Summary: This lecture discusses graph reduction, which is the basis of the most common compilation technique for lazy functional languages.
Graph reduction

- Graph reduction is the basis of the main implementation technique used in the compilation of lazy functional languages

- A program is represented as a graph

- A graph is a tree where multiple pointers can reference the same node
  - multiple pointers to a node result in sharing, which avoids repeated evaluation
  - this is in contrast to the interpreter from earlier, which represented sharing via an environment

- A graph is cyclic if there is a path from any node back to itself, otherwise it is acyclic
  - cyclic graphs can occur in infinite data structures and with recursive functions

- Evaluation proceeds by replacing part of the graph that represents a redex (e.g. a function application) with the result of reducing the redex

- In the simple case, this is more like an interpreter than a compiler
  - but later we will see that it is possible to compile the functions in a program into machine code that directly transforms the graph
Graph representation of programs

- We will have four node-types to represent expressions

1. A **leaf-node** represents a constant or variable
   - constants will include the names of user-defined functions
   - basically named $\lambda$-abstractions
2. An **@-node** (an application node) represents a function application
3. A **$\lambda$-node** (an abstraction node) represents a $\lambda$-abstraction, i.e. a function from the users program
4. A **:-node** (a constructor node) represents constructed data
   - tuples, lists, user-defined data, etc.
   - the program will already have been type-checked, so we can use the same representation for different types

- Each node in the graph is represented in memory as a contiguous sequence of words known as a **cell**
  - a cell contains a **tag** and one or more **fields**
  - each field contains either an atomic value (a number) or a pointer (an address)
Fixed- vs. variable-sized cells

- An implementation may support fixed-sized or variable-sized cells
  - fixed-sized: all cells contain exactly two fields
    - corresponds to a binary graph
  - variable-sized: cells can have any number of fields
    - corresponds to an n-ary graph

- Variable-sized cells are clearly more flexible and give a more compact representation
  - e.g. for tuples, curried functions, arbitrary-precision numbers, user-defined data

- Variable-sized cells are more efficient but are more complicated to implement
  - in particular, storage management is more complicated
Boxed vs. unboxed representations

• An implementation may support unboxed values or only boxed values

• In a boxed implementation, every value has its own cell
  – including atomic values like numbers

• In an unboxed implementation, atomic values can occur directly in the fields of other cells

• Unboxed implementations are more flexible and clearly give a more compact representation

• Unboxed implementations are faster but incur some implementation complexity
  – storage management is more complicated
  – fields must contain tags (in addition to the tags on cells) to distinguish between numbers and pointers
Evaluation

• An expression is reduced to weak-head normal form by repeated reduction of the outermost redex
  – normal-order reduction

• At each stage, we have to identify the next top-level redex

• A redex is either
  – a built-in function applied to (at least) the right number of arguments
  – an abstraction applied to one or more arguments

• We have to identify the “leftmost-lowermost” node in the program graph to identify the redex-type

• We **unwind** the spine of the graph until we reach the leftmost node
  – the spine then holds the argument(s) of the redex

• If a function is strict in an argument, the argument must be evaluated before the function is applied
\[ \beta \text{-reduction} \]

- \( \beta \)-reduction is one of the main steps in the evaluation of a functional language program

\[(\lambda x. E) \, E' \leftrightarrow_{\beta} E[E'/x]\]

- \( \beta \)-reduction in the graph reduction model has three essential parts

1. Copy the body \( E \) of the abstraction
2. Substitute a pointer to the argument \( E' \) for every instance of the bound variable \( x \)
3. Overwrite the redex with the root node of the copy
   - note that these parts are not necessarily implemented separately

- Consider the evaluation of

\[(\lambda f. \lambda x. f(f \, x)) \, (\lambda y. + \, y \, 1) \, (+ \, 2 \, 3)\]
Copying and garbage

• Copying the body of an abstraction involves allocating space for each node in the copy
  – this preserves referential transparency
  – clearly it is important that storage allocation is fast

• Only the root node is overwritten
  – always with an expression which is equivalent to the original

• Sometimes copying a graph doesn’t change it
  – i.e. if the graph doesn’t contain any instances of the variable being substituted

• Avoiding unnecessary copying saves space and time
  – if two graphs are the same, only one copy is needed
  – we can avoid the copying time
  – this also increases sharing if the graph contains any redexes
    – an implementation that maximises sharing in this way is called fully lazy

• When a reduction is performed, some nodes may become “detached” from the graph
  – if these nodes are no longer accessible, the space they occupy can be collected and recycled
  – these nodes are known as garbage nodes
  – the process of recycling garbage nodes is known as garbage collection
The reduction algorithm

repeat
    unwind the spine of the graph to the first non-@ node
    \( args = \text{stack depth} \)
    case node-type of
        leaf-node: if it is a \( \lambda \)-name
            then get the definition
        else if it is a primitive and \( args \geq \text{arity} \)
            then reduce any strict arguments
                apply the appropriate rule
                overwrite the root of the redex
        \( \lambda \)-node: if \( args > 0 \)
            then copy the body
                substitute the argument
                overwrite the root of the redex
        :-node: (don't do anything)
    end
until expression in WHNF
Projector functions

- A **projector function** is a function whose body is a single variable
  - e.g. $id$, $const$, $head$, $tail$, $fst$, $snd$

- Projector functions can cause a loss of sharing
  - consider the application $head \ [f \ E]$
  - naive overwriting causes the application $f \ E$ to be duplicated

- The problem can be solved by introducing a new node-type, the **indirection node**
  - an indirection node is a pointer to another node
  - it represents the same value as the node to which it points

- However indirections are a potential source of inefficiency because they can occur anywhere
  - every time we perform any operation, we have to test for indirections
  - chains of indirections can form
Indirections continued

• These problems can be avoided with a simple observation
  – the result of a projector function will be the next outermost redex
  – the (relevant portion of the) argument can be evaluated before applying the projector function
  – this is an example of how we can vary the reduction order for efficiency reasons without compromising the semantics of the language

• This helps both schemes (with and without indirections)
  – it saves the loss of sharing through overwriting
  – the node that we copy won’t be a redex
  – it prevents chains of indirections from forming

• It is unclear whether copying or using indirections is better overall
  – indirections use less space (the space they use can be recovered quickly)
  – but we must still test for indirections at every operation