CSC 520, Spring 2020

Principles of Programming Languages

Michelle Strout

THE UNIVERSITY OF ARIZONA

Version: 1

April 5, 2020

Plan

- **Announcements**
  - HW7 is due Wednesday April 8th
  - HW8 was posted last Friday and is due Wednesday April 15th
- **Last time**
  - What is type soundness?
  - Formation, Introduction, and Elimination Rules
  - Type checking with type constructors
- **Today**
  - Polymorphic type systems (typed uScheme)
  - Generic type representations
  - Kinds for classifying types
  - Why we want to do type inference

Monomorphic types are limiting

THE UNIVERSITY OF ARIZONA
Computer Science

- **Each new type constructor requires**
  - Special syntax
  - New type rules
  - New internal representation (type formation)
  - New code in type checker (introduction, elimination)
  - New or revised proof of soundness

Monomorphic burden: Array types

Formation:

$$\frac{\tau \text{ is a type}}{\mathsf{ARRAY}(\tau) \text{ is a type}}$$

Introduction:

$$\frac{\Gamma \vdash e_1 : \mathsf{INT} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{AMAKE}(e_1, e_2) : \mathsf{ARRAY}(\tau)}$$

Elimination:

$$\frac{\Gamma \vdash e_1 : \mathsf{ARRAY}(\tau) \qquad \Gamma \vdash e_2 : \mathsf{INT}}{\Gamma \vdash \mathsf{AAT}(e_1, e_2) : \tau}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{ARRAY}(\tau) \qquad \Gamma \vdash e_2 : \mathsf{INT} \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{APUT}(e_1, e_2, e_3) : \tau}$$

$$\frac{\Gamma \vdash e : \mathsf{ARRAY}(\tau)}{\Gamma \vdash \mathsf{ASIZE}(e) : \mathsf{INT}}$$

Monomorphism hurts programmers too

- **Leads to code duplication**

- **User-defined functions are monomorphic**

```
(define int lengthI ([xs : (list int)])
   (if (null? xs) 0 (+ 1 (lengthI (cdr xs)))))

(define int lengthB ([xs : (list bool)])
   (if (null? xs) 0 (+ 1 (lengthB (cdr xs)))))

(define int lengthS ([xs : (list sym)])
   (if (null? xs) 0 (+ 1 (lengthS (cdr xs)))))
```

Quantified types

Heart of polymorphism: $\forall \alpha_1, \ldots, \alpha_n \,.\, \tau$.

In Typed μScheme: `(forall ('a1 ... 'an) type)`

Two ideas:

- **Type variable `'a` stands for an unknown type**
- **Quantified type (with `forall`) enables substitution**

$$
\begin{array}{ll}
\texttt{length} & : \forall \alpha \,.\, \alpha\ \texttt{list} \to \texttt{int} \\
\texttt{cons} & : \forall \alpha \,.\, \alpha \times \alpha\ \texttt{list} \to \alpha\ \texttt{list} \\
\texttt{car} & : \forall \alpha \,.\, \alpha\ \texttt{list} \to \alpha \\
\texttt{cdr} & : \forall \alpha \,.\, \alpha\ \texttt{list} \to \alpha\ \texttt{list} \\
\texttt{'()} & : \forall \alpha \,.\, \alpha\ \texttt{list}
\end{array}
$$

Quantified types

Heart of polymorphism: $\forall \alpha_1, \ldots, \alpha_n \,.\, \tau$.

In Typed μScheme: `(forall ('a1 ... 'an)` *`type`*`)`

Two ideas:

- Type variable `'a` stands for an unknown type
- Quantified type (with `forall`) enables substitution

```
car    : (forall ('a) ([list 'a] -> 'a))
cdr    : (forall ('a) ([list 'a] -> [list 'a]))
cons   : (forall ('a) ('a [list 'a] -> [list 'a]))
'()    : (forall ('a) (list 'a))
length : (forall ('a) ([list 'a] -> int))
```

Type formation: Composing types

- **Punchline: Now we need a somewhat different approach to type creation, specifically using kinds**

- **Typed Impcore**
  - Closed world (no new types)
  - Simple formation rules

- **Standard ML**
  - Open world (programmers create new types)
  - How are types formed (from other types)?

- **Can't add new syntactic forms and new type formation rules for every new type**

Representing type constructors generically

THE UNIVERSITY OF ARIZONA
Computer Science

Start with monomorphic fragment (Typed μScheme):

```
datatype tyex
  = TYCON  of name
  | CONAPP of tyex * tyex list
  | FUNTY  of tyex list * tyex      (* I'm special *)
```

Examples: `bool`, `(list int)`, `(int int -> b ool)`

```
TYCON "bool"
CONAPP (TYCON "list", [TYCON "int"])
CONAPP (FUNTY [TYCON "int", TYCON "int"],
              TYCON "bool")
```

Hard to read, but easy to write code for.

Question: How would you represent an array of pairs of booleans?

```
datatype tyex
  = TYCON  of name
  | CONAPP of tyex * tyex list
  | FUNTY  of tyex list * tyex
```

`(bool * bool) array` **ML**

`(array (pair bool bool))` **Typed μScheme**

```
CONAPP (TYCON "array",
  [CONAPP (TYCON "pair", [TYCON "bool", TYCON "bool"])])
```

Well-formed types

THE UNIVERSITY OF ARIZONA
Computer Science

- **We still need to classify type expressions into:**
  - **Types** that classify terms (e.g., int)
  - **Type constructors** that build types (e.g., list)
  - **Nonsense** that means nothing (e.g., int int)
- **Idea: kinds classify types**
- **One-off type-formation rules**
- **Delta tracks type constructors, vars**

Return to quantified types

Heart of polymorphism: $\forall \alpha_1, \ldots, \alpha_n \,.\, \tau$.

In Typed μScheme: `(forall ('a1 ... 'an) type)`

Two ideas:

- Type variable `'a` stands for an unknown type
- Quantified type (with `forall`) enables substitution

$$
\begin{array}{ll}
\texttt{length} & : \forall \alpha \,.\, \alpha\ \texttt{list} \to \texttt{int} \\
\texttt{cons} & : \forall \alpha \,.\, \alpha \times \alpha\ \texttt{list} \to \alpha\ \texttt{list} \\
\texttt{car} & : \forall \alpha \,.\, \alpha\ \texttt{list} \to \alpha \\
\texttt{cdr} & : \forall \alpha \,.\, \alpha\ \texttt{list} \to \alpha\ \texttt{list} \\
\texttt{'()} & : \forall \alpha \,.\, \alpha\ \texttt{list}
\end{array}
$$

Representing quantified types

The University of Arizona
Computer Science

- **Two new alternatives for `tyex`**

```
datatype tyex
  = TYCON of name
  | CONAPP of tyex * tyex list
  | FUNTY of tyex list * tyex
  | TYVAR of name
  | FORALL of name list * tyex
```

- **Question: which are the new alternatives?**

Formation rules for quantified types

Reminder: $\Delta \vdash \tau :: *$ **means "τ is a type"**

$$\frac{\Delta\{\alpha_1 :: *, \ldots, \alpha_n :: *\} \vdash \tau :: *}{\Delta \vdash \mathsf{FORALL}([\alpha_1, \ldots, \alpha_n], \tau) :: *} \quad \textbf{(KindAll)}$$

$$\frac{\alpha \in \operatorname{dom} \Delta}{\Delta \vdash \mathsf{TYVAR}(\alpha) :: \Delta(\alpha)} \quad \textbf{(KindIntroVar)}$$

Programming with quantified types

- **Substitute for quantified variables**

```
-> length
<procedure> : (forall ('a) ((list 'a) -> int))

-> (@ length int)
<procedure> : ((list int) -> int)

-> (length '(1 2 3))
type error: function is polymorphic; instantiate before
applying

-> ((@ length int) '(1 2 3))
3 : int
```

- **The atsign (@) is instantiating the function with a type parameter**

(@ length (list int))

- **Question: how would we do a list of int lists?**

More "instantiations"

- **Explain each of the following:**

```
-> (val length-int (@ length int))
length-int : ((list int) -> int)

-> (val cons-bool (@ cons bool))
cons-bool : ((bool (list bool)) -> (list bool))

-> (val cdr-sym (@ cdr sym))
cdr-sym : ((list sym)-> (list sym))

-> (val empty-int (@ '() int))
() : (list int)
```

Create your own

- **Abstract over unknown type using type-lambda**

```
-> (val id (type-lambda ['a]
                (lambda ([x : 'a]) x )))
id : (forall ('a) ('a -> 'a))
```

- **‘a is a type parameter (an unknown type)**

- **This feature is parametric polymorphism**

Power comes at notational cost

THE UNIVERSITY OF ARIZONA. Computer Science

- **Function composition**

```
-> (val o (type-lambda ['a 'b 'c]
     (lambda ([f : ('b -> 'c)]
              [g : ('a -> 'b)])
      (lambda ([x : 'a]) (f (g x))))))
o : (forall ('a 'b 'c)
       (('b -> 'c) ('a -> 'b) -> ('a -> 'c)))
```

- **What was this in uScheme?**

```
-> (val o (lambda (f g) (lambda (x) (f (g x)))))
```

Type rules for polymorphism

Instantiate by substitution

$\forall$ elimination:

- Concrete syntax `(@ e τ1 ··· τn)`
- Rule (note new judgment form $\Delta, \Gamma \vdash e : \tau$):

$$\frac{\Delta, \Gamma \vdash e : \forall \alpha_1, \ldots, \alpha_n . \tau}{\Delta, \Gamma \vdash \text{TYAPPLY}(e, \tau_1, \ldots, \tau_n) : \tau[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]}$$

Substitution is in the book as function `tysubst`

(Also in the book: `instantiate`)

Type rules for polymorphism

THE UNIVERSITY OF ARIZONA
Computer Science

Generalize with type-lambda

$\forall$ introduction:

- Concrete syntax `(type-lambda [`$\alpha_1 \cdots \alpha_n$`]` e`)`
- Rule:

$$\frac{\Delta\{\alpha_1 :: *, \ldots \alpha_n :: *\}, \Gamma \vdash e : \tau \qquad \alpha_i \notin \mathrm{ftv}(\Gamma), \quad 1 \leq i \leq n}{\Delta, \Gamma \vdash \mathsf{TYLAMBDA}(\alpha_1, \ldots, \alpha_n, e) : \forall \alpha_1, \ldots, \alpha_n . \tau} \text{ (FORALL INTRODUCTION)}$$

Δ is kind environment (remembers α_i's are types)

What have we gained?

- No more introduction rules: instead use polymorphic functions
- No more elimination rules: instead, instantiate polymorphic functions
- But we still need formation rules, because you can’t trust code

```
-> (lambda ([a : array]) (Array.size a))
type error: used type constructor `array' as a type
-> (lambda ([x : (bool int)]) x)
type error: tried to apply type bool as type
constructor
-> (@ car list)
type error: instantiated at type constructor `list',
which is not a type
```

How can we know which types are OK?

THE UNIVERSITY OF ARIZONA
Computer Science

Return to well-formed types

To classify type expressions into:

- types that classify terms (e.g., `int`)
- type constructors that build types (e.g., `list`)
- nonsense that means nothing (e.g., `int int`)

Use judgment

$$\Delta \vdash \tau :: \kappa$$

Type formation through kinds

Each type constructor has a kind.

Type constructors of kind $\star$ classify terms

(`int` $:: *$, `bool` $:: *$)

$$\frac{}{* \text{ is a kind}} \quad (\text{KindFormationType})$$

Type constructors of arrow kinds are "types in waiting" (`list` $:: * \Rightarrow *$, `pair` $:: * \times * \Rightarrow \ *$)

$$\frac{\kappa_1, \ldots, \kappa_n \text{ are kinds} \qquad \kappa \text{ is a kind}}{\kappa_1 \times \cdots \times \kappa_n \Rightarrow \kappa \text{ is a kind}} \quad (\text{KindFormationArrow})$$

Use kinds to give arities

Examples: `int` $:: *$, `list` $:: * \Rightarrow *$, `pair` $:: * \times * \Rightarrow *$

Non-Examples: `int int` and `bool` $\times$ `list` have no kind because they are nonsense.

Kinds classify type expressions just as *types classify terms*

The kinding judgment

$\Delta \vdash \tau :: \kappa$ "Type τ has kind κ"

$\Delta \vdash \tau :: *$ Special case: "τ is a type"

Replaces one-off type-formation rules

Kind environment Δ tracks type constructor names and kinds.

Kinding rules for types

THE UNIVERSITY OF ARIZONA
Computer Science

$$\frac{\mu \in \operatorname{dom} \Delta \qquad \Delta(\mu) = \kappa}{\Delta \vdash \mathsf{TYCON}(\mu) :: \kappa} \qquad (\textsc{KindIntroCon})$$

$$\frac{\Delta \vdash \tau :: \kappa_1 \times \cdots \times \kappa_n \Rightarrow \kappa \qquad \Delta \vdash \tau_i :: \kappa_i, \quad 1 \le i \le n}{\Delta \vdash \mathsf{CONAPP}(\tau, [\tau_1, \ldots, \tau_n]) :: \kappa} \qquad (\textsc{KindApp})$$

These two rules replace all formation rules.

(Check out book functions `kindof` and `asType`)

Kinds of primitive type constructors

$\Delta(\texttt{int}) = *$

$\Delta(\texttt{bool}) = *$

$\Delta(\texttt{list}) = * \Rightarrow *$

$\Delta(\texttt{option}) = * \Rightarrow *$

$\Delta(\texttt{pair}) = * \times * \Rightarrow *$

$\Delta(\texttt{queue}) =$ **You fill in**

$\Delta(\texttt{unit}) =$ **You fill in**

Opening a closed world

- **What can a programmer add?**
- **Typed Impcore**
  - Closed world no new types
  - Simple formation rules
- **Typed uScheme**
  - Semi-closed world (new type variables)
  - Types are formed from other types through explicit instantiation
- **Standard ML**
  - Open world (programmers create new types)
  - Types are formed through implicit instantiation

Type Inference Introduction

The University of Arizona
Computer Science

- **How does the compiler know the types without annotations, or implicitly, in SML?**

```
fun append (x::xs) ys = x :: append xs ys
  | append [] ys = ys
append [1 2 3]
```

- **Questions**
  - Where do explicit types appear in C?
  - Where do explicit types appear in Typed uScheme?
- **Let's get rid of explicit types with type inference**
  - Guess a type for each formal parameter
  - Guess a return type
  - Guess a type for each use of a polymorphic type

Type Inference

- **Key Ideas**
  - Fresh type variables represent unknown types
  - Example: In (lambda (x) (+ x 3)), assign x fresh type variable alpha
  - Constraints record knowledge about type variables
  - Example: alpha ~ int
- **Why study?**
  - Useful in its own right (enables implicit typing)
  - Canonical example of static analysis, which is a key tool in cybersecurity and high performance computing

What type inference accomplishes

- **The compiler tells you useful information and there is a lower annotation burden.**

```
-> (define      double (x)          (+ x x))
double                             ;; uScheme

-> (define int double ([x : int]) (+ x x))
double : (int -> int)              ;; Typed uSch.

-> (define      double (x)          (+ x x))
double : int -> int                ;; nML
```

What else type inference accomplishes

- **No longer need to instantiate types**

```
-> ((@ cons bool) #t ((@ cons bool) #f (@ '() bool)))
(#t #f) : (list bool)     ;; typed uScheme

-> (   cons        #t (   cons         #f     '()       ))
(#t #f) : bool list       ;; nML
```

Type inference

- **How it works**
  - 1. For each unknown type, introduce a fresh type variable
  - 2. Every typing rule adds equality constraints
  - 3. Instantiate every variable automatically
  - 4. Introduce polymorphism at 'let/val' bindings
- **Plan of Study**
  - Today see a couple of examples for how to generate constraints
  - Wednesday, many more examples with you doing some
  - Wednesday, you solving constraints by hand
  - Wed and Mon, ideas for how to write constraint solver for HW8

Couple of Examples

- **Example: if**

```
(if y 1 0)
```

- **Q: What constraints are needed?**
  - Alpha ~ bool, beta ~ int, alpha is type for y, beta is type for whole expression
- **Example: sometimes can't satisfy constraints**

```
(if z z (- 0 z))
```

- **Q: What constraints? Can we solve them?**
  - Alpha is type var for z, alpha ~ bool /\ alpha ~ int