Improving Type Error Localization for Languages with Type Inference

Thomas Wies
New York University

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

Type Safety

"Well-typed programs don't go wrong."

Statically Guaranteed Type Safety

```
let rec fac  n                   =
  if n < 2 then 1
  else n * fac (n-1)
in
  fac "hello"
```

Statically Guaranteed Type Safety

```
let rec fac (n: int): int =
  if n < 2 then 1
  else n * fac (n-1)
in
  fac "hello"
```

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

Statically Guaranteed Type Safety

```
let compose  f
             g
             x           =
  g (f x)
in
  compose 3 4
```

Statically Guaranteed Type Safety

```
let compose (f: 'a -> 'b)
            (g: 'b -> 'c)
            (x: 'a): 'c =
  g (f x)
in
  compose 3 4
```

Error: This expression has type int but an expression was expected of type 'a -> 'b

Automatic Type Inference

"Have your cake and eat it too"

Automatic Type Inference

```
let compose f g x =
  g (f x)
in
  compose 3 4
```

Automatic Type Inference

```
let compose f g x =
  g (f x)
in
  compose 3 4
```

Error: This expression has type int but an expression was expected of type 'a -> 'b

Automatic Type Inference

```
let compose f g x =
  g (f x)
in
  compose 3 4
```

Inferring the type of `compose`:

Automatic Type Inference

```
let compose f g x =
  g (f x)
in
  compose 3 4
```

Inferring the type of `compose`:

```
compose: 'd -> 'e -> 'n -> 'm
f: 'd
g: 'e
x: 'n
```

Automatic Type Inference

```
let compose f g x =
  g (f x)
in
  compose 3 4
```

Inferring the type of `compose`:

```
compose: 'd -> 'e -> 'n -> 'm
f: 'd
g: 'e
x: 'n
```

```
'd = 'a -> 'b
'n = 'a
```

Automatic Type Inference

```
let compose f g x =
  g (f x)
in
  compose 3 4
```

Inferring the type of `compose`:

```
compose: 'd -> 'e -> 'n -> 'm
f: 'd
g: 'e
x: 'n
```

```
'd = 'a -> 'b
'n = 'a
'e = 'k -> 'c
'k = 'b
'm = 'c
```

Automatic Type Inference

```
let compose f g x =
  g (f x)
in
  compose 3 4
```

Inferring the type of `compose`:

```
compose: 'd -> 'e -> 'n -> 'm
f: 'd
g: 'e
x: 'n
```

```
'd = 'a -> 'b
'n = 'a
'e = 'k -> 'c
'k = 'b
'm = 'c
```

Automatic Type Inference

```
let compose f g x =
  g (f x)
in
  compose 3 4
```

Inferring the type of `compose`:

```
compose: 'd -> 'e -> 'n -> 'm
f: 'd
g: 'e
x: 'n
```

$'d \mapsto$ `'a -> 'b`
$'n \mapsto$ `'a`
$'e \mapsto$ `'k -> 'c`
$'k \mapsto$ `'b`
$'m \mapsto$ `'c`

Automatic Type Inference

```
let compose f g x =
  g (f x)
in
  compose 3 4
```

Inferring the type of `compose`:

```
compose: 'd -> 'e -> 'n -> 'm
f: 'd
g: 'e
x: 'n
```

$$
\begin{aligned}
'd &\mapsto\ 'a \to\ 'b \\
'n &\mapsto\ 'a \\
'e &\mapsto \boxed{'k} \to\ 'c \\
\boxed{'k} &\mapsto\ 'b \\
'm &\mapsto\ 'c
\end{aligned}
$$

Automatic Type Inference

```
let compose f g x =
  g (f x)
in
  compose 3 4
```

Inferring the type of `compose`:

```
compose: 'd -> 'e -> 'n -> 'm
f: 'd
g: 'e
x: 'n
```

$$
\begin{array}{lll}
'd & \mapsto & 'a \;\text{->}\; 'b \\
'n & \mapsto & 'a \\
'e & \mapsto & \boxed{'b} \;\text{->}\; 'c \\
\boxed{'k} & \mapsto & 'b \\
'm & \mapsto & 'c
\end{array}
$$

Automatic Type Inference

```
let compose f g x =
  g (f x)
in
  compose 3 4
```

Inferring the type of `compose`:

```
compose: ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
```

Automatic Type Inference

```
let compose f g x =
  g (f x)
in
  compose 3 4
```

Checking the use of `compose`:

```
compose: ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
```

Automatic Type Inference

```
let compose f g x =
  g (f x)
in
  compose 3 4
```

Checking the use of `compose`:

```
compose: ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
```

```
int = 'a -> 'b
int = 'b -> 'c
'e  = 'a -> 'c
```

Automatic Type Inference

```
let compose f g x =
  g (f x)
in
  compose 3 4
```

Checking the use of `compose`:

```
compose: ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
```

```
int = 'a -> 'b
int = 'b -> 'c
'e  = 'a -> 'c
```

Automatic Type Inference

```
let compose f g x =
  g (f x)
in
  compose 3 4
```

Checking the use of `compose`:

```
compose: ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
```

```
int = 'a -> 'b
```

```
Error: This expression has type int but an
expression was expected of type 'a -> 'b
```

Automatic Type Inference

- Type safety is guaranteed statically
- No type annotations needed
- Type parameterization comes for free
- Efficient in practice (more on that later)

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]

[Beaven & Stansifer]

[Duggan & Bent]

[J. Yang]

[O. Chitil]

[Tip & Dinesh]

[Neubauer & Thiemman]

[Stuckey, Sulzmann, & Wazny]

[Haack & Wells]

[H. Gast]

[Lerner, Flower, Grossman, & Chambers]

[Chen & Erwig]

[Zhang & Myers]

[Pavlinovic, King, & Wies]

[Zhang, Myers, Vytiniotis, & Jones]

1986

2000

2016

Defining the Problem

"A good definition is worth a thousand theorems"

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

Computing Minimum Error Sources

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

Solving the Problem

"Every CS problem can be solved by adding another level of indirection"

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
```

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

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

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

Type Inference as Constraint Solving

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

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

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

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

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

Type Inference as Constraint Solving

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

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

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

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

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

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

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
```

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

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

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

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

$$\alpha_i = \alpha_x$$

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

Type Inference as Constraint Solving

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

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

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

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

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

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

$$\alpha_i = \alpha_x$$

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

Type Inference as Constraint Solving

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

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

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

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

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

$$\alpha_i = \alpha_x$$

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

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

$$\boxed{\begin{aligned} \alpha_i &= \text{bool} \\ \alpha_o &= \text{bool} \end{aligned}}$$

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
```

$\alpha_{let} = \alpha_o$

$\alpha_x = \text{string}$

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

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

$\alpha_i = \alpha_x$

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

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

$\alpha_i = \text{bool}$

$\alpha_o = \text{bool}$

$\text{string} = \text{bool}$ ☠

Propositional Satisfiability (SAT)

- **Input**: a set of clauses in propositional logic

$$(\neg A \vee B) \wedge (\neg B \vee \neg C) \wedge A \wedge C$$

- **Output**: satisfying assignment/**unsat**

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

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

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

Satisfiability Modulo Theories (SMT)

Satisfiability Modulo Theories (SMT)

Satisfiability Modulo Theories (SMT)

on steroids

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

$$
\begin{aligned}
& f(x) \neq z \ \wedge \\
& f(y) = z \wedge \\
& w = y \wedge \\
& (x - y = 0 \vee f(w) \neq z)
\end{aligned}
$$

- **Output**: satisfying assignment/**unsat**

Satisfiability Modulo Theories (SMT)

on steroids

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

$$
\begin{aligned}
& f(x) \neq z \ \wedge \\
& f(y) = z \wedge \\
& w = y \wedge \\
& (x - y = 0 \vee f(w) \neq z)
\end{aligned}
$$

- **Output**: satisfying assignment/**unsat**

Observation:
Type Checking = Satisfiability Modulo Inductive Data Types

Weighted MaxSMT

Weighted MaxSMT

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

+ weights

$$
\begin{array}{ll}
3 & f(x) \neq z \ \wedge \\
1 & f(y) = z \wedge \\
1 & w = y \wedge \\
4 & (x - y = 0 \vee f(w) \neq z)
\end{array}
$$

- **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

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
```

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

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

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

Reduction to Weighted MaxSMT

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

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

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

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

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

Reduction to Weighted MaxSMT

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

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

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

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

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

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

Reduction to Weighted MaxSMT

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

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

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

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

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

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

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

Reduction to Weighted MaxSMT

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

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

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

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

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

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

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

Reduction to Weighted MaxSMT

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

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

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

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

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

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

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

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

Reduction to Weighted MaxSMT

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

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

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

Reduction to Weighted MaxSMT

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

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

$$\underbrace{\boxed{T_{let} \wedge T_x \wedge T_{app} \wedge T_{not}}}_{1 \quad 1 \quad 1 \quad 1} \wedge \underbrace{\boxed{T_i}}_{0} \wedge \underbrace{\boxed{T_{not\ impl}}}_{1} \quad < \infty$$

Reduction to Weighted MaxSMT

`let x = "hi" in not ?`

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

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

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 for you!

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
```

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

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

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

Let Polymorphism

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

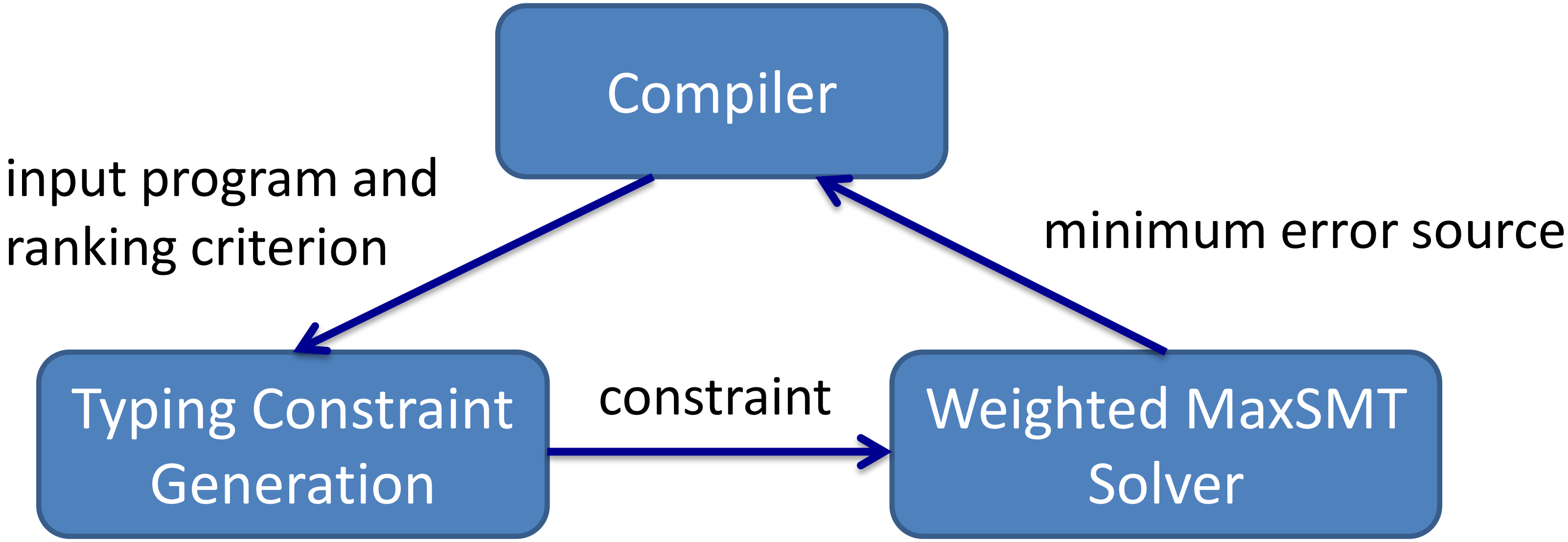
$$\alpha_{id} = \text{fun}(\alpha_x, \alpha_r)$$
$$\alpha_x = \alpha_r$$

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

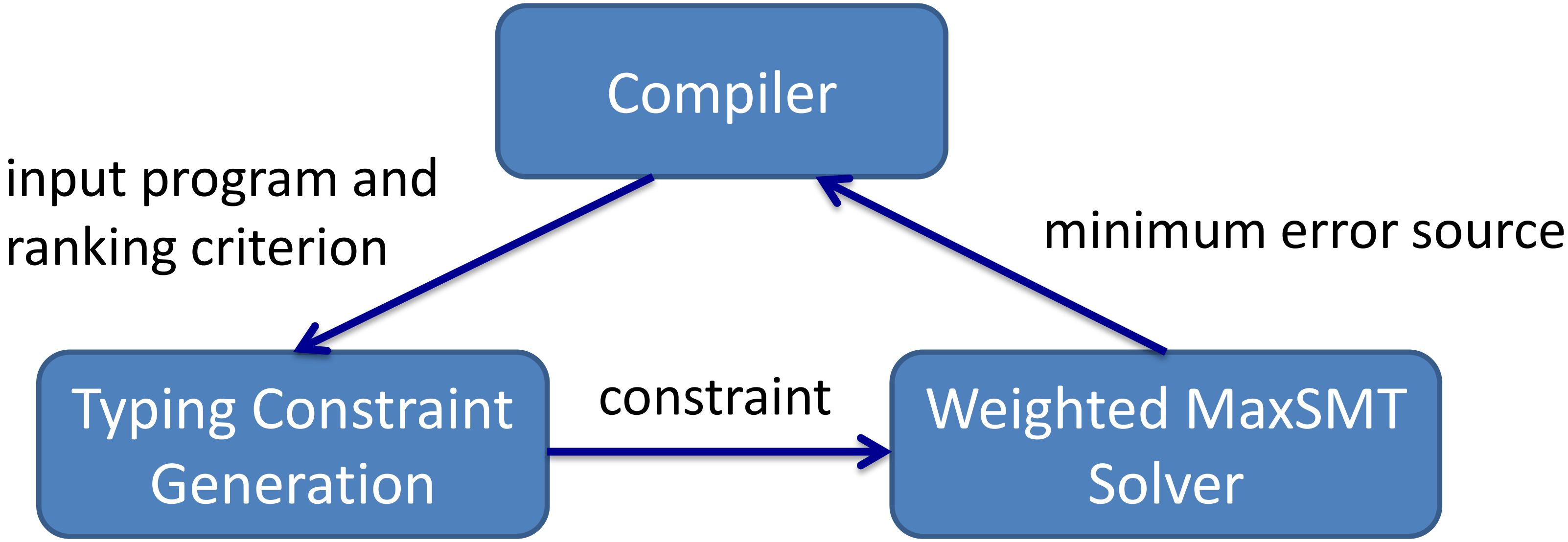
$$\alpha_{app2} = \text{fun}(\alpha_{i2}, \alpha_{o2})$$
$$\alpha_{i2} = \text{bool}$$
$$\alpha_{app2} = \alpha_{id2}$$
$$\alpha_{id2} = \text{fun}(\alpha_{x2}, \alpha_{r2})$$
$$\alpha_{x2} = \alpha_{r2}$$

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

System Architecture



System Architecture



- support various type systems due to SMT
- modest compiler modifications
- easy to change the ranking criterion

Prototype Implementation

- Subset of OCaml (roughly Caml light)
- Weighted MaxSMT procedure of νZ (branch of Z3)
- 15% more accuracy than OCaml's type checker (even with a rather simplistic ranking criterion)
- Good scalability (a few seconds for several thousand lines of code)

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, F#, Haskell, …)
- Still work to be done for more expressive type systems (GADTs, dependent types, …)