Complexity in Higher Types

Thomas Powell

University of Innsbruck

Logic, Complexity and Automation

part of Computational Logic in the Alps 2016
Obergurgl, Austria
5 September 2016

Background

What I normally research:

- Proof theory
- Strong classes of higher-order recursive functionals
- Computational interpretations of subsystems of mathematics

Background

What I normally research:

- Proof theory
- Strong classes of higher-order recursive functionals
- Computational interpretations of subsystems of mathematics

But today my talk will be a bit more practical:

1. What is the complexity of a higher-order functional program?

2. Some ideas on a general monadic denotational semantics.

3. Stuff for the future...

Warning: This is all very informal!

Throughout the talk we will work over a simple call-by-value functional language. However, the main ideas could be adapted to other settings.

Throughout the talk we will work over a simple call-by-value functional language. However, the main ideas could be adapted to other settings.

Let $e : \texttt{nat}$ be some closed expression such that $e \to^* \underline{n}$.

Normally we interpret e as the natural number represented by the numeral $\underline{n}$ i.e.

$$[\![e]\!] = n.$$

Throughout the talk we will work over a simple call-by-value functional language. However, the main ideas could be adapted to other settings.

Let $e : \texttt{nat}$ be some closed expression such that $e \to^* \underline{n}$.

Normally we interpret e as the natural number represented by the numeral $\underline{n}$ i.e.

$$[\![e]\!] = n.$$

But what if we also want information on the *cost* of evaluating e? Suppose that $e \to^k \underline{n}$.

Then we could interpret e as a pair, corresponding to a cost and a value i.e.

$$[e] = (k, n).$$

Now suppose that $t : \texttt{nat} \to \texttt{nat}$ is a closed expression and $t \to^* \lambda x.s(x)$.

Normally we interpret t as a function $f : [\![\texttt{nat}]\!] \to [\![\texttt{nat}]\!]$ such that if $e \to^* \underline{n}$ and $s(\underline{n}) \to^* \underline{m}$ then $f(n) = m$ i.e.

$$[\![te]\!] = [\![t]\!][\![e]\!] = f(n) = m.$$

Now suppose that $t : \texttt{nat} \to \texttt{nat}$ is a closed expression and $t \to^* \lambda x.s(x)$.

Normally we interpret t as a function $f : [\![\texttt{nat}]\!] \to [\![\texttt{nat}]\!]$ such that if $e \to^* \underline{n}$ and $s(\underline{n}) \to^* \underline{m}$ then $f(n) = m$ i.e.

$$[\![te]\!] = [\![t]\!][\![e]\!] = f(n) = m.$$

But what about the complexity of t? Suppose that $t \to^l \lambda x.s(x)$. Then we could define $[t] = (l, f)$.

Now suppose that $t : \texttt{nat} \to \texttt{nat}$ is a closed expression and $t \to^* \lambda x.s(x)$.

Normally we interpret t as a function $f : [\![\texttt{nat}]\!] \to [\![\texttt{nat}]\!]$ such that if $e \to^* \underline{n}$ and $s(\underline{n}) \to^* \underline{m}$ then $f(n) = m$ i.e.

$$[\![te]\!] = [\![t]\!][\![e]\!] = f(n) = m.$$

But what about the complexity of t? Suppose that $t \to^l \lambda x.s(x)$. Then we could define $[t] = (l, f)$.

But we also want information about the complexity of s. Suppose that $s(\underline{n}) \to^{c(n)} \underline{m}$. Then we define

$$[t] = (l, \underbrace{\lambda n.(1 + c(n), f(n))}_{\text{`size'}})$$

Now suppose that $t : \texttt{nat} \to \texttt{nat}$ is a closed expression and $t \to^* \lambda x.s(x)$.

Normally we interpret t as a function $f : [\![\texttt{nat}]\!] \to [\![\texttt{nat}]\!]$ such that if $e \to^* \underline{n}$ and $s(\underline{n}) \to^* \underline{m}$ then $f(n) = m$ i.e.

$$[\![te]\!] = [\![t]\!][\![e]\!] = f(n) = m.$$

But what about the complexity of t? Suppose that $t \to^l \lambda x.s(x)$. Then we could define $[t] = (l, f)$.

But we also want information about the complexity of s. Suppose that $s(\underline{n}) \to^{c(n)} \underline{m}$. Then we define

$$[t] = (l, \underbrace{\lambda n.(1 + c(n), f(n))}_{\text{`size'}})$$

In particular, this definition is *compositional* i.e. we can compute $[te]$ from $[t]$ and $[e] = (k, n)$:

$$[te] = [t] \star [e] = [t] \star (k, n) = (k + l + 1 + c(n), f(n)) = (k + l + 1 + c(n), m).$$

What is the complexity of a higher-order functional? Let's work with a concrete example $\mathtt{map} : (\mathtt{nat} \to \mathtt{nat}) \times \mathtt{nat}^* \to \mathtt{nat}^*$ defined by

$$\mathtt{map}(h, []) \to [] \qquad \mathtt{map}(h, x :: a) \to h(x) :: \mathtt{map}(h, a)$$

What is the complexity of a higher-order functional? Let's work with a concrete example $\mathtt{map} : (\mathtt{nat} \to \mathtt{nat}) \times \mathtt{nat}^* \to \mathtt{nat}^*$ defined by

$$\mathtt{map}(h, []) \to [] \qquad \mathtt{map}(h, x :: a) \to h(x) :: \mathtt{map}(h, a)$$

The term map is already in normal form, so $[\mathtt{map}] = (0, \_)$. What is the 'size' of map?

What is the complexity of a higher-order functional? Let's work with a concrete example $\mathtt{map} : (\mathtt{nat} \to \mathtt{nat}) \times \mathtt{nat}^* \to \mathtt{nat}^*$ defined by

$$\mathtt{map}(h, []) \to [] \qquad \mathtt{map}(h, x :: a) \to h(x) :: \mathtt{map}(h, a)$$

The term map is already in normal form, so $[\mathtt{map}] = (0, \_)$. What is the 'size' of map?

Suppose map takes as arguments a value $v : \mathtt{nat} \to \mathtt{nat}$ of size (c, f) and a list of numerals $[\underline{a}_1, \ldots, \underline{a}_j]$. Then

$$\mathtt{map}(v, [\underline{a}_1, \ldots, \underline{a}_j]) \to^{1+j+\sum_{i \leq j} c(a_i)} [f(a_1), \ldots f(a_j)].$$

What is the complexity of a higher-order functional? Let's work with a concrete example $\texttt{map} : (\texttt{nat} \to \texttt{nat}) \times \texttt{nat}^* \to \texttt{nat}^*$ defined by

$$\texttt{map}(h, []) \to [] \qquad \texttt{map}(h, x :: a) \to h(x) :: \texttt{map}(h, a)$$

The term map is already in normal form, so $[\texttt{map}] = (0, \_)$. What is the 'size' of map?

Suppose map takes as arguments a value $v : \texttt{nat} \to \texttt{nat}$ of size (c, f) and a list of numerals $[\underline{a}_1, \ldots, \underline{a}_j]$. Then

$$\texttt{map}(v, [\underline{a}_1, \ldots, \underline{a}_j]) \to^{1+j+\sum_{i \leq j} c(a_i)} [f(a_1), \ldots f(a_j)].$$

So we could define

$$[\texttt{map}] = (0, \lambda(c, f), \underline{a}.(1 + |\underline{a}| + \sum c(a_i), [f(a_1), \ldots, f(a_n)]))$$

and we would have $[\texttt{map}(t, e)] = [\texttt{map}] \star ([t], [e])$.

Underlying all this is the notion of a *monadic translation*. Define $[-]$ on types as

$$[D] := C \times \underbrace{[\![D]\!]}_{s(D)}$$

$$[X \to Y] := C \times (\underbrace{s(X) \to [Y]}_{s(X \to Y)})$$

For all types we have $[X] = C \times s(X)$, the idea being that the C is some structure which contains intensional information about objects $t : X$, while $s(X)$ represents a 'size' or *potential* (at ground types the usual denotation).

Underlying all this is the notion of a *monadic translation*. Define $[-]$ on types as

$$[D] := C \times \underbrace{[\![D]\!]}_{s(D)}$$

$$[X \to Y] := C \times (\underbrace{s(X) \to [Y]}_{s(X \to Y)})$$

For all types we have $[X] = C \times s(X)$, the idea being that the C is some structure which contains intensional information about objects $t : X$, while $s(X)$ represents a 'size' or *potential* (at ground types the usual denotation).

- In a traditional denotational semantics, we would have (at base types):

$$\text{Whenever } e \to^* \underline{n} \text{ then } [\![e]\!] = n.$$

- Our denotational semantics aims to capture something more, for example:

$$\text{Whenever } e \to^k \underline{n} \text{ then } [e] = (k, n).$$

Example I. A strict semantics.

$C := \{\mathbf{1}, \bot\}$, and $[t]$ is given by

$$
\begin{aligned}
[x]\,\rho &:= (\mathbf{1}, \rho(x)) \\
[\mathsf{0}]\,\rho &:= (\mathbf{1}, 0) \\
[\mathsf{s}]\,\rho &:= (\mathbf{1}, \lambda n.(\mathbf{1}, n+1)) \\
[\lambda x.t]\,\rho &:= (\mathbf{1}, \lambda a.\,[t]\,\rho^a_x) \\
[ts]\,\rho &:= (\mathsf{AND}([t]_0\,, [s]_0\,, ([t]_1\,[s]_1)_0), ([t]_1\,[s]_1)_1) \\
[fx]\,\rho &:= [r]\,\rho
\end{aligned}
$$

for recursive functions $fx \to r$.

The intensional part captures termination: If $e \to^* n$ then $[e] = (\mathbf{1}, n)$ and vice versa.

Example IIa. An exact cost semantics.

$C := \mathbb{N}_\perp$, and $[t]$ is given by

$$
\begin{aligned}
[x]\,\rho &:= (0, \rho(x)) \\
[\mathtt{0}]\,\rho &:= (0, 0) \\
[\mathtt{s}]\,\rho &:= (0, \lambda n.(0, n+1)) \\
[\lambda x.t]\,\rho &:= (0, \lambda a.\,[t]_+\,\rho^a_x) \\
[ts]\,\rho &:= ([t]_0 + [s]_0 + ([t]_1\,[s]_1)_0, ([t]_1\,[s]_1)_1) \\
[fx]\,\rho &:= [r]_+\,\rho
\end{aligned}
$$

for recursive functions $fx \to r$.

The intensional part captures cost: If $e \to^k n$ then $[e] = (k, n)$ and vice versa.

EXAMPLE IIB. A bounded cost semantics.

$C := \mathbb{N}_\perp$, and $[t]$ is given by

$$\begin{aligned}
[x]\,\rho &:= (0, \rho(x)) \\
[\mathtt{0}]\,\rho &:= (0, 0) \\
[\mathtt{s}]\,\rho &:= (0, \lambda n.(0, n+1)) \\
[\lambda x.t]\,\rho &:= (0, \lambda a.\,[t]_+\,\rho^a_x) \\
[ts]\,\rho &:= ([t]_0 + [s]_0 + ([t]_1\,[s]_1)_0, ([t]_1\,[s]_1)_1) \\
[fx]\,\rho &:= \bigvee [r]_+\,\rho
\end{aligned}$$

for recursive functions $fx \to r$.

The intensional part bounds the cost: If $e \to^k n$ then $[e] = (l, n)$ with $k \leq l$ and vice versa.

We are interested in soundness and adequacy of these kinds of translations.

We are interested in soundness and adequacy of these kinds of translations.

Strict semantics: If $[e]_0 = \mathbf{1}$ then $[e]$ can be reduced to a normal form - adequacy results of this kind are proven by Berger (2005) and are used to establish strong normalisation of λ-calculi extended with bar recursion operators.

We are interested in soundness and adequacy of these kinds of translations.

Strict semantics: If $[e]_0 = \mathbf{1}$ then $[e]$ can be reduced to a normal form - adequacy results of this kind are proven by Berger (2005) and are used to establish strong normalisation of λ-calculi extended with bar recursion operators.

Exact costs: Denotational cost semantics first explored by Sands (1990) among others, generalised and lifted to a categorical setting by Van Stone (2003).

We are interested in soundness and adequacy of these kinds of translations.

Strict semantics: If $[e]_0 = \mathbf{1}$ then $[e]$ can be reduced to a normal form - adequacy results of this kind are proven by Berger (2005) and are used to establish strong normalisation of λ-calculi extended with bar recursion operators.

Exact costs: Denotational cost semantics first explored by Sands (1990) among others, generalised and lifted to a categorical setting by Van Stone (2003).

Bounded costs: A cost semantics which is sound w.r.t. a higher-type *bounding* relation $\sqsubseteq$ is studied for variants of system T by Danner et al. (2012 & 2015). Extended to call-by-name PCF by Kim (2016).

Problem. In general, soundness and particularly adequacy seem to be difficult to prove: The more complex the relationship between $t : X$ and the component $[t]_0 \in C$, the more intricate and messy the resulting induction tends to be.

Can we give a uniform framework and adequacy proof which captures a wide range of monadic translations, including those which bound the cost of programs?

PROBLEM. In general, soundness and particularly adequacy seem to be difficult to prove: The more complex the relationship between $t : X$ and the component $[t]_0 \in C$, the more intricate and messy the resulting induction tends to be.

Can we give a uniform framework and adequacy proof which captures a wide range of monadic translations, including those which bound the cost of programs?

Proofs of this kind typically have

- an important combinatorial part - does the translation work for the building blocks of our language?
- a quite technical but rather uniform domain-theoretic part verifying that it works for arbitrary terms.

PROBLEM. In general, soundness and particularly adequacy seem to be difficult to prove: The more complex the relationship between $t : X$ and the component $[t]_0 \in C$, the more intricate and messy the resulting induction tends to be.

Can we give a uniform framework and adequacy proof which captures a wide range of monadic translations, including those which bound the cost of programs?

Proofs of this kind typically have

- an important combinatorial part - does the translation work for the building blocks of our language?
- a quite technical but rather uniform domain-theoretic part verifying that it works for arbitrary terms.

Therefore it makes sense to *seperate* these parts if possible.

$$\text{Adequacy proof} = \underbrace{\text{Combinatorial part}}_{\text{easy to check}} + \underbrace{\text{Domain-theoretic part}}_{\text{uniform}}$$

Recall that

$$[D] := C \times \underbrace{[\![D]\!]}_{s(D)}$$

$$[X \to Y] := C \times (\underbrace{s(X) \to [Y]}_{s(X \to Y)})$$

Recall that

$$[D] := C \times \underbrace{[\![D]\!]}_{s(D)}$$

$$[X \to Y] := C \times (\underbrace{s(X) \to [Y]}_{s(X \to Y)})$$

Suppose that

- $I_X(e, c)$ is an arbitrary 'cost' relation between closed terms $e : X$ and total objects of $c \in C$ while
- $S_D(v, s)$ is a 'size' relation between values of type D and $s \in [\![D]\!]$ defined at all ground types.

Recall that

$$[D] := C \times \underbrace{[\![D]\!]}_{s(D)}$$

$$[X \to Y] := C \times (\underbrace{s(X) \to [Y]}_{s(X \to Y)})$$

Suppose that

- $I_X(e, c)$ is an arbitrary 'cost' relation between closed terms $e : X$ and total objects of $c \in C$ while
- $S_D(v, s)$ is a 'size' relation between values of type D and $s \in [\![D]\!]$ defined at all ground types.

Define the relation $P_X(e, \alpha)$ between closed terms $e : X$ and $\alpha \in [X]$ as follows:

$$P_D(e, \alpha) := \alpha_0 \neq \bot \Rightarrow \exists v(e \to^* v \wedge I_D(e, \alpha_0) \wedge S_D(v, \alpha_1))$$

$$P_{X \to Y}(e, \alpha) := \alpha_0 \neq \bot \Rightarrow \exists v \left(\begin{cases} e \to^* v \wedge I_{X \to Y}(e, \alpha_0) \\ \wedge \underbrace{\forall w, \beta(S_X(w, \beta) \Rightarrow P_Y(vw, \alpha_1 \beta))}_{S_{X \to Y}(v, \alpha_1)} \end{cases} \right)$$

All previous translations are simple instances of this. In particular:

All previous translations are simple instances of this. In particular:

Strict semantics:

- $C = \{\mathbf{1}, \bot\}$
- $I_X(e, \mathbf{1})$ always true,
- $S_{\texttt{nat}}(\underline{n}, m) := (n = m)$
- $P_X(e, \alpha) \Leftrightarrow (\alpha_0 = \mathbf{1} \Rightarrow \exists v(e \to^* v \wedge \alpha_1 \approx [\![v]\!]))$

where $\alpha_1 \approx [\![v]\!]$ can be read as α_1 is 'strictly denoted' by $[\![v]\!]$.

All previous translations are simple instances of this. In particular:

Strict semantics:

- $C = \{\mathbf{1}, \bot\}$
- $I_X(e, \mathbf{1})$ always true,
- $S_{\texttt{nat}}(\underline{n}, m) := (n = m)$
- $P_X(e, \alpha) \Leftrightarrow (\alpha_0 = \mathbf{1} \Rightarrow \exists v(e \to^* v \wedge \alpha_1 \approx [\![v]\!]))$

where $\alpha_1 \approx [\![v]\!]$ can be read as α_1 is 'strictly denoted' by $[\![v]\!]$.

Bounded costs:

- $C = \mathbb{N}_\bot$
- $I_X(e, k) := \forall e'(e \to^i e' \to i \leq k)$
- $S_{\texttt{nat}}(\underline{n}, m) := (n \leq m)$
- $P_X(e, \alpha) \Leftrightarrow (\alpha_0 \neq \bot \Rightarrow \exists v(e \to^k v \wedge k \leq \alpha_0 \wedge v \sqsubseteq \alpha_1))$

where $\sqsubseteq$ is a essentially the bounding relation of Danner et al. (2012 & 2015).

AIM. A general semantics of the form

$$
\begin{aligned}
[x]\,\rho &:= (c_x, \rho(x)) \\
[\mathtt{0}]\,\rho &:= (c_{\mathtt{0}}, 0) \\
[\mathtt{s}]\,\rho &:= (c_{\mathtt{s}}, \lambda n.(c'_{\mathtt{s}}, n+1)) \\
[\lambda x.t]\,\rho &:= (c_{\lambda x.t}, \lambda a.\Phi_t([t]\,\rho^a_x)) \\
[ts]\,\rho &:= (m([t]_0\,, [s]_0\,, ([t]_1\,[s]_1)_0), ([t]_1\,[s]_1)_1) \\
[fx]\,\rho &:= \Psi_f([r]\,\rho)
\end{aligned}
$$

for recursive functions $fx \to r$, where

- c_x, $c_{\mathtt{0}}$, $c_{\mathtt{s}}$ and $c_{\lambda x.t}$ are elements of a 'cost domain' C;
- $m : C \times C \times C \to C$ is a continuous function;
- Φ_t and Ψ_f are continuous functions $[X] \to [X]$, where $r, t : X$.

AIM. A general semantics of the form

$$
\begin{aligned}
[x]\,\rho &:= (c_x, \rho(x)) \\
[\mathtt{0}]\,\rho &:= (c_{\mathtt{0}}, 0) \\
[\mathtt{s}]\,\rho &:= (c_{\mathtt{s}}, \lambda n.(c'_{\mathtt{s}}, n+1)) \\
[\lambda x.t]\,\rho &:= (c_{\lambda x.t}, \lambda a.\Phi_t([t]\,\rho^a_x)) \\
[ts]\,\rho &:= (m([t]_0\,, [s]_0\,, ([t]_1\,[s]_1)_0), ([t]_1\,[s]_1)_1) \\
[fx]\,\rho &:= \Psi_f([r]\,\rho)
\end{aligned}
$$

for recursive functions $fx \to r$, where

- c_x, $c_{\mathtt{0}}$, $c_{\mathtt{s}}$ and $c_{\lambda x.t}$ are elements of a 'cost domain' C;
- $m : C \times C \times C \to C$ is a continuous function;
- Φ_t and Ψ_f are continuous functions $[X] \to [X]$, where $r, t : X$.

We want a set of conditions on these components in terms of I_X and $S_{\mathtt{nat}}$ such that:

THEOREM. For all closed terms $e : X$ we have $P_X(e, [e])$.

The difficultly in proving a theorem of this kind for arbitrary terms lies in the fact that we allow arbitrary (potentially non-terminating) recursive functions. However, we can initially avoid this by looking at finitary systems with *bounded* recursion (via bounded fixpoints $\mathbf{fix}_n$ or stratified rewrite systems $f_n x \to r_{(n-1)}$).

The difficultly in proving a theorem of this kind for arbitrary terms lies in the fact that we allow arbitrary (potentially non-terminating) recursive functions. However, we can initially avoid this by looking at finitary systems with *bounded* recursion (via bounded fixpoints $\mathbf{fix}_n$ or stratified rewrite systems $f_n x \to r_{(n-1)}$).

Let $e_{(n)}$ denote e with all function symbols replaced by f_n.

LEMMA (COMBINATORIAL PART) For all closed terms $e_{(n)} : X$ we have $P_X(e_{(n)}, [e_{(n)}])$.

Proof. Induction on n and typing of e - it's here that we do the important work.

The difficultly in proving a theorem of this kind for arbitrary terms lies in the fact that we allow arbitrary (potentially non-terminating) recursive functions. However, we can initially avoid this by looking at finitary systems with *bounded* recursion (via bounded fixpoints $\mathbf{fix}_n$ or stratified rewrite systems $f_n x \to r_{(n-1)}$).

Let $e_{(n)}$ denote e with all function symbols replaced by f_n.

Lemma (combinatorial part) For all closed terms $e_{(n)} : X$ we have $P_X(e_{(n)}, \left[e_{(n)}\right])$.

Proof. Induction on n and typing of e - it's here that we do the important work.

Lemma (domain-theoretic part) Suppose that $[e]_0 \neq \bot$. Then there is some n such that $\left[e_{(n)}\right]_0 = [e]_0$ and $\left[e_{(n)}\right]_1 \sqsubseteq [e]_1$.

Proof. Standard.

The difficultly in proving a theorem of this kind for arbitrary terms lies in the fact that we allow arbitrary (potentially non-terminating) recursive functions. However, we can initially avoid this by looking at finitary systems with *bounded* recursion (via bounded fixpoints $\mathbf{fix}_n$ or stratified rewrite systems $f_n x \to r_{(n-1)}$).

Let $e_{(n)}$ denote e with all function symbols replaced by f_n.

LEMMA (COMBINATORIAL PART) For all closed terms $e_{(n)} : X$ we have $P_X(e_{(n)}, [e_{(n)}])$.

Proof. Induction on n and typing of e - it's here that we do the important work.

LEMMA (DOMAIN-THEORETIC PART) Suppose that $[e]_0 \neq \bot$. Then there is some n such that $[e_{(n)}]_0 = [e]_0$ and $[e_{(n)}]_1 \sqsubseteq [e]_1$.

Proof. Standard.

THEOREM. For all closed terms $e : X$ we have $P_X(e, [e])$.

Some results

- Extension of existing cost semantics. In particular we can generalise bounding relation of Danner et al. to a standard call-by-value higher order language with arbitrary recursion.
- Provide a uniform framework in which a variety of cost semantics can be understood.
- Enable one to obtain *new* monadic denotational semantics for which soundness and adequacy can be easily verified.

Some results

- Extension of existing cost semantics. In particular we can generalise bounding relation of Danner et al. to a standard call-by-value higher order language with arbitrary recursion.
- Provide a uniform framework in which a variety of cost semantics can be understood.
- Enable one to obtain *new* monadic denotational semantics for which soundness and adequacy can be easily verified.

This is all work in progress! However the main goal for the future would be to utilise the translations to *analyse* programs. For example:

- Can we automatically solve the extracted recursive equations which e.g. characterise cost of a program?
- Can we give a set of conditions which guarantee that this cost functional can be defined in a weak system?