CSCI 334:
Principles of Programming Languages

Lecture 8: Types I

Instructor: Dan Barowy

Williams

Announcements

We will go over map and fold activities from last Thursday during the next class.

---

algebraic datatypes

```
datatype treat =
   SNICKERS
 | TWIX
 | TOOTSIE_ROLL
 | DENTAL_FLOSS
```

- Each option is really a constructor in disguise.
- Those constructors can take parameters.

---

algebraic datatypes

```
datatype bag_o_treats =
   SNICKERS of int
 | TWIX of int
 | TOOTSIE_ROLL of int
 | DENTAL_FLOSS of int
```

- Each option is really a constructor in disguise.
- Those constructors can take parameters.
- ADTs are known as “disjoint unions” in SML (or “tagged unions”, or “discriminated unions”, or “variant”, or “choice type”, or “sum type”, …)

pattern matching ADTs with params

```
fun count_treats bag =
  case of bag
  SNICKERS     i => i
| TWIX         i => i
| TOOTSIE_ROLL i => i
| DENTAL_FLOSS i => i
```

pattern matching ADTs with params

or just…

```
fun count_treats (SNICKERS i)     = i
 |  count_treats (TWIX i)         = i
 |  count_treats (TOOTSIE_ROLL i) = i
 |  count_treats (DENTAL_FLOSS i) = i
```

type checking is exhaustive for ADTs

(this is occasionally exhausting for humans)

```
datatype Expr =
  Foo of int
| Bar of real
| Baz of string

fun eval (Foo f) = "foo " ^ (Int.toString f)
 |  eval (Baz b) = "baz " ^ b
```

```
stdIn:16.5-17.30 Warning: match nonexhaustive
          Foo f => ...
          Baz b => ...
```

type checking is exhaustive for ADTs

a nice trick to make warnings go away…

```
exception NotDoneYet
fun TODO() = raise NotDoneYet
```

```
datatype Expr =
  Foo of int
| Bar of real
| Baz of string

fun eval (Foo f) = "foo " ^ (Int.toString f)
 |  eval (Baz b) = "baz " ^ b
 |  eval (Bar b) = TODO()
```

(it does, however, now cause a dynamic error instead; use sparingly!)

type checking

(or, “how does ML know that my expression is wrong?”)

```
fun f(x:int) : int = “hello ” + x
```

```
stdIn:27.12-27.24 Error: operator and operand don't
agree [overload conflict]
  operator domain: [+ ty] * [+ ty]
  operand:         string * int
  in expression:
    "hello " + x
```

---

type checking

step 1: convert into lambda form

`fun f(x:int) : int = “hello ” + x`

`f = λx.“hello ” + x` convert into λ expression

`f = λx.((+ “hello ”) x)` assume + `= λx.λy.[[x + y]]`

The purpose of this step is to make all of the parts of an expression clear

(real compilers may/may not actually do this step)

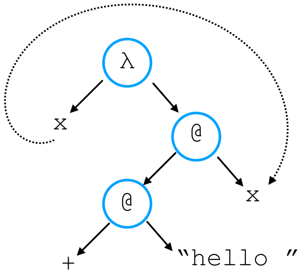
---

type checking

step 2: generate parse tree

`f = λx.((+ “hello ”) x)`

f has form `λx.((MM)M)`

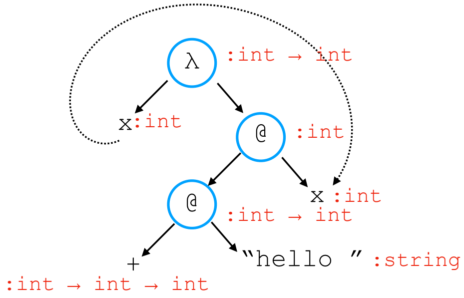


---

type checking

step 3: label parse tree with types

read “:” as “has type”



type checking

step 4: check that types are used consistently

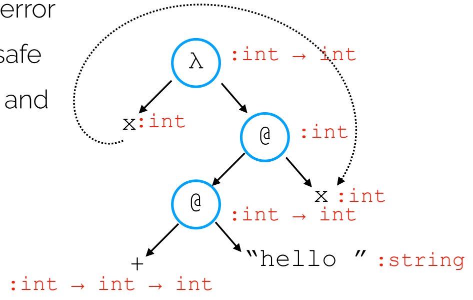
1. Start at the leaves
2. Do type mismatches arise?

   Yes = type error

   No = type safe
3. if yes, stop and report first mismatch

`int → int → int @ string`

`YES, TYPE ERROR`



---

type inference

notice that we had a typed expression

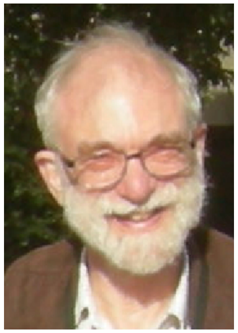
```
fun f(x:int) : int = “hello ” + x
```

what if, instead, we had

```
fun f(x) = “hello ” + x
```

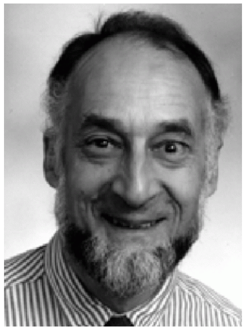
?

Hinley-Milner algorithm



J. Roger Hindley

- Hindley and Milner invented algorithm independently.
- Infers types from known data types and operations used.
- Depends on a step called “unification”.
- I will demonstrate informal method for unification; works for small examples



Robin Milner

Hinley-Milner algorithm

Has three main phases:

1. Assign type to each expression and subexpression
2. Generate type constraints based on rules of **λ** calculus:
   a. Abstraction constraints
   b. Application constraints
3. Solve type constraints using unification.

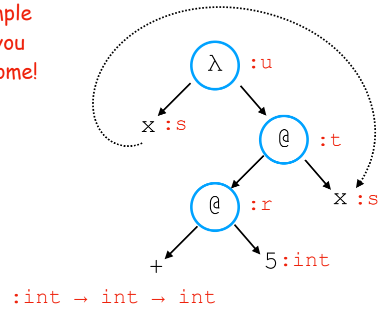
type inference

step 1: label parse tree with known/unknown types

`fun f(x) = 5 + x`

`f = λx.((+ 5) x)`

I am using the example from book so that you can follow along at home!



type inference

it is often helpful to have types in tabular form

subexpression	type
`+`	`int → int → int`
`5`	`int`
`(+5)`	`r`
`x`	`s`
`(+5)x`	`t`
`λx.((+ 5) x)`	`u`

type inference

step 2: generate type constraints using λ calculus

`M ::= x`	variable
`  | λx.M`	abstraction
`  | MM`	function application

Abstraction rule: If the type of `x` is `a` and the type of `M` is `b`, and the type of `λx.M` is `c`, then the constraint is `c = a → b`.

Application rule: If the type of M_1 is `a` and the type of M_2 is `b`, and the type of M_1M_2 is `c`, then the constraint is `a = b → c`.

type inference

subexpression	type	constraint
`+`	`int → int → int`	`n/a`
`5`	`int`	`n/a`
`(+5)`	`r`	`int → int → int = int → r`
`x`	`s`	`n/a`
`(+5)x`	`t`	`r = s → t`
`λx.((+ 5) x)`	`u`	`u = s → t`

type inference

step 3: unify

subexpression	type	constraint
+	int → int → int	n/a
5	int	n/a
(+5)	r	int → int → int = int → r
x	s	n/a
(+5)x	t	r = s → t
λx.((+ 5) x)	u	u = s → t

Start with the topmost unknown. What do we know about r?

int → int → int = int → r
r = int → int

---

type inference

step 3: unify

subexpression	type	constraint
+	int → int → int	n/a
5	int	n/a
(+5)	r = int → int	int → int → int = int → r
x	s	n/a
(+5)x	t	int → int = s → t
λx.((+ 5) x)	u	u = s → t

What do we know about s and t?

int → int = s → t
s = int
r = int

---

type inference

step 3: unify

subexpression	type	constraint
+	int → int → int	n/a
5	int	n/a
(+5)	r = int → int	int → int → int = int → r
x	s = int	n/a
(+5)x	t = int	int → int = s → t
λx.((+ 5) x)	u	u = int → int

What do we know about u?

u = int → int s = int
u = int → int

---

type inference

step 3: unify

subexpression	type	constraint
+	int → int → int	n/a
5	int	n/a
(+5)	r = int → int	int → int → int = int → r
x	s = int	n/a
(+5)x	t = int	int → int = s → t
λx.((+ 5) x)	u = int → int	u = int → int

Done when there is nothing left to do

(we will talk about polymorphic type inference next class)

completed type inference

```
fun f(x) = 5 + x
f = λx.((+ 5) x)
```