

Jane Street

Type Inference in OCaml and GHC using Levels

Richard A. Eisenberg

Jane Street

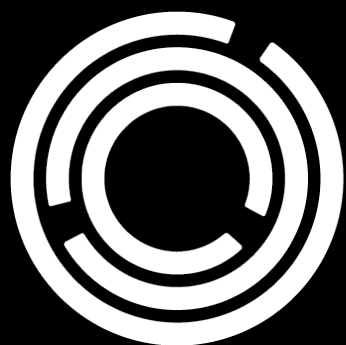
reisenberg@janestreet.com

Saturday, January 25, 2025

WITS

Denver, CO, USA

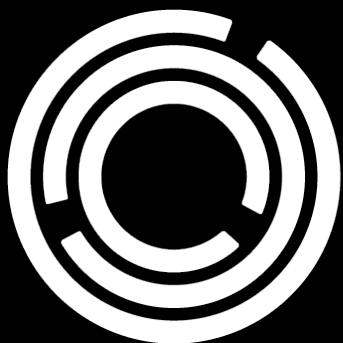
Levels are an old idea:
Rémy (1992) called them ranks.



Levels are an old idea:
Rémy (1992) called them ranks.

Structure of this talk:

- Introduce levels
- Use in OCaml
- Use in GHC



Generalization

$$\text{id } x_a = x_a$$



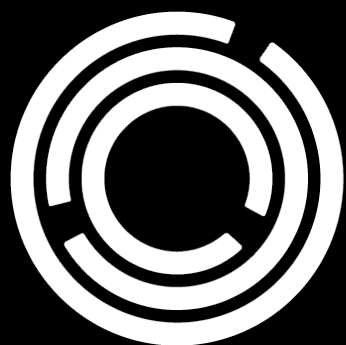
Generalization

$$\begin{array}{ccc} \text{id} & & x = x \\ a \rightarrow a & & a \\ \forall a. a \rightarrow a & & a \\ \text{inferred} & & \text{inferring} \end{array}$$



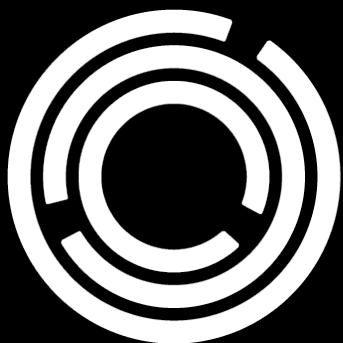
Generalization

swub $x\ y = (x, \text{not } y)$



Generalization

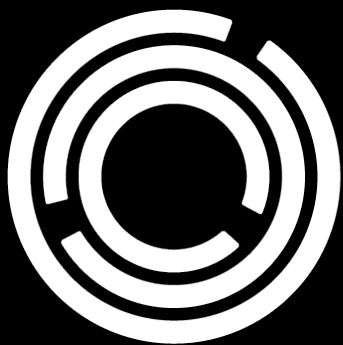
$$\begin{aligned}
 & \text{swub } \alpha \rightarrow \text{Bool} \rightarrow x_{\alpha} \ y_{\beta \text{Bool}} \\
 & \quad \alpha \times \text{Bool} \\
 & = (x_{\alpha}, \\
 & \quad \text{not Bool} \rightarrow y_{\beta \text{Bool}} \\
 & \quad \text{Bool} \\
 & \quad)_{\alpha \times \text{Bool}}
 \end{aligned}$$



Generalization

swub $\alpha \rightarrow \text{Bool} \rightarrow \quad x_\alpha \quad y \text{ Bool}$
 $\alpha \times \text{Bool}$

swub : $\forall a. \quad a \rightarrow \text{Bool}$
 $\rightarrow a \times \text{Bool}$



Generalization

```
frob x w =  
  let mk y z = ([x; y], z) in  
  (mk w 3, mk w 'z')
```



Generalization

frob $_{\alpha \rightarrow \delta \rightarrow (\text{List } \delta \times \mathbb{I})}$ x_{α} w_{δ} =
 $\times (\text{List } \delta \times \mathbb{C})$

let mk $_{\alpha \rightarrow \gamma \rightarrow \text{List } a \times \gamma}$ y_{β} z_{γ} =
 $\forall a b. a \rightarrow b \rightarrow \text{List } a \times b$

($[x_{\alpha} ; y_{\beta}] \text{List } a , z_{\gamma}$

) $\text{List } a \times \gamma$ in

(mk $_{\delta \rightarrow \mathbb{I} \rightarrow \text{List } \delta \times \mathbb{I}}$ w_{δ} \mathbb{I} ,
 mk $_{\delta \rightarrow \mathbb{C} \rightarrow \text{List } \delta \times \mathbb{C}}$ w_{δ} $z_{\mathbb{C}}$)



Generalization

but x and w are in
a list together!

frob x w =

let mk y z = $([x; y], z)$ in
 $(mk$ w 3 , mk w $'z'$)

$\alpha \rightarrow \delta \rightarrow (\text{List } \delta \times I)$
 $\times (\text{List } \delta \times C)$



Don't generalize variables
that are already in scope.



Generalization

frob $\alpha \rightarrow \alpha \rightarrow (\text{List } \alpha \times \text{I})$
 $\times (\text{List } \alpha \times \text{C})$

$\forall a. a \rightarrow a \rightarrow (\text{List } a \times \text{I})$
 $\times (\text{List } a \times \text{C})$



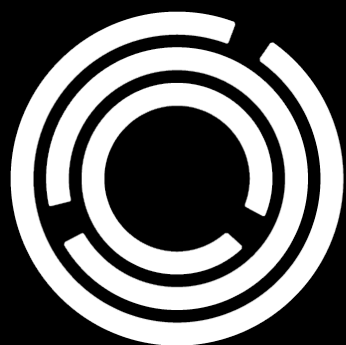
$$\text{LET}' \quad \frac{A \vdash e' : \tau' \quad A_x \cup \{x : \sigma\} \vdash e : \tau}{A \vdash \text{let } x = e' \text{ in } e : \tau}$$

Slow to compute when
the context is big.

where $\text{gen}(A, \tau)$ is defined by

$$\text{gen}(A, \tau) = \begin{cases} \forall \alpha_1 \cdots \alpha_n. \tau & \text{if } FV(\tau) \setminus FV(A) = \{\alpha_1 \cdots \alpha_n\} \\ \tau & \text{if } FV(\tau) \setminus FV(A) = \emptyset \end{cases}$$

Use levels instead.



D. Clément, T. Despeyroux, G. Kahn, and J. Despeyroux.
A simple applicative language: mini-ML. LFP '86

Oleg Kiselyov:
Generalization by levels
echoes avoiding use-after-
free errors in memory
management.



[https://okmij.org/ftp/ML/
generalization.html](https://okmij.org/ftp/ML/generalization.html)

Types Levels in OCaml



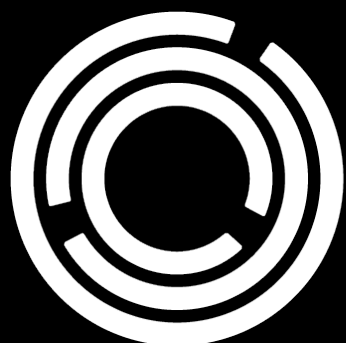
Types are graphs.

```
(* Type expressions for the core language *)
```

```
type transient_expr =  
  { mutable desc: type_desc;  
    mutable level: int;  
    mutable scope: scope_field;  
    id: int }
```

```
and type_expr = transient_expr
```

```
and type_desc =  
  Tvar of string option  
  | Tarrow of arg_label * type_expr * type_expr * commutable  
  | Ttuple of type_expr list
```



```
mutable scope! scope_field;  
id: int }
```

and type_expr = transient_expr

and type_desc =
Tvar of string option

no unique id on Tvar!

use pointer equality on the

 enclosing type_expr

```
(* Type expressions for the core language *)
```

```
type transient_expr =  
  { mutable desc: type_desc;  
    mutable level: int;  
    mutable scope: scope_field;  
    id: int }
```

```
and type_expr = transient_expr
```

```
and type_desc =  
  Tvar of string option  
  | Tarrow of arg_label * type_expr * type_expr * commutable  
  | Ttuple of type_expr list
```



```
(* Type expressions for the core
```

```
type transient_expr =  
  { mutable desc: type_desc;  
    mutable level: int;
```

Levels are mutable!

And they're stored on types.

```
and type_desc =  
  Tvar of string option  
  | Tarrow of arg_label * type_ex
```

What's the level of a *type*?

at least

The max of the levels of its ~~vars.~~
components

$\alpha:1 \rightarrow \beta:2 \rightarrow \text{int}$

This type makes sense
only at level 2 or greater.



Why are levels mutable?

Types are graphs.

Unification and generalization
change levels.



$(\alpha:1 \rightarrow \alpha:1):1$

generalizes to

$(\alpha:\infty \rightarrow \alpha:\infty):\infty$

Only a generic type can contain
generic variables.



$(\alpha:1 \rightarrow \alpha:1):1$

generalizes to

$(\alpha:\infty \rightarrow \alpha:\infty):\infty$

```
(**** Type level management ****)
```

```
let generic_level = Ident.highest_scope
```

```
let highest_scope = 100_000_000
```

```
(* assumed to fit in 27 bits, see Types.scope_field *)
```



$(\alpha:1 \rightarrow \alpha:1):1$

generalizes to

$(\alpha:\infty \rightarrow \alpha:\infty):\infty$

There is no \forall .

```
val instance: ?partial:bool -> type_expr -> type_expr  
(* Take an instance of a type scheme *)
```

copies and lowers levels



If a type's level is less than ∞ , we do not need to look inside during instantiation.



let add $x = x + 1$



let add x_{int} =

(+)
 $\text{int} \rightarrow \text{int} \rightarrow \text{int}$
 $x_{\text{int}} \ 1$

We update the level for the
 int to match α 's level.



The level differentiates
what we can be sure of
vs
what we are inferring.



```
type t1 = A | B | C  
type t2 = A | B
```

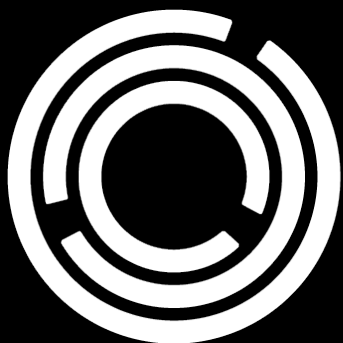
```
let f1 x = if x then C else B  
let f2 x = if x then B else C
```

f1 is accepted (warned with -principal)
f2 is rejected

inferred type **t1** is not at level ∞



Levels in GHC



```
data TcLevel = TcLevel {-# UNPACK #-} !Int
              | QInstVar
  -- See Note [TcLevel invariants] for what this Int is
  -- See also Note [TcLevel assignment]
  -- See also Note [The QInstVar TcLevel]
```

QInstVar acts like ∞
(we will ignore it)




```

data Type
  -- See Note [Non-trivial definitional equality]
  = TyVarTy Var -- ^ Vanilla type or kind variable

data Var
  = TcTyVar {
    -- Used only during type inference
    -- Used for kind variables during
    -- inference, as well
    varName      :: !Name,
    realUnique   :: {-# UNPACK #-} !Unique,
    varType      :: Kind,
    tc_tv_details :: TcTyVarDetails
  }

-- A TyVarDetails is inside a TyVar
-- See Note [TyVars and TcTyVars during type checking]
data TcTyVarDetails
  = SkolemTv      -- A skolem
    SkolemInfo    -- See Note [Keeping SkolemInfo inside a SkolemTv]
    TcLevel       -- Level of the implication that binds it
    -- See GHC.Tc.Utils.Unify Note [Deeper level on the left] for
    -- how this level number is used
    Bool          -- True <=> this skolem type variable can be overlapped
    -- when looking up instances
    -- See Note [Binding when looking up instances] in GHC.Core.InstEnv

  | RuntimeUnk    -- Stands for an as-yet-unknown type in the GHCi
    -- interactive context

  | MetaTv { mtv_info  :: MetaInfo
    , mtv_ref   :: IORef MetaDetails
    , mtv_tclvl :: TcLevel } -- See Note [TcLevel invariants]

```

Types are trees.

Levels are on variables.

(GHC loses the instantiation
optimization that OCaml has.)



$\alpha:1 \rightarrow \alpha:1$

generalizes to

$\forall \{a\}. a \rightarrow a$

```
data Type
  -- See Note [Non-trivial definitional equality]
  = TyVarTy Var -- ^ Vanilla type or kind variable (*never* a coercion variable)

  | ForAllTy -- See Note [ForAllTy]
    {-# UNPACK #-} !ForAllTyBinder
    Type          -- ^ A  $\Pi$  type.

; (binders, theta') <- chooseInferredQuantifiers residual inferred_theta
    (tyCoVarsOfType mono_ty') qtvvs mb_sig_inst

; let inferred_poly_ty = mkInvisForAllTys binders (mkPhiTy theta' mono_ty')
```



$\alpha:1 \rightarrow \alpha:1$

generalizes to

$\forall \{a\}. a \rightarrow a$

Key step implemented in
`candidateQTyVarsOfType`.




Unification

outer $x_{\alpha:1} = ()$ where

inner $y_{\beta:2} = [x, y]$

$$\alpha:1 \sim \beta:2$$

$\beta := \alpha$ 

~~$\alpha := \beta$~~



Unification

Key step implemented in
`uUnfilledVar1`.



Skolem Escape

```
data Ex where MkEx :: a -> Ex
```

```
f (MkEx y) = y
```



Skolem Escape

```
data Ex where MkEx :: a -> Ex
```

```
f arg = case arg of  
  MkEx y -> y
```



Skolem Escape

data Ex where MkEx :: a -> Ex

f arg_{Ex1} = case _{$\beta:1$} arg_{Ex1} of

MkEx y_{a:2} -> y_{a:2}

$\beta:1 \sim a:2$

$\beta:1 := a:2$ $a:2 := \beta:1$

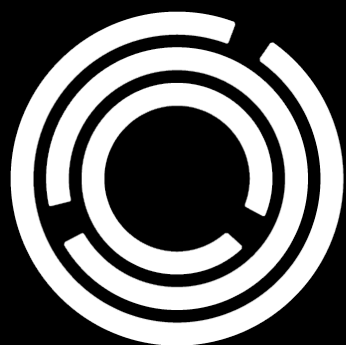
no: levels

no: skolem



In both OCaml and GHC:

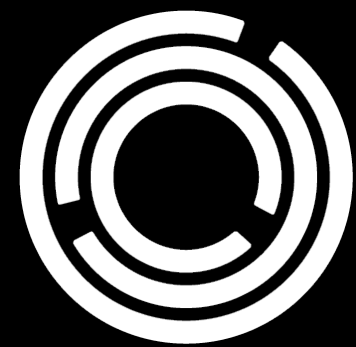
When done with a
construct, we must
generalize, promote (update
the level), or *error*.



Conclusion

Levels are a convenient mechanism in type inference, powering generalization among other inference decisions.





Jane Street

Type Inference in OCaml and GHC using Levels

Richard A. Eisenberg

Jane Street

reisenberg@janestreet.com

Saturday, January 25, 2025

WITS

Denver, CO, USA