CSC488: Type Checking

Anthony Vandikas

University of Toronto

March 22, 2018

Overview

Simple Type Checking

The simplest type checker is a recursive function that returns the type of a term.

```
type-of : term environment → type (or error)
```

Each term must contain enough information to fully determine its type in isolation.

Simple Type Checking

Terms:

- x [variables]
- n [integers]
- b [booleans]
- (λ (x : A) t) [abstraction]
- (app t_1 t_2) [application]
- (set! x t) [mutation]
- (if t_1 t_2 t_3) [branching]

Types:

- int
- bool
- unit
- (A → B)

Simple Type Checking

Problem: The return value of `set!` is undefined. No one should be able to observe it's value.

Solution: Give it a unique type `unit` that supports no special operations.

Simple Type Checking

Gamma

This means "t has type A in context Γ" (where Γ usually is a map from variables to types):

$$\Gamma \vdash t : A$$

This means "if the hypotheses hold, then the conclusion is true":

$$\frac{\text{Hypothesis 1} \qquad \ldots \qquad \text{Hypothesis n}}{\text{Conclusion}}$$

Simple Type Checking

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : \boxed{A}} \qquad \frac{n \in \mathbb{Z}}{\Gamma \vdash n : \boxed{\texttt{int}}} \qquad \frac{b \in \mathbb{B}}{\Gamma \vdash b : \boxed{\texttt{bool}}}$$

$$\frac{x : A; \Gamma \vdash t : B}{\Gamma \vdash (\lambda\ (x\ :\ A)\ t) : \boxed{(A \to B)}}$$

$$\frac{\Gamma \vdash t_1 : (A \to B) \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash (\texttt{app}\ t_1\ t_2) : \boxed{B}}$$

$$\frac{x : A \in \Gamma \qquad \Gamma \vdash t : A}{\Gamma \vdash (\texttt{set!}\ x\ t) : \boxed{\texttt{unit}}}$$

same type

$$\frac{\Gamma \vdash t_1 : \texttt{bool} \qquad \Gamma \vdash t_2 : \texttt{A} \qquad \Gamma \vdash t_3 : \texttt{A}}{\Gamma \vdash (\texttt{if}\ t_1\ t_2\ t_3) : \boxed{\texttt{A}}}$$

These rules provide an obvious implementation strategy, but this is not always the case.

Overview

Parametric Polymorphism

What type goes in the _?

```
(λ (x : _) x)
```

If we pick 'int', it only works with integers.
If we pick 'bool', it only works with booleans.
...

We would like to let functions take *types* as arguments so that they can work with values of any type.

```
(Λ (α) (λ (x : (var α)) x))
```

Two new term forms:

- `(Λ (α) t)  [type abstraction]`
- `(spec t A) [type application]`

Two new type forms:

- `(var α)    [type variables]`
- `(∀ (α) A)  [universal types]`

The expression

```
(Λ (α) (λ (x : (var α)) x))
```

gets the type

```
(∀ (α) ((var α) → (var α)))
```

Parametric Polymorphism

$$\frac{x : A \in \Gamma}{\Delta, \Gamma \vdash x : \colorbox{orange}{A}} \qquad \frac{n \in \mathbb{Z}}{\Delta, \Gamma \vdash n : \colorbox{orange}{$\mathtt{int}$}} \qquad \frac{b \in \mathbb{B}}{\Delta, \Gamma \vdash b : \colorbox{orange}{$\mathtt{bool}$}}$$

$$\frac{\Delta, x : A; \Gamma \vdash t : B}{\Delta, \Gamma \vdash (\lambda\ (x\ :\ A)\ t) : \colorbox{orange}{$(A \to B)$}}$$

$$\frac{\alpha; \Delta, \Gamma \vdash t : A \qquad {\color{red}\Delta \vdash A\ \mathtt{type}}}{\Delta, \Gamma \vdash (\Lambda\ (\alpha)\ t) : \colorbox{orange}{$(\forall\ (\alpha)\ A)$}}$$

$$\frac{\Delta, \Gamma \vdash t_1 : (A \to B) \qquad \Delta, \Gamma \vdash t_2 : A}{\Delta, \Gamma \vdash (\mathsf{app}\ t_1\ t_2) : \colorbox{orange}{B}}$$

$$\frac{\Delta, \Gamma \vdash t : (\forall\ (\alpha)\ A) \qquad \Delta \vdash B\ \mathtt{type}}{\Delta, \Gamma \vdash (\mathtt{spec}\ t\ B) : \colorbox{orange}{${\color{red}A[\alpha / B]}$}}$$

$$\frac{x : A \in \Gamma \qquad \Delta, \Gamma \vdash t : A}{\Gamma \vdash (\mathtt{set!}\ x\ t) : \colorbox{orange}{$\mathtt{unit}$}}$$

$$\frac{\Delta, \Gamma \vdash t_1 : \mathtt{bool} \qquad \Delta, \Gamma \vdash t_2 : \mathtt{A} \qquad \Delta, \Gamma \vdash t_3 : \mathtt{A}}{\Delta, \Gamma \vdash (\mathtt{if}\ t_1\ t_2\ t_3) : \colorbox{orange}{$\mathtt{A}$}}$$

Parametric Polymorphism

The judgement

$$\Delta \vdash A \ \texttt{type}$$

means "A is a well-scoped type in context Δ".

$$\frac{}{\Delta \vdash \texttt{int type}} \qquad \frac{}{\Delta \vdash \texttt{bool type}} \qquad \frac{}{\Delta \vdash \texttt{unit type}}$$

$$\frac{\alpha \in \Delta}{\Delta \vdash (\texttt{var}\ \alpha)\ \texttt{type}} \qquad \frac{\alpha; \Delta \vdash A\ \texttt{type}}{\Delta \vdash (\forall\ (\alpha)\ A)\ \texttt{type}}$$

$$\frac{\Delta \vdash A\ \texttt{type} \qquad \Delta \vdash B\ \texttt{type}}{\Delta \vdash (A \to B)\ \texttt{type}}$$

Parametric Polymorphism

$$\frac{\Delta, \Gamma \vdash t : (\forall\ (\alpha)\ A) \qquad \Delta \vdash B\ \mathtt{type}}{\Delta, \Gamma \vdash (\mathtt{spec}\ t\ B) : A[\alpha/B]}$$

We need to specialize ∀-types for specific values of α. For example

```
(spec (Λ (α) (λ (x : (var α)) (var x)))
      int)
```

should have type

```
(int → int)
```

which we obtain by substituting `int` for `α` in `((var α) → (var α))`.

Parametric Polymorphism

What happens when we substitute `((var α) → int)` for β in `(∀ (α) ((var β) → (var α)))`? Naive substitution gives us:

```
(∀ (α) (((var α) → int) → (var α)))
```

The correct result should be:

```
(∀ (γ) (((var α) → int) → (var γ)))
```

Solution: rename all bound variables during substitution.

Parametric Polymorphism

```
(define (rename A α β)
  (define (rename′ A) (rename A α β))
  (match A
    [`(,B → ,C)    `(,(rename′ B) → ,(rename′ C))]
    [`(∀ (,γ) ,B) `(∀ (,γ) ,(if (equal? α γ) B (rename′ B)))]
    [_             (if (equal? A α) β A)]))

(define (subst A α B)
  (define (subst′ A) (subst A α B))
  (match A
    [`(,C → ,D)    `(,(subst′ C) → ,(subst′ D))]
    [`(∀ (,β) ,C) (define γ (gensym))
                   (define C′ (rename C β γ))
                   `(∀ (,γ) (subst′ C′))]
    [_             (if (equal? A α) B A)]))
```

Overview

Simple Type Checking

Parametric Polymorphism

Type Inference

Let Generalization

Conclusion

Type Inference

We want to infer all Λ and `spec` forms as well as λ type annotations. Unfortunately, **type inference is undecidable for the previous type system**.

Restriction: only let ∀ appear in the outermost part of a type.

We distinguish between *mono-types*

- `(var α)`
- `int`
- `bool`
- `unit`
- `(A → B)`

and *poly-types*

- `(∀ (α ...) A)`

Type Inference

Without type annotations, it is impossible to determine the type of an expression without looking at the surrounding code. We will split type checking into three parts:

```
infer : term environment → mono-type (list-of constraint)
solve : (list-of constraint) → assignment
generalize : mono-type → poly-type
```

Type Inference

The `solve` function is implemented as a unification algorithm.

Robinson's unification algorithm:

```
G ∪ { A ≡ A }                 ⇒ G
G ∪ { (A → B) ≡ (C → D) } ⇒ G ∪ { A ≡ C, B ≡ D }
G ∪ { (A → B) ≡ c }         ⇒ error where c ∈ {int,bool,unit}
G ∪ { c ≡ (A → B) }         ⇒ error where c ∈ {int,bool,unit}
G ∪ { x ≡ A }                 ⇒ G[x/A] if  x ∉ vars(A)
                                           and x ∈ vars(G)
                                                free variables of G
G ∪ { x ≡ A }                 ⇒ error if x ∈ vars(A)
G ∪ { A ≡ x }                 ⇒ G ∪ { x ≡ A }
```

Break down equations into smaller constraints until one side is a variable, then perform substitution.

Overview

Let Generalization

This expression

```
(let ([id (λ (x) x)])
  (id 3)
  (id #t))
```

is equivalent to

```
((λ (id)
    ((λ (_) (id #t))
     (id 3)))
 (λ (x) x))
```

Problem: above expression does not pass type checker.

1. `id` gets type `(α → α)`
2. `id` gets passed 3, so `α ≡ int`
3. `id` gets passed `#t`, so `α ≡ bool`
4. Conflict!

Let Generalization

`id` *should* get type `(∀ (α) (α → α))`

Solution: handle `let` as a special case and generalize during the `infer` step. before the rest is generalized

Be careful not to generalize variables that don't belong to you!

```
(λ (x)
  (let [f (λ (y) x)]
    ...))
```

Before generalization, `x : α`, `y : β`, and `f : (β → α)`.

After generalization, we want `f : (∀ (β) (α → β))`, **not** `f : (∀ (α β) (α → β))`. x's type responsible

Only quantify over variables that do not appear in the surrounding environment.

Overview

Conclusion

Stuff we didn't get to cover:

- Static Analysis
  - Dataflow Analysis: approximating the set of values a variable can take at some point in the program
- Program Optimization
  - Inlining: reduce function call overhead
  - Register allocation: keeping relevant data in registers
  - Strength reduction/Peephole Optimization: replacing known sets of instructions with faster ones
  - Dead code elimination: removing unreachable code
  - Deforestation/Fusion: removing redundant intermediate data structures (e.g. `(map f (map g l)) = (map (compose f g) l)`)
  - Common Sub-expression Elimination: factoring out common code so that it's not evaluated multiple times
  - Constant Folding/Partial Evaluation: evaluating code ahead of time
  - ...
- Nanopass Framework

Conclusion

Questions?