Functional Programming

Christopher D. Clack

FUNCTIONAL PROGRAMMING

Lecture 21

GRAPH REDUCTION continued

FUNCTIONAL PROGRAMMING GRAPH REDUCTION continued

CONTENTS

This lecture continues our exploration of the implementation technique of Graph Reduction – specifically, interpretive graph reduction

The lecture will explain two different approaches to traversing the spine of application cells, and discuss the issue of lazy versus strict evaluation

This will be followed by a discussion of combinator graph reduction and how that differs from graph reduction of λ expressions

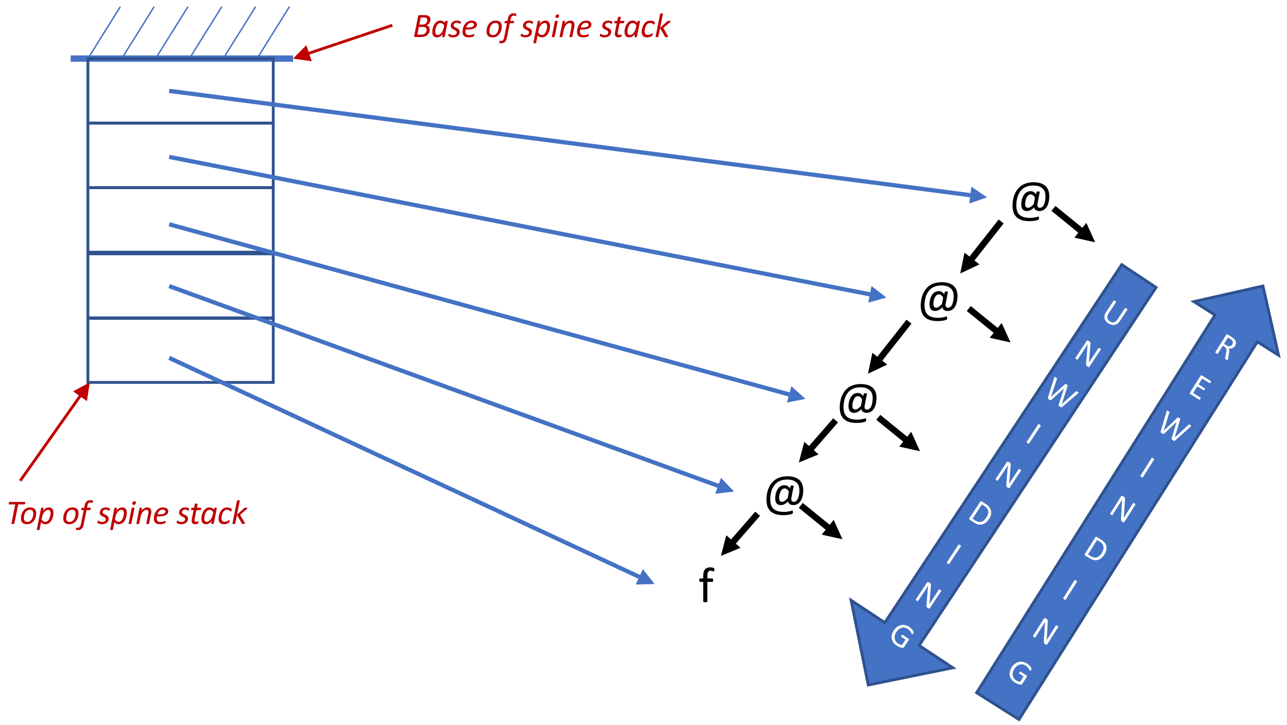
Supercombinator graph reduction is a further step in the development of Graph Reduction technology, and we explain how it provides benefits over previously covered techniques

As for the last lecture, students are expected to read Chapters 10, 11, 12 and 13 of "The Implementation of Functional Programming Languages", by Simon Peyton Jones, available for free download at this link: https://www.microsoft.com/en-us/research/uploads/prod/1987/01/slpj-book-1987.pdf

FUNCTIONAL PROGRAMMING GRAPH REDUCTION continued

SPINE TRAVERSAL – the Spine Stack

- "descending" ("unwinding") and "ascending" ("rewinding") the spine
- Using a stack of pointers to the vertebrae



In the previous lecture we mentioned that interpretive graph reduction would "descend the spine" . Notice that it will also be necessary to "ascend the spine" to find the correct vertebra node to overwrite with the result of a reduction

But how is "descending" and "ascending" (also known as "unwinding" and "rewinding") actually achieved? Here we give two mechanisms – first, using a Spine Stack, and then using a technique called Pointer Reversal

The Spine Stack

The code that performs interpretive graph reduction uses a stack data structure, where each item on the stack is the memory address of (a pointer to) a vertebra on the spine

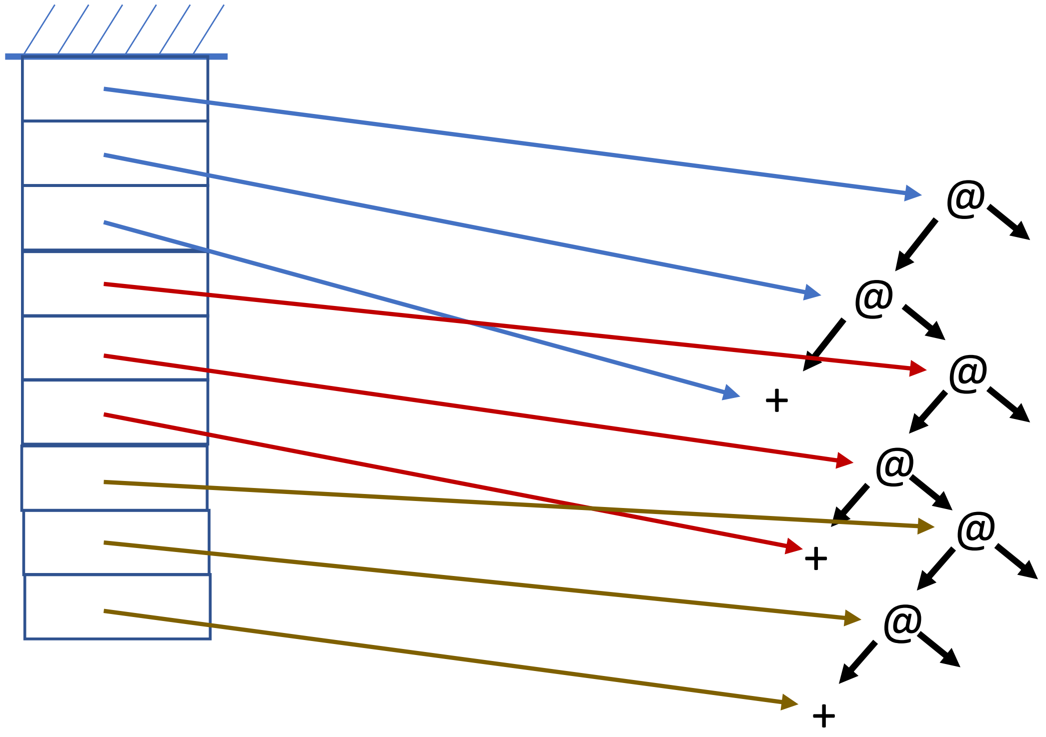
Unwinding: The first item at the base of the stack (by convention shown growing downwards in memory) points to the top vertebra – this is set to be the root of the program when graph reduction starts. By following the pointer, the address held in the left field of the vertebra node can be found, and a copy is placed next on the stack. By following the pointer just placed on the stack, the address in the left field of the next lower vertebra node can be found, and a copy placed on the stack. This continues until the tip is found

Rewinding is trivial now that the pointers to the vertebra are on the spine stack

FUNCTIONAL PROGRAMMING GRAPH REDUCTION continued

SPINE TRAVERSAL – the Spine Stack

- Recursively evaluate strict arguments to built-in functions
- Each recursively-searched spine can use the same Spine Stack



An advantage of the Spine Stack is that the graph reduction code now not only has access to every vertebra on the spine (so that it can ascend and descend the spine) but also, via the vertebrae, to the ribs and therefore the arguments (which is necessary for both β and δ reduction)

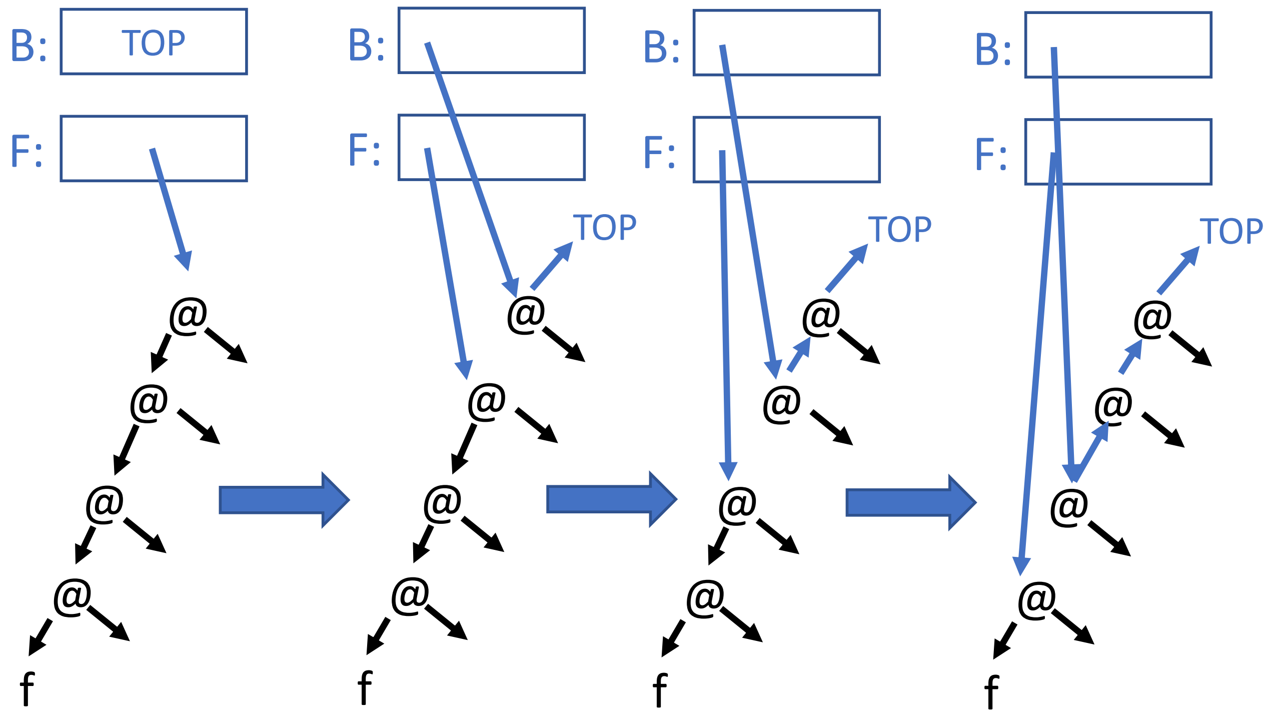
When recursively evaluating strict arguments to built-in functions such as "+", the pointers for traversing the argument spine can be built directly on the Spine Stack, as illustrated in this diagram. Notice that the existing Spine Stack entries (e.g. the blue and red pointers in this diagram) will not change while a more deeply-nested argument (e.g. the brown pointers) is being evaluated. Also notice that the Spine Stack for any spine can be discarded once the entire spine has been evaluated (and so the brown Spine Stack entries can be discarded – popped from the stack - after that subgraph is evaluated, leaving the red Spine Stack entries to be considered next, and so on)

When recursively evaluating an argument, the entirety of the existing Spine Stack must be retained so that it can be returned to when the argument has been evaluated. Essentially, this is the implementation technique of "stack frames" often used in imperative language runtime systems

FUNCTIONAL PROGRAMMING GRAPH REDUCTION continued

SPINE TRAVERSAL – Pointer Reversal

- Uses finite memory – just two locations B and F
- Using a stack of pointers to the vertebrae



A Spine Stack is simple to implement, but has the disadvantage that it requires an unknown amount of memory (we don't know in advance how deep the stack will become)

An alternative is to use Pointer Reversal (independently invented by Deutsch and by Schorr & Waite) This technique only requires two memory locations that we shall call B (for Back) and F (for Forward)

To start, B holds a unique pointer called TOP and F holds a pointer to the root vertebra. To unwind one step down the spine, two changes occur:

- swap the contents of B and the left field of the vertebra pointed to by F
- then swap the contents of B and F

A common technique for swapping the contents of two memory locations (without needing a third intermediate location) is to apply a bitwise XOR function three times

This process repeats for each unwind step, and the result is that the spine is split into two parts:

1. pointed to by B, wherein the pointers in the left fields of the vertebrae get reversed, so that they point UP the spine rather than down
2. pointed to by F, wherein the pointers in the left fields of the vertebrae point DOWN the spine, unchanged

Rewinding the spine is the reverse process, stopping when B=TOP

FUNCTIONAL PROGRAMMING GRAPH REDUCTION continued

SPINE TRAVERSAL – Pointer Reversal

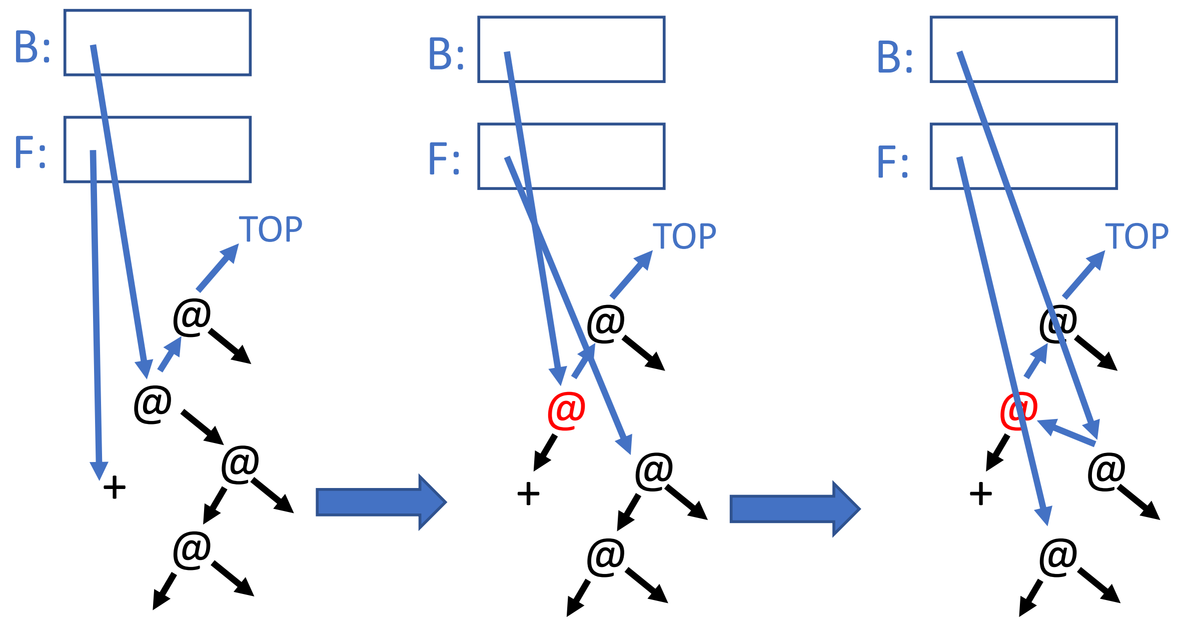
- Evaluating strict arguments for operators

When a built-in operator needs a strict argument to be evaluated in advance, the pointer-reversal technique requires the use of a new tag (a modified application node tag, indicated here by "@" in red)

Pointer reversing into an argument requires the following changes:

- swap the contents of F and the right field of the vertebra pointed to by B
- then change the tag of the node pointed to by B from @ to @

The reason for changing the tag is so that when rewinding out of an argument spine it is clear when B has reached the parent spine (since in this case we cannot rely on the special value TOP). At this point the next rewind step must reverse the changes stated above, so that the pointers are reset correctly and the tag is set back to normal



f

FUNCTIONAL PROGRAMMING GRAPH REDUCTION continued

SPINE TRAVERSAL – Comparison

- Using a Spine Stack is very fast, with far fewer accesses to heap memory locations, but uses an unbounded amount of memory
- Using Pointer Reversal is very frugal with memory, but is very slow (unless implemented in hardware)
- Using Pointer Reversal is not merely useful for traversing spines (and argument spines) but also stores information about how far evaluation has progressed – this can be useful when evaluating multiple strict arguments (with a Spine Stack after evaluating each argument the parent spine is viewed entirely afresh, with no such information)

Using a Spine Stack is very fast, with far fewer accesses to heap memory locations, but uses an unbounded amount of memory

Using Pointer Reversal is very frugal with memory, but is very slow (unless implemented in hardware)

Using Pointer Reversal is not merely useful for traversing spines (and argument spines) but also stores information about how far evaluation has progressed – this can be useful when evaluating multiple strict arguments (with a Spine Stack after evaluating each argument the parent spine is viewed entirely afresh, with no such information)

FUNCTIONAL PROGRAMMING GRAPH REDUCTION continued

LAZY AND STRICT EVALUATION

- Normal Order evaluation is the process already described
- Strict (Applicative Order) evaluation uses the same procedure except that all arguments to all functions and built-in operators are treated in the same way as strict arguments for the built-in operators (i.e. evaluate them first before performing either a β or δ reduction)
- Lazy Evaluation is a combination of Normal Order evaluation and the fact that each subgraph is evaluated at most once – which is what we achieve by (i) copying pointers to arguments during β reduction and (ii) overwriting the root node of each redex with an indirection to the result

Normal Order evaluation is the process already described

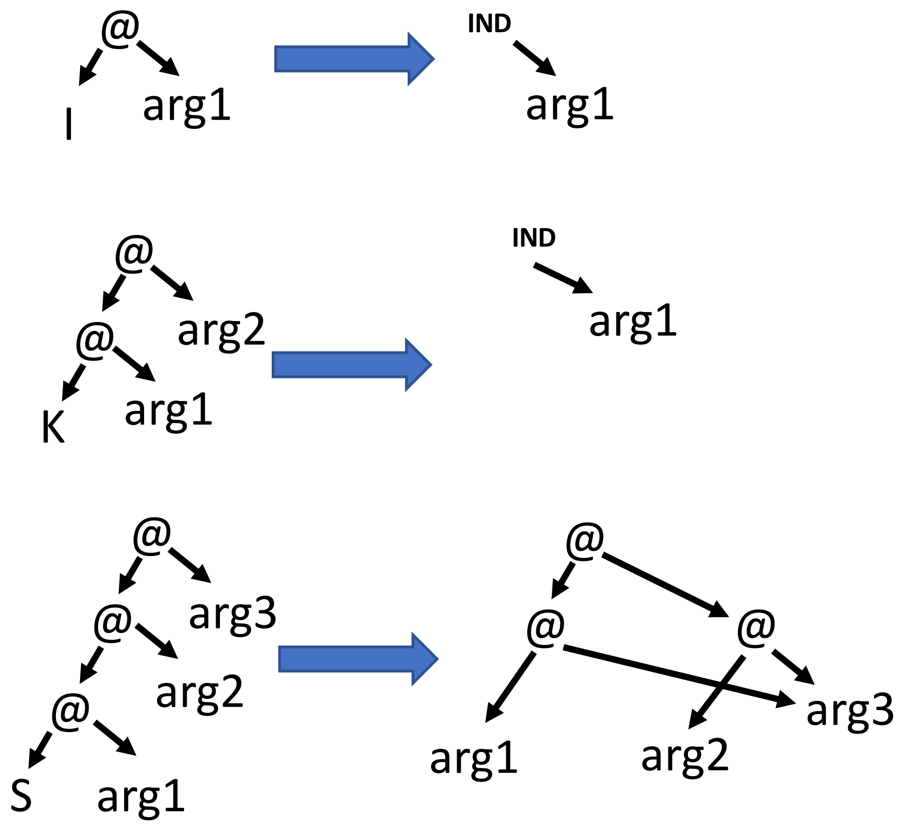
Strict (Applicative Order) evaluation uses the same procedure except that all arguments to all functions and built-in operators are treated in the same way as strict arguments for the built-in operators (i.e. evaluate them first before performing either a β or δ reduction)

Lazy Evaluation is a combination of Normal Order evaluation and the fact that each subgraph is evaluated at most once – which is what we achieve by (i) copying pointers to arguments during β reduction and (ii) overwriting the root node of each redex with an indirection to the result

FUNCTIONAL PROGRAMMING GRAPH REDUCTION continued

COMBINATOR GRAPH REDUCTION

- Fixed-set combinators – e.g. S, K and I
- No β reduction, only δ reduction
- δ rules:



The method used for combinator graph reduction depends on whether a fixed set or a variable set of combinators is used

Assuming a fixed set of combinators, e.g. S, K and I, these are viewed as built-in functions with rules for δ reduction. There are no λ abstractions and so no β reductions

The δ rules for S, K and I are easily expressed as graph manipulations and derive directly from the combinator definitions:

I x = x	*(replace root with IND to x)*
K x y = x	*(replace root with IND to x)*
S f g x = f x (g x)	*(see diagram)*

Notice that none of these combinators need to know the value of their arguments, so they are not strict in their arguments (just as the built-in operator "if" is not strict in its second and third arguments)

Because we are dealing with a fixed set of combinators, it is easy to have these δ rules hard-wired into the graph reduction implementation (the runtime system code). Thus, the symbols S, K and I (and perhaps others, if a larger set of combinators is used) each cause some (usually small amount of) native code to run

FUNCTIONAL PROGRAMMING GRAPH REDUCTION continued

SUPERCOMBINATOR GRAPH REDUCTION

- Fixed combinators are treated like operators (not evaluated until all arguments are present)
- This will also apply to Super-Combinators
- We know that Super-Combinators have no free variables, and if they are not evaluated until all arguments are present then there can be no need for an *environment* of bindings to be managed at runtime
- Thus, Super-Combinators lead to super-efficient implementation using compiled graph reduction

We start with an observation that the (fixed) combinator K is treated like a built-in function rather than a user-defined function, and in particular it is not evaluated until it has **both** of its arguments

- this has nothing to do with the strictness of those arguments – e.g. the built-in operator "if" is only strict in its first argument but is not δ-reduced until all three arguments are present
- nor does this prevent partial applications – it is only about the *timing* of when evaluation occurs at runtime

This approach will also apply to Super-Combinators – although their definitions can't be known in advance (as there will be a different set of Super-Combinators for each program), we can treat them like operators and only evaluate them when all arguments are available

Notice that Super-Combinators are (by definition) combinators and therefore have no free variables, and if they are never evaluated until all arguments are present (and their body is not a λ abstraction) this solves the problem seen in the previous lecture that required an *environment* of bindings to be managed at runtime

Thus, Super-Combinators lead to super-efficient implementation using compiled graph reduction

FUNCTIONAL PROGRAMMING GRAPH REDUCTION continued

SUPERCOMBINATOR GRAPH REDUCTION

DEFINITION

A supercombinator *$S* of arity *n* is a λ expression of the form $\lambda x_1.\lambda x_2 \ldots \lambda x_n.E$

where *E* is not a λ abstraction (this just ensures that all the 'leading lambdas' are accounted for by $x_1 \ldots x_n$) such that

(i) *$S* has no free variables,

(ii) any λ abstraction in *E* is a supercombinator,

(iii) $n = 0$; that is, there need be no lambdas at all.

A supercombinator redex consists of the application of a supercombinator to *n* arguments, where *n* is its arity

A supercombinator reduction replaces a supercombinator redex by an instance of the supercombinator body with the arguments substituted for free occurrences of the corresponding formal parameter

This slide gives the definition of Super-Combinators appearing on Page 223 (Section 13.2) of the required-reading text book "The Implementation of Functional Programming Languages", by Simon Peyton Jones:

A supercombinator *$S* of arity *n* is a λ expression of the form $\lambda x_1.\lambda x_2 \ldots \lambda x_n.E$

where *E* is not a λ abstraction (this just ensures that all the 'leading lambdas' are accounted for by $x_1 \ldots x_n$) such that

(i) *$S* has no free variables,

(ii) any λ abstraction in *E* is a supercombinator,

(iii) $n = 0$; that is, there need be no lambdas at all.

A supercombinator redex consists of the application of a supercombinator to *n* arguments, where *n* is its arity

A supercombinator reduction replaces a supercombinator redex by an instance of the supercombinator body with the arguments substituted for free occurrences of the corresponding formal parameter

FUNCTIONAL PROGRAMMING GRAPH REDUCTION continued

SUPERCOMBINATORS OF NON-ZERO ARITY

Supercombinators of non-zero arity (that is, having at least one λ at the front) have no free variables (clause (i) of the definition) and we can therefore compile a fixed code sequence for them

Furthermore, clause (ii) of the definition ensures that any lambda abstractions in the body have no free variables, and hence do not need to be copied when instantiating the supercombinator body

Supercombinators of non-zero arity (that is, having at least one λ at the front) have no free variables (clause (i) of the definition) and we can therefore compile a fixed code sequence for them

Furthermore, clause (ii) of the definition ensures that any lambda abstractions in the body have no free variables, and hence do not need to be copied when instantiating the supercombinator body

FUNCTIONAL PROGRAMMING GRAPH REDUCTION continued

SUPERCOMBINATORS OF ZERO ARITY

A supercombinator with arity zero (that is, having no λs at the front) is just a constant expression (remember that it has no free variables)

These supercombinators are often called "constant applicative forms" or CAFs. For example:

- the constant 3
- the constant expression 4 + 5
- the constant function (+3)

The last example shows that CAFs can still be functions. Since a CAF has no λs at the front, it is never instantiated. Hence, no code need be compiled for it, and a single instance of its graph can freely be shared

A supercombinator with arity zero (that is, having no λs at the front) is just a constant expression (remember that it has no free variables)

These supercombinators are often called "constant applicative forms" or CAFs. For example:

- the constant 3
- the constant expression 4 + 5
- the constant function (+3)

The last example shows that CAFs can still be functions. Since a CAF has no λs at the front, it is never instantiated. Hence, no code need be compiled for it, and a single instance of its graph can freely be shared

FUNCTIONAL PROGRAMMING GRAPH REDUCTION continued

SUPERCOMBINATORS CODE

What should supercombinator code do?

- Keep the body of the supercombinator as a tree, and use template-copying interpretation
- Let the "code" be a graph held in a contiguous block of store, instantiated with a (fast) block copy
- Compile the body to a linear sequence of *intermediate code* graph-manipulation instructions to direct the operation of a simple run-time interpreter to create an instance of the body when executed
- Compile the body to a linear sequence of native code

With either intermediate code or native code we can introduce optimisations such as using fast stack allocation rather than allocating graph nodes in a heap

There are many options for implementing supercombinators:

- We could keep the body of the supercombinator as a tree, and use template-copying interpretation
- Because supercombinators are constructed once and for all at compile-time, rather than being generated on the fly at run-time, the "code" could just be a graph held in a contiguous block of store, instantiated with a (fast) block copy (but this also misses opportunities for optimisation)
- Compile the body to a linear sequence of *intermediate code* graph-manipulation instructions to direct the operation of a simple run-time interpreter to create an instance of the body when executed
- Compile the body to a linear sequence of native code

With either intermediate code or native code we can introduce optimisations such as using fast stack allocation rather than allocating graph nodes in a heap

FUNCTIONAL PROGRAMMING
GRAPH REDUCTION continued

SUMMARY

- Spine traversal
- Lazy and strict evaluation
- Combinator Graph Reduction
- Supercombinator Graph Reduction

In summary, this lecture has continued our exploration of the implementation technique of interpretive Graph Reduction

Two different approaches were covered for traversing the spine of application cells, and the issue of lazy versus strict evaluation was discussed

This was followed by a discussion of combinator graph reduction and supercombinator graph reduction and how they differ from graph reduction of λ expressions

As for the last lecture, students are expected to read Chapters 10, 11, 12 and 13 of "The Implementation of Functional Programming Languages", by Simon Peyton Jones