Functional Programming

Christopher D. Clack

FUNCTIONAL PROGRAMMING

Lecture 19

INTRODUCTION TO IMPLEMENTATION

FUNCTIONAL PROGRAMMING INTRODUCTION TO IMPLEMENTATION

This lecture will give an overview of implementation issues for functional programming, including for example issues that relate to the λ calculus, string tree and graph reduction, combinators, abstract machines and memory management

These issues will be revisited in future lectures

CONTENTS

FUNCTIONAL PROGRAMMING
INTRODUCTION TO IMPLEMENTATION

WHAT IS IMPLEMENTATION?

Issues:

- strict and lazy languages
- compiler technology
- type systems
- intermediate representations
- runtime systems
- reduction and rewriting systems
- compiled and interpreted execution
- abstract machines
- strict and lazy execution
- memory management
- garbage collection

When we talk about “implementation” of functional languages, we typically include a range of issues such as:

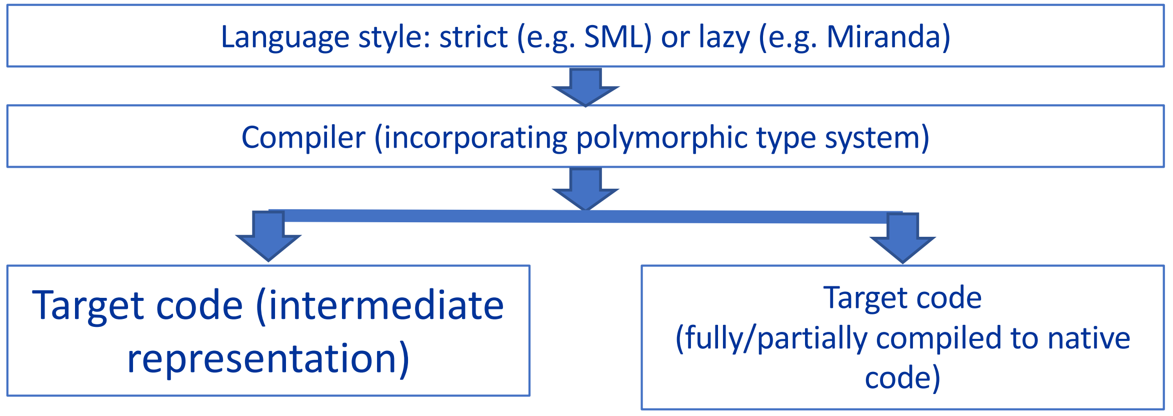
- strict and lazy languages
- compiler technology
- type systems
- intermediate representations
- runtime systems
- reduction and rewriting systems
- compiled and interpreted execution
- abstract machines
- strict and lazy execution
- memory management
- garbage collection

At first sight it can be difficult to link these together, so this lecture will attempt to do just that – to give a “big picture” of how these issues are connected to each other

Later lectures will cover some of these issues in more depth

FUNCTIONAL PROGRAMMING INTRODUCTION TO IMPLEMENTATION

WHAT IS IMPLEMENTATION?


Language style: strict (e.g. SML) or lazy (e.g. Miranda)
Compiler (incorporating polymorphic type system)
Target code (intermediate representation)
Target code (fully/partially compiled to native code)

We start with the fact that there are two different styles of functional language:

- Strict languages, such as SML, use Applicative Order reduction
- Lazy languages, such as Miranda and Haskell, use Normal Order reduction and sharing

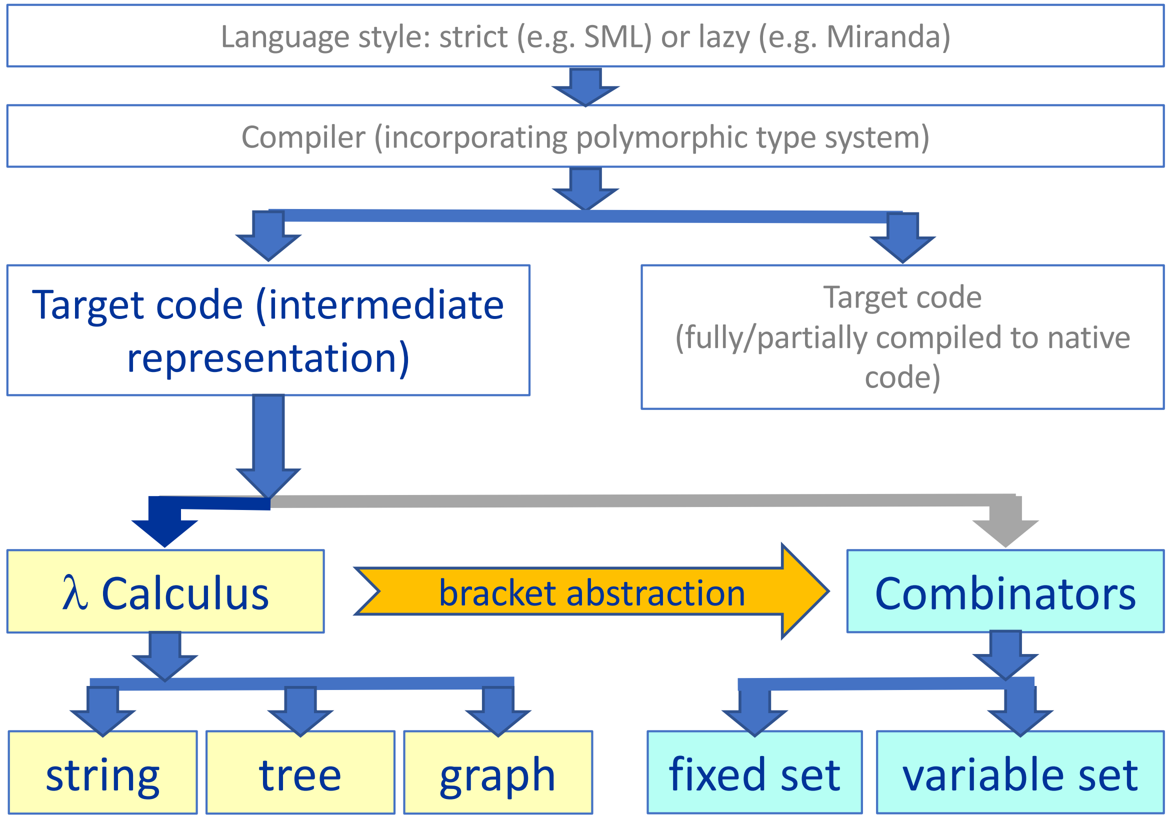
Both styles of language will be converted by a compiler into a target code. The compiler will typically provide a polymorphic type system (such as the Hindley-Milner type system, or an extension of that system)

The target code may be an intermediate representation requiring further interpretation by a runtime system, or may be completely or partially compiled into native code

Historically, implementation schemes for native code (we call these *compiled* implementations, whether wholly or partially in native code) were developed as optimisations of the schemes based on *interpretation* of intermediate representations. In this module we focus on the *interpretive* implementations

FUNCTIONAL PROGRAMMING INTRODUCTION TO IMPLEMENTATION

WHAT IS IMPLEMENTATION?



Language style: strict (e.g. SML) or lazy (e.g. Miranda)
Compiler (incorporating polymorphic type system)
Target code (intermediate representation)
Target code (fully/partially compiled to native code)
λ Calculus
bracket abstraction
Combinators
string
tree
graph
fixed set
variable set

The implementation route via an intermediate representation is interesting in that two possible representations are:

- the λ Calculus
- combinators

Both of these representations can be further sub-divided into alternatives:

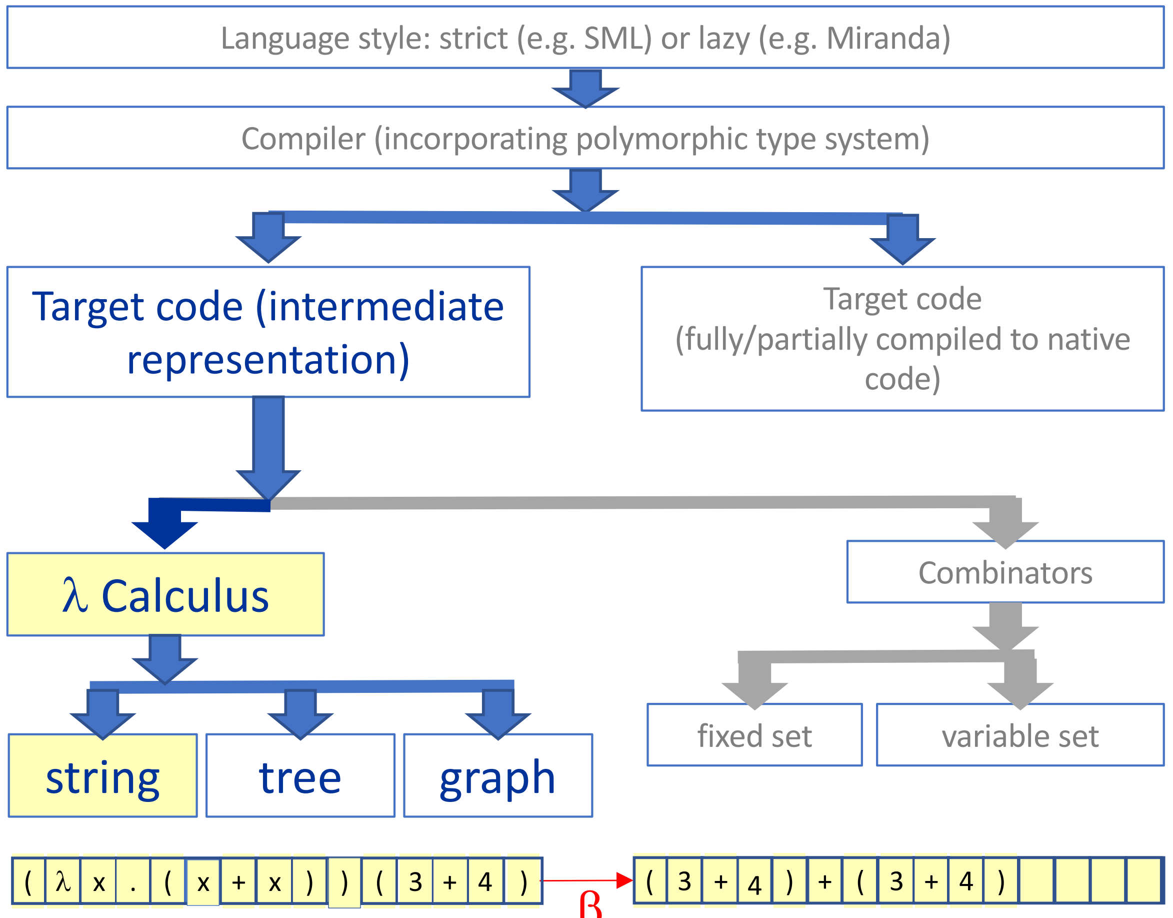
- The λ Calculus can be represented as a string, as a tree or as a graph
- Combinators may either be a fixed set of combinators or a variable set of combinators

Examples of these alternative representations will be given in the following slides

Another implementation route for generating combinators has been first to generate the λ Calculus and then to apply "abstraction elimination" using a "bracket abstraction" algorithm to generate combinators

FUNCTIONAL PROGRAMMING
INTRODUCTION TO IMPLEMENTATION

WHAT IS IMPLEMENTATION?



Language style: strict (e.g. SML) or lazy (e.g. Miranda)
Compiler (incorporating polymorphic type system)
Target code (intermediate representation)
Target code (fully/partially compiled to native code)
λ Calculus
Combinators
string
tree
graph
fixed set
variable set
(λ x . (x + x)) (3 + 4)
β
(3 + 4) + (3 + 4)

string reduction

λ Calculus string representation

Consider the following λ Calculus expression (with constants & operators):

(λx.(x + x)) (3+4)

A simple "string" representation uses a sequence of characters set out **contiguously** in memory:

(	λ	x	.	(	x	+	x	)	)	(	3	+	4	)

Performing a β reduction involves overwriting the string (that part representing the redex to be reduced) with the result of the reduction:

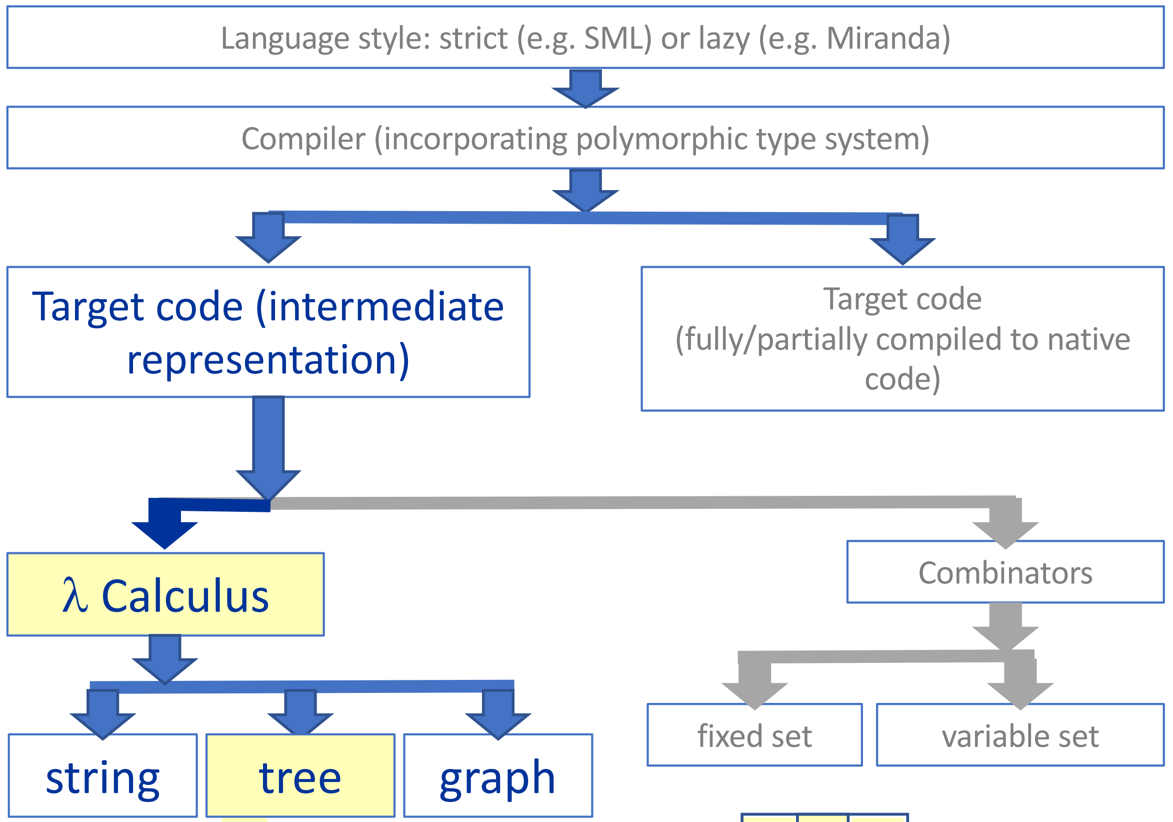
(	3	+	4	)	+	(	3	+	4	)				

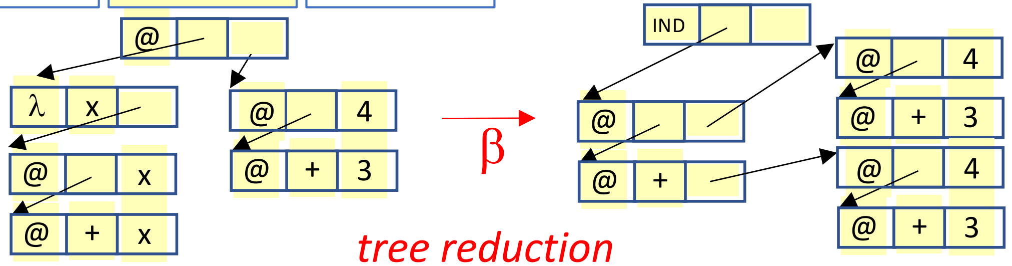
If the result of the reduction is smaller than the redex, this is straightforward. If however the result is larger than the redex (e.g. if the argument expression were large, and copied multiple times into the function body) then the string of characters must be lengthened and existing characters moved sideways to give room for the result to be stored in the correct place in the string – this is very slow!

Also notice that for string reduction Applicative Order Reduction is generally more efficient than Normal Order Reduction (which is one reason for the emphasis in early years on designing "strict" functional languages)

FUNCTIONAL PROGRAMMING
INTRODUCTION TO IMPLEMENTATION

WHAT IS IMPLEMENTATION?


Language style: strict (e.g. SML) or lazy (e.g. Miranda)
Compiler (incorporating polymorphic type system)
Target code (intermediate representation)
Target code (fully/partially compiled to native code)
λ Calculus
Combinators
string
tree
graph
fixed set
variable set



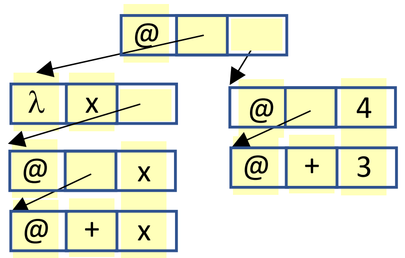
tree reduction

λ Calculus tree representation

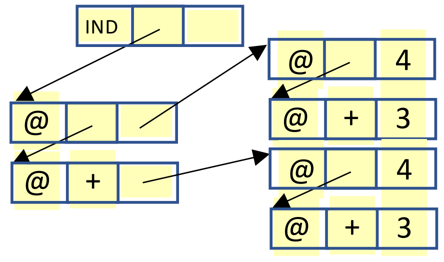
Consider the following λ Calculus expression (with constants & operators):

(λx.(x + x)) (3+4)

A simple "tree" representation uses a dynamic data structure with pointers (memory addresses) connecting data in **discontiguous** memory cells



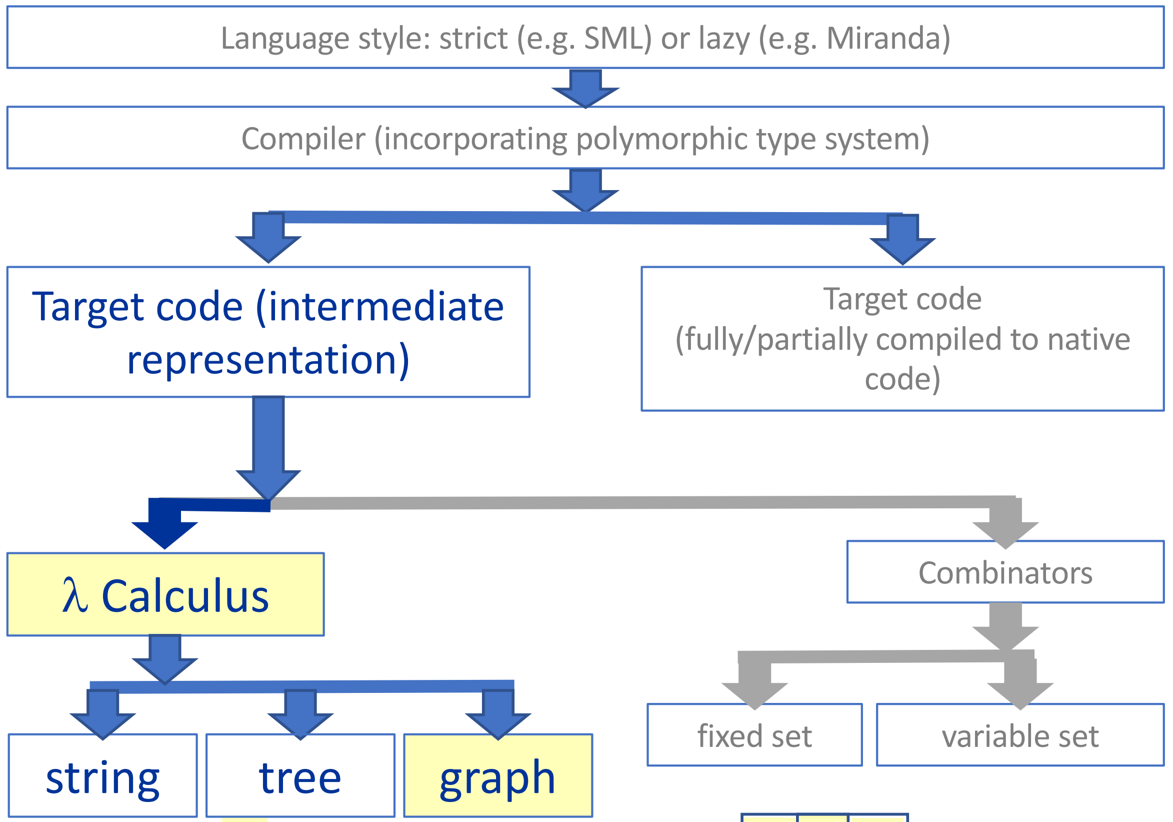
Performing a β reduction involves copying the function body, replacing free occurrences of "x" with pointers to copies of the argument, and setting the original top node of the redex to be an indirection to the result:



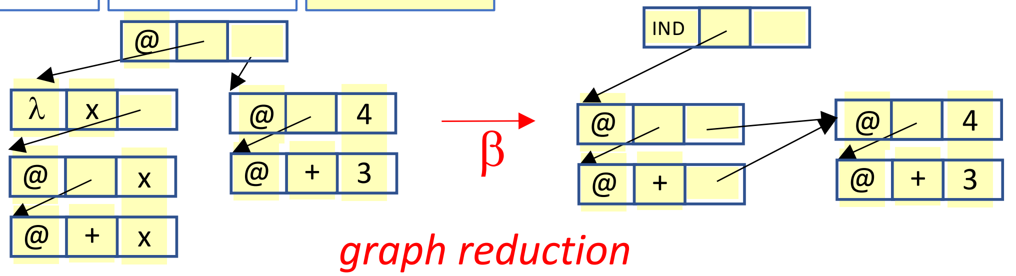
If the result of the reduction is larger than the redex, this is much faster to modify than string reduction. For Normal Order Reduction it still has the disadvantage of potentially copying an argument multiple times

FUNCTIONAL PROGRAMMING INTRODUCTION TO IMPLEMENTATION

WHAT IS IMPLEMENTATION?



Language style: strict (e.g. SML) or lazy (e.g. Miranda)
Compiler (incorporating polymorphic type system)
Target code (intermediate representation)
Target code (fully/partially compiled to native code)
λ Calculus
Combinators
string
tree
graph
fixed set
variable set



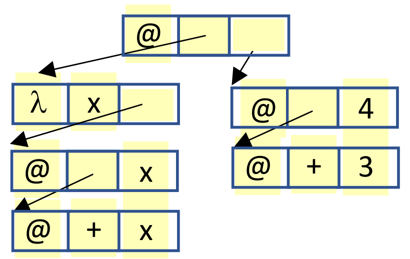
graph reduction

λ Calculus graph representation

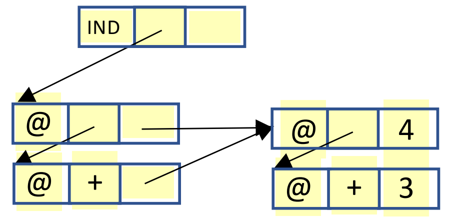
Consider the following λ Calculus expression (with constants & operators):

(λx.(x + x)) (3+4)

A simple "graph" representation uses a dynamic data structure with pointers (memory addresses) connecting data in memory cells, with sharing and cyclic pointers permitted:



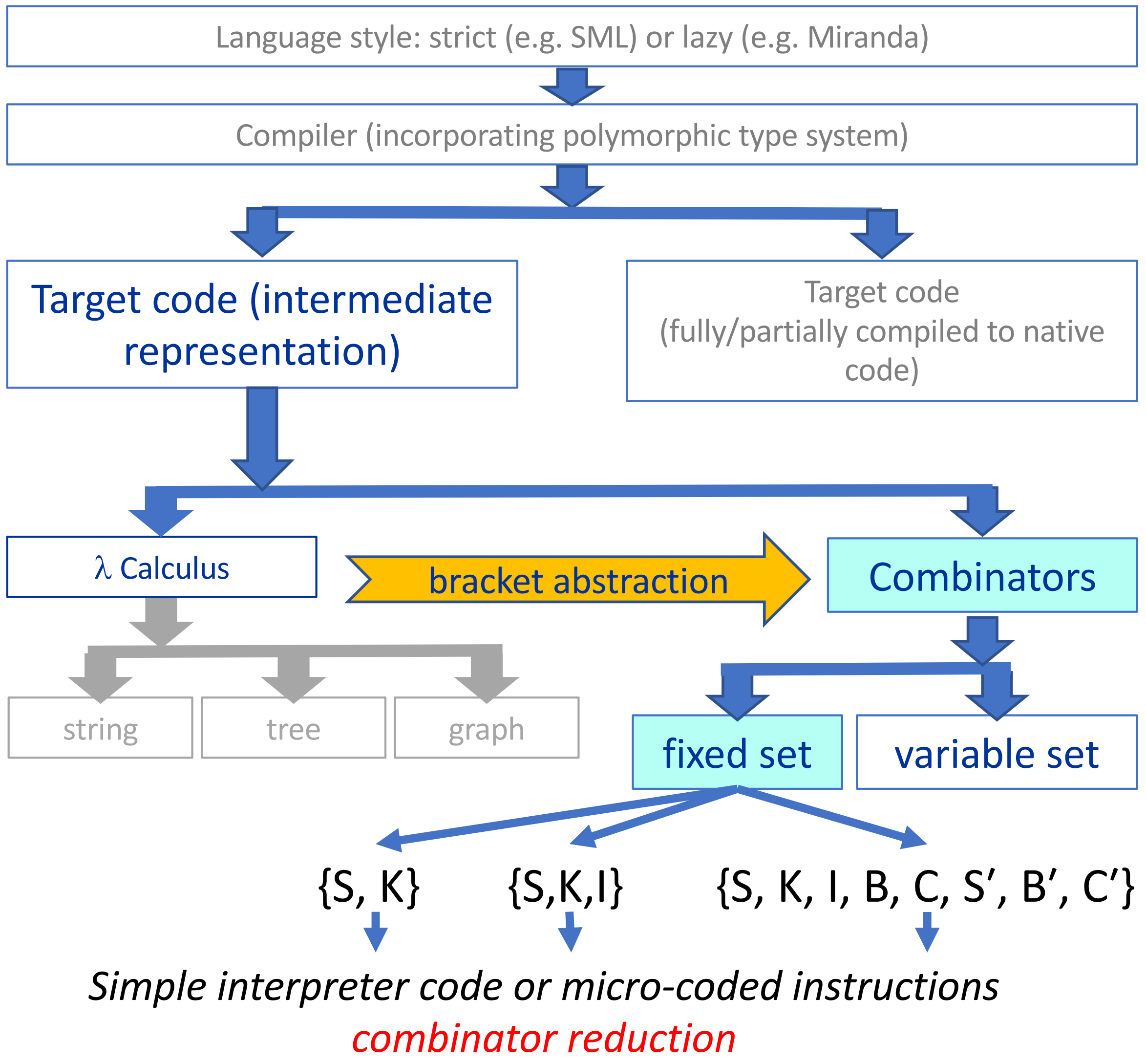
Performing a β reduction involves copying the function body, replacing free occurrences of "x" with pointers to the original argument **with sharing**, and setting the original top node of the redex to be an indirection to the result:



This has all the advantages of tree reduction, plus the advantage that Normal Order Reduction does not potentially create multiple copies of an argument. We'll see later how recursion is also more optimal with graph reduction

FUNCTIONAL PROGRAMMING
INTRODUCTION TO IMPLEMENTATION

WHAT IS IMPLEMENTATION?


Language style: strict (e.g. SML) or lazy (e.g. Miranda)
Compiler (incorporating polymorphic type system)
Target code (intermediate representation)
Target code (fully/partially compiled to native code)
λ Calculus
bracket abstraction
Combinators
string
tree
graph
fixed set
variable set
{S, K}
{S,K,I}
{S, K, I, B, C, S′, B′, C′}
Simple interpreter code or micro-coded instructions
combinator reduction

Combinator representation: fixed set

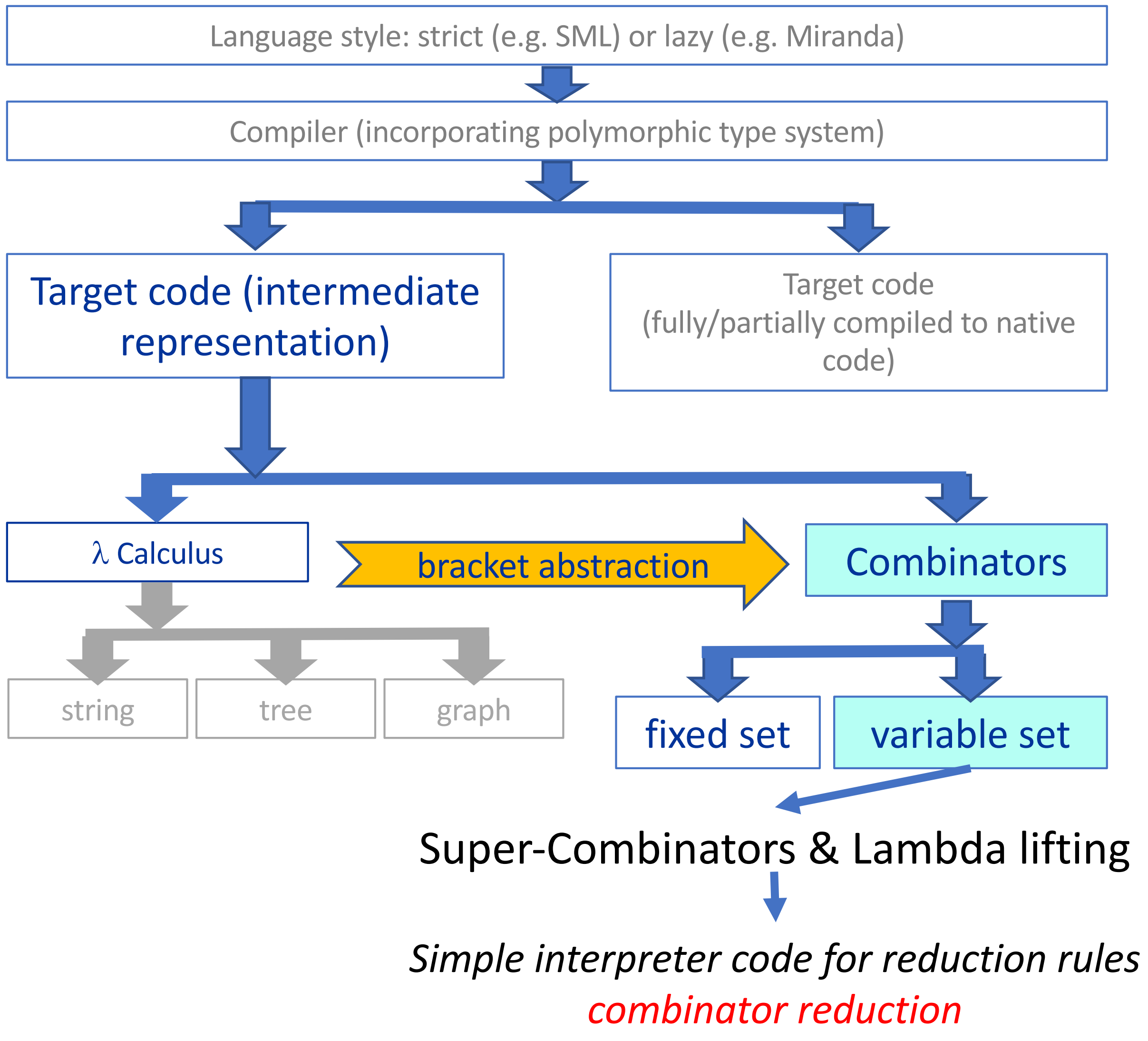
We have explained previously how the two combinators S and K are *computationally complete* (with just these two functions, we can compute anything that is computable). A compiler could therefore convert from a language such as Miranda into a target code that is a single expression using just the function names S and K. That is, it uses the fixed set of two combinators {S, K} as its basis

The set {S, K} is not the only set of combinators that could be used. It is theoretically possible (but impractical) to use just one combinator – more helpfully, we can add more combinators to reduce the overall size of expressions in the target code (much larger than λ expressions), and improve efficiency. Examples are I, B, C etc. David Turner's bracket abstraction algorithm uses a fixed-set basis of 8 combinators

Writing interpreter code to implement the reduction rules for a fixed set of combinators is straightforward (e.g. K a b → a), and for Turner's set it gives good performance (better than λ calculus). These rules are so simple that they can also be micro-coded as machine instructions for bespoke hardware – as was done for the set {S,K,I} for the SKIM architecture (*Clarke et al, 1980, Proc. ACM Lisp and Functional Programming Conference*)

FUNCTIONAL PROGRAMMING
INTRODUCTION TO IMPLEMENTATION

WHAT IS IMPLEMENTATION?



Language style: strict (e.g. SML) or lazy (e.g. Miranda)
Compiler (incorporating polymorphic type system)
Target code (intermediate representation)
Target code (fully/partially compiled to native code)
λ Calculus
bracket abstraction
Combinators
string
tree
graph
fixed set
variable set
Super-Combinators & Lambda lifting
Simple interpreter code for reduction rules
combinator reduction

Combinator representation: variable set

A fixed set of combinators has a number of drawbacks (*Hughes "Super-Combinators", Proc. ACM Lisp & Functional Programming Conference 1982*):

- target code and source code are very unalike, making debugging difficult
- compilation is slow due to the multiple optimisations and multiple passes over expressions nested within multiple λ expressions
- execution takes many tiny steps, with overhead linking each to the next

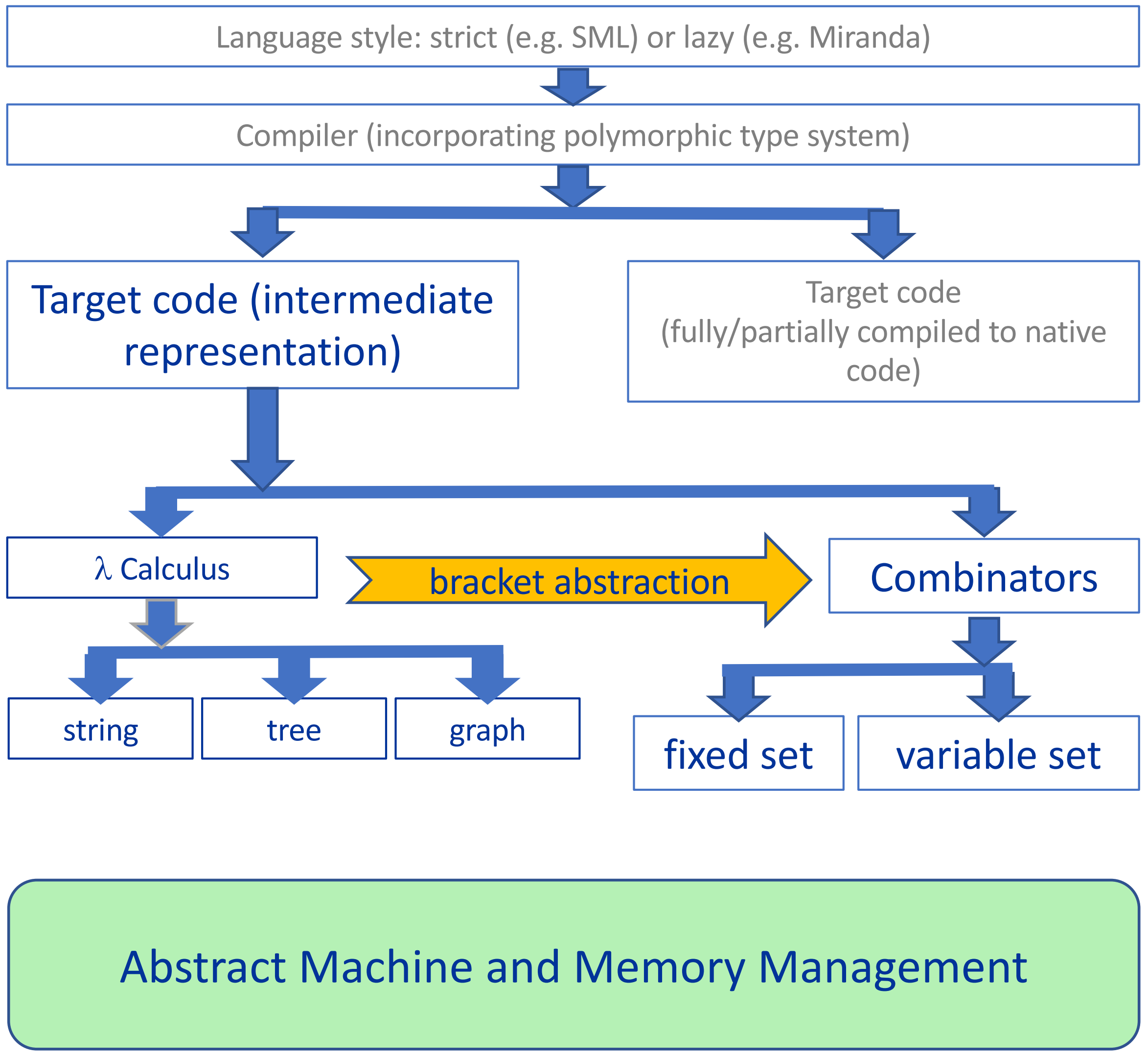
Hughes solved this problem by creating "Super-Combinators" directly from the source code – each has the properties of a combinator, each with its own reduction rule, but at larger granularity. This is done by identifying the free variables of a function body and creating new parameters for the function – one for each free variable. Now the function is a combinator, but we must also find all the applications of the function and add the extra parameters it needs

"Lambda lifting" is a transformation technique for generating supercombinators (*Lecture Notes in Computer Science 201:190-203 1985*)

With either approach, the result is that for each program there is a different set of combinators, with a different number of combinators, and each has its own reduction rule

FUNCTIONAL PROGRAMMING INTRODUCTION TO IMPLEMENTATION

WHAT IS IMPLEMENTATION?



Language style: strict (e.g. SML) or lazy (e.g. Miranda)
Compiler (incorporating polymorphic type system)
Target code (intermediate representation)
Target code (fully/partially compiled to native code)
λ Calculus
bracket abstraction
Combinators
string
tree
graph
fixed set
variable set
Abstract Machine and Memory Management

Abstract Machines and Memory Management

Underlying all of this is the Abstract Machine that is the runtime system for executing the various different systems:

- string reduction
- tree reduction
- graph reduction
- combinator reduction

of λ expressions

It is in the abstract machine that the low-level machinery exists to determine what reduction order is used and whether the system will provide strict or lazy evaluation

Even where the target code is fully compiled into native code (a route that we have not explored), a runtime system will be necessary

Furthermore, an important part of the runtime system is the memory management subsystem that provides dynamic memory allocation and garbage collection in order to make the most efficient use (and re-use) of memory – later we will explore memory management issues such as memory allocation, garbage identification, garbage collection and control of fragmentation

FUNCTIONAL PROGRAMMING INTRODUCTION TO IMPLEMENTATION

In summary, this lecture has given an overview of implementation issues for functional programming, including for example issues that relate to the λ calculus, string tree and graph reduction, combinators, abstract machines and memory management

These issues will be revisited in future lectures

SUMMARY

- What is “implementation" ?
    - λ calculus
    - String, Tree and Graph Reduction
    - Combinators
    - Abstract machines
    - Memory allocation
    - Garbage Collection