_2 \texttt{ -> } t_3 = t_3 \texttt{ -> } t_4, \quad (t_2 \texttt{ -> } t_3) \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Simplify:

(3) $\boxed{t_2 = t_3, \quad t_3 = t_4, \quad (t_2 \texttt{ -> } t_3) \texttt{ -> } t_2 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Define small substitution $t_2 := t_3$ and apply to remaining constraints:

(4) $\boxed{t_3 = t_4, \quad (t_3 \texttt{ -> } t_3) \texttt{ -> } t_3 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Define small substitution $t_3 := t_4$ and apply to remaining constraints:

(5) $\boxed{(t_4 \texttt{ -> } t_4) \texttt{ -> } t_4 \texttt{ -> } t_4 = (\texttt{Int -> Int -> Int}) \texttt{ -> } t_5}$

Simplify:

(6) $\boxed{t_4 \texttt{ -> } t_4 = \texttt{Int -> Int -> Int}, \quad t_4 \texttt{ -> } t_4 = t_5}$

Simplify:

(7) $\boxed{t_4 = \texttt{Int -> Int}, \quad t_4 = \texttt{Int}, \quad t_4 \texttt{ -> } t_4 = t_5}$

Define small substitution $t_4 := \texttt{Int -> Int}$ and apply to remaining constraints:

(8) $\boxed{\texttt{Int -> Int} = \texttt{Int}, \quad (\texttt{Int -> Int}) \texttt{ -> Int -> Int} = t_5}$

Contradiction: the top (i.e. leftmost above) constraint $\texttt{Int -> Int} = \texttt{Int}$ cannot be satisfied. **There is no solution and N is not typable**.