CSCI 334:
Principles of Programming Languages

Lecture 9: Types II

Instructor: Dan Barowy

Williams

Announcements

Graded HW2 back this weekend

(look for email from GitHub)

Activity Solution

```
fun is_older (y,m,d) (y',m',d') =
  y < y'
  orelse (y = y' andalso m < m')
  orelse (y = y' andalso m = m' andalso d < d')
```

Activity

Write a function `num_before_sum` that takes an `int` called `sum` (assume `sum` is positive) and an `int list` (assume all positive) and `returns an int`. The return value is an `n` such that the sum of the first `n` elements is < `sum` and the sum of the `n + 1` elements is >= `sum`. Assume that the sum of the entire list > `n`. Summing goes from left to right.

E.g.,
`num_before_sum 3 [0,1,2,3]` returns 2

Activity

```
fun num_before_sum sum xs =
  #1(
    foldl (fn (x,(n,acc)) =>
      if acc + x >= sum then
        (n, acc + x)
      else
        (n + 1, acc + x)
    ) (0,0) xs
  )
```

type inference

Recall motivation: type annotations are ugly and hard to get right.

Instead of `fun addOne(x: int) = x + 1`

Write `fun addOne x = x + 1`

polymorphic type inference

Type inference should work for “generic” code, too!

```
fun apply f x = f x
```

```
- apply addOne 1;
val it = 2 : int
```

```
fun prependHello x = "Hello " ^ x;
```

```
- apply prependHello "Dan";
val it = "Hello Dan" : string
```

polymorphic type inference

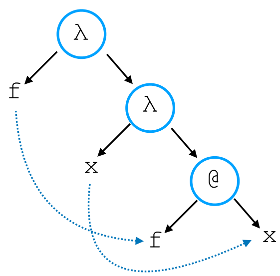
```
fun apply f x = f x
```

same basic procedure as “monomorphic” (i.e., non-generic) code:

1. convert to λ expression
2. label with type variables
3. generate constraints
4. unify
5. rename variables

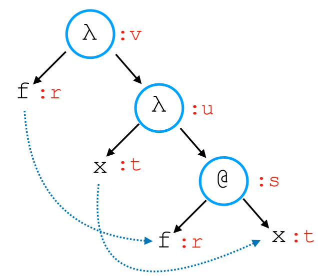
1. convert to λ expression

```
fun apply f x = f x
f = λf.λx.f x
```



2. label with type variables

```
fun apply f x = f x
f = λf.λx.f x
```



3. generate constraints

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

Variable rule: No constraint.

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.

3. generate constraints

x:a, M:b, c:λx.M => c = a → b

M_1:a, M_2:b, c:M_1M_2 => a = b → c

subexpression	type	constraint
f	r	n/a
x	s	n/a
f x	t	r = s → t
λx.f x	u	u = s → t
λf.λx.f x	v	v = r → u

4. unify

x:a, M:b, c:λx.M => c = a → b

M_1:a, M_2:b, c:$M_1 M_2$ => a = b → c

subexpression	type	constraint
f	s → t	n/a
x	s	n/a
f x	t	
λx.f x	u	u = s → t
λf.λx.f x	v	v = s → t → u

4. unify

x:a, M:b, c:λx.M => c = a → b

M_1:a, M_2:b, c:$M_1 M_2$ => a = b → c

subexpression	type	constraint
f	s → t	n/a
x	s	n/a
f x	t	
λx.f x	s → t	
λf.λx.f x	v	v = s → t → s → t

4. unify

x:a, M:b, c:λx.M => c = a → b

M_1:a, M_2:b, c:$M_1 M_2$ => a = b → c

subexpression	type	constraint
f	s → t	n/a
x	s	n/a
f x	t	
λx.f x	s → t	
λf.λx.f x	s → t → s → t	

5. rename variables in alpha order

x:a, M:b, c:λx.M => c = a → b

M_1:a, M_2:b, c:$M_1 M_2$ => a = b → c

subexpression	type	constraint
f	'a → t	n/a
x	'a	n/a
f x	t	
λx.f x	'a → t	
λf.λx.f x	'a → t → 'a → t	

5. rename variables in alpha order

x:a, M:b, c:λx.M => c = a → b

M_1:a, M_2:b, c:M_1M_2 => a = b → c

subexpression	type	constraint
f	'a → 'b	n/a
x	'a	n/a
f x	'b	
λx.f x	'a → 'b	
λf.λx.f x	'a → 'b → 'a → 'b	

5. rename variables in alpha order

x:a, M:b, c:λx.M => c = a → b

M_1:a, M_2:b, c:M_1M_2 => a = b → c

subexpression	type	constraint
f	'a → 'b	n/a
x	'a	n/a
f x	'b	
λx.f x	'a → 'b	
λf.λx.f x	'a → 'b → 'a → 'b	

Is this the right answer?

```
- fun apply f x = f x;
val apply = fn : ('a -> 'b) -> 'a -> 'b
```

activity

```
fun f g x = g(g x)
```

polymorphic type inference: a problem

```
let val id = fn x => x in
(id "hi", id true)
```

1. convert to lambda expression

```
let val id = fn x => x in
(id "hi", id true)
```

```
let val id = λx.x in
(id "hi", id true)
```

`let z = U in V` is the same as `(λz.V)U`

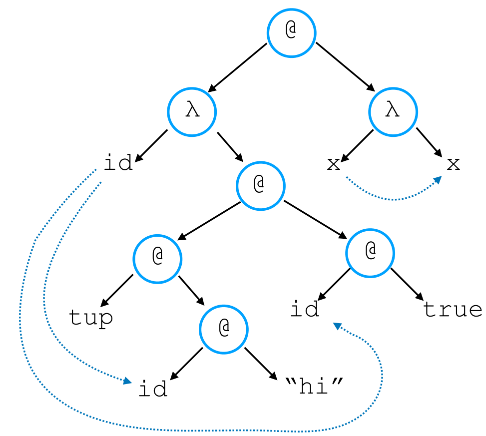
`(λid.(id "hi", id true))(λx.x)`

`tup: 'a → 'b → 'a * 'b`

`(λid.((tup (id "hi"))(id true))(λx.x)`

1. convert to lambda expression

`(λid.((tup (id "hi"))(id true))(λx.x)`



2. label types

	A	B	C
1	let val id = fn x => x in (id "hi", id true)		
2	**expr**	**type**	**constraint**
3	id		
4	"hi"		
5	tup		
6	(id "hi")		
7	(tup (id "hi"))		
8	TRUE		
9	(id true)		
10	((tup ...)(id true))		
11	(lam id.___)		
12	x		
13	(lam x.x)		
14	(lam id.___)(lam x.x)		

2. label types

	A	B	C
1	let val id = fn x => x in (id "hi", id true)		
2	**expr**	**type**	**constraint**
3	id	string -> e	
4	"hi"	string	
5	tup	'a -> 'b -> 'a * 'b	
6	(id "hi")	e	
7	(tup (id "hi"))	f	
8	TRUE	bool	
9	(id true)	g	
10	((tup ...)(id true))	h	
11	(lam id.___)	i	
12	x	j	
13	(lam x.x)	k	
14	(lam id.___)(lam x.x)	l	

3. generate constraints

x:a, M:b, c:λx.M => c = a → b

M_1:a, M_2:b, c:$M_1 M_2$ => a = b → c

3. generate constraints

	A	B	C
1	let val id = fn x => x in (id "hi", id true)		
2	**expr**	**type**	**constraint**
3	id	string -> e	
4	"hi"	string	
5	tup	'a -> 'b -> 'a * 'b	
6	(id "hi")	e	
7	(tup (id "hi"))	f	
8	TRUE	bool	
9	(id true)	g	
10	((tup ...)(id true))	h	
11	(lam id.___)	i	
12	x	j	
13	(lam x.x)	k	
14	(lam id.___)(lam x.x)	l	

3. generate constraints

	A	B	C
1	let val id = fn x => x in (id "hi", id true)		
2	**expr**	**type**	**constraint**
3	id	d	
4	"hi"	string	
5	tup	'a -> 'b -> 'a * 'b	
6	(id "hi")	e	d = string -> e
7	(tup (id "hi"))	f	'a -> 'b -> 'a * 'b = e -> f
8	TRUE	bool	
9	(id true)	g	d = bool -> g
10	((tup ...)(id true))	h	f = g -> h
11	(lam id.___)	i	i = d -> h
12	x	j	
13	(lam x.x)	k	k = j -> j
14	(lam id.___)(lam x.x)	l	i = k -> l

4. unify

	A	B	C
1	let val id = fn x => x in (id "hi", id true)		
2	**expr**	**type**	**constraint**
3	id	d	
4	"hi"	string	
5	tup	'a -> 'b -> 'a * 'b	
6	(id "hi")	e	d = string -> e
7	(tup (id "hi"))	f	'a -> 'b -> 'a * 'b = e -> f
8	TRUE	bool	
9	(id true)	g	d = bool -> g
10	((tup ...)(id true))	h	f = g -> h
11	(lam id.___)	i	i = d -> h
12	x	j	
13	(lam x.x)	k	k = j -> j
14	(lam id.___)(lam x.x)	l	i = k -> l

4. unify

	A	B	C
1	let val id = fn x => x in (id "hi", id true)		
2	**expr**	**type**	**constraint**
3	id	string -> e	
4	"hi"	string	
5	tup	'a -> 'b -> 'a * 'b	
6	(id "hi")	e	
7	(tup (id "hi"))	f	'a -> 'b -> 'a * 'b = e -> f
8	TRUE	bool	
9	(id true)	g	string -> e = bool -> g
10	((tup ...)(id true))	h	f = g -> h
11	(lam id.___)	i	i = string -> e -> h
12	x	j	
13	(lam x.x)	k	k = j -> j
14	(lam id.___)(lam x.x)	l	i = k -> l

replace d with `string -> e`

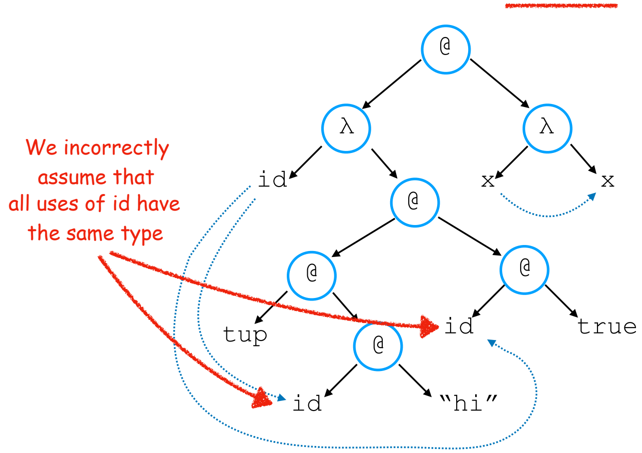
4. unify

	A	B	C
1	let val id = fn x => x in (id "hi", id true)		
2	**expr**	**type**	**constraint**
3	id	string -> e	
4	"hi"	string	
5	tup	'a -> 'b -> 'a * 'b	
6	(id "hi")	e	
7	(tup (id "hi"))	f	'a -> 'b -> 'a * 'b = e -> f
8	TRUE	bool	
9	(id true)	g	string -> e = bool -> g
10	((tup ...)(id true))	h	f = g -> h
11	(lam id.___)	i	i = string -> e -> h
12	x	j	
13	(lam x.x)	k	k = j -> j
14	(lam id.___)(lam x.x)	l	i = k -> l

Wait, what?!! **string = bool ?**

what went wrong?

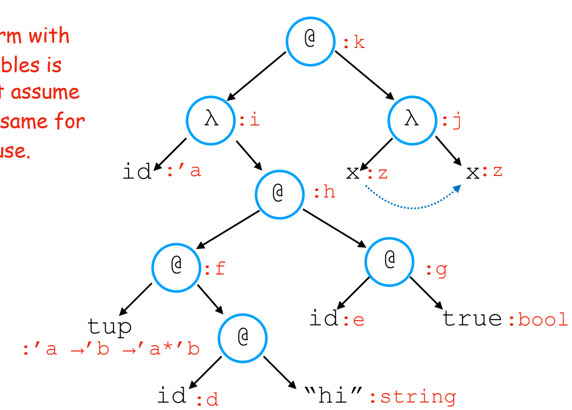
```
let val id = fn x => x in (id "hi", id true)
```



let-polymorphism

```
let val id = fn x => x in (id "hi", id true)
```

When a term with type variables is bound, don't assume that type is same for every use.



new language

programming is too complicated…

…because there are too many letters in a program!

Williams introduces

the Breph programming language

Breph language

only 8 commands:

```
e
p
h
r
a
i
m
!
```
(for extra effect)

programming now much simpler!

Breph language

we will be switching CS134 to Breph

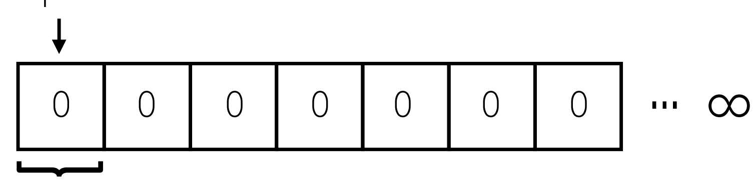
we need your help writing tutorial materials

i.e., hello world

(*for some reason* we can't figure it out)

Breph abstract machine

data pointer



1 byte

instruction pointer

```
hhhhhhhhhhhhhhhhhhhhhhhhhh
hhhhhhhhhhhhhhhhhhhhhhhh!
```

Breph language

```
e   move data pointer to the right
p   move data pointer to the left
h   increment byte at data pointer by 1
r   decrement byte at data pointer by 1
a   accept one byte of input; store @ data pointer loc
i   while loop start
m   while loop end
!   print byte @ data pointer loc using ascii table
```


Breph language

```
e   move data pointer to the right
p   move data pointer to the left
h   increment byte at data pointer by 1
r   decrement byte at data pointer by 1
a   accept one byte of input; store @ data pointer loc
i   while loop start
m   while loop end
!   print byte @ data pointer loc using ascii table
```

Breph language

```
e   move data pointer to the right
p   move data pointer to the left
h   increment byte at data pointer by 1
r   decrement byte at data pointer by 1
a   accept one byte of input; store @ data pointer loc
i   while loop start
m   while loop end
!   print byte @ data pointer loc using ascii table
```

Breph language

```
e   move data pointer to the right
p   move data pointer to the left
h   increment byte at data pointer by 1
r   decrement byte at data pointer by 1
a   accept one byte of input; store @ data pointer loc
i   while loop start
m   while loop end
!   print byte @ data pointer loc using ascii table
```

Breph language

e move data pointer to the right
p move data pointer to the left
h increment byte at data pointer by 1
r decrement byte at data pointer by 1
a accept one byte of input; store @ data pointer loc
i while loop start
m while loop end
! print byte @ data pointer loc using ascii table

Breph language

e move data pointer to the right
p move data pointer to the left
h increment byte at data pointer by 1
r decrement byte at data pointer by 1
a accept one byte of input; store @ data pointer loc
i while loop start
m while loop end
! print byte @ data pointer loc using ascii table

Breph language

e move data pointer to the right
p move data pointer to the left
h increment byte at data pointer by 1
r decrement byte at data pointer by 1
a accept one byte of input; store @ data pointer loc
i while loop start
m while loop end
! print byte @ data pointer loc using ascii table

Breph language

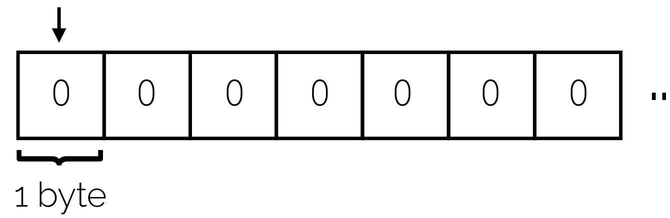
e move data pointer to the right
p move data pointer to the left
h increment byte at data pointer by 1
r decrement byte at data pointer by 1
a accept one byte of input; store @ data pointer loc
i while loop start
m while loop end
! print byte @ data pointer loc using ascii table

Breph language

e	move data pointer to the right
p	move data pointer to the left
h	increment byte at data pointer by 1
r	decrement byte at data pointer by 1
a	accept one byte of input; store @ data pointer loc
i	while loop start
m	while loop end
!	print byte @ data pointer loc using ascii table

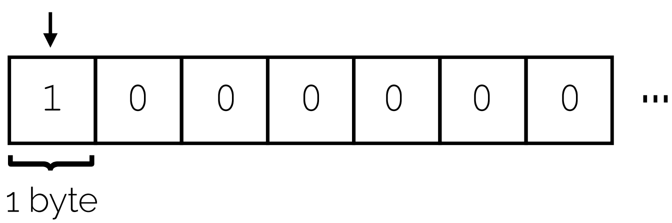
all commands advance instruction pointer
(except i and m, which *may* jump it elsewhere)

Breph abstract machine



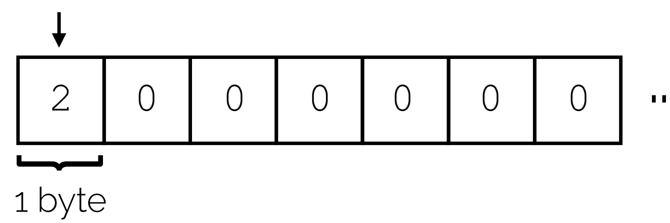

```
hhhhhhhhhhhhhhhhhhhhhhhhh
hhhhhhhhhhhhhhhhhhhhhhhh!
```

Breph abstract machine



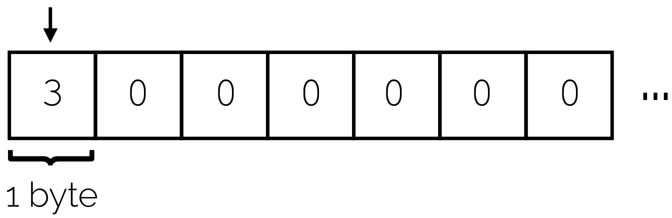

```
hhhhhhhhhhhhhhhhhhhhhhhhh
hhhhhhhhhhhhhhhhhhhhhhhh!
```

Breph abstract machine



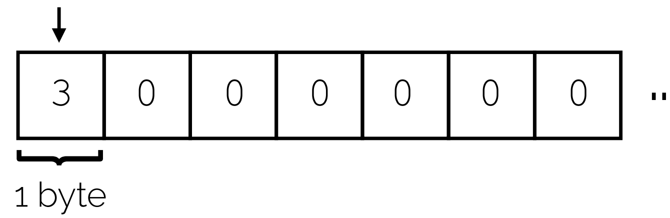

```
hhhhhhhhhhhhhhhhhhhhhhhhh
hhhhhhhhhhhhhhhhhhhhhhhh!
```

Breph abstract machine



hhhhhhhhhhhhhhhhhhhhhhhhh
hhhhhhhhhhhhhhhhhhhhhhh!

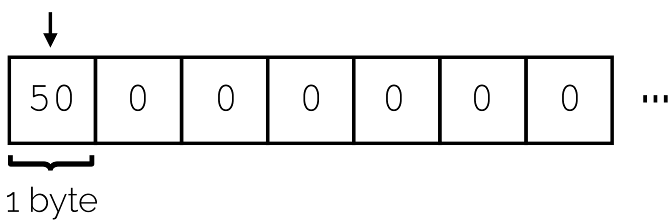
Breph abstract machine



hhhhhhhhhhhhhhhhhhhhhhhhh
hhhhhhhhhhhhhhhhhhhhhhh!

(skipping ahead)

Breph abstract machine

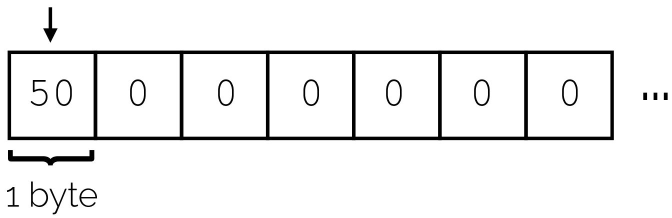


hhhhhhhhhhhhhhhhhhhhhhhhh
hhhhhhhhhhhhhhhhhhhhhhh!

ASCII conversion chart

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Breph abstract machine



"2"

```
hhhhhhhhhhhhhhhhhhhhhhhhhh
hhhhhhhhhhhhhhhhhhhhhhhh!
```

Breph abstract machine

Demo

```
hhhhhhhhhhhhhhhhhhhhhhhhhh
hhhhhhhhhhhhhhhhhhhhhhhh!
```

```
$ ./breph.rb program.moo
$ ruby breph.rb program.moo
```

Breph abstract machine

Challenge

Top 5 people who write the "brephest" program that prints "Hello Dan Barowy" will earn extra points

("brephest" = shortest)