Finding Minimum Type Error Sources

Thomas Wies

NYU | COURANT

joint work with Zvonimir Pavlinovic (NYU) and Tim King (Google)

Type Error Localization

```
let f x y =
  let yi = int_of_string y in
  x + yi
in
f "1" "2" + f "3" "4"
```

Type Error Localization

```
let f x y =
  let yi = int_of_string y in
  x + yi
in
f "1" "2" + f "3" "4"
```

```
Error: This expression has type string but
an expression was expected of type int
```

Type Error Localization

```
let f x y =
  let yi = int_of_string y in
  x + yi
in
f "1" "2" + f "3" "4"
```

Who should be blamed for a type mismatch?

```
Error: This expression has type string but
an expression was expected of type int
```

Type Error Localization

```
let f(lst:move list): (float*float) list =
          ...
  let rec loop lst x y dir acc =
    if lst = [] then
      acc
    else
      print_string "foo"
  in
  List.rev
    (loop lst 0.0 0.0 0.0 [(0.0,0.0)])
```

Type Error Localization

```
let f(lst:move list): (float*float) list =
           ...
  let rec loop lst x y dir acc =
    if lst = [] then
      acc
    else
      print_string "foo"
  in
  List.rev
    (loop lst 0.0 0.0 0.0 [(0.0,0.0)])
```

acc must have type unit

Type Error Localization

```
let f(lst:move list): (float*float) list =
          ...
  let rec loop lst x y dir acc =
    if lst = [] then
      acc
    else
      print_string "foo"
  in
  List.rev
    (loop lst 0.0 0.0 0.0 [(0.0,0.0)])
```

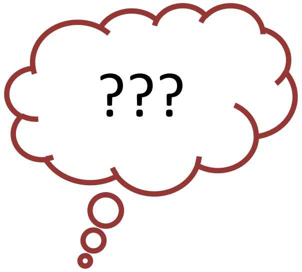
acc must have type unit

```
Error: This expression has type 'a list but
an expression was expected of type unit
```

Type Error Localization

```
let f(lst:move list): (float*float) list =
           ...
  let rec loop lst x y dir acc =
    if lst = [] then
      acc
    else
      print_string "foo"
  in
  List.rev
    (loop lst 0.0 0.0 0.0 [(0.0,0.0)])
```

acc must have type unit



```
Error: This expression has type 'a list but
an expression was expected of type unit
```

Type Error Localization

```
let f(lst:move list): (float*float) list =
            ...
  let rec loop lst x y dir acc =
    if lst = [] then
      acc
    else
      print_string "foo"
  in
  List.rev
    (loop lst 0.0 0.0 0.0 [(0.0,0.0)])
```

A-ha!

Error: This expression has type unit but an expression was expected of type (float*float) list

Challenges

- Can we find good heuristics to rank type error sources by their usefulness?
- Can we find a solution that is agnostic to the specific type system?
- Can we implement that solution without substantial compiler modifications?
- Can we provide formal quality guarantees?

Is this not a solved problem by now?

Is this not a solved problem by now?

[M. Wand]

1986

[Beaven & Stansifer]

[Duggan & Bent]

[J. Yang]

[O. Chitil]

[Tip & Dinesh]

2000

[Neubauer & Thiemann]

[Stuckey, Sulzmann, & Wazny]

[Haack & Wells]

[H. Gast]

[Lerner, Flower, Grossman, & Chambers]

[Chen & Erwig] [Zhang & Myers] [Pavlinovic, King, & Wies]

2019

[Zhang, Myers, Vytiniotis, & Jones]

Defining the Problem

Error Sources

```
let x = "hi" in not x
```

Error Sources

```
let x = "hi" in not x
```

Error Sources

```
let x = "hi" in not ?
```

Error Sources

```
let x = "hi" in not ?
```

Error Source

An error source is a set of program expressions that, once corrected, yield a well-typed program

Minimum Error Sources

```
let x = "hi" in not x
```

Minimum Error Sources

```
let x = "hi" in ?
```

Minimum Error Sources

```
let x = "hi" in ?
```

- Rank sources by some *useful* criterion
  - by assigning weights to expressions

Minimum Error Source

An error source with minimum cumulative weight

Ranking Criteria - Example

Ranking Criteria - Example

- Prefer error sources that require fewer code modifications?

Ranking Criteria - Example

- Prefer error sources that require fewer code modifications?
  - assign weights according to expression's size

Ranking Criteria - Example

- Prefer error sources that require fewer code modifications?
  - assign weights according to expression's size

```
let x = "hi" in not x
```

Ranking Criteria - Example

- Prefer error sources that require fewer code modifications?
  - assign weights according to expression's size

```
let x = "hi" in not
```

? (1)

Ranking Criteria - Example

- Prefer error sources that require fewer code modifications?
  - assign weights according to expression's size

```
let x = "hi" in not x
```

Ranking Criteria - Example

- Prefer error sources that require fewer code modifications?
  - assign weights according to expression's size

```
let x = "hi" in ? (1) x
```

Ranking Criteria - Example

- Prefer error sources that require fewer code modifications?
  - assign weights according to expression's size

```
let x = "hi" in not x
```

Ranking Criteria - Example

- Prefer error sources that require fewer code modifications?
  - assign weights according to expression's size

```
let x = "hi" in  ? (3)
```

Ranking Criteria - Example

- Prefer error sources that require fewer code modifications?
  - assign weights according to expression's size

```
let x = "hi" in not x
```

Ranking Criteria - Example

- Prefer error sources that require fewer code modifications?
  - assign weights according to expression's size

? (5)

Ranking Criteria - Example

- Prefer error sources that require fewer code modifications?
  - assign weights according to expression's size

```
let x = "hi" in not x
```

Ranking Criteria - Example

- Prefer error sources that require fewer code modifications?
  - assign weights according to expression's size

```
let x = "hi" in not x
```

Problem Definition

[Pavlinovic, King, Wies OOPSLA'14]

Computing Minimum Error Sources

Given a program and a ranking criterion, find a minimum error source subject to that criterion

Solving the Problem

Type Inference as Constraint Solving

```
let x = "hi" in not x
```

Type Inference as Constraint Solving

```
let x = "hi" in not x
```

$$\alpha_{let} = \alpha_o$$

Type Inference as Constraint Solving

```
let x = "hi" in not x
```

$$\alpha_{let} = \alpha_o$$

$$\boxed{\alpha_x = \text{string}}$$

Type Inference as Constraint Solving

`let x = "hi" in not x`

$$\alpha_{let} = \alpha_o$$

$$\alpha_x = \text{string}$$

$$\alpha_{app} = \text{fun}(\alpha_i\ ,\ \alpha_o)$$

Type Inference as Constraint Solving

`let x = "hi" in not x`

$$\alpha_{let} = \alpha_o$$

$$\alpha_x = \text{string}$$

$$\alpha_{app} = \text{fun}(\alpha_i \, , \, \alpha_o)$$

$$\alpha_{not} = \alpha_{app}$$

Type Inference as Constraint Solving

```
let x = "hi" in not x
```

$$\alpha_{let} = \alpha_o$$

$$\alpha_x = \text{string}$$

$$\alpha_{app} = \text{fun}(\alpha_i\ ,\ \alpha_o)$$

$$\alpha_{not} = \alpha_{app}$$

$$\alpha_i = \alpha_x$$

Type Inference as Constraint Solving

```
let x = "hi" in not x
```

$$\alpha_{let} = \alpha_o$$

$$\alpha_x = \text{string}$$

$$\alpha_{app} = \text{fun}(\alpha_i\ ,\ \alpha_o)$$

$$\alpha_{not} = \alpha_{app}$$

$$\alpha_i = \alpha_x$$

$$\boxed{\alpha_{not} = \text{fun}(\text{bool},\ \text{bool})}$$

Type Inference as Constraint Solving

```
let x = "hi" in not x
```

$$\alpha_{let} = \alpha_o$$

$$\alpha_x = \text{string}$$

$$\alpha_{app} = \text{fun}(\alpha_i \, , \, \alpha_o)$$

$$\alpha_{not} = \alpha_{app}$$

$$\alpha_i = \alpha_x$$

$$\alpha_{not} = \text{fun}(\text{bool}, \, \text{bool})$$

program is well-typed

if and only if

constraints are satisfiable

Type Inference as Constraint Solving

```
let x = "hi" in not x
```

$$\alpha_{let} = \alpha_o$$

$$\alpha_x = \text{string}$$

$$\alpha_{app} = \text{fun}(\alpha_i \, , \, \alpha_o)$$

$$\alpha_{not} = \alpha_{app}$$

$$\alpha_i = \alpha_x$$

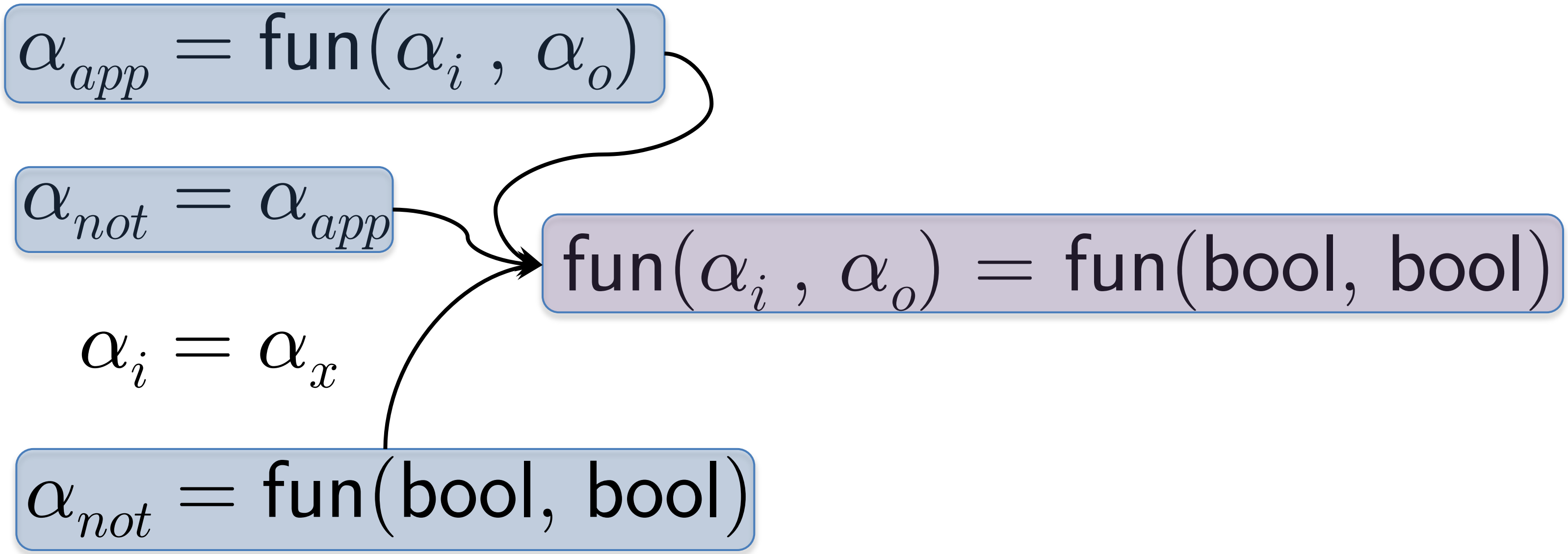
$$\alpha_{not} = \text{fun}(\text{bool}, \text{ bool})$$

Type Inference as Constraint Solving

```
let x = "hi" in not x
```

$$\alpha_{let} = \alpha_o$$

$$\alpha_x = \text{string}$$



$$\alpha_{app} = \text{fun}(\alpha_i \,, \alpha_o)$$

$$\alpha_{not} = \alpha_{app}$$

$$\text{fun}(\alpha_i \,, \alpha_o) = \text{fun}(\text{bool}, \text{bool})$$

$$\alpha_i = \alpha_x$$

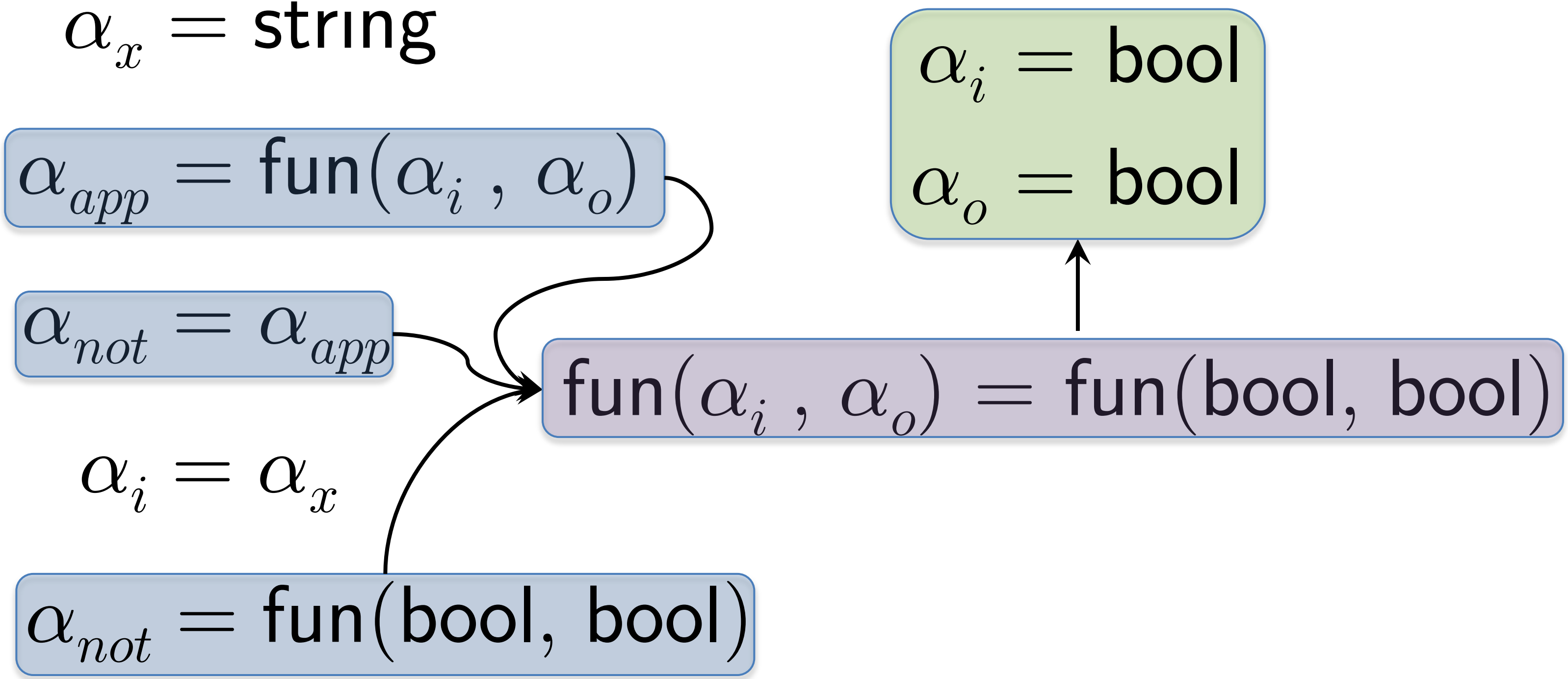
$$\alpha_{not} = \text{fun}(\text{bool}, \text{bool})$$

Type Inference as Constraint Solving

```
let x = "hi" in not x
```

$$\alpha_{let} = \alpha_o$$

$$\alpha_x = \text{string}$$



$$\alpha_{app} = \text{fun}(\alpha_i\ ,\ \alpha_o)$$

$$\alpha_{not} = \alpha_{app}$$

$$\alpha_i = \alpha_x$$

$$\alpha_{not} = \text{fun}(\text{bool},\ \text{bool})$$

$$\text{fun}(\alpha_i\ ,\ \alpha_o) = \text{fun}(\text{bool},\ \text{bool})$$

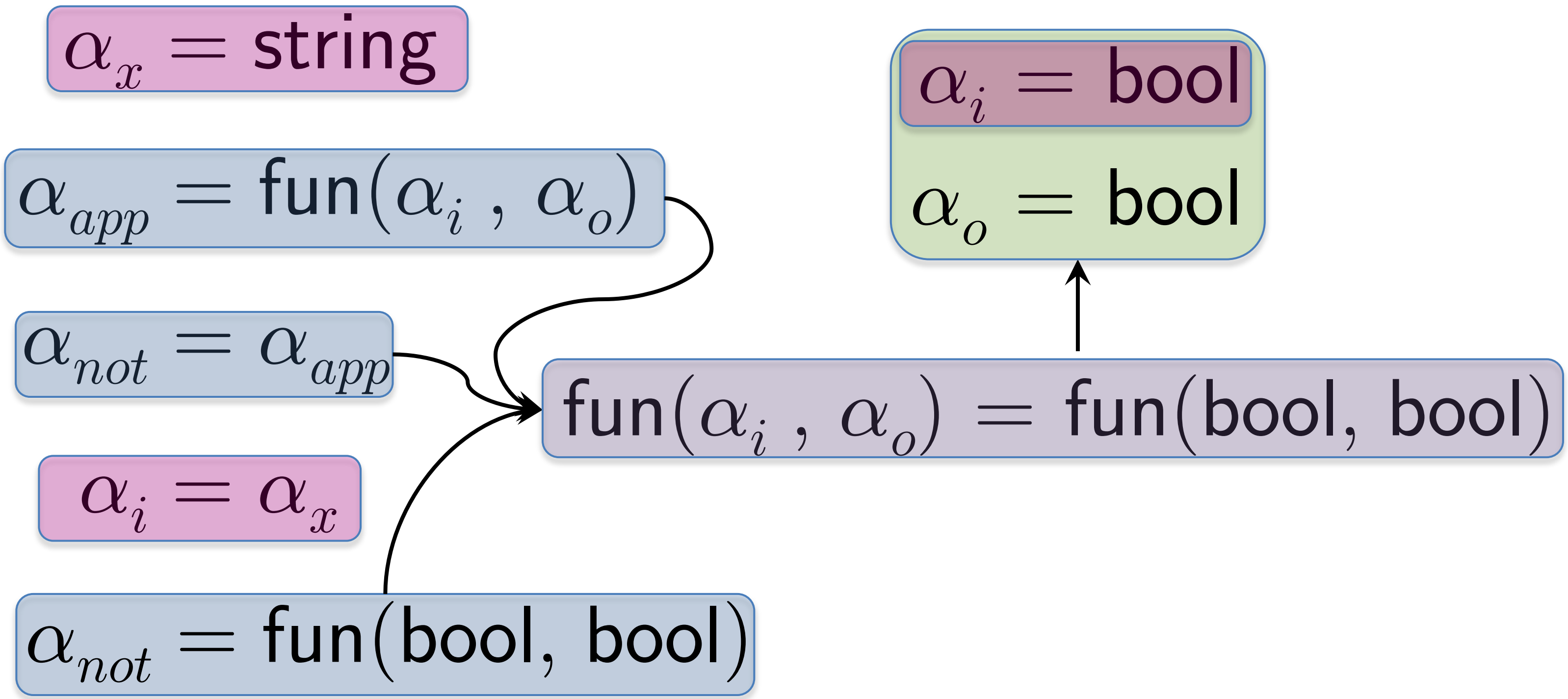
$$\alpha_i = \text{bool}$$

$$\alpha_o = \text{bool}$$

Type Inference as Constraint Solving

```
let x = "hi" in not x
```

$\alpha_{let} = \alpha_o$



$\alpha_x = \text{string}$

$\alpha_{app} = \text{fun}(\alpha_i\ ,\ \alpha_o)$

$\alpha_{not} = \alpha_{app}$

$\alpha_i = \alpha_x$

$\alpha_{not} = \text{fun}(\text{bool}, \text{bool})$

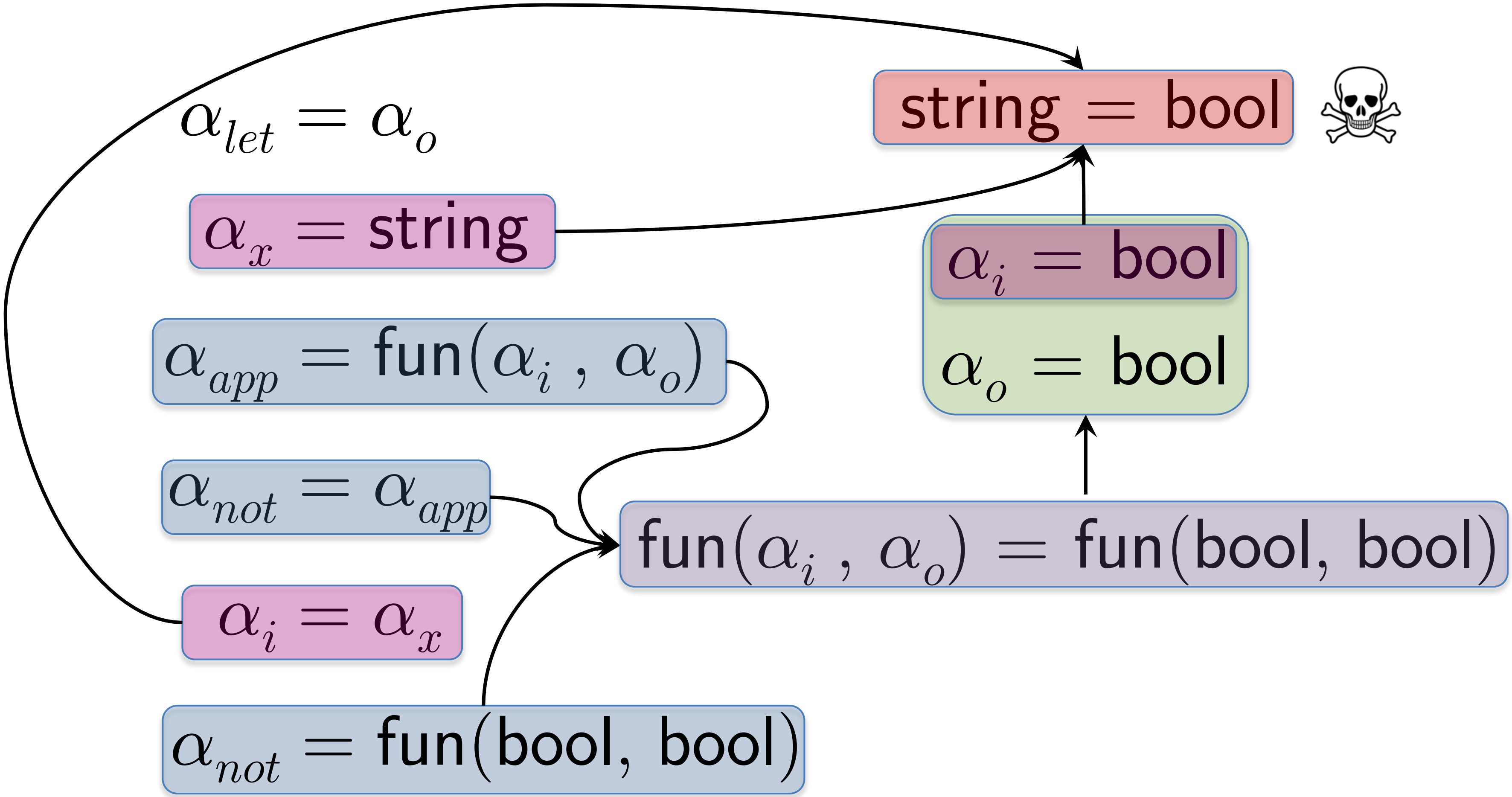
$\text{fun}(\alpha_i\ ,\ \alpha_o) = \text{fun}(\text{bool}, \text{bool})$

$\alpha_i = \text{bool}$

$\alpha_o = \text{bool}$

Type Inference as Constraint Solving

```
let x = "hi" in not x
```



$\alpha_{let} = \alpha_o$

$\alpha_x = \text{string}$

$\alpha_{app} = \text{fun}(\alpha_i\ ,\ \alpha_o)$

$\alpha_{not} = \alpha_{app}$

$\alpha_i = \alpha_x$

$\alpha_{not} = \text{fun}(\text{bool},\ \text{bool})$

$\text{fun}(\alpha_i\ ,\ \alpha_o) = \text{fun}(\text{bool},\ \text{bool})$

$\alpha_i = \text{bool}$

$\alpha_o = \text{bool}$

$\text{string} = \text{bool}$ ☠

Weighted MaxSAT

- **Input**: a set of clauses in propositional logic
  + a positive weight for each clause

$$\underset{2}{(\neg A \lor B)} \land \underset{1}{(\neg B \lor \neg C)} \land \underset{3}{A} \land \underset{3}{C}$$

- **Output**: satisfiable subset of input clauses with maximum cumulative weight

Weighted MaxSAT

- **Input**: a set of clauses in propositional logic
  + a positive weight for each clause

$$\underset{2}{(\neg A \lor B)} \land \underset{1}{(\neg B \lor \neg C)} \land \underset{3}{A} \land \underset{3}{C}$$

- **Output**: satisfiable subset of input clauses with maximum cumulative weight

Weighted MaxSMT

Weighted MaxSMT

- **Input**: a set of clauses in (quantifier-free) first-order logic interpreted in a specified theory

+ weights

3 $f(x) \neq z \wedge$
1 $f(y) = z \wedge$
1 $w = y \wedge$
4 $(x - y = 0 \vee f(w) \neq z)$

- **Output**: satisfiable subset of input clauses with maximum cumulative weight

Weighted MaxSMT

- **Input**: a set of clauses in (quantifier-free) first-order logic interpreted in a specified theory

+ weights

3 $f(x) \neq z \ \wedge$

1 $f(y) = z \wedge$

1 $w = y \wedge$

4 $(x - y = 0 \vee f(w) \neq z)$

- **Output**: satisfiable subset of input clauses with maximum cumulative weight

Weighted MaxSMT

- **Input**: a set of clauses in (quantifier-free) first-order logic interpreted in a specified theory

+ weights

3 $f(x) \neq z \ \wedge$

1 $f(y) = z \ \wedge$

1 $w = y \ \wedge$

4 $(x - y = 0 \ \vee \ f(w) \neq z)$

- **Output**: satisfiable subset of input clauses with maximum cumulative weight

Observation:

Type Checking = Satisfiability Modulo Inductive Data Types

Reduction to Weighted MaxSMT

```
let x = "hi" in not x
```

Reduction to Weighted MaxSMT

`let x = "hi" in not x`

$$\alpha_{let} = \alpha_o \;\; \wedge$$

Reduction to Weighted MaxSMT

`let x = "hi" in not x`

$$\alpha_{let} = \alpha_o \ \wedge$$

$$\alpha_x = \text{string} \wedge$$

Reduction to Weighted MaxSMT

```
let x = "hi" in not x
```

$$\alpha_{let} = \alpha_o \ \wedge$$

$$\alpha_x = \text{string} \wedge$$

$$\alpha_{app} = \text{fun}(\alpha_i, \alpha_o) \ \wedge$$

Reduction to Weighted MaxSMT

```
let x = "hi" in not x
```

$$\alpha_{let} = \alpha_o \;\wedge$$

$$\alpha_x = \text{string} \wedge$$

$$\alpha_{app} = \text{fun}(\alpha_i, \alpha_o) \;\wedge$$

$$\alpha_{not} = \alpha_{app} \;\wedge$$

Reduction to Weighted MaxSMT

```
let x = "hi" in not x
```

$$\alpha_{let} = \alpha_o \quad \wedge$$

$$\alpha_x = \text{string} \wedge$$

$$\alpha_{app} = \text{fun}(\alpha_i, \alpha_o) \wedge$$

$$\alpha_{not} = \alpha_{app} \quad \wedge$$

$$\alpha_i = \alpha_x \qquad \wedge$$

Reduction to Weighted MaxSMT

`let x = "hi" in not x`

$$\alpha_{let} = \alpha_o \quad \wedge$$

$$\alpha_x = \mathsf{string} \wedge$$

$$\alpha_{app} = \mathsf{fun}(\alpha_i, \alpha_o) \wedge$$

$$\alpha_{not} = \alpha_{app} \quad \wedge$$

$$\alpha_i = \alpha_x \qquad \wedge$$

$$\alpha_{not} = \mathsf{fun}(\mathsf{bool}, \mathsf{bool})$$

Reduction to Weighted MaxSMT

```
let x = "hi" in not x
```

$$\boxed{\alpha_{let} = \alpha_o} \wedge$$

$$\boxed{\alpha_x = \mathsf{string}} \wedge$$

$$\boxed{\alpha_{app} = \mathsf{fun}(\alpha_i, \alpha_o)} \wedge$$

$$\boxed{\alpha_{not} = \alpha_{app}} \wedge$$

$$\boxed{\alpha_i = \alpha_x} \wedge$$

$$\boxed{\alpha_{not} = \mathsf{fun}(\mathsf{bool}, \mathsf{bool})}$$

Reduction to Weighted MaxSMT

```
let x = "hi" in not x
```

$$T_{let} \implies (\alpha_{let} = \alpha_o \ \wedge$$

$$T_x \implies \alpha_x = \textsf{string} \wedge$$

$$T_{app} \implies (\alpha_{app} = \textsf{fun}(\alpha_i, \alpha_o) \wedge$$

$$T_{not} \implies \alpha_{not} = \alpha_{app} \ \wedge$$

$$T_i \implies \alpha_i = \alpha_x \ )) \ \wedge$$

$$T_{not\ impl} \implies \alpha_{not} = \textsf{fun}(\textsf{bool}, \textsf{bool}) \ \wedge$$

$$T_{let} \wedge T_x \wedge T_{app} \wedge T_{not} \wedge T_i \wedge T_{not\ impl}$$

Reduction to Weighted MaxSMT

```
let x = "hi" in not x
```

$$\left.\begin{array}{l}
T_{let} \implies (\, \alpha_{let} = \alpha_o \;\wedge \\
\quad T_x \implies \alpha_x = \text{string} \wedge \\
\quad T_{app} \implies (\, \alpha_{app} = \text{fun}(\alpha_i, \alpha_o) \wedge \\
\qquad\qquad T_{not} \implies \alpha_{not} = \alpha_{app} \;\wedge \\
\qquad\qquad T_i \implies \alpha_i = \alpha_x \;)) \;\;\wedge \\
T_{not\ impl} \implies \alpha_{not} = \text{fun}(\text{bool}, \text{bool}) \;\;\wedge
\end{array}\right\} \infty$$

$$T_{let} \wedge T_x \wedge T_{app} \wedge T_{not} \wedge T_i \wedge T_{not\ impl} \quad < \infty$$

Reduction to Weighted MaxSMT

```
let x = "hi" in not ?
```

$$
\left.
\begin{array}{l}
T_{let} \implies (\alpha_{let} = \alpha_o \;\wedge \\
\quad T_x \implies \alpha_x = \text{string} \wedge \\
\quad T_{app} \implies (\alpha_{app} = \text{fun}(\alpha_i, \alpha_o) \wedge \\
\qquad\quad T_{not} \implies \alpha_{not} = \alpha_{app} \;\wedge \\
\qquad\quad \boxed{T_i} \implies \alpha_i = \alpha_x \;)) \;\wedge \\
T_{not\ impl} \implies \alpha_{not} = \text{fun}(\text{bool}, \text{bool}) \;\wedge
\end{array}
\right\} \infty
$$

$$
\boxed{T_{let} \wedge T_x \wedge T_{app} \wedge T_{not}} \wedge \boxed{T_i} \wedge \boxed{T_{not\ impl}} \quad < \infty
$$

1 1 1 1 0 1

Reduction to Weighted MaxSMT

```
let x = "hi" in not ?
```

$$\left.\begin{aligned}
&T_{let} \implies (\, \alpha_{let} = \alpha_o \;\wedge \\
&\quad T_x \implies \alpha_x = \text{string} \wedge \\
&\quad T_{app} \implies (\, \alpha_{app} = \text{fun}(\alpha_i, \alpha_o) \wedge \\
&\qquad\quad T_{not} \implies \alpha_{not} = \alpha_{app} \;\wedge \\
&\qquad\quad \boxed{T_i} \implies \alpha_i = \alpha_x \;)) \;\wedge \\
&T_{not\ impl} \implies \alpha_{not} = \text{fun}(\text{bool}, \text{bool}) \;\wedge
\end{aligned}\right\} \infty$$

$$\boxed{\underset{w_{let}}{T_{let}} \wedge \underset{w_x}{T_x} \wedge \underset{w_{app}}{T_{app}} \wedge \underset{w_{not}}{T_{not}}} \wedge \boxed{\underset{w_i}{T_i}} \wedge \boxed{\underset{w_{not\ impl}}{T_{not\ impl}}} \quad < \infty$$

Prototype Implementation

- Supports subset of OCaml (roughly Caml light)
- Evaluated on benchmark suite of more that 700 OCaml programs
- 15% more accuracy than OCaml's type checker
  (even with a rather simplistic ranking criterion)
- Good scalability (a few seconds for several K lines of code)
  - achieved by efficiently encoding types of polymorphic functions [ICFP'15]

First **scalable** type error localization tool that provides **formal optimality guarantees**.

Conclusions

- Practical algorithm for localizing type errors
- Finds the "best" source of a type error
- Abstracts from the definition of "best"
- Works well for Hindley-Milner type systems (OCaml, SML, Haskell, …)
- Still work to be done for more expressive type systems (unrestricted polymorphism, refinement types, …)

Exponential Complexity of Type Checking for the ML Language Family

```
let pair f x = f x x in
let f x = pair x in
let f x = f (f x) in
let f x = f (f x) in
let f x = f (f x) in
let f x = f (f x) in
let f x = f (f x) in
let f x = f (f x) in
let f x = f (f x) in
let f x = f (f x) in
let f x = f (f x) in
let f x = f (f x) in
let f x = f (f x) in
let f x = f (f x) in
let f x = f (f x) in
let f x = f (f x) in
let f x = f (f x) in
let f x = f (f x) in
fun z -> f (fun x -> x) z
```

I'll reserve a table at Miliways!

Let Polymorphism

```
let id x = x in
id 1, id true
```

$$\alpha_{id} = \mathsf{fun}(\alpha_x, \alpha_r)$$

$$\alpha_x = \alpha_r$$

Let Polymorphism

```
let id x = x in
id 1, id true
```

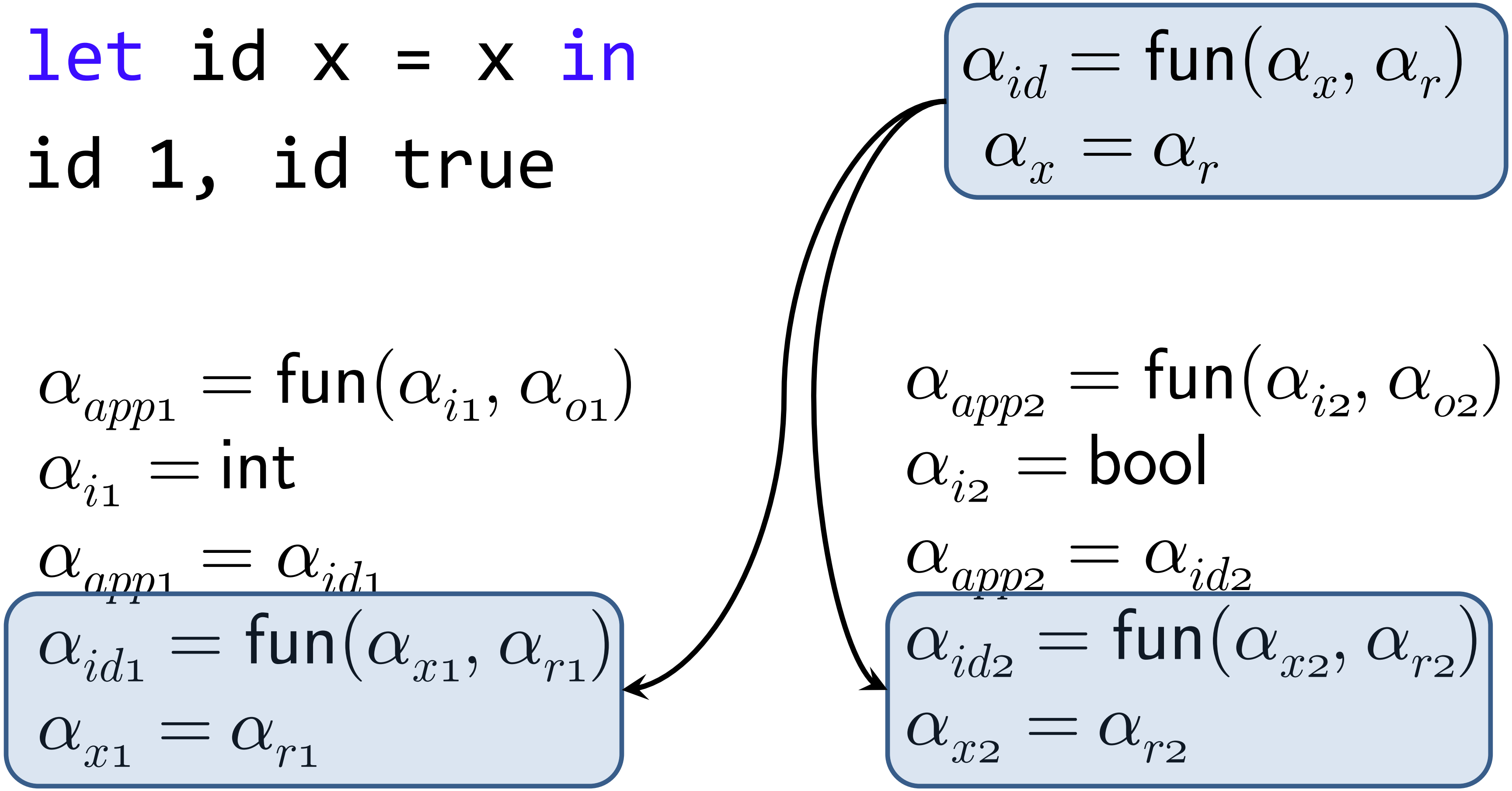
$$\alpha_{id} = \text{fun}(\alpha_x, \alpha_r)$$
$$\alpha_x = \alpha_r$$

$$\alpha_{app1} = \text{fun}(\alpha_{i1}, \alpha_{o1})$$
$$\alpha_{i1} = \text{int}$$
$$\alpha_{app1} = \alpha_{id1}$$
$$\alpha_{id1} = \text{fun}(\alpha_{x1}, \alpha_{r1})$$
$$\alpha_{x1} = \alpha_{r1}$$

$$\alpha_{app2} = \text{fun}(\alpha_{i2}, \alpha_{o2})$$
$$\alpha_{i2} = \text{bool}$$
$$\alpha_{app2} = \alpha_{id2}$$
$$\alpha_{id2} = \text{fun}(\alpha_{x2}, \alpha_{r2})$$
$$\alpha_{x2} = \alpha_{r2}$$

Let Polymorphism

```
let id x = x in
id 1, id true
```



$$\alpha_{id} = \text{fun}(\alpha_x, \alpha_r)$$
$$\alpha_x = \alpha_r$$

$$\alpha_{app1} = \text{fun}(\alpha_{i1}, \alpha_{o1})$$
$$\alpha_{i_1} = \text{int}$$
$$\alpha_{app1} = \alpha_{id1}$$
$$\alpha_{id_1} = \text{fun}(\alpha_{x1}, \alpha_{r1})$$
$$\alpha_{x1} = \alpha_{r1}$$

$$\alpha_{app2} = \text{fun}(\alpha_{i2}, \alpha_{o2})$$
$$\alpha_{i_2} = \text{bool}$$
$$\alpha_{app2} = \alpha_{id2}$$
$$\alpha_{id_2} = \text{fun}(\alpha_{x2}, \alpha_{r2})$$
$$\alpha_{x2} = \alpha_{r2}$$

Constraint size grows exponentially with the nesting depth of `let`s

Taming Constraint Explosion

[Pavlinovic, King, Wies ICFP'15]

- How do we tame blow-up?

Taming Constraint Explosion

[Pavlinovic, King, Wies ICFP'15]

- How do we tame blow-up?

```
let first  (a, b, _) = a
let second (a, b, _) = b

let f x =
  let first_x  = first x in
  let second_x = int_of_string (second x) in
  first_x + second_x

f ("1", "2", f ("3", "4", 5))
```

Taming Constraint Explosion

[Pavlinovic, King, Wies ICFP'15]

- How do we tame blow-up?

```
let first  (a, b, _) = a
let second (a, b, _) = b

let f x =
  let first_x  = first x in
  let second_x = int_of_string (second x) in
  first_x + second_x

f ("1", "2", f ("3", "4", 5))
```

Taming Constraint Explosion

[Pavlinovic, King, Wies ICFP'15]

- How do we tame blow-up?

```
let first  (a, b, _) = a
let second (a, b, _) = b

let f x =
  let first_x  = first x in
  let second_x = int_of_string (second x) in
  first_x + second_x

f ("1", "2", f ("3", "4", 5))
```

Principal Type Abstraction

`; first  :` $\forall \alpha_a, \alpha_b, \alpha_c\ \mathsf{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_a)$

`; second :` $\forall \alpha_a, \alpha_b, \alpha_c\ \mathsf{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_b)$

`; f :` $\forall \alpha_a\ \mathsf{fun}(\mathsf{int} * \mathsf{string} * \alpha_a, \mathsf{int})$

`f ("1", "2", f ("3", "4", 5))`

Principal Type Abstraction

`; first  :` $\forall \alpha_a, \alpha_b, \alpha_c\ \mathsf{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_a)$

`; second :` $\forall \alpha_a, \alpha_b, \alpha_c\ \mathsf{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_b)$

`; f :` $\forall \alpha_a\ \mathsf{fun}(\mathsf{int} * \mathsf{string} * \alpha_a, \mathsf{int})$

`f ("1", "2", f ("3", "4", 5))`

$$T_{f_2} \implies \gamma = \mathsf{fun}(\mathsf{int} * \mathsf{string} * \beta, \mathsf{int})$$

Principal Type Abstraction

`; first  :` $\forall \alpha_a, \alpha_b, \alpha_c\ \mathsf{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_a)$

`; second :` $\forall \alpha_a, \alpha_b, \alpha_c\ \mathsf{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_b)$

`; f :` $\forall \alpha_a\ \mathsf{fun}(\mathsf{int} * \mathsf{string} * \alpha_a, \mathsf{int})$

```
f ("1", "2", f ("3", "4", 5))
```

$$T_{f_2} \implies \gamma = \mathsf{fun}(\mathsf{int} * \mathsf{string} * \beta, \mathsf{int})$$

Principal Type Abstraction

`; first  :` $\forall \alpha_a, \alpha_b, \alpha_c\ \mathsf{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_a)$

`; second :` $\forall \alpha_a, \alpha_b, \alpha_c\ \mathsf{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_b)$

`; f :` $\forall \alpha_a\ \mathsf{fun}(\mathsf{int} * \mathsf{string} * \alpha_a, \mathsf{int})$

`f ("1", "2", f ("3", "4", 5))`

$$T_{f_2} \implies \gamma = \mathsf{fun}(\mathsf{int} * \mathsf{string} * \beta, \mathsf{int})$$

- Guard each usage of a function's principal type

Principal Type Abstraction

`; first  :` $\forall \alpha_a, \alpha_b, \alpha_c\ \text{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_a)$

`; second :` $\forall \alpha_a, \alpha_b, \alpha_c\ \text{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_b)$

`; f :` $\forall \alpha_a\ \text{fun}(\text{int} * \text{string} * \alpha_a, \text{int})$

`f ("1", "2", f ("3", "4", 5))`

$$T_{f_2} \implies \gamma = \text{fun}(\text{int} * \text{string} * \beta, \text{int})$$

- Guard each usage of a function's principal type
  - with the minimum weight in its defining expression

Principal Type Abstraction

`; first  :` $\forall \alpha_a, \alpha_b, \alpha_c\ \text{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_a)$

`; second :` $\forall \alpha_a, \alpha_b, \alpha_c\ \text{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_b)$

`; f :` $\forall \alpha_a\ \text{fun}(\text{int} * \text{string} * \alpha_a, \text{int})$

`f ("1", "2", f ("3", "4", 5))`

$$T_{f_2} \Longrightarrow \gamma = \text{fun}(\text{int} * \text{string} * \beta, \text{int})$$

- Guard each usage of a function's principal type
  - with the minimum weight in its defining expression

$$T_{f_2} \Longrightarrow P_{let\ f} \Longrightarrow \gamma = \text{fun}(\text{int} * \text{string} * \beta, \text{int})$$

Principal Type Abstraction

`; first  :` $\forall \alpha_a, \alpha_b, \alpha_c\ \mathsf{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_a)$

`; second :` $\forall \alpha_a, \alpha_b, \alpha_c\ \mathsf{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_b)$

`; f :` $\forall \alpha_a\ \mathsf{fun}(\mathsf{int} * \mathsf{string} * \alpha_a, \mathsf{int})$

```
f ("1", "2", f ("3", "4", 5))
```

$$T_{f_2} \implies \gamma = \mathsf{fun}(\mathsf{int} * \mathsf{string} * \beta, \mathsf{int})$$

- Guard each usage of a function's principal type
  - with the minimum weight in its defining expression

$$T_{f_2} \implies P_{let\ f} \implies \gamma = \mathsf{fun}(\mathsf{int} * \mathsf{string} * \beta, \mathsf{int})$$

Principal Type Abstraction

`; first  :` $\forall \alpha_a, \alpha_b, \alpha_c\ \mathsf{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_a)$

`; second :` $\forall \alpha_a, \alpha_b, \alpha_c\ \mathsf{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_b)$

`; f :` $\forall \alpha_a\ \mathsf{fun}(\mathsf{int} * \mathsf{string} * \alpha_a, \mathsf{int})$

```
f ("1", "2", f ("3", "4", 5))
```

$$T_{f_2} \implies \gamma = \mathsf{fun}(\mathsf{int} * \mathsf{string} * \beta, \mathsf{int})$$

- Guard each usage of a function's principal type
  - with the minimum weight in its defining expression

$$T_{f_2} \implies P_{let\ f} \implies \gamma = \mathsf{fun}(\mathsf{int} * \mathsf{string} * \beta, \mathsf{int})$$

Incremental Expansion

`; first  :` $\forall\alpha_a, \alpha_b, \alpha_c\ \mathsf{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_a)$
`; second :` $\forall\alpha_a, \alpha_b, \alpha_c\ \mathsf{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_b)$

```
let f x =
  let first_x  = first x in
  let second_x = int_of_string (second x) in
  first_x + second_x

f ("1", "2", f ("3", "4", 5))
```

Incremental Expansion

`; first  :` $\forall \alpha_a, \alpha_b, \alpha_c\ \mathsf{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_a)$

`; second :` $\forall \alpha_a, \alpha_b, \alpha_c\ \mathsf{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_b)$

```
let f x =
  let first_x  = first x in
  let second_x = int_of_string (second x) in
  first_x + second_x

f ("1", "2", f ("3", "4", 5))
```

Incremental Expansion

`; first  :` $\forall \alpha_a, \alpha_b, \alpha_c\ \mathsf{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_a)$

`; second :` $\forall \alpha_a, \alpha_b, \alpha_c\ \mathsf{fun}(\alpha_a * \alpha_b * \alpha_c, \alpha_b)$

```
let f x =
  let first_x  = first x in
  let second_x = int_of_string (second x) in
  first_x + second_x

f ("1", "2", f ("3", "4", 5))
```