Exercise Session Informatik III

2. Lambda Calculus and
SML Introduction

The Remains of Last Week

- Well-Formed vs. Well-Typed
  - $(9 = 3) + 27$
  - $(9 - 3) + 27$
- Unicity of Typing
  - Consider the statement $loc_1 := @loc_2$ in L_c with booleans.
  - Is this statement in violation with the property of Unicity of Typing?

Lambda Calculus

$$
\begin{aligned}
& ((\lambda z.(za))(\lambda x.x)) && [(\lambda x.x)/z] \\
\equiv\ & (\lambda x.x)a && [a/x] \\
\equiv\ & a
\end{aligned}
$$

Lambda Calculus

$$
\begin{aligned}
& (\lambda y.(\lambda x.yx)y)(\lambda z.x) && \\
\equiv\ & (\lambda y.(\lambda u.[u/x]yx)(\lambda z.x) && \\
\equiv\ & (\lambda y.(\lambda u.yu)y)(\lambda z.x) && [(\lambda z.x)/y] \\
\equiv\ & (\lambda u.(\lambda z.x)u)(\lambda z.x) && [(\lambda z.x)/u] \\
\equiv\ & (\lambda z.x)(\lambda z.x) && [(\lambda z.x)/z] \\
\equiv\ & (\lambda z.x) && \\
\equiv\ & x &&
\end{aligned}
$$

Lambda Calculus

$$
\begin{aligned}
& (\lambda y.yy)((\lambda x.x)(\lambda z.z)) && [((\lambda x.x)(\lambda z.z))/y] \\
\equiv\ & ((\lambda x.x)(\lambda z.z))((\lambda x.x)(\lambda z.z)) && [(\lambda z.z)/x] \\
\equiv\ & (\lambda z.z)((\lambda x.x)(\lambda z.z)) && [((\lambda x.x)(\lambda z.z))/z] \\
\equiv\ & ((\lambda x.x)(\lambda z.z)) && [(\lambda z.z)/x] \\
\equiv\ & (\lambda z.z) &&
\end{aligned}
$$

SML Tutorial

- **`andalso`** and **`orelse`**
  - SML uses short-circuit evaluation
  - E_1 **`andalso`** E_2
  - E_1 **`orelse`** E_2
- Operator Precedence

  ~

  mod, div

  *, /

  +, –

SML Tutorial

The difference of trunc and floor is in the handling of negative numbers, i.e. **`floor(~1.5)`** returns **`~2`** whereas **`trunc(~1.5)`** yields **`~1`**.

```
fun trunc x = if x < 0.0 then ~1 * floor(~x)
                         else floor(x);
val trunc = fn : real -> int

fun sign 0 = "0"
  | sign x = if x < 0 then "-"
                      else "+";
val sign = fn : int -> string

fun power2 0 = false
  | power2 1 = true
  | power2 n = (n mod 2 = 0) andalso power2(n div 2);
val power2 = fn : int -> bool
```

SML Tutorial

```
fun third nil = ~1
  | third (st::nil) = ~1
  | third (st::nd::nil) = ~1
  | third (st::nd::rd::tl) = rd;
val third = fn : int list -> int

fun nth _ nil = ~1
  | nth n (hd::tl) = if n = 1 then hd
                              else nth (n - 1) tl;
val nth = fn : int -> int list -> int

fun third list = nth 3 list;
val third = fn : int list -> int
```

SML Tutorial

```
fun append left nil = left
  | append nil right = right
  | append (hd::tl) right = hd::append tl right;
val append = fn : 'a list -> 'a list -> 'a list
```

Note: There is an in-built operator @ that concatenates two lists. Instead of **`append(left, right)`** it is thus possible to use **`left @ right`**

```
fun reverse nil = nil
  | reverse (hd::tl) = (reverse tl) @ [hd];
val reverse = fn : 'a list -> 'a list
```

SML Tutorial

```
type date = {day:int, month:int, year:int};
type date = {day:int, month:int, year:int}

fun age(birth:date, today:date) =
        if #year birth > #year today
        then 0
        else ((#day today +
                 #month today * 30 +
                 #year today * 365)
              -(#day birth +
                 #month birth * 30 +
                 #year birth * 365)
              ) div 365;
val age = fn : date * date -> int
```

SML Tutorial

```
val today = {day=8, month=11, year=2001}
val today = {day=8, month=11, year=2001} :
                          {day:int, month:int, year:int}

age({day=22, month=6, year=1976}, today);
val it = 25 : int
```

That's all folks!

Vorbereitung für nächste Woche:
- Kapitel 2 des SML Tutorial
- Type Inferencing