Combinators and supercombinators in the implementation of functional programming languages.

Introduction

In this essay we will discuss historical background of combinatory and compilation using supercombinators and combinators S, K and I.

The historical background of combinatory logic

Combinatory logic was introduced by Moses Schönfinkel (1) and Haskell Curry (2) in the 1920s. They aimed to describe the most fundamental properties of function-abstraction, application and substitution in a very general setting. It based on combinators which were first introduced by Schönfinkel (1). His idea was to provide an analogous way to develop functions and to eliminate the use of bound variables. To do achieve this, he introduced five operators Z, T, I, C and S which became the modern basic combinators B, C, I, K and S respectively (3).

The Lambda-calculus was invented in about 1928 by Alonzo Church (4). It is on the concept of function, and the primitives included abstraction Lambda x[M] and application {F}(X), which we shall call here “Lambda x.M” and “(FX)”.

Compilation using supercombinators

This section is referenced from (6) .

The problem of compilation of lambda terms

Transform a lambda expression into a form in which the lambda abstractions are particularly easy to instantiate. These special lambda abstractions are called supercombinators and the transformation is called lambda-lifting.

Compilation is process of translating computer code written in one source programming language into a target programming language. This is normally executed by a compiler which is a computer program, that translate a source code from a high level programming language to a lower level language to create an executable program.

In compilation of functional programming, we associate with each lambda body a fixed sequence of instructions which will construct an instance of the lambda body. Subsequently, the operation of instantiating a lambda body would consist of following the sequence of instructions associated with the lambda body. As this instruction sequence can be developed beforehand by a compiler and consists of the knowledge about the shape of the body and where the formal parameter occurs, the compiled code is expected to run much faster than other means for example the Instantiate method. Furthermore, compilation opens up the opportunity for optimization and further increases efficiency. However, not all lambda abstractions are amenable to compilation.

For example a Lambda term that contains free variable in the body of abstrations, $\lambda x.\ (\lambda y.\ y\ x)$

Combinators and supercombinators in the implementation of functional programming languages.

This problem is the free variable x occurred in $\lambda y.\ y\ x$, the x then becomes bounded to $\lambda\ x$. Therefore, when we apply $\lambda\ x.\ (\lambda y.\ y\ x)$ to an argument, e.g. "*N*", we instantiate the body $\lambda y.\ y\ x$, and creating a completely new lambda abstraction $\lambda y.\ y\ Z$ instead of $\lambda y.\ y\ x$. This is not in line with the aim to compile a single fixed code sequence for each lambda abstraction. On the other hand, this problem would not appear when there is no free variable in the λy abstraction.

To solve the problem above, solutions had been proposed instead supercombinator graph reduction. The idea is to transform the program into an equivalent one in which all the lambda abstractions are amenable to compilation, by a transformation algorithm called lambda-lifting.

Supercombinators

In this section, we will talk about the definition of supercombinators.

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

where *E* is not a lambda abstraction such that
1. *$S* has no free variables,
2. any lambda abstraction in *E* is a supercombinator
3. *n* is greater or equal than 0.

A supercombinator redex consists of the application of a supercombinator to *n* argurments, where *n* is its arity. Meanwhile, 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 parameters.
$\lambda f.f\ (\lambda x.x)$ is an example of supercombinators as there is no free variable. In contrast $\lambda f.f\ (\lambda x.\ xf)$ is not a supercombinator as λx abstraction contains a free variable *f*.

Without free variable, supercombinator solves the problem mentioned in the previous section. Supercombinator is amenable to compilation.

Supercombinator and combinator

As supercombinator is based on combinator, we will also discuss the definition of combinator. Combinator is a lambda expression which contains no occurrences of a free variable (5). A combinator is a 'pure' function. The value of a combinator applied to some arguments depends only on the values of the arguments but not on any free variables.
Therefore supercombinator is a subset of combinator, while combinator is a subset of lambda expression.

Supercombinator-based compilation strategy and implementation

In order to compile programs containing lambda abstractions which are not supercombinators, we can transform the programs so that it contains only supercombinators.

Strategy is to trasnsform the lambda expression into
1. a set of supercombinator definitions – e.g. $XY x y = - y x
2. an expression to be evaluated – e.g. $XY 3 4

Combinators and supercombinators in the implementation of functional programming languages.

Note that supercombinator can only go through reduction when all artuments are present. This is defined as a series of the rewrite rules. For example in $XY = *λx . λy . – y x , ($XY 4 5)* can undertake reduction but *($XY 4)* cannot.

Therefore the implementation of the strategy will be:
1. a translation algorithm (called lambda-lifting) which transforms all the lambda abstractions in the program into supercombinators.
2. an execution of supercominator reduction (this will not be discussed, as is outside of the scope of this essay.

To understand lambda-lifting, consider this example

(λx .(λy . + y x) x) 4

λy abstraction has a free variable *x (*which is bound to *λx)*, so its not a supercombinator. With a transformation:
"make each free variable into an extra parameter, aka abstracting the free variable.

Then we would transform
from λy . y x
to (λx . λy . + y x) x

This process is essential beta-abstraction. Subsequently, this clarifies the distinction between the two xs in the original expression *(λx .(λy . + y x) x) 4 .* Now we have *(λx .(λx λy . + y x) x x) 4 . T*he original free variable x in the inner abstraction is now bound to inner *λx* instead of outer *λx*. Therefore, this is now a supercombinator and is ready to be executed.

The process of compilation and execution can be visualised as a box

```
_____________________________
| supercombinator definitions:
| $Y x y = + y x
|_____________________________
| Expression to be evaluated:
| (λx .$Y x x ) 4
|_____________________________
```

Where $Y represents the supercombinator

Then

```
_____________________________
| supercombinator definitions:
| $Y x y = + y x
| $x X = $y x x
|_____________________________
| Expression to be evaluated:
| $X 4
|_____________________________
```

Upon completion of compilation. The program can now be executed by performing supercombinator reduction.

$X 4 >> $Y 4 4 >> + 4 4 >> 8

Combinators and supercombinators in the implementation of functional programming languages.

Overall the algorithm is described as below (6):

UNTIL there are no more lambda abstractions:

1. Choose any lambda abstraction which has no inner lambda abstraction in its body
2. Take out all its free variable as extra parameters
3. Give an arbitrary name to the lambda abstraction (e.g. $X *4* in above example)
4. Replace the occurrence of the lambda abstraction by the name applied to the free variables
5. Compile the lambda abstraction and associate the name with the compiled code

END

With this approach, the reduction rules became easier and faster however the disadvantage is that the program increases in size during the transformation.

Combinators S, K, I

This section is referenced from (7).
Combinators S, K and I are examples of supercombinators. They are used in graph reduction technique. This method is advantageous because it enables a simple reduction machine that only supports built-in operators without the need of template-instantiation mechanism.
The strategy aims to transform the program into one containing only the built-in operators and constants, along with the combinators S, K and I. Each of them has transformation rules and the reduction rules.

The combinators S, K and I reduction rules are defined as (7).:

S f g x >> f x (g x), where f g x implies (((f)g)x)
K x y >> x
I x >> x
Where the lowercase letters represent lambda expressions. Where >> means run time reduction.

The combinators S, K and I transformation rules are defined as (7).:
λx.x -> I
λx.c -> K c, where c does not equals to x
λx.e_1. e_2 -> S (λx.e_1) (λx.e_2)
Where -> means compile time transformation.

Compilation of lambda expressions using combinators S, K and I

The S, K and I transformations rules all together form a complete compilation algorithm which is known as the SK compilation algorithm. This algorithm will transform any lambda expression into an expression involving only S, K, I and constants (including built-in functions).
The process of the SK compilation algorithm to compile an expression e is described by a pseudocode below (7).

WHILE *e* contains a lambda abstraction *DO*
1. Choose any innermost lambda abstraction of e
2. If its body is an application, apply the S-transformation rule.

3. Otherwise its lambda body must be a variable or constant, so apply the K or I transformation rules respectively as appropriate.

END of algorithm

As the innermost lambda abstractions is transformed first, it is made certain that the body of the chosen lambda abstraction contains no lambda. This prevented the free variable problem and alpha-conversion problems mentioned previously, during compilation and evaluation processes of the combinator expression. The evaluation process, which follows the completion of compilation, is not discussed here due to the scope of this essay.

Conclusion

We have discussed brief historical background of combinatory logic, supercombinators and definition and compilation process of combinators S, K and I.

Combinators and supercombinators in the implementation of functional programming languages.

References

1. Schönfinkel, M., 1924. "Über die Bausteine der mathematischen Logik" (PDF). Mathematische Annalen. 92 (3–4): 305–316. doi:10.1007/bf01448013. Translated by Stefan Bauer-Mengelberg as "On the building blocks of mathematical logic" in Jean van Heijenoort, 1967. A Source Book in Mathematical Logic, 1879–1931. Harvard Univ. Press: 355–66.
2. Curry, H. B. 1930. "Grundlagen der Kombinatorischen Logik" [Foundations of combinatorial logic]. American Journal of Mathematics (in German). The Johns Hopkins University Press. 52 (3): 509–536. doi:10.2307/2370619. JSTOR 2370619.
3. Curry, H. B. and Feys, R., 1958. Combinatory Logic, Volume I.North-Holland Co., Amsterdam. (3rd edn. 1974).
4. Church, A. 1932 A set of postulates for the foundation of logic. Annals of Mathematics, Series 2, 33:346–366.
5. Barendregt, H. P., 1984. The Lambda Calculus: its syntax and semantics, volume 103 of Studies in Logic. North Holland
6. Peyton-Jones, S., 1987. The Implementation of Functional Programming Languages. Prentice-Hall, Chapter 13
7. Peyton-Jones, S., 1987. The Implementation of Functional Programming Languages. Prentice-Hall, Chapter 16