Functional Design and Programming

Lecture 1

version 1.0
The course

♦ Teachers:
  • Lars Birkedal (birkedal@itu.dk)
  • Troels C. Damgaard (tcd@itu.dk)
  • Søren Debois (debois@itu.dk)

♦ Course homepage: http://www.it-c.dk/courses/FDP/E2004/

♦ Course plan:
  http://www.it-c.dk/courses/FDP/E2004/plan.html

♦ The lecture slides will be available from the course plan (on the web)

♦ Book: Hansen and Rischel’s *Introduction to Programming Using SML*
Lars Birkedal

♦ Cand. Scient. from University of Copenhagen, 1994
  • Advisors: Neil Jones and Mads Tofte
  • Partial Evaluation of SML
  • The ML Kit with Regions

♦ Ph.D. from Carnegie Mellon University, 1999
  • Advisor: Dana Scott
  • Realizability models of type theories and logics

♦ Started at ITU March 2000 as Assistant Professor

♦ Now Associate Professor and Head of Theory Department, one of four research departments at ITU

♦ Head of LaCoMoCo research theme http://lacomoco.itu.dk
♦ B.Sc. from RUC


♦ Main interests: High-level programming languages, and calculi and models for mobility and concurrency. (Machine learning and adaptivity in general.)

♦ Works on Bigraphical Programming Languages (BPL) research project.

♦ Implemented SML modules for BPL project. See BPL homepage: http://www.itu.dk/research/theory/bpl/seminar/

♦ Homepage: www.itu.dk/people/tcd
Søren Debois

♦ B.Sc. from DIKU


♦ Works on Bigraphical Programming Languages research project.

♦ Implemented a compiler for a hardware-specification language in SML. This is still in use at DIKU: http://www.diku.dk/undervisning/2004e/dat-ark/Kreds/kreds.html

♦ Implemented program transformation tools in SML for recent research in compiler optimization techniques. (See homepage.)

♦ Homepage: http://www.itu.dk/people/debois
Exercise classes: Wednesday 13:00–16:00 in room 4A 54.

Weekly exercises are given, some of which can be handed in; see separate exercise sheet.

Solve most exercises at home, and discuss them Wednesday afternoon.

Some exercises are singled out. In the lectures we assume that you have solved these exercises.
15 Hours

♦ 3 hours lectures
♦ 3 hours exercises with supervision
♦ 3 hours reading in the book
♦ 6 hours exercise solving on your own
Why this course?

♦ Learn a new programming paradigm
♦ Learn to write beautiful and correct code
♦ Learn a new programming language
♦ Learn a *modern* programming language
♦ Learn to attack problems in a structured manner
Listen to the wise

Simplicity and elegance are unpopular because they require hard work and discipline to achieve and education to be appreciated.

Dijkstra
Simplicity and elegance are unpopular because they require hard work and discipline to achieve and education to be appreciated.

*Dijkstra*

Coding is like Trivial Pursuit: if you don’t know something, it’s not because you’re stupid.

*Rusty Russell, in his 2003 OLS Keynote*
Programming paradigms

♦ Imperative programming
  • Procedural
  • Object-oriented
♦ Functional programming
♦ (Logic programming)
♦ Machine has a *state* (memory, state of devices)

♦ State is acted upon by commands

♦ Programs describe sequences of state changes by simple commands and constructs for composing those
Procedural programming

♦ Central concept: procedure. Purpose: Code factoring, reuse.

♦ A procedure is a named piece of code (sequence of commands), parameterised by \textit{formal arguments}.

♦ A procedure $P$ can be \textit{called (invoked)} by referring to its name and providing \textit{actual arguments (values)} that are assigned to the corresponding formal arguments.
Central concept: object. Purpose: Code factoring, abstraction (safety), reuse.

Encapsulation of data and the procedures (methods) that operate on those data.
Central concepts: value (including function), compositionality, and types.

Computation by calculation

What not how

Any expression can be replaced with its value (referential transparency).

Special case: A function always yields the same answer when given the same inputs.
Example: What not How

♦ Imperative:

```c
int gcd(int m, int n) {
    int prevm;
    while(m != 0) {
        prevm = m; m = n % m; n = prevm;
    }
    return n;
}
```

♦ Functional

```plaintext
fun gcd (0, n) = n
| gcd (m, n) = gcd(n mod m, m)
```
Imperative or Functional:

```c
int gcd(int m, int n) {
    if (m == 0) return n;
    else return gcd(n % m, m);
}
```

Functional programming is not (just) about the choice of programming language
Easy to define powerful data structures; for example, lists and trees.

First-class functions: functions are values (can be passed as arguments to and returned from other functions, can be stored in data structures).

Recursion: recursively defined functions and types.

Pattern matching: convenient and concise access to components of data structures.
Parametric polymorphism: convenient and concise definition of generic functions and data structures.

Safe composition: support for modular composition

High-level programming: many details about the machine has been abstracted away; example: automatic memory management.
Some Related Research at ITU

♦ The ML Definition (Tofte)
♦ The ML Kit with Regions (Tofte, Birkedal, Elsman, Hallenberg)
♦ SML Server (Elsman, Hallenberg)
♦ Moscow ML (Sestoft)
♦ Bigraphical Programming Languages (Birkedal, Hildebrandt, Elsman, Niss, Glenstrup, Christoffersen, Debois, Elsborg)
♦ Peer-to-peer layer for persistent tree-structured data (Niss)
♦ Parametric polymorphism (Birkedal, Møgelberg, Petersen)
Values

Values have no identity, no birth date, no expiration date, and they can’t change (ever heard of a 5 turning into a 6?); they are unchanging and immortal — *immutable*.

Examples of values: the string “Franz”, the number denoted by the Arabic numeral 5, the list of natural numbers from zero to five: [0,1,2,3,4,5].
Names refer to things (values, people, . . . )

Names can be *bound* to values in constructs called *declarations*.

Some values have predefined names (*constants*) that cannot be bound to other values; for example, 60 is the standard name (“numeral”) for a particular number.

Names are *not* values. Franz as a name is not the same as the string “Franz”, which is a value.
Identifiers

Those names that can be bound to values in declarations:

♦ Alphabetic names (e.g., size_of_tree, fact, x1)
♦ Symbolic names (e.g., @, :-|, --->)
♦ Long identifiers (e.g., Math.pi, String.sub)
SML distinguish between integers (type `int`) and real numbers (type real).

- Integers: 42, ~23, 0
- Reals: 0.0, ~42.23, 10E6
Computation by calculation

\[ 3 \times (9 + 5) \]

\[ \rightarrow \quad 3 \times (9 + 5) \]

\[ \rightarrow \quad 3 \times 14 \]

\[ \rightarrow \quad 3 \times 14 \]

\[ \rightarrow \quad 42 \]
fun fact 0 = 1
  | fact n = n * fact(n-1)
val fact = fn : int -> int
fun fact 0 = 1  (* 1 *)
| fact n = n * fact(n - 1)  (* 2 *)

fact 3

\[3 \times \text{fact}(3 - 1)\]

\[3 \times \text{fact}(2)\]

\[3 \times (2 \times \text{fact}(2 - 1))\]

\[3 \times (2 \times \text{fact}(1))\]

\[3 \times (2 \times (1 \times \text{fact}(1 - 1)))\]

\[3 \times (2 \times (1 \times \text{fact}(0)))\]

\[3 \times (2 \times (1 \times 1))\]

\[3 \times (2 \times 1)\]

\[3 \times 2\]

6
Bindings and environments

♦ Bind the identifier \texttt{five} to the integer value 5

\begin{verbatim}
val five = 5

val five = 5 : int
\end{verbatim}

♦ Bind the identifier \texttt{$$$}$

\begin{verbatim}
val $$$ = 1000000.0

val $$$ = 1000000.0 : real
\end{verbatim}

♦ The environment for these two bindings

\[
\begin{array}{c c}
\text{five} & \rightarrow 5 \\
\text{$$$}$ & \rightarrow 1000000.0 \\
\end{array}
\]
Strings and characters

♦ String constant: "Franz"

♦ Character constant: \"a\"

♦ Escape sequences: \"\", "I’m here.\nWhere?"

Some string functions:

♦ size s is the number of characters in s

  size "1234" → 4

♦ s1 ^ s2 forms a new string by placing s2 after s1

  "One," ^ " two" → "One, two"
Truth values

♦ The two values of type bool: true and false

♦ Conditional expression: if $e_1$ then $e_2$ else $e_3$

♦ Example:

\[
\text{fun cmp (m, n) = }
\]

\[
\begin{align*}
\text{if } m < n & \text{ then } \sim 1 \\
\text{else if } m = n & \text{ then } 0 \\
\text{else } (* m > n *) & \text{ 1}
\end{align*}
\]
Type summary

♦ Integers has type int
♦ Real numbers has type real
♦ Text strings has type string
♦ Characters has type char
♦ Functions has a type with an arrow ->
   For example: string -> int
   We say that -> is a type constructor