The Semantics of Higher Order Algorithms
Lecture I
May 27 - 2013

Dag Normann
The University of Oslo
Department of Mathematics

Introduction

- By an higher order algorithm we will mean any algorithm where the input data may themselves be functions, functionals or any other kind of infinite entity.
- When this is made more concrete with e. g. the use of a programming language, the operational semantics of this language will induce some kind of algebra on the typed set of programs, but there may be numerous choices for a good denotational semantics for the language in question.

Introduction

- There may even be ways to interpret an algorithm that are partly operational and partly denotational.
- In these lectures we will consider approaches to such models, both explicit examples and more abstract classes of models.

Introduction

- Today we will survey the history of our subject, introducing the Scott model and the path that led to it.
- We will let Plotkin's PCF be our key example of a concretization of higher order algorithms.
- On Thursday we will consider more abstract approaches to higher order models, letting Kleene's first model and the finitary sequential procedures be our two key examples.
- On Friday we will present the sequential procedures and the sequential functionals as our favorite model, and show in what sense domain theory is inadequate for modeling higher order deterministic algorithms.

Introduction

- These lectures are to some extent based on the forthcoming

  Higher Order Computability

  by John Longley and Dag Normann, to appear in the CiE Book Series on Computability, published by Springer Verlag.

Prerequisites

We will not assume any deep knowledge of computability theory, domain theory, topology etc., but it will certainly be an advantage to be familiar with:

1. Basic set theory as given in a course on discrete mathematics.
2. Basic knowledge of formal languages, with distinction between free and bounded variables.
3. The basic definition of the computable functions, e. g. as defined via Turing Machines.
   The universal Turing Machine.
4. Basic intuition about algorithms and the operational semantics.

DO NOT HESITATE TO ASK QUESTIONS.

A brief historical introduction - λ-calculus

- The historical origin of a subject in science is normally vague, but we have to start somewhere.
- It is tedious to be historically correct, so we will look at the history using, to some extent, today's terminology.
- We choose to start with the λ-calculus in a modernized version.

A brief historical introduction - λ-calculus

- λ-calculus is given as a term language with variables and rewriting (conversion) rules.
- The underlying intuition is that all objects are functions, and the terms will be generated from the variables using two constructors:
  - If M and N are terms, then (MN) is a term, thought of as M *operating on* N.
  - If M is a term and x is a variable then $(\lambda x.M)$ is a term, thought of as denoting the function that to an interpretation of x gives the associated interpretation of M.
  - For instance, $\lambda x.x$ denotes the identity function in an abstract sense.

A brief historical introduction - λ-calculus

- λ-abstraction is a binding of variables introducing the usual distinction between free and bounded variables, and then also defining what we mean by *a substitutable term*.
- The α*-rule* allows shifts of bounded variables

$$\lambda x.M \to \lambda y.M^x_y \ .$$

- The β*-rule* links the two syntactic constructors:

$$(\lambda x.M)N \to M^x_N \ .$$

- In both cases we assume that the substitution is legal.
- We let $\to^*$ be the reflexive and transitive closure of $\to$.

A brief historical introduction - λ-calculus

- The *Church numerals* are defined by

$$\hat{n} = \lambda x.\lambda y.x^n(y)$$

  where we drop unnecessary parentheses.
- A closed term N *defines* a partial function $f : \mathbb{N} \hookrightarrow \mathbb{N}$ if for all n and m, the following are equivalent:
  1. $f(n)$ is defined and is equal to m.
  2. $N\hat{n} \rightarrow^* \hat{m}$.
- It is a fact that a partial function f is λ-definable if and only if it is computable by a Turing Machine.

A brief historical introduction - λ-calculus

- Turing's model was immediately accepted as conceptually sounder than λ-calculus as a foundation of computability and decidability.
- The calculus was nevertheless favoured by computer scientists in the 50'ies and 60'ies as a semantical tool in the theory of programs.

A brief historical introduction - λ-calculus

- Mathematicians were more skeptical, due to the fact that λ-calculus is pure syntax - there are apparently no natural mathematical models.
- Mathematicians preferred to study models of computability relevant for mathematical structures while computer scientists preferred to study mathematical structures illuminating calculi of computations.
- This led to different approaches, of which we can harvest the best from both.

A brief historical introduction - Combinators

- There is an alternative approach to λ-calculus using *combinators*.
- The set of combinator terms are generated from the constants K and S by the rule

$$M, N \mapsto (MN)$$

Introduction

- In order to improve readability, we drop parentheses according to certain conventions, e. g. MNL actually means $((MN)L)$.
- Then the conversion rules of the combinators are:
  1. $KMN \to M$
  2. $SMNL \to (ML)(NL)$

  The combinators have exactly the same expressive power as the closed λ-terms.

A brief historical introduction - Kleene

- λ-calculus is at the same time first order and higher order, all objects may represent both arguments and algorithms.
- In 1959, Kleene introduced higher order algorithms for typed functionals.

Introduction

- He let $Tp(0) = \mathbb{N}$ and $Tp(k+1)$ consist of all set-theoretic functions from $Tp(n)$ to $\mathbb{N}$. Via nine schemes, S1 - S9, he defined a relation

$$\{e\}(\Phi_1, \ldots, \Phi_n) \simeq a$$

  meaning that algorithm number e with functional inputs $\Phi_1, \ldots, \Phi_n$ terminates with output $a \in \mathbb{N}$.
- We will not go into detail, since the main applications were to definability theory rather than to computability theory.

A brief historical introduction - Kleene and Kreisel

- In 1959 Kleene and Kreisel independently introduced another typed structure of functionals.
- Kleene called these functionals *countable* since we need a countable amount of information to fully describe each of them.
- Kreisel called them *continuous* because of his construction via formal neighborhoods.

A brief historical introduction - Kleene and Kreisel

- Their constructions were not equivalent, but were soon considered to be alternative constructions of what we now call the *Kleene-Kreisel continuous functionals*
- Kleene's S1-S9 make sense also for the typed structure of KK-continuous functionals.
- What is significant to us is that there is an alternative way to define what we may mean with a *computable continuous functional*, as *codable by a computable function*.
- Some mathematical effort were put into the analysis of the relation between the two definitions of being computable.

A brief historical introduction - Platek

- The main problem with Kleene's approach was that it only dealt with total functionals at all types.
- This somehow bars the use of partial functions at intermediate steps in a computation.

A brief historical introduction - Platek

- In 1963 Platek, in his thesis, gave an alternative construction of
  1. Functionals of finite types
  2. Computable functionals
- It is time to be a bit more precise than what we have been up to now.
- For the sake of simplicity, we continue to sacrifice historical correctness.

Typed structures

We have now seen enough examples to justify the following definition:

Definition

a) The *finite types* is the language generated by the following grammar

Type σ
$\sigma ::= \text{Nat} \mid (\sigma \to \sigma)$

b) We simplify the notation by dropping the outmost brackets and rewriting $\sigma \to (\tau \to \delta)$ to $\sigma, \tau \to \delta$.

Typed structures

Any type can be written on the *normal form*

$$\sigma = \tau_1, \ldots, \tau_n \to \text{Nat}$$

where $n \geq 0$.

Typed structures

Definition

An extensional *typed structure* is a map $\sigma \mapsto T(\sigma)$, where $T(\sigma)$ is a set for each type σ, such that

$T(\mathsf{Nat}) \subseteq \mathbb{N} \cup \{\bot\}$

$T(\sigma)$ is a set of functions defined on $T(\tau)$, and with values in $T(\delta)$, when $\sigma = (\tau \to \delta)$.

Typed structures

- A typed structure is *total* if $\bot \notin T(\mathrm{Nat})$, otherwise it is *partial*.
- Sometimes one will be primarily interested in a total typed structure, but will need a partial counterpart in order to interpret algorithms.

A brief historical introduction - Platek

Definition

We define Platek's functionals (*hereditarily consistent partial functionals*) as a pair $(P(\sigma), \sqsubseteq_\sigma)$ for each type σ as follows:

- $P(\mathsf{Nat}) = \mathbb{N} \cup \{\bot\}$ with

$$a \sqsubseteq_{\mathsf{Nat}} b \Leftrightarrow a = b \vee a = \bot \, .$$

- $P(\sigma)$ is the set of monotonously increasing functions from $P(\tau)$ to $P(\delta)$ when $\sigma = (\tau \to \delta)$.
- $P(\sigma)$ is then ordered by the pointwise ordering.

A brief historical introduction - Platek

Platek's calculus is based on constants for basic arithmetical functions together with typed combinators:

1. $K_{\sigma,\delta}$ of type $\sigma, \delta \to \sigma$.
2. $S_{\sigma,\delta,\tau}$ of type $(\sigma \to (\delta \to \tau)), (\sigma \to \delta), \sigma \to \tau$.
3. Y_σ of type $(\sigma \to \sigma) \to \sigma$.

where we have

- Application terms must be correctly typed.
- Conversion rules for K and S with indices are as before.
- The conversion rule for Y_σ is $Y_\sigma N \to N(Y_\sigma N)$.

A brief historical introduction - Platek

- Every functional Φ in $P(\sigma \to \sigma)$ will have a least fixed point $Fix(\Phi)$ in $P(\sigma)$ and the map $\Phi \mapsto Fix(\Phi)$ is an element of $P((\sigma \to \sigma) \to \sigma)$
- If we let $[[Y_\sigma]]$ be this map, we have an interpretation $[[M]]$ of each correctly typed term in Platek's calculus, such that

$$M \to^* N \Rightarrow [[M]] = [[N]] \, .$$

- This defines the *Platek-computable* functionals directly using the denotational semantics, without any explicit operational semantics. This was the style of mathematicians in 1963.

Platek's Thesis

Platek's thesis is hard to get at, but a readable account can be found in

Johan Moldestad

Computations in Higher Types

Lecture Notes in Mathematics 574

Springer Verlag 1977

A brief historical introduction - Scott

- In 1969 Dana Scott combined the ideas of Kleene/Kreisel and Platek - added a few of his own - and made our subject a part of Computer Science.
- He formed the language/logic LCF with a new kind of semantics consisting of
  1. A term language very much like the one suggested by Platek.
  2. A full language containing relation symbols for $\sqsubseteq_\sigma$ and $=_\sigma$.
  3. A formal logic.

A brief historical introduction - Scott

- Adding a further restriction of continuity to Platek's construction, he defined the *hereditarily partial and continuous functionals* of finite types, and gave an interpretation of each term sound with respect to the calculus and logic.

A brief historical introduction - Scott

- Scott's motivation was to replace the untyped λ-calculus with a typed variant with a sound semantics, claiming that in essence, all data are typed.
- Within months, however, he discovered that he had the tools needed for constructing a mathematical model for the original λ-calculus.

A brief historical introduction - Scott

- This led to the use of topology, via domains and complete lattices, for semantical purposes in the analysis of programs and algorithms.
- One question we will ask ourselves is if Scott's domain theory is the best tool for giving an interpretation of the calculus LCF.

A brief historical introduction - Plotkin

- In a seminal paper from 1977 Plotkin linked this development of the understanding of higher order computability even closer to the theory of programming.
- By going back to traditional typed λ-calculus, he reformulated LCF to PCF.

A brief historical introduction - Plotkin

- He devised a deterministic, or sequential, evaluation strategy for PCF-terms of base types.
- The strategy will give a terminating evaluation of all closed terms that are interpreted as an integer in Scott's model.
- PCF is a standalone programming language for higher order algorithms, and has inspired e. g. languages like Haskell.

PCF - The calculus

- Our prototype for higher order algorithms will be those dealing with functionals of finite types, and expressible in a reduced version of PCF.
- We may extend our universe of types with a type for the booleans and close under products $\times$, disjoint unions $\oplus$, lists, trees etc.
- We may even introduce types using strictly positive induction or W-kinds of types, and preserve much of our analysis.

PCF - The calculus

- We will not do any of this here, in order to be able to focus on distinctions like continuity vs. sequentiality, effectivity vs. completeness and partiality vs. totality.
- We will later see a construction, the *Karoubi envelope*, that will enable us to view a lot of extra datatypes as represented in a simple typed structure.

PCF - The calculus

- Our language will be the language of typed λ-calculus with the following constants:
  1. A constant $\underline{0}$ of type Nat.
  2. A constant *Suc* of type Nat $\to$ Nat.
  3. A constant *Pre* of type Nat $\to$ Nat.
  4. A constant $\supset$ of type Nat, Nat, Nat $\to$ Nat.
  5. Constants Y_σ of type $(\sigma \to \sigma) \to \sigma$.
- We close under application and λ-abstraction as before.

PCF - The calculus

- Our conversion rules will be the α- and β-rules of typed λ-calculus together with:
  1. $\mathit{Pre}(\mathit{Suc}\ x) \to x$.
  2. $\supset \underline{0}NM \to N$.
  3. $\supset \underline{k+1}NM \to M$.
  4. $Y_\sigma N \to N(Y_\sigma N)$.
- where we define the *PCF-numerals* by the recursion $\underline{k+1} = \mathit{Suc}(\underline{k})$.
- A partial function f of type Nat $\to$ Nat is *PCF-computable* if there is a closed term M_f such that $f(n) = m$ if and only if $M_f\underline{n} \to^* \underline{m}$.
- It may come as a surprise that all Turing-computable functions are PCF-computable.

PCF - The calculus

In a term, we may give the type of a variable as an index of the first occurrence, but never at later occurrences.

Example
Consider the term

$$\lambda x_{\mathsf{Nat}}.\lambda y_{\mathsf{Nat}}.(\,Y_{\mathsf{Nat}}\lambda f_{\mathsf{Nat}\to\mathsf{Nat}}.\lambda z_{\mathsf{Nat}} \supset zySuc(f(Pre\ z)))x\ .$$

This shows that $x, y \mapsto x + y$ is PCF-computable.
Any other function defined by primitive recursion is definable by the same method.

PCF - The calculus

- *Minimization*, or the μ-operator, is a functional of type $(\text{Nat} \to \text{Nat}) \to \text{Nat}$ defined by
- $\mu(f) = n$ if
  - $f(n) = 0$.
  - $f(m)$ is defined and $f(m) > 0$ for all $m < n$.
- Minimization is PCF-definable. Let us look at the next slide:

Minimization in PCF

We define minimization informally by the equation

$$\mu(f) = \begin{cases} 0 & \text{if} \quad f(0) = 0 \\ \mu(\lambda x.f(x+1)) + 1 & \text{if} \quad f(0) > 0 \end{cases}$$

If F is a variable of type $(\mathrm{Nat} \to \mathrm{Nat}) \to \mathrm{Nat}$ and f is a variable of type $\mathrm{Nat} \to \mathrm{Nat}$ we consider the term

$$\lambda F.\lambda f.\supset (f\underline{0})\underline{0}\mathit{Suc}(F\lambda x_{\mathrm{Nat}}.f(\mathit{Suc}\ x))\ .$$

We then need $Y_{(\mathrm{Nat}\to\mathrm{Nat})\to\mathrm{Nat}}$ in front of this to give a PCF-term for minimization.

The Scott interpretation

- The Scott model is a natural completion of the finitary versions of the Platek model.
- By recursion on the type σ, we define the finite partial ordering $(D_n(\sigma), \sqsubseteq_{\sigma,n})$ as follows:
  - $D_n(\mathsf{Nat}) = \{\bot, 0, \ldots, n\}$
  - $a \sqsubseteq_{\mathsf{Nat},n} b \Leftrightarrow a = \bot \vee a = b$ for $a, b \in D_n(\mathsf{Nat})$.
  - $D_n(\sigma \to \tau)$ is the set of monotonously increasing functions from $D_n(\sigma)$ to $D_n(\tau)$.
  - $\sqsubseteq_{\sigma\to\tau,n}$ is the pointwise ordering.

The Scott interpretation

- These typed structures can be linked via a system of embedding-projection pairs:
- If $n \leq m$ and σ is a type, we define the embedding $\varepsilon^{\sigma}_{n,m} : D_n(\sigma) \to D_m(\sigma)$ and the projection $\pi^{\sigma}_{n,m} : D_m(\sigma) \to D_n(\sigma)$ by recursion on σ as follows:
  - $\varepsilon^{\mathsf{Nat}}_{n,m}(a) = a$
  - $\pi^{\mathsf{Nat}}_{n,m}(a) = a$ when $a \in D_n(\mathsf{Nat})$ and $\perp$ otherwise.
  - If $\sigma = \tau \to \delta$, $f \in D_n(\sigma)$ and $y \in D_m(\tau)$ we let

$$\varepsilon^{\sigma}_{n,m}(f)(y) = \varepsilon^{\delta}_{n,m}(f(\pi^{\tau}_{n,m}(y))) \, .$$

  - If $\sigma = \tau \to \delta$, $g \in D_m(\sigma)$ and $x \in D_n(\tau)$ we let

$$\pi^{\sigma}_{n,m}(g)(n) = \pi^{\delta}_{n,m}(f(\varepsilon^{\tau}_{n,m}(y))) \, .$$

The Scott interpretation

- The crucial property, easily proved by induction on the type, is
  1. $\pi^{\sigma}_{n,m} \circ \varepsilon^{\sigma}_{n,m}$ is the identity function $id_{n,\sigma}$ on $D_n(\sigma)$.
  2. $\varepsilon^{\sigma}_{n,m} \circ \pi^{\sigma}_{n,m} \sqsubseteq id_{m,\sigma}$.
- This shows that we have defined a *chain* of typed structures, and this chain will have a limit.
- This limit is known as the Scott model, see the next slide:

The Scott interpretation

- For each type σ and $n < m$, we may consider $D_n(\sigma)$ as a subset of $D_m(\sigma)$ identifying $x \in D_n(\sigma)$ with $\varepsilon^{\sigma}_{n,m}(x) \in D_m(\sigma)$.
- If we let $D_\omega = \bigcup_{n\in\mathbb{N}} D_n(\sigma)$ for each type σ, we have produced a typed structure of finitary partial functionals.
- Since the ε-functions preserve $\sqsubseteq$ (exercise), we inherit the ordering $\sqsubseteq$ in $D_\omega(\sigma)$.
- This typed structure is the backbone of Scott's model, which may be viewed as the completion, much like going from $\mathbb{Q}$ to $\mathbb{R}$.

The Scott interpretation

- We do not intend to introduce domain theory in full in this course, just enough to enable us to define the Scott model.
- An *ideal* in $D_\omega(\sigma)$ will be a set of the form

$$\{y \in D_\omega(\sigma) \mid \exists n\ y \sqsubseteq x_n\}$$

  where $\{x_n\}_{n\in\mathbb{N}}$ is an increasing sequence from $D_\omega(\sigma)$.
- The Scott-interpretation $D(\sigma)$ of the type σ will be the set of ideals, ordered by inclusion.

The Scott interpretation

- We consider $D_\omega(\sigma)$ as a subset of $D(\sigma)$ by identifying $x \in D_\omega(\sigma)$ with the ideal generated from the constant x sequence.
- We define application on the typed sets of ideals via pointwise application of generating sequences.
- An ideal is *computable* by definition if it is generated from a computable sequence. This corresponds to being c. e. .

The Scott interpretation

- It is easy to prove
  a) If $\sigma = \tau \to \delta$, then every element in $D(\sigma)$ commutes with least upper bounds of directed sets.
  b) There is an ideal representing the least fixed point operator on each $D(\sigma \to \sigma)$.
- We can use this to interpret every PCF-term N of type σ with free variables among $x_{\tau_1}, \ldots, x_{\tau_n}$ as an element $[[N]]$ in $D(\tau_1, \ldots, \tau_n \to \sigma)$.
- This is, slightly modified, the classical approach. The approach links computability and continuity in a strong sense.

The Scott interpretation

- On Friday, we will question if this link is as natural as originally believed, or if we should look for alternatives to continuity as abstractions of computability.
- Scott's model does not capture that there are underlying *evaluations* of functions on inputs.

Plotkin's adequacy Theorem

- Plotkin's sequential strategy for evaluating closed PCF-terms of type Nat is simple:
- Apply one of the conversion rules at the leftmost position possible.
- Then a closed term N of type Nat rewrites to a numeral $\underline{n}$ if and only if $[[N]] = n$.
- The proof borrows ideas from the normalization theorem for simply typed λ-calculus.
- We say that he model is *adequate* for the calculus.

Non-determinism in the model

- One problem with the model, already known to Platek, is that it models more than the PCF-definable functions, even at the finite level.
- Let $f(n, m) = 0$ if $n = 0$ or $m = 0$, $f(1, 1) = 1$ and $f(n, m) = \bot$ in all other cases.
- $f \in D_{\mathsf{Nat},\mathsf{Nat}\to\mathsf{Nat},k}$ for $k \geq 1$, but here will be no closed PCF-term N such that $f \sqsubseteq [[N]]$.
- f is non-deterministic, and cannot be evaluated by a sequential procedure.

Summarizing

- Today we have been looking at the historical development of higher order computability.
- On Thursday we will consider extensional and intensional typed structures at a more general level, including what we call *the sequential functionals*.
- On Friday we will mainly discuss properties of these sequential functionals, and to some extent, challenges related to models for higher order algorithms.