Compiler Construction, a course summary

Frank Wijmans

April 12, 2012

Abstract

This is a summary of the course on compiler construction, as taught by Atze Dijkstra. This is written during the compiler construction course given in the begin of the year 2012. Containing all subjects as presented during the 'hoorcolleges' and maybe most significant, summarizing the capita selecta lectures. The summary is based on the presentations and notes, and the capita selecta is from notes made during lectures.

1 Introduction

Compilers change programs based on their language.

Compiler construction has come a long way, currently trying to map advanced high-level language concepts onto native machine languages.

2 TDiagrams

TDiagrams visualize interactions between programs, platforms, interpreters and compilers. Programs can be compiled to other programs, and interpreters can allow programs to run on different platforms.

3 Structure of a compiler

Compilers are often a pipeline of components (or phases). A distinction between the front end and back end can be made, respectively the analysis of the input, and the synthesis of the target program.

Sometimes a compiler has a middle end in which the program is optimized.

During the front end a symbol table is created to contain incrementally obtained information about the source.

By loosely coupling different phases it is possible to gain modularity in the compiler.

This component way of running a compiler can be formalised in the Feedback monad. Components can be run in the feedback monad. Component resembles (is) an arrow, having the same structure. With the functions:

1. pure
2. (>>>)
3. first

4. second

it is possible to chain components to create computations.

3.1 Trees

A symbol table, or representation of the source typically take the form of trees, thus a transformation from and to trees and flat data streams is needed.

Reading and showing data is not enough, we need an data structure that complements this transformation.

3.2 ATerms

An ATerm (for annotated term) is such an structure. With two functions, fromTree and toTree, we transform a `Tree a` to `a`. An ATerm can contain any data type.

4 Expressions

Next we introduce a expression language for simple arithmetic expressions. A small language will be enough to show the problems encountered in simple productions.

4.1 Syntax

Abstract syntax is the tree form, concrete syntax is the string representation. Four ways of defining a language syntax.

- **Inductive definition** Let N be natural numbers, and:

    1. if $n \in N$ then $n \in Tm$
    2. if $t_1 \in Tm$ and $t_2 \in Tm$, then t_1+ or $*t_2 \in Tm$

    while no other elements be in Tm.

- **Inference rules** Natural deduction style is having premises and conclusions. Rules allow to rewrite to more specific conclusions. Rules for numerals, addition and multiplication.

- **Generation procedure** Defining the following: $Tm_0 = 0$
    $Tm_{i+1}) = N \cup \{t_1 + t_2, t_1 * t_2 | t_1, t_2 \in Tm_i\}$
    and let $Tm = \cup_i Tm_i$

- **BNF**

    Nackus-Naur form we can define abstract syntax:

    1. $n \in Num = N$ numerals
    2. $t \in Tm$ terms

    With its syntax

$$t ::= n | t_1 + t_2 | t_1 * t_2$$

4.2 Semantics

With defining sematics we need to distinguish object language and metalanguage. With the metalanguage you can argue about the object language. Three ways to define a simple language:

- **Axiomatic** semantics is the way of defining using equalities between object and meta language.
- **Denotational** semantics provides a mapping between sets of one to sets of the other language.
- **Operational** semantics describes a machine that evaluates rules, simplifying terms to values.
    1. Small-step operational semantics (or structural semantics) are stepwise reductions. Congruence rules are added to guide reduction of a compound term.
    2. Big-step (or natural) semantics formalises the concept of evaluation directly. It is similar to denotational

5 Types

In the following chapter we introduce simple types in our expression language.

5.1 Syntax and semantics

In the language we now have numerals, terms and values. Furthermore we have numerals ($n \in \mathbb{N}$) and booleans as values. Given this language we can define plus, multiply, less than, equals, greater than and conditionals. *In the slides are all the small-step semantic rules.*

A type system will catch the cases where the semantic rules are not followed. Not all rules will evaluate to meaningful values, a type system can check this.

A type system guarantees absence of some runtime errors, it provides static semantics. Higher abstraction has been made clear. Overall it is a safe way for computers to say something about the validity of a program.

A normal form is a irreducible term, and these terms can get stuck. Stuck terms are terms which are not values in the language.

Checking on stuck terms will solve some run-time errors. A type system checks a program to tell a programmer about stuck terms.

Well-typed programs share the following properties:

- Progress
- Preservation

These two properties together establish type safety.

An algorithm that assigns types needs to be sound and complete. Such that it assigns correctly, and all terms are assigned the right type.

The slides gives explanation about the actual implementation.

6 Attribute grammars

6.1 Catamorphisms

The example used is a the problem of calculating a maximum segment sum. Given some sequence of changes, which segment would give the highest value. This problem asks for a minimum and maximum of the sequence. These are computations that can be done sequentially while running through the data tree only once.

A catamorphism is a function that consumes objects X of some algebraic data-type τ and produces objects of some ρ by

- destructing X according to the structure of τ
- calling itself recursively on any components of X that are themselves also of type τ
- combining the recursively obtained results, and such, creating the result type ρ

In Haskell we would use a data type with a function over it, creating such catamorphism means writing boilerplate code.

Creating the algebra for the catamorphism follows the constructors, a definition for the cons, and for nil. If you have a catamorphism over lists, use the data of the cons, recur over the list part of the cons, while doing nothing on the nil. This is the `signature` of the algebra. Combine the data and the recursive parts to obtain the result. The result type is called the `carrier` When having other data-types you will have to look at different kinds of data and recursive steps.

Algebras can be used for almost anything from length, size, sum or product. It is familiar to a fold, but different that it can do multiple things for the same tree more easily.

6.2 Syntax-directed computation

The UU Attribute Grammar system facilitates a way of defining catamorphisms in a different way.

Attribute grammars allow you to write catamorphisms indirectly and using the preprocessor, it will generate the code for these data-types and functions. The three steps of defining an ag:

1. Define a grammar
2. Declare attributes
3. Define attribute equations

There are two different attributes, namely **synthesised** which travel upward (to the root) and **inherited** attributes that travel from parent to child (to the leafs).

6.2.1 Defining a grammar

A grammar consists of a set of Haskell-like data-type definitions where each data constructor has a label. These get stripped from the processed Haskell source-code, but are of use inside the equations.

6.2.2 Declaring attributes

A attribute is declared as an attribute on a data-type definition. Declare which labelled part is synthesised or inherited, with `syn` or `ihn` respectively.

6.2.3 Define attribute equations

Adding the semantic equations is done by the keyword `sem`. Where you use the labels and keywords like prev and lhs to define what to do at every node of the walk over the data.

The copy rule can be used when we need to produce a synthesised attribute, to produce code to copy the value of the local attribute to the synthesised attribute.

Sometimes it is needed to generate wrappers to combine inherited and synthesised values to the result object (ρ).

6.3 Case Study: AG for typechecking

The initial grammar is straight forward, as can be read in the second slide of the seventh presentation. Source positions are added to help with the error messages. The attributes in the grammar are as follows:

- Terms, `Tm` and `Tm_` get a type `Ty` and type error `[TyErr]`. Both are synthesised.

Type errors apply the `use` keyword, which takes precedence over the copy rule where you don't want to copy.

Please see the code examples in the slide show mentioned above.

7 Abstraction

General purpose programming languages need abstraction if they aspire to be useful. Next is the syntax, semantics and implementation of some abstraction mechanisms.

7.1 Local definitions

Variables are added to the syntax to allow the programmer to use those. A countable infinite set of variables are assumed.

Local definitions using variables can be: `let` $x = t_1$ `in` t_2 `ni`. The = in that sequence is a variable binder. The concepts of free variables, alpha-equivalence, shadowing of variables and beta-substitution are of effect when adding variables to a language. When doing beta-substitution, it is often defined as explicit substitution. Where every substitution is not defined directly, but it builds up an environment, a map-like data structure with substitution values.

An environments domain and codomain, are the keys, and substitutions. When rewriting, you lookup bindings from variable to value. When doing natural semantics, the environment is the premiss on which a evaluation of a variable is done. For the type system the approach is similar, define type environments. Based on a `TyEnv` it is possible to map `Var` to a `Ty`.

7.2 Mutuable state

When dealing with a mutuable variable environment (unlike Haskell), where variables can change of value, you'd need to allow rewriting. Then, a term to rewrite variables is added. Also, sequencing has to be made clear, which is often a ;. To do this, there are two approaches:

1. Each assignment can evaluate to a fixed trivial value, or

2. an assignment evaluates to the value of the assigned value.

The side effects of assignments do not only need to be passed down, towards the leafs, but it has to be chained, it is passed up as well as down. In an attribute grammar this means the attribute is inherited and synthesised. This means that the environment, and type environment are chained. To fix problems for typing rules, we need to allow scoping in environments. *The example in the uses an if statement with different assignments in the else and then clauses.*

It is possible to create local variables for every scoped variable, so that every scope gets its own environment bindings, though this is often not applied in programming languages.

8 Type systems

Lambda calculus is a idealised and minimalistic functional programming language.

8.1 Untyped lambda-calculus

It contains alpha-equivalence and beta-substitution. Untyped lambda contains only variablesand terms, and terms can be variables, lambda and application `let .. in ..` is a derived construct, and is written like $(\lambda x.t_2)t_1$.

8.2 Simply typed lambda-calculus

But in simply typed l-c you have naturals and bool. Next to variables and terms, it has types. An environment maps variables to types, using the type rules for lambda and application.

8.3 System F

System F adds polymorphism to the typed lambda calculus.

Adding type variables tyvar to the var, ty and tm.

When functions can take polymorphic functions as arguments, then you enter higher-rank polymorphism. A function that takes a normal polymorphic function as argument is rank 2, because it takes a normal, rank 1 function.

8.4 Hindley-Milner typing

HM is a compromise between System F and the desire to leave out type annotations.

Two restrictions apply:

1. All types are at most rank 1.
2. Functions can only have a polymorphic type if they are directly bound in a local definition.

These restrictions allow for all principal types to be inferred.

Because let is very important, they are now a part of the language.

To restrict rank 1 polymorphism, types and type schemes are formed.

8.5 Algorithm W

This algorithm finds the types by adding fresh types and trying to refine these each step. Each step more information about the use of parameters is found. Algorithm W uses a syntax directed variation of the HM type rules, which means that for every term at most one rule applies. Generalisation occurs at let-bindings and instantiation only at use-sites of variables.

Using type substitutions, type schemes are transformed and thus refined. Unification of two fresh types is important to find the most general unifier.

See the slides for detailed code of the algorithm.

9 Capita selecta - Super combinators

9.1 Rewriting Haskell basics in lambda-calculus

9.1.1 Functional representations of data structures

We can write data structures like functions, we gain a continuation programming style.

```
data Pair a b = Pair a b
pair a b = \\f. f a b -- more intuitive
pair a b f = f a b

fst(pair a b) = a
fst pab = pab (\ a b. a)
snd pab = pab (\ a b. b)
```

Normally we would ask a pair for a first or second element. `fst` calls the function `f` in `pair`. This is a continuation function, pair 'stores' values and returns them to the function `f`. Each example begins stating the normal data structure

```
data Bool = True | False

if c t e
true = \t e. t
false = \t e. e
```

```
if true  t e => t
if false t e => e
```

```
data Mayba a = Just a | Nothing

just   a j n = \f j a
nothing j n = \f n

--example:
case ma of
  Nothing = e1
  Just = e2

maybe ma e1 e2 = ma e1 e2
```

They are equal, but e2 actually is a function that takes an a, because the Just carries an a that is still in scope. This can also be done for lists.

```
data List a = Cons a (List a) | Nil

cons a as c n = c a as
nil       c n = n
```

The steps you take are, first, you create functions out of the constructors. Secondly you give the functions the arguments they need. Next you add arguments containing functions for every constructor. Then apply the functions with the arguments.

9.1.2 Remove (recursive) let/in structures

```
let x = e1 in e2 = (\xdef. e2) e1

Recursive let
x = e1 in e2 -- in e1 an x is bound.

let x = .. x ..

let x = rx
                where rx = x rx | rx = fix rx
 x x = .. x ..
        ...(...x...)...

fix x = let rx = x rx
  in rx
```

In recursive let constructions you need to rewrite the x such that it is not recursive. Fix ix a recursive pattern, that will evaluate a part of it self plus itself. We can write fix as a lambda expression.

$(\lambda x.xx)(\lambda x.xx)$ rewrites to itself. Another expression will rewrite to itself: $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))F$ and gives back the F. Recursion over the F

```
fix  f = \xdef.  f(xx)(\xdef.f(xx))
```

Recursion can be written in many ways, it's normally named Y (capital).

In lambda calculus we can formalize all the data structures we use in Haskell.

9.2 Combinator Reduction SKI

In the following example the y+2 is already bound by the lambda.

```
(\x  ..  x  x  x)  ()
(\x  \y   x       x)  (y+2)
    (\y   y+2   y+2)
```

This can be done by beta reduction.

But SKI combinators can build lambda expressions without having to compute all parts of an expression, but combine them together.

Now we will rewrite lambda expressions, such that it doesn't contain a lambda. Using a data structure, we can rewrite an example lambda.

I for Identity, K for constant and S is

```
data Lambda = App Lambda Lambda
                              | Var String
                              | Lambda String Lambda


\x. e1
\x. x -> I
\x. y -> K y
K y x = y
```

When $e1$ is rewritten e1 contains no variable or lambda, so it is a application $e1e2$. Next we need to substitute the x in $e1$.

```
e1 = (e1 e2)
\x. e1 = \x. ((\x. e1) x) ((\x. e2) x)
              ( re1   )       ( re2   )


(\f g x. (f x) (g x)) re1 re2
```

For a function that takes the n^{th} element from a list, we have the following.

```
el n = \l. if (=n 1) (hd l) (el (-n 1) (tl l))
      = S (\l if (=n i) (hd l)) (\l el (-n 1) (tl l))
           (S (\l if (=n 1)) (\l hd l)) (S (\l. el (-n 1)) (\l tl l))


(\l. hd l) = (S (\l. hd) (\l l))
              (S (K hd) I)
```

By rewriting all the way, you see that you do much work, and at the end you throw it away. The expression blows up if you do it this way. You will create a tree of S K I. S means send the argument in the function and the argument part. K means don't use it, it's not needed. I means, here you can use the lambda, apply it.

9.2.1 SKI ⇒ G

This section is not needed.

9.2.2 SKI-BC-S'B'C'

$Bfgx = f(gx)$ Remember that C stands not for compose, so B does. B has a function and a argument part, and the x is only needed in the argument part. And c is equal to flip, $Cfgx = fxg$. C takes an x, and applies it to the f, instead of a g. B and C are like each other.

NExt, the `app` is a smart constructor.

```
data Lambda = S | K | I | B | C | App L L

S K K x = (K x) (K x) = x
S K K = I

app S (K f) (
app      (App (K f)))(K g) = App K (App f g)

app (App (S (K f) g)) = App (App B f) g [??]

app (App (S f (K g))) = App f (App C g) [??]
```

In other words you can use R and L for B and C respectively. R for right, because the argument goes into the right, and with L the argument goes to the left.

Expressions grow exponentially because the transformations require that all arguments are passed to all parts of the expression, even though you don't need it. Introducing the S'

```
S' c f g x = c (f x) (g x)
B' c f g x = c f     (g x)
C' c f g x = c (f x)  g
```

The *c* argument is a constant part of the expression. A director string is the part of combinators that states the number of arguments times the number of application nodes in the graph. This string tells evaluation which path to take.

```
foldr op e []      = e
                (x:Xs) = x `op` foldr op e xs

sum = foldr (+) 0

sum [] = 0
sum (x:xs) = x + (sum xs)
```

foldr passes three parameters, the 'normal' way only passes one.

```
foldr op e = go
  where go [] = e
            go (x:xs) = x `op` go xs

--using combinators
foldr = \op e. Y go'

go' go = \l. if (nil l) e (op (hd l) (go (tl l)))
          = S (L (R if nil) e) (S (R op hd) (R go tl))
```

```
--take out go

go' = R' S (L (R if nil) e) (R' (S (R op hd))(L' (R I tl)))
```

Doing graph rewriting this way, you get self optimizing programs. In Haskell you want to program as abstractly as possible, but you don't want to pay for abstraction. Clean uses partial evaluation and graph rewriting to evaluate a code.

```
Y f = f (Y f)
```

9.2.3 Super Combinator

Define a special purpose set of combinators accustomed to the program at hand.

```
\x. x .. x .. x .. x
```

These x's are reachable by subtrees in the expression, and some trees have no x inside them. In the graph, the substitution is visible, you see parts where x's are not needed. The sub expressions without x named e_1, e_2, e_n. A function that will build a graph with x's given the parts without x.

```
f x = spnx e1 e2 en x
```

Not all combinators are interpreted, but the substitution is clear. By looking at free expressions, expressions that are not using an x, you can make a function that takes free expressions and substitutes x between them. The free expressions don't depend on a argument, so they don't have to be evaluated as if the were using it. The function is always the same, given any argument.

What about ordering e_1, e_2, e_n..?

10 Capita selecta - Garbage collection

10.1 Memory management

Managing can be done manual or via garbage collection. The classical example is a free list. A free list is a data structure that allows for dynamic memory allocation, allowing (de)allocation very simple by taking free space from, and linking to a free space in the list. Reference counting allows you to count all references such that you can reallocate memory when it is not referenced. It is simple and lightweight, but also naÃ¯ve and inefficient.

10.1.1 Mark-sweep

Mark-sweep algorithms first finds out what is free and what isn't to use this info later on. A variation on this algorithm is black-grey-white. Where black contains marked nodes with all pointers visited, grey nodes with some pointers visited and white nodes which are unvisited. The problem right now is running a program and garbage collection in parallel. It might cause for inconsistencies because pointers might change while some of those areas were already checked. It is solved by just taking all memory, and after that do garbage collection. The problem is that a program might stutter at such moment where it needs to clear up memory. In the next subsection even more strategies are stated, but c-like memory management becomes infeasible. It allows pointers to be modified like pointers.

10.1.2 Mark-compact

When using the mark-compact algorithm, you need moving memory, which results in no fragmentation nor administration. Moving memory is tricky and very expensive, while most problems are circumvented when using a language that doesn't contain expressions using pointers, moving memory all the time is to expensive. The copying garbage collector uses two halves of the memory. It uses one as a working part, and the second to de-fragment. It is fast and particularly good for short-lived objects. Problems with cycles are solved by replacing the old address with a link to the new address. A solution to get it working with long lived objects, is to save those in a third part of the memory. The problem is that you don't yet know when objects are long lived. When observing the lifespan of an object you can generate a garbage collector.

10.1.3 Generational garbage collector

Each object gets an age, and gets put in the right area of the memory. The bigger or long lived the objects are, the smaller the chance that it gets collected as garbage. The smaller or short lived, it gets collected earlier. Not good for real time systems, but good for interactive systems. This does pose problems like: finalization, stable and weak pointers.

11 Capita selecta - Smalltalk

11.1 Objects and classes

Smalltalk is the first (real) object oriented language, where every thing is an **object**. The programming paradigm is that objects are used to send messages. Other languages split up between primitives and objects. And object has an **class**, and class holds the following meta information:

1. How big is an instance
2. Which mehods can be called (mappings from methods to code)
3. Etc.

But we said everything is an object, so class also is an object and has an class. And the end there is an 'class-sink' called **metaclass**.

Methods are found using **inheritance**, if the class doesn't contain it, ask the class' class. This is called **indirection**. An indirection solves any problem, but is inefficient. **Caching** is used to find methods faster, using a hash table.

Because of the object/class structure, the code can be modified during run-time. Also it's compiler, since it is also made of objects.

11.2 Simple language

Smalltalk is a simple language. It can use **libraries** and is very extensible, but the code also becomes **complex** if you want to express complex problems. Even though it uses libraries, there is no package support. This means that you have a closed world, you have all objects in one big file. This makes **deployment** hard, dependencies are hard to control. It is only possible to deploy by sharing source code, which is not suitable for all commercial purposes.

Furthermore, Smalltalk uses pointers to methods, which in Java for example is changed to names. The problem of pointers is, that in different builds of a source, pointers may differ, whereas names won't change.

11.3 Variants

Java is based on the language SELF, Javascript uses **prototypes**. Prototypes contain a dictionary of the methods, that might refer to own or other prototypes' methods.

Prototyping languages have a sharing problem, which means that prototypes share methods from (upper)classes.

Prototyping is slow but can be **optimized** by inlining and specializing. These techniques are used in modern JIT-compilers.

Smalltalk is a simple language, which implies that it is easy to write but suffers on modularity and reduces efficiency

12 Capita selecta - Utrecht Haskell compiler

The following text is not part of the exam

12.1 Introduction

There have been many functional programming languages, Haskell next to ML came out. It was intended that Haskell was a small languages, but has grown a lot lately. Thus classes and instances where added. An overview of the compiler process is as follows:

1. Haskell
2. Essential Haskell, which rewrites all the syntactic sugar. Errors produced here, need to be written back to the source.
3. Core language builds up information for later use.
4. Grin removes laziness and links all modules together.
5. Silly (is used sometimes)
6. C code
7. Executable

It also happens that a transformation happens that stays to the core. A part is then transformed, but not compiled to another language. The first step is desugaring Haskell. All statements are rewritten to lets, case statements, using full pathnames for datatypes and inserting implicit functions like fromInteger. The notation gets more involved with every step. The core language also resolves class instances, when using fromInteger for instance, it find the correct overloaded version for the current case. GHC will insert the code for the right instance and take out the extra variables that gave information about which instance to use. Case matching is rewritten to numbers, and there it finds equality for constructors. Recursion has been made explicit to provide the information, later on, that the function is recursive. Case statements force values to be evaluated. In lazy languages, values are not evaluated, so at every point where you use a lazy value, you need to check wether it is evaluated.

12.2 Tools

There are many tools that can be used to create compilers.

- **Shuffle** helps to organize code. It has a set (or variants) of compilers.
- **Attribute grammer compiler** builds up a tree of catamorphisms.
- **Ruler** is a tool for specifying type rules.
- **ghc**