Analysing PCF and Kleene computability via sequential procedures

John Longley
Laboratory for Foundations of Computer Science
University of Edinburgh

Theory Seminar, University of Birmingham
6 July 2012

Introduction: The language PCF

PCF is a prototypical higher order sequential and purely functional programming language (Plotkin 1977). Basically a toy version of Haskell (without monads).

We'll take PCF to be a simply typed lambda calculus over a single ground type $\mathtt{N}$, with constants

$$\begin{array}{rcll} \widehat{n} & : & \mathtt{N} & \text{for each } n \in \mathbb{N} \\ suc,\ pre & : & \mathtt{N} \to \mathtt{N} & \\ ifzero & : & \mathtt{N} \to \mathtt{N} \to \mathtt{N} \to \mathtt{N} & \\ Y_\sigma & : & (\sigma \to \sigma) \to \sigma & \text{for each } \sigma \end{array}$$

and suitable (call-by-name) reduction rules.

Pure types: we'll write $\overline{0} = \mathtt{N}$, $\overline{k+1} = \overline{k} \to \mathtt{N}$.

‘Algorithms’ for PCF programs

Consider the following two (sugared) PCF programs for computing the minimization operator $f^{\overline{1}} \mapsto \mu n.\, f(n) = 0$.

$$
\begin{array}{lcl}
M_0\ f & = & \mathit{if}\ \ f\,\widehat{0} = \widehat{0}\ \ \mathit{then}\ \ \widehat{0}\ \ \mathit{else}\ \ \mathit{suc}(M_0(\lambda n.f(\mathit{suc}\,n))) \\
M_1\ f & = & \mathit{if}\ \ f\,\widehat{0} = \widehat{0}\ \ \mathit{then}\ \ \widehat{0}\ \ \mathit{else} \\
 & & \mathit{if}\ \ f\,\widehat{1} = \widehat{0}\ \ \mathit{then}\ \ \widehat{1}\ \ \mathit{else}\ \ \mathit{suc}(\mathit{suc}(M_1(\lambda n.f(\mathit{suc}(\mathit{suc}\,n)))))
\end{array}
$$

In terms of their interactions with their argument f, these embody the same ‘algorithm’, which we can display as an infinite decision tree:

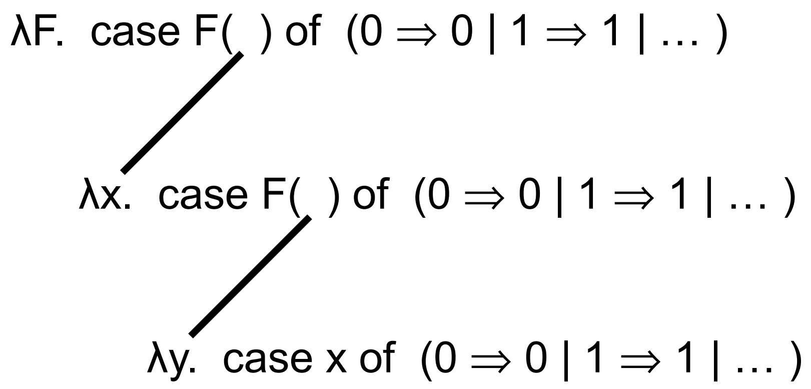
$$
\begin{array}{lll}
\lambda f.\ \mathtt{case}\ f(0)\ \mathtt{of} & (0 \Rightarrow 0 & \\
 & \mid 1 \Rightarrow \mathtt{case}\ f(1)\ \mathtt{of} & (0 \Rightarrow 0 \\
 & & \mid 1 \Rightarrow \mathtt{case}\ f(2)\ \mathtt{of}\ \ \cdots \\
 & & \mid \cdots) \\
 & \mid \cdots) &
\end{array}
$$

Nested calls

We can loosely think of bound variables like f as ‘oracles’ received from the outside world.

In general, in a higher order setting, the arguments we feed to these oracles will themselves be ‘algorithms’ of a similar kind.

Example: Kierstead functional $\lambda F^{\overline{2}}.\, F(\lambda x.\, F(\lambda y.x))$.


λF. case F( ) of (0 ⇒ 0 | 1 ⇒ 1 | …)
λx. case F( ) of (0 ⇒ 0 | 1 ⇒ 1 | …)
λy. case x of (0 ⇒ 0 | 1 ⇒ 1 | …)

Nested sequential procedures

We'll refer to trees of the above kind as (nested) sequential procedures (NSPs). They're typically infinitely broad (with $\mathbb{N}$-indexed conditional branching), and can be infinitely deep.

The minimization example illustrates one kind of infinite depth.

Another (more serious) kind is illustrated by the tree for $Y_{\overline{1}}$. This is $\lambda F.\mathcal{T}$, where '$\mathcal{T} = \lambda x.\, F\mathcal{T}x$'. More formally:

$$\mathcal{T} \;=\; \lambda x.\, \mathsf{case}\; F\,\mathcal{T}\,(\mathsf{case}\; x\; \mathsf{of}\; (j \Rightarrow j))\; \mathsf{of}\; (i \Rightarrow i)$$

These trees are what we shall study, in relation both to PCF and to Kleene (S1–S9) computability (1959) for total functionals. [Idea: NSPs capture what is common to these two notions.]

History of ideas

- Sazonov (1970s) used Turing machines with oracles to identify sequentially computable elements of Scott model.
- Ideas also implicit in late work of Kleene and Gandy.
- Explicitly defined in papers on game semantics of PCF (1993), where they played an ancillary role. (AJM: evaluation trees. HO: canonical forms.)
- Amadio and Curien (1998) study NSPs as PCF Böhm trees, showing they form a model of PCF in their own right.
- Sazonov (2007) used same ideas to give a standalone characterization of the sequentially computable *functionals*.
- Normann and Sazonov (2012) considered them under the name sequential procedures. They used this model to obtain order-theoretic properties of the sequential functionals.

Aside: NSPs and game models (Take 1)

In game models, interactions between programs and arguments are captured via a dialogue between Player and Opponent. A program's behaviour is modelled by a strategy for Player.

Both NSPs and game strategies are definitely *intensional*: they each allow many representations of the same (extensional) function. (E.g. multiple NSPs/strategies for $+ : \mathtt{N} \rightarrow \mathtt{N} \rightarrow \mathtt{N}$.)

In fact, there's a perfect bijection between NSPs and e.g. the innocent strategies of Hyland-Ong. However:

- A game strategy can *prima facie* only be applied to another strategy — not to a function considered as 'pure oracle'. So less well suited e.g. to Kleene computability.
- On the other hand, game models of PCF fit into a broader semantic framework (Abramsky et al). NSPs don't seem to.

Present contributions

1. We give a semi-novel construction of the NSP model offering various advantages. In particular, we show that much of the theory of NSPs works in a completely untyped setting.

2. We identify substructures of the NSP model corresponding to interesting ‘sub-PCF’ notions of computability. We use these to give semantic proofs of some new expressivity results. (E.g. bar recursion is not definable in System T+min.)

3. Finally, we look briefly at the light shed by NSPs on Kleene computability, e.g. in the full set-theoretic (total) model.

PART 1. Untyped and typed NSPs

Untyped NSPs are defined *coinductively* as follows:

$$\begin{array}{lrcl} \textit{Procedures:} & p, q & ::= & \lambda x_0 x_1 \cdots . \, e \\ \textit{Expressions:} & e & ::= & \bot \;\mid\; n \;\mid\; \texttt{case}\ a\ \texttt{of}\ (0 \Rightarrow e_0 \mid 1 \Rightarrow e_1 \mid \cdots) \\ \textit{Applications:} & a & ::= & x\, q_0 q_1 \cdots \end{array}$$

Every λ binds an infinite sequence of variables, and every application involves an infinite sequence of arguments.

For typed NSPs: use typed variables, consider *finite* abstractions/applications, and impose the obvious typing constraints. Alternatively, can retrieve the typed model from the untyped one via its Karoubi envelope.

The typed/untyped dichotomy is pretty much orthogonal to everything else involved. Here we'll follow the untyped route, to see how much can be done in a type-free setting.

Defining application for NSPs

In Amadio/Curien and Normann/Sazonov, application is defined first for *finite* NSP, then extended to general ones via continuity. However, even showing that finite NSPs are closed under application is surprisingly non-trivial, and the existing proofs require at least Π^0_2 induction.
By contrast, our treatment . . .

- requires no non-trivial theorems for the mere construction of the model;
- requires only 'finitistic' reasoning to obtain key properties;
- gives a way of computing directly with NSPs. This makes it clear that NSP application is itself 'sequentially computable' (e.g. it's realizable over van Oosten's B).

The computation calculus

We shall explain how a procedure p 'interacts with' a sequence of arguments $q_0 q_1 \cdots$ to yield a ground type result. To represent the 'intermediate forms', we expand our language to a calculus of *meta-terms*. The choice of both this extended language and its reduction system are somewhat delicate.

$$\begin{array}{rrcl}
\textit{Meta-procedures:} & P, Q & ::= & \lambda x_0 x_1 \cdots . E \\
\textit{Meta-expressions:} & E & ::= & \bot \mid n \mid \mathsf{case}\ G\ \mathsf{of}\ (0 \Rightarrow E_0 \mid 1 \Rightarrow E_1 \mid \cdots) \\
\textit{Ground meta-terms:} & G & ::= & E \mid x\, Q_0 Q_1 \cdots \mid P Q_0 Q_1 \cdots
\end{array}$$

Top-level reductions:

$$\begin{array}{rcl}
(\lambda x_0 x_1 \cdots . E) Q_0 Q_1 \ldots & \rightsquigarrow & E[\overline{x} \mapsto \overline{Q}] \\
\mathsf{case}\ \bot\ \mathsf{of}\ (i \Rightarrow E_i) & \rightsquigarrow & \bot \\
\mathsf{case}\ n\ \mathsf{of}\ (i \Rightarrow E_i) & \rightsquigarrow & E_n \\
\mathsf{case}\ (\mathsf{case}\ G\ \mathsf{of}\ (i \Rightarrow E_i))\ \mathsf{of}\ (j \Rightarrow F_j) & \rightsquigarrow & \mathsf{case}\ G\ \mathsf{of}\ (i \Rightarrow \mathsf{case}\ E_i\ \mathsf{of}\ (j \Rightarrow F_j))
\end{array}$$

The computation calculus (continued)

The foregoing reductions may be applied in ‘head position’, as specified by:

- If $G \rightsquigarrow G'$ and G is not a case meta-term, then
$$\texttt{case } G \text{ of } (i \Rightarrow E_i) \;\rightsquigarrow\; \texttt{case } G' \text{ of } (i \Rightarrow E_i)$$
- If $E \rightsquigarrow E'$ then $\lambda\overline{x}.E \;\rightsquigarrow\; \lambda\overline{x}.E'$.

General reduction proceeds by reducing a meta-term T to ‘head normal form’, and then recursively reducing its subterms, Böhm tree style. (Formal details omitted.) Since the calculus is infinitary, we then define the ‘value’ of T by

$$\ll T \gg \;=\; \bigsqcup \{t \text{ finite } \mid \exists T'.\, T \rightsquigarrow^* T' \;\wedge\; t \sqsubseteq T'\}$$

Finally, if $p = \lambda x_0 x_1 x_2 \cdots .e$, we define

$$p \cdot q \;=\; \lambda x_1 x_2 \cdots . \ll e[x_0 \mapsto q] \gg$$

What this gives us

The set of closed untyped NSPs under $\cdot$ forms a $\lambda\eta$-algebra USP^0. (Requires real work, but entirely 'elementary'.) Our construction also yields a simulation (i.e. applicative morphism) $\mathsf{USP}^0 \longrightarrow\!\!\triangleright\, \mathsf{B}$.

By standard results, the Karoubi envelope of USP^0 is a cartesian closed category. Moreover, the idempotent

$$\lambda x y_0 y_1 \cdots .\, x \bot\bot \cdots$$

gives an object playing the role of $N_\bot$, so we obtain a simply typed model SP^0. This coincides exactly with the typed NSP model given by direct definition. So the explicit typing constraints fall out naturally from a general abstract construction.

Furthermore, the untyped route lets us glimpse a wider category that SP^0 naturally sits inside.

What else this gives us

As is already known, SP^0 provides an adequate model of PCF^Ω (i.e. PCF with an 'oracle constant' C_f for each $f : \mathbb{N} \rightharpoonup \mathbb{N}$).

Furthermore, for each σ, there's a PCF program $I_\sigma : \overline{1} \rightarrow \sigma$ such that for every $p \in \mathsf{SP}^0(\sigma)$ we have $[\![I_\sigma]\!] \cdot \lceil p \rceil = p$, where $\lceil p \rceil \in \mathsf{SP}^0(\overline{1})$ is the 'type 1 code' for p. Thus, every $p \in \mathsf{SP}^0(\sigma)$ is definable on the nose by a term of PCF^Ω.

Likewise for effective procedures and PCF, modulo a minor and idiosyncratic deficiency of standard (call-by-name) PCF.

Various standard results about PCF (e.g. Context Lemma; absence of parallel operations) can now be treated slightly more abstractly in terms of NSPs.

Finally, most other reasonable interpretations of PCF (e.g. the one in the Scott model) factor readily through SP^0.

The extensional quotient

As with game models, we can pass from SP^0 to the model SF of sequential functionals by taking an 'observational quotient' at each type σ.

Alternatively, we can take the quotient of USP^0 by a single 'applicative equivalence' relation:

$$p \ \sim_{app} \ p' \quad \text{iff} \quad \forall q_0, q_1, \ldots \in \mathsf{USP}^0. \ll p q_0 q_1 \cdots \gg = \ll p' q_0 q_1 \cdots \gg$$

This yields an extensional λ-model, and from its Karoubi envelope we can retrieve SF.

So this part of the story, at least, can be told in a 'type-free' way. Of course, that doesn't alter the fact that the equivalence relation in question is intractable (Loader's theorem)!

PART 2: Substructures. Finite sequential procedures

‘Finite NSPs’ may be construed as NSPs in our sense by ‘adding $\perp$ everywhere else’. A known but non-trivial fact is that the finite sequential procedures in SP^0 are closed under application. (Our development so far has avoided relying on this fact.) The ‘natural’ proof is by induction on types and uses Π^0_2 induction.

A symptom of the non-triviality is that the size of $p \cdot q$ is in general super-elementary in the sizes of p, q. (Closely analogous to normalization in pure simply typed lambda calculus.)

Surprisingly, perhaps, the same results also hold in the untyped setting, though they require a little more work there.

This contrasts starkly with the situation for lambda calculus: e.g. $\lambda x.f(xx)$ applied to itself generates an infinite Böhm tree. The secret is that any untyped finite procedure can be annotated with indices that play a role similar to ‘type levels’.

Well-founded sequential procedures

Finite NSPs don't form a model by themselves, since k, s are infinite.

However, the above results carry over easily to well-founded NSPs, i.e. those generated by our original grammar construed *inductively*.

In SP^0, well-founded procedures are closed under application and include k, s. Indeed, they form a model for Gödel's System T and other 'total type theories' in a similar spirit.

Even in the untyped setting, the well-founded procedures are closed under application (though they don't include k, s). So again, the basic phenomenon isn't dependent on the presence of typing. (Proof here uses transfinite 'level annotations'.)

Left-well-founded sequential procedures

Now things get interesting.

Say a procedure p is left-well-founded (LWF) if the tree of application subterms $xq_0q_1\cdots$ (ordered by syntactic inclusion) is well-founded. (LWF is weaker than WF.)

Again, the LWF procedures are closed under application, both in SP^0 and USP^0. In SP^0, they form a model e.g. for System $\mathsf{T}+min$, and embody a seemingly natural notion of what can be computed without 'nesting to infinite depth'.

This structural condition on NSPs can be used to give semantic proofs of some new results (cf. Berger, *Minimization vs. recursion on the partial continuous functionals*, 2001). . . .

Application 1: Bar recursion

Notation: If $x = \langle x_0, \ldots, x_{r-1} \rangle \in \mathbb{N}$, write $x.z$ for $\langle x_0, \ldots, x_{r-1}, z \rangle$, and take $\phi_{x,t}(i) = x_i$ for $i < r$, $\quad t$ for $i \geq r$.

Say $B : \overline{2}, \overline{1}, \overline{2} \rightarrow \overline{1}$ is a bar recursor if for all total F, L, G we have

$$\begin{array}{lcll} B(F,L,G)(x) & = & L(x) & \text{if } F(\phi_{x,0}) = F(\phi_{x,1}) \\ B(F,L,G)(x) & = & G(\lambda z.\, B(F,L,G)(x.z)) & \text{otherwise.} \end{array}$$

Here F specifies a well-founded tree, L defines the behaviour at leaf nodes, and G defines the behaviour at branch nodes.

These clauses fairly directly yield a definition of a bar recursor in PCF. However, the corresponding NSP is clearly not LWF.

In fact, we can prove that in SP^0, *no* bar recursor can be LWF (even under a very conservative notion of 'total' element). So no bar recursor is definable in T $+$ min.

Sketch of proof

We show bar recursion isn't LWF-computable even if the leaf function L is specialized to $x \mapsto 2x+1$, and the node x to $\langle\rangle$.

Suppose $B, B' \in \mathsf{SP}^0(\overline{2},\overline{2} \to \overline{0})$ where B is a (specialized) bar recursor and B' is an LWF procedure. We cook up $F, G \in \mathsf{SP}^0(\overline{2})$ such that $B \cdot F \cdot G \neq B' \cdot F \cdot G$.

Essentially, we choose F sufficiently 'deep' that computing $B \cdot F \cdot G$ requires deeper nesting than B' is willing to perform.

Problem: Which part of B' does F need to go deeper than? This itself seems to depend on knowing F and G!

Idea: start with $G_0 = \lambda g.2g(0)$. Build a suitable F by a process of successive approximation in tandem with our analysis of the computation of $B' \cdot F \cdot G_0$. Now choose G_1 sufficiently 'close to' G_0 that $B' \cdot F \cdot G_1 = B' \cdot F \cdot G_0$, but carefully designed so that $B \cdot F \cdot G_1$ is something else entirely.

Some related history (encroaching on PART 3)

Let Ct denote the total continuous functionals (Kleene-Kreisel). It was an open problem in the 1970s (posed by Kreisel) whether any elements of Ct were Kleene computable (via S1–S9) but not μ-computable (via S1–S8 + S10).

This was answered positively by Bergstra (1976) by means of an *ad hoc* example involving degree theory.

At the same time, however, the bar recursor $BR \in \mathrm{Ct}(\overline{2}, \overline{1}, \overline{2} \rightarrow \overline{1})$ (Scarpellini 1971) was well known as a natural example of a total continuous (and Kleene computable) functional!

It follows easily from our theorem that BR is not μ-computable.

Application 2: Sublanguages of $\mathsf{T} + min$

In System T, we have a 'primitive recursor' for each type σ:

$$rec_\sigma \;:\; \sigma \to (\sigma \to \mathtt{N} \to \sigma) \to (\mathtt{N} \to \sigma)$$

By restricting rec_σ to types σ of level $\leq k$, get sublanguages T_k.

It's well known that $\mathsf{T}_0, \mathsf{T}_1, \ldots$ form a strict hierarchy. Berger (2001) asks: is the same true for the languages $\mathsf{T}_k + min$?

Using NSPs, we can prove $\mathsf{T}_0 + min \prec \mathsf{T}_1 + min$. [Idea: any $F \in \mathsf{SP}^0(\overline{2})$ definable in $\mathsf{T}_0 + min$ has call tree of height $< \omega^\omega$.]

Related fact: there's no universal program $U : \overline{0} \to \overline{2}$ in $\mathsf{T}_0 + min$.

I expect all this to extend easily to arbitrary k.

An open problem?

The following very natural question about PCF was posed in (Berger 2001). To my knowledge, it is still open.

Let PCF_k *be the sublanguage of* PCF *with operators* Y_σ *only for* σ *of level* $\leq k$*. Do the* PCF_k *form a strict hierarchy as regards expressivity?*

At first sight, the NSP model would seem ideally suited to addressing this: we just need some structural property of PCF_k-definable procedures not shared by $Y_{\overline{k+1}}$. But I haven't succeeded in finding such a property.

I now suspect this may be a hard problem . . . perhaps the most obvious outstanding question about PCF?

Aside: NSPs and game models (Take 2)

Before we leave PCF, some tentative remarks NSPs in relation to game models. Do these approaches offer different things?

In Hyland-Ong parlance, an innocent function on Player views is very close to a de Bruijn representation of an NSP.

OQ	$\sim$	occurrence of λ
PQ	$\sim$	occurrence of head variable
OA	$\sim$	case branch numeral
PA	$\sim$	leaf numeral
justification sequence	$\sim$	upward path in tree

However, the definitions of application look quite different, and the equivalence isn't quite trivial. Intuitively, in game models everything is more 'local'.

Probably all our proofs could be translated into a game framework, though at the cost of some extra clutter. Conversely, there's a result from games that I can't prove with NSPs: *The type structure* SF *has no universal type.*

PART 3 proper: Kleene S1–S9 computability

Let A be a (suitable) total type structure over $\mathbb{N}$ (e.g. the full set-theoretic one S, or Ct.) Kleene's schemas S1–S9 give a monster inductive definition of a relation '$\{e\}(\vec{x}) = n$', where $e, n \in \mathbb{N}$ and the x_i are elements of A.

If '$\{e\}(\vec{x})$' is defined for all $\vec{x}$ of appropriate types, say e is the index of a total Kleene-computable function $\{e\} \in A(\sigma)$. Whether this is so may depend on A.

However, if e defines a total function of type σ in S, it does so in any other (suitable) A, by an easy logical relation argument.

If σ has level ≥ 2, the set of total indices of type σ (w.r.t. S) is Π^1_1-complete (essentially Moldestad and Normann).

Kleene computability and NSPs

Kleene indices e are very ‘syntactic’. What underlying mathematical objects do they represent?

For any σ and any index $e \in \mathbb{N}$, it’s possible to ‘grow’ a procedure $[e]_\sigma \in \mathsf{SP}^0(\sigma)$, independently of A. The interpretation of Kleene indices in *any* suitable A then factors through a (partial) interpretation of SP^0 in A.

So we get a finitely branching tree for e itself. (Kleene considered only the infinitary computation trees arising from apply e to given elements $x_i \in A$.

We can now investigate the character of these trees . . .

Kleene computability and well-foundedness

On the strength of the last two slides, one might conjecture:

Every Kleene computable functional in S *is definable by a well-founded NSP.*

However, there is a counterexample:

$$\lambda F^{\overline{2}}.\ \mathit{if}\ \ F(\lambda n.n) = F(\lambda n.F(\lambda x.n))\ \ \mathit{then}\ \mathsf{0}\ \ \mathit{else}\ \ \mathit{min}\ n.\ F(\lambda x.n) \neq n$$

I believe there are even total Kleene computable functionals in S not definable by any *left-well-founded* NSP.

Conceptually, this sheds some light on the nature of Kleene computability: ‘demand-driven’, not ‘supply-driven’. However, the WF- and LWF- submodels seem to represent natural ‘sub-Kleene’ computability notions. More to explore here!

Conclusions

The NSP model may seem rather ‘syntactic’ for some tastes.

But it does what we want denotational models to do:

- Raises the level of abstraction of proofs, by comparison with reasoning based directly on operational semantics.
- Provides mathematical structure which gives us new leverage on properties of programs.

Though closely related to game models, NSPs appear to offer their own distinctive slant on PCF.

Furthermore, NSPs convincingly capture the notion of ‘algorithm’ common to both PCF and Kleene computability.

I expect more applications to follow!