CSC488: Type Checking

Anthony Vandikas

University of Toronto

March 22, 2018
Overview

Simple Type Checking

Parametric Polymorphism

Type Inference

Let Generalization

Conclusion
Simple Type Checking

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

\[
\text{type-of} : \text{term environment} \rightarrow \text{type (or error)}
\]

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

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

Types:
- int
- bool
- unit
- $(A \rightarrow B)$
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

This means “$t$ has type $A$ in context $\Gamma$” (where $\Gamma$ usually is a map from variables to types):

$$\Gamma \vdash t : A$$

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

Hypothesis 1 \hspace{1cm} \ldots \hspace{1cm} Hypothesis n

Conclusion
Simple Type Checking

\[
\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \quad \frac{n \in \mathbb{Z}}{\Gamma \vdash n : \text{int}} \quad \frac{b \in \mathbb{B}}{\Gamma \vdash b : \text{bool}}
\]

\[
\frac{x : A; \Gamma \vdash t : B}{\Gamma \vdash (\lambda (x : A) t) : (A \to B)} \quad \frac{\Gamma \vdash t_1 : (A \to B) \quad \Gamma \vdash t_2 : A}{\Gamma \vdash (\text{app} \ t_1 \ t_2) : B}
\]

\[
\frac{x : A \in \Gamma}{\Gamma \vdash t : A} \quad \frac{\Gamma \vdash t : A}{\Gamma \vdash (\text{set!} \ x \ t) : \text{unit}}
\]

\[
\frac{\Gamma \vdash t_1 : \text{bool} \quad \Gamma \vdash t_2 : A \quad \Gamma \vdash t_3 : A}{\Gamma \vdash (\text{if} \ t_1 \ t_2 \ t_3) : A}
\]

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

Simple Type Checking

Parametric Polymorphism

Type Inference

Let Generalization

Conclusion
Parametric Polymorphism

What type goes in the \(_\)?

\((\lambda (x : _) x)\)

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 \textit{types} as arguments so that they can work with values of any type.

\((\Lambda (\alpha) (\lambda (x : (\text{var} \ \alpha)) x))\)
Two new term forms:

- \((\Lambda (\alpha) \ t)\) [type abstraction]
- \((\text{spec} \ t \ A)\) [type application]

Two new type forms:

- \((\text{var} \ \alpha)\) [type variables]
- \((\forall (\alpha) \ A)\) [universal types]

The expression

\[(\Lambda (\alpha) (\lambda (x : (\text{var} \ \alpha)) \ x))\]

gets the type

\[(\forall (\alpha) ((\text{var} \ \alpha) \to (\text{var} \ \alpha)))\]
**Parametric Polymorphism**

\[ \frac{x : A \in \Gamma}{\Delta, \Gamma \vdash x : A} \quad \frac{n \in \mathbb{Z}}{\Delta, \Gamma \vdash n : \text{int}} \quad \frac{b \in \mathbb{B}}{\Delta, \Gamma \vdash b : \text{bool}} \]

\[ \frac{\Delta, x : A; \Gamma \vdash t : B}{\Delta, \Gamma \vdash (\lambda (x : A) \ t) : (A \to B)} \]

\[ \frac{\alpha; \Delta, \Gamma \vdash t : A \quad \Delta \vdash A \text{ type}}{\Delta, \Gamma \vdash (\forall (\alpha) \ t) : (\forall (\alpha) \ A)} \]

\[ \frac{\Delta, \Gamma \vdash t_1 : (A \to B) \quad \Delta, \Gamma \vdash t_2 : A}{\Delta, \Gamma \vdash (\text{app} \ t_1 \ t_2) : B} \]

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

\[ \frac{x : A \in \Gamma \quad \Delta, \Gamma \vdash t : A}{\Gamma \vdash (\text{set!} \ x \ t) : \text{unit}} \]

\[ \frac{\Delta, \Gamma \vdash t_1 : \text{bool} \quad \Delta, \Gamma \vdash t_2 : A \quad \Delta, \Gamma \vdash t_3 : A}{\Delta, \Gamma \vdash (\text{if} \ t_1 \ t_2 \ t_3) : A} \]
Parametric Polymorphism

The judgement

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

means “$A$ is a well-scoped type in context $\Delta$.”

- $\Delta \vdash \text{int type}$
- $\Delta \vdash \text{bool type}$
- $\Delta \vdash \text{unit type}$

- $\alpha \in \Delta \quad \frac{}{\Delta \vdash (\text{var } \alpha) \text{ type}}$
- $\Delta \vdash A \text{ type} \quad \frac{}{\Delta \vdash (\forall (\alpha) \ A) \text{ type}}$

- $\Delta \vdash A \text{ type} \quad \Delta \vdash B \text{ type} \quad \frac{}{\Delta \vdash (A \rightarrow B) \text{ type}}$
Parametric Polymorphism

\[
\Delta, \Gamma \vdash t : (\forall (\alpha) A) \quad \Delta \vdash B \text{ type}
\]
\[
\Delta, \Gamma \vdash (\text{spec } t \ B) : A[\alpha/B]
\]

We need to specialize \(\forall\)-types for specific values of \(\alpha\). For example

\[
(\text{spec } (\Lambda (\alpha) (\lambda (x : (\text{var } \alpha)) (\text{var } x))) \ \text{int})
\]

should have type

\[
(\text{int } \rightarrow \text{int})
\]

which we obtain by substituting \(\text{int}\) for \(\alpha\) in \(((\text{var } \alpha) \rightarrow (\text{var } \alpha))\).
What happens when we substitute \(((\text{var } \alpha) \to \text{int})\) for \(\beta\) in \((\forall (\alpha) ((\text{var } \beta) \to (\text{var } \alpha)))\)? Naive substitution gives us:

\[(\forall (\alpha) (((\text{var } \alpha) \to \text{int}) \to (\text{var } \alpha)))\]

The correct result should be:

\[(\forall (\gamma) (((\text{var } \alpha) \to \text{int}) \to (\text{var } \gamma)))\]

**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 $\Lambda$ and spec forms as well as $\lambda$ type annotations. Unfortunately, **type inference is undecidable for the previous type system**.

**Restriction:** only let $\forall$ appear in the outermost part of a type.

We distinguish between *mono-types*

- $(\text{var } \alpha)$
- int
- bool
- unit
- $(A \rightarrow B)$

and *poly-types*

- $(\forall (\alpha \ldots) A)$
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:

\[
\begin{align*}
\text{infer} &: \text{term environment} \to \text{mono-type (list-of constraint)} \\
\text{solve} &: (\text{list-of constraint}) \to \text{assignment} \\
\text{generalize} &: \text{mono-type} \to \text{poly-type}
\end{align*}
\]
Type Inference

The solve function is implemented as a unification algorithm.

Robinson’s unification algorithm:

\[
\begin{align*}
G \cup \{ A \equiv A \} & \Rightarrow G \\
G \cup \{ (A \rightarrow B) \equiv (C \rightarrow D) \} & \Rightarrow G \cup \{ A \equiv C, B \equiv D \} \\
G \cup \{ (A \rightarrow B) \equiv c \} & \Rightarrow \text{error where } c \in \{\text{int, bool, unit}\} \\
G \cup \{ c \equiv (A \rightarrow B) \} & \Rightarrow \text{error where } c \in \{\text{int, bool, unit}\} \\
G \cup \{ x \equiv A \} & \Rightarrow G[x/A] \text{ if } x \notin \text{vars}(A) \text{ and } x \in \text{vars}(G) \\
G \cup \{ x \equiv A \} & \Rightarrow \text{error if } x \in \text{vars}(A) \\
G \cup \{ A \equiv x \} & \Rightarrow G \cup \{ x \equiv A \}
\end{align*}
\]

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

Simple Type Checking

Parametric Polymorphism

Type Inference

Let Generalization

Conclusion
Let Generalization

This expression

\[
(\text{let (id (λ (x) x))})
\]
\[
(id 3)
\]
\[
(id \#t))
\]

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 \((\forall (\alpha) (\alpha \to \alpha))\)

Solution: handle let as a special case and generalize during the infer step.

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

\((\lambda (x)
\quad (\text{let } [f (\lambda (y) x)]
\quad \ldots))\)

Before generalization, \(x : \alpha\), \(y : \beta\), and \(f : (\beta \to \alpha)\).

After generalization, we want \(f : (\forall (\beta) (\alpha \to \beta))\), not \(f : (\forall (\alpha \beta) (\alpha \to \beta))\).

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

Simple Type Checking

Parametric Polymorphism

Type Inference

Let Generalization

Conclusion
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. \((\text{map } f (\text{map } g l)) = (\text{map } (\text{compose } f \ g) l)\))
  ▶ 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?