Slide 1

# Compilation of Functional Languages

Advanced Functional Programming (04s1)

Manuel M. T. Chakravarty
University of New South Wales

Slide 2

## Functional Languages & Compilation

Challenges for FL Compilers:

- ➜ Ample use of higher-order functions
- ➜ Ample use of polymorphism
- ➜ Currying
- ➜ Laziness
- ➜ Intermediate data structures (e.g., "point-free programs")
  ✧(IDS.hs)

Slide 3

FL Properties Facilitating Compilation:

- ➜ Simple core language:
  - Simplifies most compiler phases
  - Simplifies reasoning about the correctness of the compiler
- ➜ Clean functional semantics:
  - Facilitates many optimisations
  - Facilitates compilation-by-program-transformation
  - Simplifies reasoning about the correctness of the compiler

Slide 4

## Core Languages

What is a Core Language (or Kernel Language)?

- ➜ Smallest subset of the source language that still distinguishes all operationally distinct language features, such that
- ➜ all programs of the source language can be readily transformed into the core language

For example,

- ➜ the Haskell 98 language definition defines the Haskell Kernel as

  "...essentially a slightly sugared variant of the lambda calculus with a straightforward denotational semantics."

- ➜ It is used in the language definition to define the semantics of Haskell 98

Slide 5

## Enriched Lambda Calculus:

| | | | |
|---|---|---|---|
| $exp$ | $\rightarrow$ | $var$ | (variable) |
| | $\mid$ | $lit$ | (literal) |
| | $\mid$ | $exp_1\ exp_2$ | (application) |
| | $\mid$ | `\`$var$ `->` $exp$ | (abstraction) |
| | $\mid$ | `let {` $binds$ `} in` $exp$ | (recursive bindings) |
| | $\mid$ | `case` $exp$ `of {` $alts$ `}` | (case selection) |
| $binds$ | $\rightarrow$ | $var_1$ `=` $exp_1$`; ...;` $var_n$ `=` $exp_n$ | (bindings) |
| $alts$ | $\rightarrow$ | $pat_1$ `->` $exp_1$ `; ...;` $pat_n$ `->` $exp_n$ | (alternatives) |
| $pat$ | $\rightarrow$ | $lit$ | |
| | $\mid$ | $con\ var_1\ \ldots\ var_n$ | (flat pattern) |

Slide 6

## Translating the Enriched to the Plain Lambda Calculus:

- From a denotational point of view, `let` and `case` are not essential
- The untyped lambda calculus (aka $\lambda$-calculus) is Turing complete

### Non-recursive `let`:

`let` $v$ `=` $e_1$ `in` $e_2$

$\Rightarrow$ `(\`$v$ `->` $e_2$`)` $e_1$

### Recursive `let` (one binding):

`let` $v$ `=` $e_1$ `in` $e_2$

$\Rightarrow$ `let` $v$ `= fix (\`$v$ `->` $e_1$`) in` $e_2$ (now it's non-recursive)

Multiple, mutually recursive bindings? Think about it.



Slide 7

## `case` expressions:

- Let's start with something simple: pairs
- In plain Haskell, rewrite the following to not use explicit patterns:

```
sumTwo x = case x of
             (x1, x2) -> x1 + x2
```

➨

```
sumTwo x = fst x + snd x
```

- But what to do about pair construction? E.g., `sumTwo (2, 3)`
- Approach: We need a function `pair`, such that

  `fst (pair e1 e2)` $=$ `e1`

  `snd (pair e1 e2)` $=$ `e2`

  without using `(,)` or `case` in the definition of `pair`, `fst`, or `snd`.

  ✧`(Pair.hs)`
- How about types with alternatives? E.g., implement lists (`cons`, `nil`, `head`, `tail`, and `ifNull`) without `case` and constructors?

Slide 8

## So, why have `let` and `case` in the core language?

- On standard hardware, it is more efficient to implement them directly (rather than via the encoding into plain lambda calculus)
- Just consider pairs:
  - Returning a pair from a function, should ideally translate into using two registers to return the two components of the result
  - The higher-order functions `pair`, `fst`, and `snd` require closure creation and additional function calls (including indirect jumps)
- Similar situation with `let`s

## Compilation by Transformation

Typical structure of a compiler:

$$\text{Source} \rightarrow \text{IL}_1 \rightarrow \text{IL}_2 \rightarrow \cdots \rightarrow \text{IL}_n \rightarrow \text{Object Code}$$

(IL = Intermediate Language)

**Slide 9**

What is the disadvantage?

- ➜ Different intermediate languages need different representations
- ➜ Code is needed to transform one language into another
- ➜ Some compiler changes will require changes to many ILs
- ➜ Reasoning about compiler phases can be hard

Can't we unify some of the intermediate languages?

---

Compilation by core-to-core transformations:

- ➜ Idea: replace multiple ILs by one core language
- ➜ The core language may be adapted by varying it:
  - Varying sets of primitive operations
  - Syntactic restrictions

Varying sets of primitive operations:

**Slide 10**

- ➜ Breaking complex operations down into simple ones
- ➜ Example: unboxing

```
addInt :: Int -> Int -> Int
addInt (I# x) (I# y) = I# (x +# y)
```

Syntactic restrictions:

- ➜ Brings the core language closer to the machine
- ➜ Examples: Continuations passing style & administrative normal form



---

**Slide 11**

Advantages of core-to-core transformations over multiple ILs:

- ➜ Only one data structure in the compiler (consider support code)
- ➜ Most compilation phases change only some aspects of a program
- ➜ Some extensions affect less code
- ➜ Semantic clarity

---

## Typed Core Languages

Recently, typed core languages have become popular:

- ➜ Functional core language with explicit type information that is preserved by transformations
- ➜ Examples: TILT Compiler (TIL language) & GHC (Core language)

**Slide 12**

Why is this helful?

- ➜ Typechecking after each major compilation step finds bugs in the compiler quickly
- ➜ Type information can help to optimise the code
- ➜ A proof (formal or informal) asserting that a compiler phase preserves types says a lot about the correctness of that phase
- ➜ Typed assembly language & proof carrying code

Slide 13

## Compiled Graph Reduction

### Graph reduction:

- The conceptual basis for the implementation of lazy functional languages
- Expressions, such as $(\lambda x.x + 1)\ 2$ are represented as a (possibly cyclic) graph
- Program execution corresponds to the application of reduction rules to this graph
- E.g., function application involves copying the piece of the graph containing the lambda abstractions representing that function
- Disadvantages:
  - Not very efficient
  - Even simple operations require costly graph manipulations
- Still useful for interpreters, such as, Hugs

Slide 14

### What do we want instead?

- Use conventional hardware features:
  - Function bodies should be vanilla object code
  - Function applications should translate to normal function calls
  - Want to use registers for local values and function arguments where possible
  - Don't want to store simple values (like integers) always in the heap
- Problem: We seem to be able to compute functions on the fly

1. `map (+ 1) [1..10]`
2.

```
adder   :: Int -> (Int -> Int)
adder x = let
              doAdd y = x + y
            in
            doAdd
```

◇(Adder.hs)



Slide 15

### Towards Compiled Graph Reduction:

- Closures: Pair of a code pointer plus a set of valuations for the free variables in the code
- In the example,

```
adder   :: Int -> (Int -> Int)
adder x = let doAdd y = x + y
          in
          doAdd
```

the result of `adder 1` is represented by

$$(\texttt{doAdd}, \{x \mapsto 1\})$$

- The code portion of a closure can be regarded as a function getting an implicit argument: a pointer to the heap-allocated closure
- This is not unlike the implicit "self" pointer passed to methods in the implementation of OO languages

Slide 16

### Tagless Heap:

- Usually, objects in the heap have to be tagged to distinguish between closures, integers, floats, and objects representing constructors (such as list cells)
- Idea: Use an uniform closure representation for everything
  - Every heap object consist of a code pointer (first word) followed by a set of values for free variables in that code (all other words)
  - Integer objects are represented by

    ```
    mkInt i = let intCode () = i
              in
              intCode
    ```

    that is, `mkInt 42` gives

    $$(\texttt{intCode}, \{i \mapsto 42\})$$
- This idea is used in the Spineless Tagless G-machine on which GHC is based.

**Slide 17**

For more details, see

- *The Implementation of Functional Programming Languages*, Simon Peyton Jones, Prentice-Hall, 1987 and
- *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine*, Simon Peyton Jones, Journal of Functional Programming 2(2), 1992.