(first, some unfinished business…)
The Span is...?

What's the span of $\text{Msort } T$?
when $T$ is balanced, depth $d$
Span of Ins

fun Ins (x, Empty) = Node(Empty, x, Empty)
| Ins (x, Node(t_1, y, t_2)) =
  case compare(x, y) of
    GREATER => Node(t_1, y, Ins(x, t_2))
    | _        => Node(Ins(x, t_1), y, t_2)

(no parallelism!)

For a balanced tree of depth d>0,
depth t_i ≤ d-1, so

\[ S_{\text{Ins}}(d) = 1 + S_{\text{Ins}}(d-1) \]

\[ S_{\text{Ins}}(d) \text{ is } O(d) \]

Span of SplitAt

\[
\text{fun SplitAt}(y, \text{Empty}) = (\text{Empty}, \text{Empty})
\]
\[
\mid \text{SplitAt}(y, \text{Node}(t_1, x, t_2)) =
\]
\[
\begin{cases}
\text{case compare}(x, y) \, \text{of} \\
\quad \text{GREATER} \Rightarrow \text{let val } (l_1, r_1) = \text{SplitAt}(y, t_1) \text{ in } (l_1, \text{Node}(r_1, x, t_2)) \text{ end} \\
\quad _{-} \Rightarrow \text{let val } (l_2, r_2) = \text{SplitAt}(y, t_2) \text{ in } (\text{Node}(t_1, x, l_2), r_2) \text{ end}
\end{cases}
\]

For a balanced tree of depth \(d>0\),

\[
S_{\text{SplitAt}}(d) = 1 + S_{\text{SplitAt}}(d-1)
\]

\(S_{\text{SplitAt}}(d)\) is \(O(d)\)
Span of Merge

\[
\text{fun Merge (Empty, } t_2) = t_2 \\
\quad \text{let val (} l_2, r_2) = \text{SplitAt}(x, t_2) \text{ in } \text{Node(Merge}(l_1, l_2), x, \text{Merge}(r_1, r_2)) \text{ end}
\]

For balanced trees of depth \(d > 0\),

the trees got by splitting have depth \(\leq d-1\), so

\[
S_{\text{Merge}}(d) = S_{\text{SplitAt}}(d) + \max(S_{\text{Merge}}(d-1), S_{\text{Merge}}(d-1)) = S_{\text{SplitAt}}(d) + S_{\text{Merge}}(d-1) = O(d) + S_{\text{Merge}}(d-1)
\]

\(S_{\text{Merge}}(d)\) is \(O(d^2)\)
Span of Msort

\textbf{fun} MsortEmpty = Empty
\begin{itemize}
\item Msort (Node(t_1, x, t_2)) =
\item \text{Ins} (x, \text{Merge}(\text{Msort} t_1, \text{Msort} t_2))
\end{itemize}

For a balanced tree of depth \(d\)

\[ S_{\text{Msort}}(d) = \max(S_{\text{Msort}}(d-1), S_{\text{Msort}}(d-1)) + S_{\text{Merge}}(d) + S_{\text{Ins}}(2d) \]

\[ = S_{\text{Msort}}(d-1) + O(d^2) \]

\( S_{\text{Msort}}(d) \) is \( O(d^3) \)
oops

• Our calculations assumed that trees obtained by splitting, merging, inserting were **balanced**

• That’s **NOT** true!
losing balance

Msort can produce badly balanced trees
• *Merge, Ins* don’t preserve balance!

• **We could** design a *balancing* function...

```
fun Msort Empty = Empty
| Msort (Node(t1, x, t2)) =
  balance(Ins (x, Merge(Msort t1, Msort t2)))
```

• Or implement new versions of *Ins* and *Merge* that do *preserve balance*
balanced vs sorted

• **Msort** produces a sorted tree

• Restoring balance is a lot of extra work!

• Later we will see how to build *nearly-balanced* sorted trees…

• …with the same asymptotic behavior as *perfectly-balanced* sorted trees
Type checking
Polymorphism
Type inference
type benefits

... a static check provides a runtime guarantee

If e has type t,
then e evaluates to a value of type t

If d declares x : t,
then d binds x to a value of type t

<table>
<thead>
<tr>
<th>static property</th>
<th>runtime guarantee</th>
</tr>
</thead>
<tbody>
<tr>
<td>e has type t</td>
<td>if e =&gt;* v then v : t</td>
</tr>
<tr>
<td>d declares x : t</td>
<td>if d =&gt;* [x : v] then v : t</td>
</tr>
</tbody>
</table>
advantages

Type analysis is easy, static, cheap

• A type error indicates a bug, detected, and prevented, without running code

• An unexpected type may also indicate a bug!

Values of a given type have predictable form

• We can use appropriate patterns for the type and design code accordingly

• Type information can also guide specifications and proofs
Referential transparency for types

How to tell \textit{statically} when $e : t$

- The type of an expression depends on the types of its sub-expressions

  \[ x + x \text{ has type } \text{int} \quad \text{if } x \text{ has type } \text{int} \]
  \[ x + x \text{ has type } \text{real} \quad \text{if } x \text{ has type } \text{real} \]
type analysis

can be done *statically*, at *parse time*

- There are *syntax-directed* rules for figuring out when $e$ has type $t$

  e is well-typed, with type $t$, if and only if *provable* from these rules

We say “$e$ has type $t$” or write “$e : t$”

... possibly with assumptions like “$x$:*int and $y$:*int”
There are static (syntax-directed) rules for

\[ e \text{ has type } t \]

\[ d \text{ declares } x_1 : t_1 \ldots x_k : t_k \]

\[ p \text{ matches type } t \text{ and binds } x_1 : t_1 \ldots x_k : t_k \]

under appropriate assumptions about the free variables of \( e \) and \( d \)
arithmetic

- n has type int
- \( e_1 + e_2 \) has type int if \( e_1 \) and \( e_2 \) have type int
- Similarly for \( e_1 \times e_2 \) and \( e_1 - e_2 \)

<table>
<thead>
<tr>
<th>static property</th>
<th>runtime behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 21 + 21 ) has type int</td>
<td>( 21 + 21 =&gt; 42 ) : int</td>
</tr>
</tbody>
</table>
booleans

- **true** and **false** have type **bool**
- \( e_1 \text{ andalso } e_2 \) has type **bool** if \( e_1 \) and \( e_2 \) have type **bool**
- \( e_1 < e_2 \) has type **bool** if \( e_1 \) and \( e_2 \) have type **int**

<table>
<thead>
<tr>
<th>Static property</th>
<th>Runtime behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (3+4 &lt; 1+7) : \text{bool} )</td>
<td>( (3+4 &lt; 1+7) =&gt;^* \text{true} : \text{bool} )</td>
</tr>
</tbody>
</table>
conditional
(for each type $t$)

- $\text{if } e \text{ then } e_1 \text{ else } e_2$ has type $t$
  if $e$ has type $\text{bool}$ and $e_1, e_2$ have type $t$

  test must be a boolean,
  both branches must have the same type

<table>
<thead>
<tr>
<th>static</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{if } x &lt; y \text{ then } x \text{ else } y$ has type $\text{int}$</td>
<td></td>
</tr>
<tr>
<td>if $x: \text{int}$ and $y: \text{int}$</td>
<td>$\text{if } x &lt; y \text{ then } x \text{ else } y = \ast \ 4 : \text{int}$</td>
</tr>
<tr>
<td></td>
<td>if $x:4$ and $y:5$</td>
</tr>
</tbody>
</table>
tuples
(for all types $t_1$ and $t_2$)

- $(e_1, e_2)$ has type $t_1 \times t_2$
  if $e_1$ has type $t_1$ and $e_2$ has type $t_2$

<table>
<thead>
<tr>
<th>static</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(x+2, y)$ has type int * bool when $x$:int and $y$:bool</td>
<td>$(x+2, y) =&gt;* (4, true) : int * bool when $x$:2 and $y$:true</td>
</tr>
</tbody>
</table>

Similarly for $(e_1, ..., e_k)$ when $k>0$
Also $(\ )$ has type **unit**
lists

(for each type t)

- \([e_1, ..., e_n]\) has type \(t\) list if for each \(i\), \(e_i\) has type \(t\)

- \(e_1::e_2\) has type \(t\) list if \(e_1\) has type \(t\) and \(e_2\) has type \(t\) list

- \(e_1@e_2\) has type \(t\) list if \(e_1\) and \(e_2\) have type \(t\) list

\([1+2, 3+4]\) has type \(\text{int list}\)

\([1+2, 3+4]\) \(\Rightarrow^*\) \([3, 7]\) : \(\text{int list}\)
functions

- \texttt{fn x => e} has type \( t_1 \rightarrow t_2 \) if \( e \) has type \( t_2 \) when \( x : t_1 \)

\textbf{the type of a function ensures type-safe application}

\begin{align*}
\text{fn x => x+x} & \quad \text{has type } \text{int} \rightarrow \text{int} \\
\text{fn x => x+x} & \quad \text{has type } \text{real} \rightarrow \text{real} \\
\text{fn y => x+y} & \quad \text{has type } \text{int} \rightarrow \text{int} \quad \text{when } x:\text{int}
\end{align*}
application

- \( e_1 \ e_2 \) has type \( t_2 \)
  if \( e_1 \) has type \( t_1 \rightarrow t_2 \) and \( e_2 \) has type \( t_1 \)

argument \( e_2 \) must have correct type for function \( e_1 \)

\[
(fn \ x \Rightarrow x+x) \ (10+11) \quad \text{has type } \text{int}
\]

\[
(fn \ x \Rightarrow x+x) \ (1.0+1.1) \quad \text{has type } \text{real}
\]
The example function:

\[
\text{fn } x \Rightarrow \text{ if } x=0 \text{ then } 1 \text{ else } f(x-1)
\]

has type **int** $\rightarrow$ **int** if $f : \text{int} \rightarrow \text{int}$

by rules for

\[
\text{fn } x \Rightarrow e
\]

if-then-else

application

\[
\ldots
\]
declarations

- **val** \(x = e\) declares \(x : t\) if \(e\) has type \(t\)

  - **val** \(x = 42\) declares \(x : \text{int}\)
  - **val** \(x = y + y\) declares \(x : \text{int}\) if \(y : \text{int}\)
  - **val** \(f = \text{fn}\ x \Rightarrow x + 1\) declares \(f : \text{int} \rightarrow \text{int}\)
declarations

If

\[ d_1 \text{ declares } x_1:t_1 \]

and (with this type for \( x_1 \))

\[ d_2 \text{ declares } x_2:t_2 \]

then

\[ d_1;d_2 \text{ declares } x_1:t_1, x_2:t_2 \]

\begin{verbatim}
val y = 21;
val x = y+y
\end{verbatim}
declares \( y: \text{int}, x: \text{int} \)
declarations

• **fun** \( f \ x = e \) declares \( f : t_1 \rightarrow t_2 \)
  
  if, assuming \( x : t_1 \) and \( f : t_1 \rightarrow t_2 \), \( e \) has type \( t_2 \)

  assuming that \( f \) is applied to an argument of type \( t_1 \)

  and recursive calls to \( f \) in \( e \) have type \( t_1 \rightarrow t_2 \)

  the result of application will have type \( t_2 \)

**fun** \( f \ x = \text{if } x = 0 \text{ then } 1 \text{ else } f(x - 1) \)

declares \( f : \text{int} \rightarrow \text{int} \)

… binds \( f \) to a function value of type \( \text{int} \rightarrow \text{int} \)
let expressions

- **let d in e end** has type \( t \)
  if \( d \) declares \( x_1 : t_1, \ldots, x_k : t_k \)
  and, in the scope of these bindings
  \( e \) has type \( t \)

**let val x = 21 in x + x end** has type \( \text{int} \)
and evaluates to \( 42 : \text{int} \)

**let fun f x = if x=0 then 1 else f(x-1) in f 42 end** has type \( \text{int} \)
and evaluates to \( 1 : \text{int} \)
patterns

when \( p \) matches type \( t \)

- \( _ \) matches \( t \) always
- \( 42 \) matches \( t \) iff \( t \) is \( \text{int} \)
- \( x \) matches \( t \) always

- \( (p_1, p_2) \) matches \( t \) iff \( t \) is \( t_1 \times t_2 \), \( p_1 \) matches \( t_1 \), \( p_2 \) matches \( t_2 \)

- \( p_1::p_2 \) matches \( t \) iff \( t \) is \( t_1 \text{ list} \), \( p_1 \) matches \( t_1 \), \( p_2 \) matches \( t_1 \text{ list} \)
examples

• Pattern \texttt{x::R} matches type \texttt{int list} and binds \texttt{x:int, R:int list}

• Pattern \texttt{x::R} matches type \texttt{bool list} and binds \texttt{x:bool, R:bool list}

• Pattern \texttt{42::R} matches type \texttt{int list} and binds \texttt{R:int list}
clausal functions

- \textbf{fn} \: p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k \text{ has type } t_1 \rightarrow t_2 \\
  \text{if for each } i, \: p_i \text{ matches } t_1 \\
  \text{and its bindings give } e_i \text{ the type } t_2

Each clause \( p_i \Rightarrow e_i \) must have \textit{same} type \( t_1 \rightarrow t_2 \)

- Each \( p_i \) must \textit{match} type \( t_1 \)
- Each \( e_i \) must have type \( t_2 \)

\textbf{fn} \: 0 \Rightarrow 0 \mid n \Rightarrow f(n - 1) \text{ has type } \text{int} \rightarrow \text{int} \\
  \text{if } f \text{ has type } \text{int} \rightarrow \text{int}
clausal declarations

- **fun** $f \ p_1 = e_1 \mid ... \mid f \ p_k = e_k$ declares $f : t_1 \to t_2$
  iff for $i = 1$ to $k$, $p_i$ matches $t_1$, giving type bindings for which, assuming $f : t_1 \to t_2$, $e_i$ has type $t_2$

  each clause $p_i => e_i$ must have *same* type $t_1 \to t_2$
  assuming recursive calls to $f$ in $e_i$ have this type

**fun** $f \ 0 = 0 \mid f \ n = f \ (n - 1)$ declares $f : \text{int} \to \text{int}$

  ... and binds $f$ to a *value* of type $\text{int} \to \text{int}$
fun f n = if n=0 then 1 else n + f (n - 1)

declares f : int -> int

because, assuming n : int and f : int -> int,

if n=0 then 1 else n + f (n - 1)

has type int
Polymorphic types

- ML has type variables
  • ′a, ′b, ′c
- A type with type variables is polymorphic
  • ′a list -> ′a list
- A polymorphic type has instances
  • int list -> int list
  • real list -> real list
  • (int * real) list -> (int * real) list
  • ... instances of ′a list -> ′a list

substitute a type for each type variable
fun split [ ] = ([ ], [ ])
  | split [x] = ([x], [ ])
  | split (x::y::L) =
    let val (A,B) = split L in (x::A, y::B) end

also (more generally!) declares
split : int list -> int list * int list

also (more generally!) declares
split : 'a list -> 'a list * 'a list

(the most general type is polymorphic)
sorting

Assuming

\[\text{split} : \text{'}a\ \text{list} \to \text{'}a\ \text{list} \times \text{'}a\ \text{list}\]
\[\text{merge} : \text{int list} \times \text{int list} \to \text{int list}\]

\[
\text{fun msort} \ [\ ] = [\ ]
\mid \ msort \ [x] = [x]
\mid \ msort \ L = \textbf{let}
\quad \textbf{val} \ (A,B) = \text{split} \ L
\quad \textbf{in}
\quad \text{merge} \ (\text{msort} \ A, \text{msort} \ B)
\quad \textbf{end}
\]

declares \text{msort} : \text{int list} \to \text{int list}
**sorting**

Assuming

- `split : 'a list -> 'a list * 'a list`
- `merge : int list * int list -> int list`

```ml
fun msort [] = []

| msort L = let

  val (A,B) = split L

  in

  merge(msort A, msort B)

  end

end
```

declares `msort : 'a list -> int list`

... there's a bug in the code!
polymorphic values?

• Every type has a set of syntactic values

• What are the values of type ‘a -> ’a?
  (all are equivalent to) fn x => x or fn x => loop()

• What are the values of type ’a?
  There are none!

Reasons:
  the type guarantee
typability

- \( t \) is a type for \( e \)
  iff \( (e \text{ has type } t) \) is \textit{provable}

- In the scope of \( d \), \( x \) has type \( t \)
  iff \( (d \text{ declares } x:t) \) is \textit{provable}

\[
\text{int list -> int list} \quad \text{is a type for} \quad \text{rev}
\]
\[
\text{real list -> real list} \quad \text{is a type for} \quad \text{rev}
\]
\[
\text{'a list -> 'a list} \quad \text{is a type for} \quad \text{rev}
\]
Instantiation

- If $e$ has type $t$, and $t'$ is an instance of $t$, then $e$ also has type $t'$

An expression can be used at any instance of its type
Most general types

Every well-typed expression has a *most general* type

$t$ is a *most general type for* $e$

iff $t$ is a type for $e$

& every type for $e$ is an instance of $t$

$rev$ has most general type `a list -> 'a list`
type inference

• ML computes **most general types**
  • statically, using syntax as guide

Standard ML of New Jersey v110.75
- fun rev [] = [] | rev (x::L) = (rev L) @ [x];
val rev = fn : 'a list -> 'a list
datatypes

- ML allows *parameterized* datatypes

```ml
datatype 'a tree = Empty
  | Node of 'a tree * 'a * 'a tree
```

a type constructor `tree`

and *polymorphic* value constructors

- `Empty : 'a tree`
- `Node : 'a tree * 'a * 'a tree -> 'a tree`
fun trav Empty = []
| trav (Node(t1, x, t2)) = (trav t1) @ x :: (trav t2)

declares trav : 'a tree -> 'a list
options

datatype 'a option = NONE | SOME of 'a

fun try (f, [ ]) = NONE
  | try (f, x::L) = case (f x) of
    NONE => try (f, L)
    | y => y

try : ('a -> 'b) -> ('a option -> 'b option)
equality

• ML allows use of $=\,$ only on certain types

• These are called equality types
  
  • int
  
  • tuples and lists built from equality types
  
  • not real and not function types

• ML uses type variables "$a", "b", "c" to stand for equality types

• must be instantiated with an equality type
example

fun mem (x, [ ]) = false
| mem (x, y::L) = (x=y) orelse mem (x, L)

declares mem : "a * "a list -> bool

OK instances include

  int * int list -> bool

but not real * real list -> bool