Teaching Denotational Semantics

Achim Jung

September 18, 2014

1 Introduction

In 1969 Dana Scott suggested, [Sco93], that a Tarskian semantics could be given to programming languages by employing ordered structures of a certain kind, now known as *domains*. One of his key insights was that recursion can be modelled via the *least fixpoint* of endofunctions on domains. Shortly afterwards Gordon Plotkin, explored this proposal further; among the many results of his landmark paper [Plo77] he formulated and proved *computational adequacy*, which can be said to justify Scott's least-fixpoint semantics. Much of the development of denotational semantics since then can be traced back to these two papers and few would contest their status as "classics" of our subject.

50 years later, one would expect to see Scott's and Plotkin's work reflected in the standard syllabus of computer science degrees around the world, on a par with other classic results of our subject, such as the relationship between automata and languages, the undecidability of the halting problem, and NP-completeness. This, however, is not the case, despite a clutch of excellent textbooks appearing in the late 1980s and early 1990s: [Sch86, Ten91, NN91, Gun92, Win93, Mit96]. Of the many possible reasons for this state of affairs, I believe the following are the most important: (a) the subject is notationally subtle and heavy at the same time, (b) students do not have enough mathematical background, (c) the "rewards" for the time and effort invested appear limited to the student.

In this note I want to report on my own experiences of teaching denotational semantics, which has taken place in a variety of contexts and to some very different audiences, with my primary reference point being a five-lecture course delivered as part of the Midlands Graduate School in the Foundations of Computing Science (MGS), which since 2001 has taken place annually at one of the four partner universities: Birmingham, Leicester, Nottingham, and Sheffield.

My aim here is to present my choice of topics and methodology for these five lectures in the light of the problems listed above, and to report on the specific experience in that context. So the focus is on reflection, not on presenting the material in detail and what follows are not course notes, but I still hope that I will give enough detail so that a knowledgeable reader may be able to adopt some ideas for their own teaching.

I am grateful to Mike Mislove for giving me the opportunity to present my musings in the newly established SIGLog Newsletter and I hope that my example will encourage others to use this venue to discuss matters of *communicating research* as well as research itself.

2 Approach

As I said at the beginning, denotational semantics is a challenging subject to teach, dealing as it does with an unusually wide range of mathematical ideas and complex notations. Students of computer science are typically poorly prepared for such a study, and students of mathematics are not interested. Both constituencies have difficulties seeing the point of it all, and this phenomenon is my first main issue: How to motivate the subject and how to pique students' interest?

Many texts choose to start with a simple `while`-language, introduce its denotational semantics as state-transforming functions and explain the meaning of a `while`-loop as a least fixpoint. From a pedagogical point of view, it seems appropriate and correct to begin with a formalism that is familiar to the students but I'd like to propose that this is also a key problem: The language is simple and the semantics is hard. However, if we are to "sell" denotational semantics, then it should be the other way round!

Alternatively, one could begin immediately with PCF; it can be argued, after all, that it is nothing more than the core of any modern functional programming language and students can be assumed to have taken a course in that subject before. Thomas Streicher's excellent book [Str06] does exactly that. I have found, however, that in practice the learning curve is too steep for many in my target audience (which includes mathematicians as well as computer scientists).

The approach that I have adopted and that I want to advertise here, is the one that Carl Gunter chose for his textbook [Gun92]. In a nutshell, the idea is to discuss the semantics of the simply typed lambda calculus as a primer for PCF. Apart from offering a gentler path towards domain semantics and the adequacy proof, it lays the foundation for other subjects that students may want to study, in

particular, type theory.

No doubt, from a pedagogical perspective Gunter's approach has many inherent problems of its own — and one aim of this note is to discuss those — but there are many attractive features which tip the balance in its favour. Foremost, the presentation can be motivated by the desire to *prove theorems about calculi*. Without going into any detail, these concern compositionality, soundness, completeness, definability, adequacy, and extensionality. The methods employed in the proofs are themselves rich and varied, and they are ubiquitous in denotational semantics: proof by structural induction, term model construction, logical relations and domain theory. In fact, their applicability is not confined to denotational semantics and one of the joys of teaching at the Midlands Graduate School is to see these themes and methods emerge in different courses.

This emphasis on theorems marks me out as a mathematician, I admit, and it does not work for everyone. As an alternative motivation that may work better for more practically-minded computer science students, I'd like to advertise the approach that Martín Escardó has successfully tried at the Midlands Graduate School. He begins by writing down the PCF term for the "Gandy–Berger functional", [Ber90], which tests whether a predicate on Cantor space (the latter implemented as int $\to$ bool) is satisfiable. The term is small but quite incomprehensible and Martín's point is that it is not clear at all that it must terminate for total predicates. He then goes fairly deeply into the domain semantics of PCF and develops the topological machinery that one needs for the termination proof. The Gandy-Berger functional is not just a pedagogical device; it appears, for example, in Alex Simpson's implementation of exact real number integration, [Sim98].

Returning to my approach, a second key argument for beginning with the lambda calculus is that despite its similarity to functional programming, still appears sufficiently "foreign" to students so that giving a semantics seems appropriate; in other words, the language is "hard," and it remains to make sure that the semantics is (or appears) "easy." Here we take advantage of the fact that the simply-typed lambda calculus only requires ordinary functions for its semantics, and that the discussion of partiality and fixpoints can be deferred to the second part when we talk about PCF. This goes some way to address Problem (b) mentioned in the Introduction, the lack of mathematical knowledge in our target audience.

Disappointingly, for the first problem — the notational complexity so typical of denotational semantics — I cannot offer a silver bullet, as that's just the way things are. However, we have an advantage when we are giving lectures as often some detail can be said rather than written down, and sometimes, it can even be suppressed entirely. Since I don't use slides but always present with the help of

the board only, it is absolutely essential for me to simplify as much as possible. I'd like to propose that this is actually *a very good thing*, because students need to see how we practitioners organise the material for ourselves, how we keep on top of the notations, what we emphasise and what we neglect.

3 Lecture I: The simply typed λ-calculus

I usually begin the lectures with a discussion of different approaches to semantics, relating them to the ways in which a child learns the meaning of words. The denotational approach is perhaps the most natural one: pointing to a dog and simultaneously saying the word "doggie" to the child. There is already an interesting point to be made, in that the "objects" which our chosen calculus denotes are sets and functions, and these do not have any physical representation we can point to but are construction of our (mathematical) mind expressed in mathematical language. It follows that denotational semantics does not feel denotational at all but in essence comes down to a translation from a syntactic calculus to mathematical language. (Translation is what we use when we learn a foreign language but clearly not when we learn the first words from our parents.) I think it is important to confront this issue head on at the very beginning because it is one that tends to confuse and even confound learners when they encounter semantic definitions for the first time. One can then make the point that even straightforward translation can be useful if the languages involved are sufficiently different, and that one should feel entitled to question why the particular translation that denotational semantics offers provides any insight at all. I also stress that it is theorems we are after, not just alternative descriptions, and that the onus is on me to demonstrate that the theorems we will prove are interesting and useful.

At the Midlands Graduate School, my course is usually complemented with a concurrent one on the lambda calculus, so my introduction to the language can be brief. But even without that support, it is not too hard to present the syntax to an audience of beginning PhD students. There are, after all just two clauses to the definition of simple types and three to the definition of terms.

However, the issue of *variable binding* is much more subtle than it may appear to the students, and we know now that our best semantic accounts require a formidable mathematical apparatus. We don't yet know whether it is possible to present these in the context of an introductory lecture course, so for the time being we must stick to the tried and tested ways of keeping the problems of binding under control.

$$\frac{}{x^\sigma : \sigma} \qquad \frac{M : \sigma \to \tau \quad N : \sigma}{MN : \tau} \qquad \frac{M : \tau}{\lambda x^\sigma . M : \sigma \to \tau}$$

Figure 1: The term formation rules of the simply typed lambda calculus in Church style

Given the constraints of a rather short lecture course, I have come to the conclusion that Church's original formalism is the most appropriate. In it each variable has a *fixed type*, which may be part of the name (as in x^σ) or which may be accessed via a global *type assignment* function: type$(x) = \sigma$. The only proviso one needs to make (and the reason for it becomes clear early on in the course) is that there should be an unbounded number of variables for every type. This is the approach that one finds in [Win93, Chapter 11] and also [Plo77], for example.

While it is clear that Church's calculus is the most economical available, see Figure 1, it may still be worthwhile to spend some time here to ponder its pros and cons more generally as well as those of the alternatives.

One of its disadvantages is that apart from FORTRAN it has not been adopted in any practical programming language. A more serious objection is that the approach does not generalise to more sophisticated type systems. Regarding the last point, there is little I can say in defence of my choice other than to reiterate the point that formalisms should be introduced as and when they are needed, and they should not be allowed to dominate the presentation.

If nevertheless a Curry-style presentation with explicit type environments is chosen, then it seems to me that one should go the full mile and deal with them as one does in actual programming languages and in type theory, that is, type environments are lists and later declarations of a variable override those that come earlier, a phenomenon known as *shadowing*. The machinery required to make this work is formidable but at least it is honest. One has to talk explicitly about "structural rules", that is, weakening, exchange and duplication of type assumptions although that may not be apparent from the term formation rules, Figure 2, but they will be necessary for a discussion of the equational theory.

I don't know of any textbook on denotational semantics that goes down this route, and I agree that dealing with type contexts is probably not a core concern of denotational semantics, but the various compromises that one finds are not very convincing either: One common assumption, for example, is that terms are really only representatives of their α-equivalence classes and semantics is given to the

$$\frac{x \notin \mathrm{dom}(\Delta)}{\Gamma, x : \sigma, \Delta \vdash x : \sigma} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}$$

Figure 2: The term formation rules of the simply typed lambda calculus in Curry style

latter rather than the former. Pedagogically, this seems to me problematic as it amounts to the admission that we can't model the original concrete language, and it means that semantic considerations have encroached on the syntax and made it less realistic. Another device that appears in the literature is to say that typing contexts are *sets* and to require that $x \notin \mathrm{dom}(\Gamma)$ in the formation rule of lambda abstraction, but this is credible only if one is dealing with α-equivalence classes, as nested occurrences of the same variable may appear after β-reduction even if this was not the case at the outset.

After this digression, let us return to the material that I do present. The semantics of types is the usual one: an arbitrary set A_ι is chosen for the ground type ι and function types are interpreted by the set of all functions. Semantic environments ρ are functions from a set of variables to their respective semantic domains. The semantic function $[\![-]\!]_\rho$ is defined for those terms of the language whose set of free variables is contained in the domain of ρ.

As another aside for the cognoscenti, in the chosen presentation the semantic value of a term M of type σ in a semantic environment ρ is an *element* of the set A_σ. For the completeness proof that follows, and also for the discussion of PCF, this is the most convenient approach. One could call this the *model-theoretic* point of view. If typing contexts are made explicit, then we give semantics to *judgements* $\Gamma \vdash M : \sigma$, not to terms, and the semantic values are always *functions* from $[\![\Gamma]\!]$ to $[\![\sigma]\!] = A_\sigma$; we could call this the *categorical* point of view.

The semantic clause for lambda abstraction reads

$$[\![\lambda x : \sigma.M]\!]\rho \quad = \quad a \in A_\sigma \mapsto [\![M]\!](\rho, x \mapsto a)$$

which more obviously alludes to the mathematical view of a function as a set of ordered pairs than the "semantic lambda" that one sometimes finds. On the other hand, this plays down the (important) view that a denotational semantic function should be a *homomorphism* from the language to the model. This is a subtle point and I wish I were able to express more clearly and convincingly why I find my formulation more appropriate.

After presenting the three semantic clauses one has the first opportunity to mention the principle of *compositionality* as the semantics is pieced together from the semantic values of the parts. It is important to point out, however, that the matter is slightly more subtle than this because the semantics of a lambda abstraction $\lambda x.M$ makes reference to the semantics of M in a *different* semantic environment.

Also, this is a good time to pause and to explain the two views of the semantics that has been defined:

- If our primary interest is in the calculus, then we have given a *model* for the calculus employing sets and mathematical functions. Once we have introduced equations for the calculus, we can then ask whether the model validates the equations (*soundness*) and whether the equations capture equality in the model (*completeness*).
- If the primary interest is in mathematical functions, then we may view the simply-typed lambda calculus as a *language* for these. The natural question then is how *expressive* this language is.

4 Lecture II: Soundness

Presenting the eight rules for deriving equalities between lambda terms is straightforward, except that one has to be able to write very fast on the board. However, not having to deal with typing contexts helps, and even the types of the terms themselves can be suppressed because there is no need to repeat the rules for typing a term. Knowing the type of a term is necessary only in the case of (η).

The rules come in three groups; first the three rules that establish that $\approx$ is an equivalence relation:

$$\frac{}{M \approx M} \qquad \frac{M \approx N}{N \approx M} \qquad \frac{M \approx N \quad N \approx P}{N \approx M}$$

Then the two congruence rules:

$$\frac{M \approx M' \quad N \approx N'}{MN \approx M'N'} \qquad \frac{M \approx M'}{\lambda x.M \approx \lambda x.M'}$$

Finally, the three rules specific to the lambda calculus:

$$\frac{y \notin \mathrm{var}(M)}{\lambda x.M \approx \lambda y.M[y/x]}\,(\alpha) \qquad \frac{}{(\lambda x.M)N \approx M[N/x]}\,(\beta) \qquad \frac{M : \sigma \to \tau \quad x^\sigma \notin \mathrm{var}(M)}{M \approx \lambda x^\sigma.Mx}\,(\eta)$$

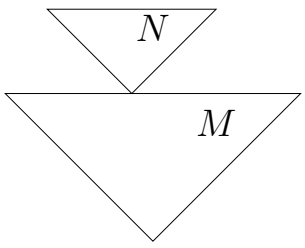
Again, since α is not "built into" my version of the calculus, we have a rule for it; in a first course on semantics, I view this as an advantage.

One can then start to prove the soundness theorem and find that neither (α) nor (β) have an obvious proof because substitution is not a connective of the calculus. This is yet another opportunity to return to the theme of the semantic function as a homomorphism.

I then present a precise definition of substitution for terms (taken from [HS86]) and go through some of the cases of the proof of the substitution lemma. The critical one is of course when a bound variable needs to be renamed, that is, we want to compute $[\![(\lambda y.M)[N/x]]\!]$ in the situation that $y \in \mathrm{FV}(N)$. The substitution rules stipulate that $(\lambda y.M)[N/x]$ be rewritten to $\lambda z.M[z/y][N/x]$ where $z \notin \{x\} \cup \mathrm{FV}(M) \cup \mathrm{FV}(N)$. The computation of the semantics of this takes a number of steps but it is satisfying for the students to see how everything fits together:

$$
\begin{array}{lll}
 & [\![\lambda z.M[z/y][N/x]]\!]\rho & \\
= & a \mapsto [\![M[z/y][N/x]]\!](\rho, z \mapsto a) & \text{(definition of } [\![\lambda \ldots]\!]) \\
= & a \mapsto [\![M[z/y]]\!](\rho, z \mapsto a, x \mapsto [\![N]\!](\rho, z \mapsto a)) & \text{(induction hypothesis, } x \neq z) \\
= & a \mapsto [\![M[z/y]]\!](\rho, z \mapsto a, x \mapsto [\![N]\!]\rho) & \text{(Lemma, } z \notin \mathrm{FV}(N)) \\
= & a \mapsto [\![M]\!](\rho, z \mapsto a, x \mapsto [\![N]\!]\rho, & \\
 & \qquad\qquad y \mapsto [\![z]\!](\rho, z \mapsto a, x \mapsto [\![N]\!]\rho)) & \text{(induction hypothesis)} \\
= & a \mapsto [\![M]\!](\rho, z \mapsto a, x \mapsto [\![N]\!]\rho, y \mapsto a) & \text{(definition of } [\![z]\!]) \\
= & a \mapsto [\![M]\!](\rho, x \mapsto [\![N]\!]\rho, y \mapsto a) & (z \notin \mathrm{FV}(M)) \\
= & [\![\lambda y.M]\!](\rho, x \mapsto [\![N]\!]\rho) & \text{(definition of } [\![\lambda \ldots]\!])
\end{array}
$$

The computation shows that the identity of the fresh variable z is immaterial for the result to hold, which goes some way to address any concerns that students might have over the fact that substitution *on terms* is not a deterministic operation. If there is time, then one can usefully insert a brief discussion of *binding diagrams* (where bound variables are replaced by arrows to the binding lambda) as an alternative "syntax." The substitution lemma is then almost a complete triviality and its proof immediately apparent from the following picture



One can use this visualisation to re-enforce the principle of compositionality.

The soundness proof is then soon completed and there is usually still time left to introduce general Henkin models, the idea being that we may not need all set-theoretic functions for the interpretation of lambda abstractions, although the students have no reason to believe this at this point. One can also come back to the view of the semantic function as a homomorphism, as Henkin models are defined as certain multi-sorted algebras with explicit application operators. (Here I am following [Mit96, Section 4.5].)

5 Lecture III: Completeness

The aim of this lecture is to prove *Friedman's completeness theorem* which states that the full set-theoretic model built over an infinite set (for example $\mathbb{N}$) is complete with respect to $\alpha\beta\eta$-equality. It proceeds in two stages, first a term model construction is used to show that the equational rules are complete with respect to general Henkin models, and then a relation between the term model and the full set-theoretic model establishes completeness with respect to the latter.

In the construction of the term model $\mathcal{T}$ our choice of a Church-style presentation comes in handy: The "material" from which the semantic domains are built are truly the terms of our calculus, that is, T_σ consists of equivalence classes of (open) terms of type σ, and for every σ there are at least the (countably infinitely many) variables of that type to start from. We write the equivalence classes with respect to the basic $\alpha\beta\eta$ theory as $\langle M \rangle$ and define application by

$$\mathrm{app}\ \langle M \rangle\ \langle N \rangle = \langle MN \rangle$$

One now has to show that a Henkin model is thus obtained, that is, we need to show extensionality and "richness" of the function types. For the former, we exploit the fact that we have an unbounded supply of variables, and we use the η-law:

$$\begin{array}{lrcll}
\forall N: & \mathrm{app}\ \langle M \rangle\ \langle N \rangle & = & \mathrm{app}\ \langle M' \rangle\ \langle N \rangle & \text{assumption} \\
& \mathrm{app}\ \langle M \rangle\ \langle x \rangle & = & \mathrm{app}\ \langle M' \rangle\ \langle x \rangle & \text{choose } N = x \text{ fresh} \\
& \langle Mx \rangle & = & \langle M'x \rangle & \text{definition of app} \\
& \langle \lambda x.Mx \rangle & = & \langle \lambda x.M'x \rangle & \text{congruence rule for lambda} \\
& \langle M \rangle & = & \langle M' \rangle & \eta
\end{array}$$

Showing that for every type $\sigma \to \tau$ we have enough elements in $A_{\sigma\to\tau}$ requires us

to show that the semantics in this model amounts to simultaneous substitution:

$$[\![M]\!]\rho = \langle M[r]\rangle$$

where r is a map that picks out an element of $\rho(x)$ for every variable x in the domain of ρ. The proof is by induction on the structure of M. Here is the case of lambda abstraction: Assume type$(x) = \sigma$; we have

$$[\![\lambda x.M]\!]\rho \quad = \quad \langle N\rangle \in T_\sigma \mapsto [\![M]\!](\rho, x \mapsto \langle N\rangle)$$

by definition of $[\![-]\!]$. We need to show that the function on the right hand side behaves the same as the element $\langle(\lambda x.M)[r]\rangle$ to which end we apply it to an element $\langle N\rangle \in T_\sigma$:

$$\begin{array}{lll} & \text{app } \langle(\lambda x.M)[r]\rangle\ \langle N\rangle & \\ = & \text{app } \langle\lambda x'.M[x'/x][r, x' \to x']\rangle\ \langle N\rangle & x' \text{ fresh} \\ = & \langle(\lambda x'.M[x'/x][r, x' \to x'])N\rangle & \text{definition of app in } \mathcal{T} \\ = & \langle M[x'/x][r, x' \to x'][N/x']\rangle & \beta \\ = & \langle M[r, x \to N]\rangle & \end{array}$$

and we see that despite our attempts to simplify things we have plenty of notational fiddliness still to deal with!

Completeness now follows easily by considering the semantic environment that maps every variable x to its $\alpha\beta\eta$ equivalence class $\langle x\rangle$.

Students are unlikely to have seen a completeness proof in full before and one should therefore take the opportunity to point out that the generality of the method. Also, although one has to be careful with the details, it is in my view a very worthwhile and satisfying exercise to see how each and every equation has its role to play in the proof.

For the second stage of the proof of Friedman's completeness theorem we return to the full set-theoretic model $\mathcal{N}$ built over $[\![\iota]\!] = \mathbb{N}$. Friedman's proof uses a "partial surjective homomorphism h" from $\mathcal{N}$ to the term model $\mathcal{T}$ but a more fruitful perspective is to regard the graph of h as a *logical relation*. So at this point I introduce logical relations between two Henkin models and prove the *fundamental lemma*, which doesn't take very long.

If there is sufficient time, one can generalise this to logical relations of arbitrary arity, and discuss the *lambda definability problem* (and Ralph Loader's undecidability result). If there isn't, then this is still a worthwhile topic to explore in the exercises.

$$\begin{array}{rcl} \underline{0}, \underline{1}, \ldots & : & \text{int} \\ \text{succ}, \text{pred} & : & \text{int} \to \text{int} \\ \text{zero?} & : & \text{int} \to \text{bool} \\ \text{if}_\sigma & : & \text{bool} \to \sigma \to \sigma \to \sigma \\ Y_\sigma & : & (\sigma \to \sigma) \to \sigma \end{array}$$

Figure 3: The constants of PCF

In any case, having established the fundamental lemma for logical relations, one may now swiftly finish Friedman's proof of the completeness of $\mathcal{N}$ by showing that a logical relation between $\mathcal{N}$ and $\mathcal{T}$, which happens to be a surjective function at ground type, will be functional and surjective at every type (though it will be a *partial* function).

6 Lecture IV: PCF

From the discussion of expressivity in the previous lecture, it is easy to reiterate the point that the simply typed lambda calculus is far too parsimonious to be considered a language for describing functions (and higher-order functions) on the natural numbers, making the step to PCF easy and natural. All the constants of PCF are readily introduced, see Figure 3. However, I usually defer talking about Y until I have presented the rewrite rules for the other constants. Again, these are easily motivated by the desire to compute normal forms. I use the small-step presentation from Plotkin's original paper [Plo77], where each rule may be seen as a refinement of an equation as discussed previously. In fact, small-step semantics appears so natural that it is worthwhile for the students to get a chance — perhaps in the exercises — to reflect on how one might implement the rewriting process in practice (and to contrast it with the algorithm derived from the big-step presentation). In the lectures we use $M \Downarrow \underline{n}$ purely as a shorthand for a finite sequence of rewrite steps that ends in $\underline{n}$.

As we arrive at the heart of the course — the semantics of recursion — we take time to discuss the constant(s) Y and how they are used to represent recursion. I begin by pointing out that the way we get a recursive program in ordinary programming languages is always by *naming terms*, something that PCF does not provide. For non-recursive terms, naming would be just a convenience — a case of Landin's *syntactic sugar* — but in a recursive term this is not so. A recursive

term declaration such as

$$f = \lambda x.\ \text{if zero? } x \text{ then } \underline{1} \text{ else } x * f(\text{pred } x)$$

is really an equation for the unknown f. It is not a general equation $M = N$, but one where the left-hand side is the unknown itself, that is, it has the shape of a *fixpoint equation*. Making this explicit amounts to introducing a function on the right-hand side:

$$f = (\lambda g.\lambda x.\ \text{if zero? } x \text{ then } 1 \text{ else } x * g(\text{pred } x))\ f$$

If this function is abbreviated to M, then the fixpoint equation has the form $f = Mf$. We may now introduce YM as a *purely formal name* for the solution to this equation, just like the imaginary constant i is a purely formal name for a solution to $x^2 = -1$. Nothing needs to be known about YM other than that it may be replaced by $M(YM)$ wherever it is encountered; again, this is in analogy to the way students learned to manipulate i in high school: Nothing needs to be known about it except that i^2 may always be replaced by -1.

I believe it is very useful for students to see how this works in practice, by reducing a term such as the one above for some concrete value of x, for example, $YM\,\underline{2}$. In the exercises one can then go a bit further and ask the students to implement primitive recursive functions in PCF, and likewise μ-recursion. It is a useful insight for them to see that the former requires terms of rank 2 only, while μ-recursion requires rank 3. This opens up an interesting line of thought regarding the differences between Turing-machine computability (which the students should be familiar with) and higher-order computability.

To continue the motivational journey from recursive definitions to the constant Y, one now needs to realise that the way we treat the "formal terms" YM is captured by a rewrite rule for Y that is no different in structure from those for the other PCF constants. One then sees that the *semantics* of this constant amounts to a functional that returns a fixpoint for every (definable) endofunction. I believe it is important to spend some time on this point and to make it clear that by introducing Y we are *postulating* that every (fixpoint) equation has a solution, something that is certainly not true in mathematics. One can quite usefully contemplate why this should be the case in computer science, and one valid point of view is indeed that it doesn't hold there either.

So now that we have all components of the language, we turn to the semantics. Given the training the students have had with the simply typed lambda calculus, very little time needs to be spent on the semantics of PCF minus Y, in other

words, one can cut straight to the chase and explain the difficulties one has with interpreting Y in the usual full set hierarchy $\mathcal{N}$ where functions do not need to have fixpoints.

This brings us to domain theory, introduced as an abstraction of the idea of interpreting function types as partial rather than total functions. Without going into the detail of this "partial function model," one can introduce the order between them (as containment of the function graphs) and observe that a programmable function should preserve it.

One then shows how partial functions can be replaced by total ones into a "lifted set," that is, a flat domain. But then all sets should be lifted and by looking at the Booleans, one finds that while there are only *nine* partial functions from $\mathbb{B}$ to $\mathbb{B}$, there are *11* monotone and total functions from $\mathbb{B}_\perp$ to $\mathbb{B}_\perp$. This provides an opportunity to talk about *strictness* and CBV vs CBN.

Next we look at $[\mathbb{N}_\perp \to \mathbb{N}_\perp]$, again comparing this to the partial function space $[\mathbb{N} \rightharpoonup \mathbb{N}]$ and see that its order structure is much richer. This is the moment to define abstractly the notion of an *ω-chain-complete partial order*, or *ccpo*, and that of an ω-chain-continuous function.

The four key theorems to prove are (a) that the function space of two ccpos is again a ccpo, (b) that application and abstraction are continuous functions, (c) that every continuous function has a fixpoint, and (d) that the fixpoint map itself is continuous. I am embarrassed to admit that of these I only ever manage to prove (c) in any detail. It is good material for the exercise class, though.

I stress to the students that because of (a) and (b), ccpos and ω-chain-continuous functions are the *third* example of a Henkin model for the simply typed lambda calculus, after the full set hierarchy and the term model.

As an aside, it is a fact that continuity is not necessary for the adequacy proof but it is of course much harder to prove that monotone functions have fixpoints, which would require either transfinite induction or otherwise the clever argument of Pataraia. Both can be considered in the exercises.

7 Lecture V: Adequacy and the Context Lemma

This lecture starts with pointing out that one should not expect an axiomatisation of the equational theory induced by the domain model for PCF, as this would entail solving the halting problem. On the other hand, the equations that we do have, all have been refined to rewrite rules, so we can talk about computation as reduction to normal form (again, it may be helpful to point out that we could have

done the same for the simply typed lambda calculus). Correctness is still an issue but it can be dealt with swiftly. Completeness, on the other hand, is available at ground type only and now takes the rather interesting form of *adequacy*:

Theorem. *If P is a closed term of type int and if $[\![P]\!] = n \in \mathbb{N}$ then P reduces to $\underline{n}$ in finitely many steps.*

In class, the proof is given in full and uses the well-known "logical relation" technique (which, I am told, has indeed a long history). For a closed term M of ground type (let's restrict attention to the integers from now on) and $a \in \mathbb{N}_\perp$, one defines

$$(a, M) \in R_{\text{int}} \text{ iff } a = \perp \text{ or } (a = n \text{ and } M \Downarrow \underline{n})$$

This is extended to higher types as usual:

$$(f, M) \in R_{\sigma\to\tau} \text{ iff } \forall N : \sigma \text{ closed and } a \in A_\sigma.\ (a, N) \in R_\sigma \Longrightarrow (f(a), MN) \in R_\tau$$

We now proceed by the following simple steps:

Lemma. *$\perp\ R\ M$ always.*

Lemma. *If $M' \to M$ and $a\ R\ M$ then $a\ R\ M'$.*

Lemma. *If $a_i\ R\ M$ for the elements of a chain $a_0 \sqsubseteq a_1 \sqsubseteq \ldots$ then $\bigsqcup a_i\ R\ M$.*

It is in the proof of the last lemma where we notice that we only need that sups of functions are computed pointwise but not that functions are continuous. In other words, the proof would go through if we built our model with functions which are only required to be monotone, an observation which I learned from Alley Stoughton.

The three lemmas allow us to show the following, which has the same shape as the fundamental lemma for logical relations:

Lemma. *If M is closed, then $[\![M]\!]\ R\ M$.*

The proof is by induction over the structure of M (so we need to extend R to open terms) whereas the others are by induction over the type, the key steps being lambda abstraction and the fixpoint combinator. Adequacy then follows immediately from this and the definition of R.

The final highlight of the course is a demonstration that the semantics can be invoked to show *Milner's context lemma*. So we introduce *contexts* as "terms with holes" and the contextual preorder $\lesssim$. The key to the argument is to employ the relation R from the adequacy proof:

Lemma. *Let M, M' be closed and of type $\sigma = \sigma_1 \to \sigma_2 \to \ldots \to \sigma_k \to \mathrm{int}$. Then the following are equivalent:*

1. $M \lesssim M'$
2. $\forall P$ *closed*. $PM \Downarrow \underline{n} \Longrightarrow PM' \Downarrow \underline{n}$
3. $\forall N_1, \ldots, N_k$ *closed*.$MN_1 \ldots N_k \Downarrow \underline{n} \Longrightarrow M'N_1 \ldots N_k \Downarrow \underline{n}$
4. $[\![M]\!]\ R\ M'$

The proof is so elegant that it is worthwhile to spell it out in full:

Proof. $(1) \Leftrightarrow (2)$ It is possible to treat the "hole" in a context as if it were a variable because M and M' are assumed to be closed.

(3) is a special case of (2), consider $P = \lambda x.xN_1 \ldots N_k$.

$(3) \Rightarrow (4)$ Use the definition of the logical relation: Let $a_i\ R\ N_i$ for $i \in \{1, \ldots, k\}$ and consider $M'N_1 \ldots N_k$ and $[\![M]\!](a_1) \ldots (a_k)$. We want to show that $[\![M]\!](a_1) \ldots (a_k)$ is in relation to $M'N_1 \ldots N_k$. We know that $[\![M]\!]\ R\ M$ and hence $[\![M]\!](a_1) \ldots (a_k)\ R\ MN_1 \ldots N_k$. There are two cases:

- If $[\![M]\!](a_1) \ldots (a_k) = \bot$ then $[\![M]\!](a_1) \ldots (a_k)\ R\ M'N_1 \ldots N_k$ by definition.
- If $[\![M]\!](a_1) \ldots (a_k) = n$ then again by the definition of R: $MN_1 \ldots N_k \Downarrow \underline{n}$, and hence $M'N_1 \ldots N_k \Downarrow \underline{n}$ by assumption. In this case, too, $[\![M]\!](a_1) \ldots (a_k)\ R\ M'N_1 \ldots N_k$ follows.

$(4) \Rightarrow (2)$ Let P be closed. We have $[\![P]\!]\ R\ P$, hence $[\![PM]\!] = [\![P]\!]([\![M]\!])\ R\ PM'$ by assumption. Now if $PM \Downarrow \underline{n}$ then by correctness $[\![PM]\!] = n$. By the definition of R, $PM' \Downarrow \underline{n}$ follows. □

The course ends by pointing the students to the classic texts by Scott [Sco93] and Plotkin [Plo77], as well as to the textbooks mentioned above, and especially [Str06] for a continuation of the story begun in this course.

8 Concluding remarks

If you have taught denotational semantics yourself, you will have made different choices and you will have had different experiences. I'd love to hear from you! If you agree or disagree with the particular choices I have presented here, then I'd

love to hear from you, too! Perhaps you have suggestions for making the subject more interesting or more tractable still; that would also be welcome.

I should like to end by expressing my gratitude to the many colleagues and students (especially here at Birmingham) with whom I have discussed the contents of my course over the years, and I ask for forgiveness from those from whom I have taken an idea without giving proper credit. Most of all, I am indebted to Dana Scott and Gordon Plotkin for laying the foundation for this beautiful subject.

References

[Ber90] U. Berger. *Totale Objekte und Mengen in der Bereichstheorie*. PhD thesis, Ludwig-Maximilians-Universität München, 1990.

[Gun92] C. Gunter. *Semantics of Programming Languages. Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[HS86] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.

[Mit96] J.C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[NN91] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction for Computer Science*. Wiley, 1991.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Sch86] D. A. Schmidt. *Denotational Semantics*. Allyn and Bacon, 1986.

[Sco93] D. S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 1993. Reprint of a manuscript written in 1969.

[Sim98] A. Simpson. Lazy functional algorithms for exact real functionals. In *Mathematical Foundations of Computer Science 1998*, volume 1450 of *Lecture Notes in Computer Science*, pages 456–464. Springer Verlag, 1998.

[Str06] Th. Streicher. *Domain-Theoretic Foundations of Functional Programming*. World Scientific, 2006. 132pp.

[Ten91] R. D. Tennent. *Semantics of Programming Languages*. Prentice Hall, 1991.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages. An Introduction*. MIT Press, 1993.