Coinduction and program extraction in computable analysis

Ulrich Berger - Swansea

CLMPS 2011

Nancy, France

Outline

Introduction

A coinductive description of approximable real numbers

Program extraction in computable analysis

Conclusion

The Curry-Howard correspondence (or Brouwer-Heyting-Kolmogorov interpretation)

Formulas correspond to *data types*
Proofs correspond to *programs*

$A \vee B$	disjoint sum
$A \wedge B$	cartesian product
$A \to B$	function space
$\exists x\, A$	(dependent) cartesian product
$\forall x\, A$	(dependent) function space

The Curry-Howard correspondence (or Brouwer-Heyting-Kolmogorov interpretation)

Formulas correspond to *data types*
Proofs correspond to *programs*

$A \vee B$	disjoint sum
$A \wedge B$	cartesian product
$A \to B$	function space
$\exists x\, A$	(dependent) cartesian product
$\forall x\, A$	(dependent) function space

A proof of a formula A corresponds to a program constructing an element of A.

The Curry-Howard correspondence (or Brouwer-Heyting-Kolmogorov interpretation)

Formulas correspond to *data types*
Proofs correspond to *programs*

$A \vee B$	disjoint sum
$A \wedge B$	cartesian product
$A \to B$	function space
$\exists x\, A$	(dependent) cartesian product
$\forall x\, A$	(dependent) function space

A proof of a formula A corresponds to a program constructing an element of A.

- What is a function?
- What if the quantified x ranges over abstract objects?
- How do we interpret logical axioms, e.g. $A \vee \neg A$?
- How do we interpret maths axioms, e.g. induction, choice?
- Why is it interesting and useful?

Why Curry-Howard is interesting and useful

Foundations

Constructive foundation of Mathematics (Brouwer, Heyting, Kolmogorov, Gödel, Kleene, Kreisel, Martin-Löf). Properties of logical and mathematical systems (Realizability $\Rightarrow$ existence and disjunction property; Dialectica Interpretation $\Rightarrow$ consistency)

Why Curry-Howard is interesting and useful

Foundations

Constructive foundation of Mathematics (Brouwer, Heyting, Kolmogorov, Gödel, Kleene, Kreisel, Martin-Löf). Properties of logical and mathematical systems (Realizability $\Rightarrow$ existence and disjunction property; Dialectica Interpretation $\Rightarrow$ consistency)

Programming

Program extraction (Minlog, Coq, Isabelle, Agda). In Minlog, realizability is used to automatically extract from a proof a program and its correctness proof.

Why Curry-Howard is interesting and useful

Foundations

Constructive foundation of Mathematics (Brouwer, Heyting, Kolmogorov, Gödel, Kleene, Kreisel, Martin-Löf). Properties of logical and mathematical systems (Realizability $\Rightarrow$ existence and disjunction property; Dialectica Interpretation $\Rightarrow$ consistency)

Programming

Program extraction (Minlog, Coq, Isabelle, Agda). In Minlog, realizability is used to automatically extract from a proof a program and its correctness proof.

Mathematics

Approximation-, fixedpoint-, ergodic-theory (Kohlenbach, Avigad, . . . , using DI). The study of function spaces led to new developments in computability theory, topology, domain theory. The problem of C-H interpreting classical choice axioms has led to new recursion principles such as bar recursion and products of selection functions (see recent work by Martin Escardo and Paulo Oliva).

What is a function and when is it a proof of an implication?

BHK-interpretation: A proof of $A \to B$ is a function f mapping proofs of A to proofs of B.

What is a function and when is it a proof of an implication?

BHK-interpretation: A proof of $A \to B$ is a function f mapping proofs of A to proofs of B.

- f should be computable. What does this mean if A itself consists of functions? ($\Rightarrow$ computability in higher types)
- Don't we need a *proof* that f does it's job? (circularity!)

Realizing an implication

Realizability (Kleene, Kreisel)

$$f\,\mathbf{r}\,(A \to B) \quad \equiv \quad \forall a\,(a\,\mathbf{r}\,A \to f(a)\,\mathbf{r}\,B)$$

Realizing an implication

Realizability (Kleene, Kreisel)

$$f\,\mathbf{r}\,(A \to B) \;\equiv\; \forall a\,(a\,\mathbf{r}\,A \to f(a)\,\mathbf{r}\,B)$$

Dialectica Interpretation (Gödel)

Realizing an implication

Realizability (Kleene, Kreisel)

$$f\,\mathbf{r}\,(A \to B) \quad \equiv \quad \forall a\,(a\,\mathbf{r}\,A \to f(a)\,\mathbf{r}\,B)$$

Dialectica Interpretation (Gödel)

Define realizabilty such that $a\,\mathbf{r}\,A$ always is a purely universal formula $\forall u\,(a\,\mathbf{r}_u\,A)$ with quantifier free kernel $a\,\mathbf{r}_u\,A$.

Realizing an implication

Realizability (Kleene, Kreisel)

$$f\,\mathbf{r}\,(A \to B) \quad \equiv \quad \forall a\,(a\,\mathbf{r}\,A \to f(a)\,\mathbf{r}\,B)$$

Dialectica Interpretation (Gödel)

Define realizabilty such that $a\,\mathbf{r}\,A$ always is a purely universal formula $\forall u\,(a\,\mathbf{r}_u\,A)$ with quantifier free kernel $a\,\mathbf{r}_u\,A$.

$$\begin{aligned} f\,\mathbf{r}\,(A \to B) \quad &\equiv \quad \forall a\,(\forall u\,(a\,\mathbf{r}_u\,A) \to \forall v\,(b\,\mathbf{r}_v\,B)) \\ &\equiv \quad \forall a \forall v\,(\forall u\,(a\,\mathbf{r}_u\,A) \to b\,\mathbf{r}_v\,B) \end{aligned}$$

Realizing an implication

Realizability (Kleene, Kreisel)

$$f \, \mathbf{r} \, (A \to B) \quad \equiv \quad \forall a \, (a \, \mathbf{r} \, A \to f(a) \, \mathbf{r} \, B)$$

Dialectica Interpretation (Gödel)

Define realizabilty such that $a \, \mathbf{r} \, A$ always is a purely universal formula $\forall u \, (a \, \mathbf{r}_u \, A)$ with quantifier free kernel $a \, \mathbf{r}_u \, A$.

$$\begin{aligned} f \, \mathbf{r} \, (A \to B) \quad &\equiv \quad \forall a \, (\forall u \, (a \, \mathbf{r}_u \, A) \to \forall v \, (b \, \mathbf{r}_v \, B)) \\ &\equiv \quad \forall a \forall v \, (\forall u \, (a \, \mathbf{r}_u \, A) \to b \, \mathbf{r}_v \, B) \end{aligned}$$

By a continuity argument the premise, $\forall u \, (a \, \mathbf{r}_u \, A)$, is needed for finitely many u only. In fact, classically, a single u, to be computed from a and v, suffices:

Realizing an implication

Realizability (Kleene, Kreisel)

$$f \,\mathbf{r}\,(A \to B) \quad \equiv \quad \forall a\,(a\,\mathbf{r}\,A \to f(a)\,\mathbf{r}\,B)$$

Dialectica Interpretation (Gödel)
Define realizabilty such that $a\,\mathbf{r}\,A$ always is a purely universal formula $\forall u\,(a\,\mathbf{r}_u\,A)$ with quantifier free kernel $a\,\mathbf{r}_u\,A$.

$$\begin{aligned} f\,\mathbf{r}\,(A \to B) &\equiv \forall a\,(\forall u\,(a\,\mathbf{r}_u\,A) \to \forall v\,(b\,\mathbf{r}_v\,B)) \\ &\equiv \forall a \forall v\,(\forall u\,(a\,\mathbf{r}_u\,A) \to b\,\mathbf{r}_v\,B) \end{aligned}$$

By a continuity argument the premise, $\forall u\,(a\,\mathbf{r}_u\,A)$, is needed for finitely many u only. In fact, classically, a single u, to be computed from a and v, suffices:

$$(f,g)\,\mathbf{r}\,(A \to B) \quad \equiv \quad \forall a,v\,(a\,\mathbf{r}_{g(a,v)}\,A \to f(a)\,\mathbf{r}_v\,B)$$

Soundness

Both interpretations, Realizability and the Dialectica Interpretation, extract from a proof of A a term M and a proof of $M\,\mathbf{r}\,A$ (Soundness Theorem).

Soundness

Both interpretations, Realizability and the Dialectica Interpretation, extract from a proof of A a term M and a proof of $M\,\mathbf{r}\,A$ (Soundness Theorem).

In the Dialectica Interpretation the proof of $M\,\mathbf{r}\,A$ takes place in a quantifier free system!

Soundness

Both interpretations, Realizability and the Dialectica Interpretation, extract from a proof of A a term M and a proof of $M \mathbf{r} A$ (Soundness Theorem).

In the Dialectica Interpretation the proof of $M \mathbf{r} A$ takes place in a quantifier free system!

In the following we will work with realizability.

Realizing quantifiers

Traditionally:

$$
\begin{aligned}
(x, a)\, \mathbf{r}\, \exists x\, A(x) &\equiv a\, \mathbf{r}\, A(x) \\
f\, \mathbf{r}\, \forall x\, A(x) &\equiv \forall x\, (f(x)\, \mathbf{r}\, A(x))
\end{aligned}
$$

Realizing quantifiers

Traditionally:

$$\begin{aligned}(x, a)\,\mathbf{r}\,\exists x\, A(x) &\equiv a\,\mathbf{r}\,A(x)\\ f\,\mathbf{r}\,\forall x\, A(x) &\equiv \forall x\,(f(x)\,\mathbf{r}\,A(x))\end{aligned}$$

x may range over abstract object (reals, real functions, ...).

This seems to require a realizing programming language with data types for such abstract objects.

Realizing quantifiers

Traditionally:

$$
\begin{aligned}
(x, a)\,\mathbf{r}\,\exists x\, A(x) &\equiv a\,\mathbf{r}\,A(x) \\
f\,\mathbf{r}\,\forall x\, A(x) &\equiv \forall x\,(f(x)\,\mathbf{r}\,A(x))
\end{aligned}
$$

x may range over abstract object (reals, real functions, ...).

This seems to require a realizing programming language with data types for such abstract objects.

Alternative: uniform realization of quantifiers

$$
\begin{aligned}
a\,\mathbf{r}\,\exists x\, A(x) &\equiv \exists x\,(a\,\mathbf{r}\,A(x)) \\
a\,\mathbf{r}\,\forall x\, A(x) &\equiv \forall x\,(a\,\mathbf{r}\,A(x))
\end{aligned}
$$

Relativized quantifiers

For concrete objects we may relativize the quantifiers.

For example, “every natural number can be approximately halved” can be expressed by

$$\forall x\,(\mathbb{N}(x) \to \exists y\,(\mathbb{N}(y) \wedge (x = 2y \vee x = 2y + 1)))$$

where the predicate $\mathbb{N}$ is defined in such a way that $n\,\mathbf{r}\,\mathbb{N}(x)$ means that n is a representation of the natural number x.

Relativized quantifiers

For concrete objects we may relativize the quantifiers.

For example, "every natural number can be approximately halved" can be expressed by

$$\forall x\, (\mathbb{N}(x) \to \exists y\, (\mathbb{N}(y) \land (x = 2y \lor x = 2y + 1)))$$

where the predicate $\mathbb{N}$ is defined in such a way that $n\,\mathbf{r}\,\mathbb{N}(x)$ means that n is a representation of the natural number x.

From a (constructive) proof of this formula, realizability extracts a program of type $\mathbf{N} \to \mathbf{nat} \times \mathbf{B}$ computing integer division by 2.

Program extraction and the law of excluded middle (LEM)

Realizing, say, $\forall x\,(\mathbb{N}(x) \to A(x) \vee \neg A(x))$ would mean to construct a program computing for every (representation of) a natural number x a realizer of $A(x)$ or a realizer or $\neg A(x)$. This is impossible, in general.

Program extraction and the law of excluded middle (LEM)

Realizing, say, $\forall x\,(\mathbb{N}(x) \to A(x) \vee \neg A(x))$ would mean to construct a program computing for every (representation of) a natural number x a realizer of $A(x)$ or a realizer or $\neg A(x)$. This is impossible, in general.

But, one can eliminate all uses of LEM in proofs of formulas of the form

$$\forall x\,(\mathbb{N}(x) \to \exists y\,(\mathbb{N}(y) \wedge A_0(x,y))$$

where $A_0(x,y)$ is decidable, using Gödel's negative translation and the Friedman/Dragalin A-translation.

Other approaches to program extraction from classical proofs

- ϵ-substitution calculus (Hilbert).
- Interpretation of $\neg\neg A \to A$ by continuations (Felleisen).
- Direct computational interpretation of classical sequent calculus ($\lambda\mu$-calculus, Parigot).
- Interpretation of restricted forms of LEM by learning based realizability (Berardi, Aschieri)
- Realizability interpretation of classical systems via stacks and processes (Krivine).

Interpreting induction

Induction on natural numbers

$$A(0) \wedge \forall x\,(A(x) \to A(x+1)) \to \forall x\,(\mathbb{N}(x) \to A(x))$$

is a special case of induction on an inductively defined predicate:

Interpreting induction

Induction on natural numbers

$$A(0) \wedge \forall x \, (A(x) \to A(x+1)) \to \forall x \, (\mathbb{N}(x) \to A(x))$$

is a special case of induction on an inductively defined predicate:
Set $\Phi(X) := \{0\} \cup \{x + 1 \mid x \in X\}$, then $\mathbb{N} = \mu\Phi = \mu X.\Phi(X)$

Interpreting induction

Induction on natural numbers

$$A(0) \wedge \forall x\,(A(x) \to A(x+1)) \to \forall x\,(\mathbb{N}(x) \to A(x))$$

is a special case of induction on an inductively defined predicate: Set $\Phi(X) := \{0\} \cup \{x+1 \mid x \in X\}$, then $\mathbb{N} = \mu\Phi = \mu X.\Phi(X)$

In general, one has for a monotone predicate transformer Φ an induction schema for its least fixed point $\mu\Phi$:

$$\Phi(\mathcal{P}) \subseteq \mathcal{P} \to \mu\Phi \subseteq \mathcal{P}$$

The data type associated with $\mu\Phi$ is the initial algebra $\mathrm{In}_\varphi : \varphi(\mu\varphi) \to \mu\varphi$ of a functor φ derived from Φ. The induction scheme is realized by the iterator $\mathbf{It}_\varphi$ that iterates any "step function" (i.e. φ-algebra) $f : \varphi(\alpha) \to \alpha$ to an algebra morphism $\mathbf{It}_\varphi(f) : \mu\varphi \to \alpha$ with computation rule (i.e. morphism equation)

$$\mathbf{It}_\varphi(f)\,\mathrm{In}_\varphi(m) = f(\mathbf{map}_\varphi(\mathbf{It}_\varphi(f))(m))$$

Example: Natural numbers

Recall $\mathbb{N} = \mu\Phi$ where

$$\begin{aligned} \Phi(X) &= \{0\} \cup \{x + 1 \mid x \in X\} \\ &= \{y \mid y = 0 \vee \exists x\,(y = x + 1 \wedge x \in X)\} \end{aligned}$$

Example: Natural numbers

Recall $\mathbb{N} = \mu\Phi$ where

$$\begin{aligned} \Phi(X) &= \{0\} \cup \{x + 1 \mid x \in X\} \\ &= \{y \mid y = 0 \vee \exists x\,(y = x + 1 \wedge x \in X)\} \end{aligned}$$

The functor associated with Φ is obtained by removing all first-order parts from Φ: $\varphi(\alpha) = 1 + \alpha$. The initial algebra $\mathrm{In}_\varphi : \varphi(\mu\varphi) \to \mu\varphi$ is the familiar structure of unary natural numbers $\mathbb{N} := \mu\varphi$ generated by zero and successor.

Example: Natural numbers

Recall $\mathbb{N} = \mu\Phi$ where

$$\begin{aligned}\Phi(X) &= \{0\} \cup \{x+1 \mid x \in X\} \\ &= \{y \mid y = 0 \vee \exists x\,(y = x+1 \wedge x \in X)\}\end{aligned}$$

The functor associated with Φ is obtained by removing all first-order parts from Φ: $\varphi(\alpha) = 1 + \alpha$. The initial algebra $\mathrm{In}_\varphi : \varphi(\mu\varphi) \to \mu\varphi$ is the familiar structure of unary natural numbers $\mathbb{N} := \mu\varphi$ generated by zero and successor.

A step function $f : \varphi(\alpha) \to \alpha$ consists of $f_0 : \alpha$ and $f_1 : \alpha \to \alpha$. The iteration $g := \mathbf{It}_\varphi(f) : \mathbb{N} \to \alpha$ is defined recursively by $g(0) = f_0$, $g(\mathrm{S}(n)) = f_1(g(n))$.

Example: Natural numbers

Recall $\mathbb{N} = \mu\Phi$ where

$$\begin{aligned}\Phi(X) &= \{0\} \cup \{x+1 \mid x \in X\} \\ &= \{y \mid y = 0 \vee \exists x\,(y = x+1 \wedge x \in X)\}\end{aligned}$$

The functor associated with Φ is obtained by removing all first-order parts from Φ: $\varphi(\alpha) = 1 + \alpha$. The initial algebra $\mathrm{In}_\varphi : \varphi(\mu\varphi) \to \mu\varphi$ is the familiar structure of unary natural numbers $\mathbb{N} := \mu\varphi$ generated by zero and successor.

A step function $f : \varphi(\alpha) \to \alpha$ consists of $f_0 : \alpha$ and $f_1 : \alpha \to \alpha$. The iteration $g := \mathbf{It}_\varphi(f) : \mathbb{N} \to \alpha$ is defined recursively by $g(0) = f_0$, $g(\mathrm{S}(n)) = f_1(g(n))$.

Remarks: 1. The variables x, y may range over abstract objects, for example the real numbers. 2. Category theory is only needed to explain realizability. The "user" doesn't have to know anything about this.

Reals as processes

We view a real number x as a *process* emitting digits that provide better and better approximations to x.

Processes are conveniently modelled by *final coalgebras*.

Realizability naturally associates final coalgebras with *coinductive definitions*, i.e. greatest fixed points of monotone predicate transformers (in the same way as it associates initial algebras with inductive definitions.

Hence, we use coinductive definitions to model a digital approach to computable analysis.

Coinduction

Coinduction is dual to induction. Given a monotone predicate transformer Φ we have a coinduction scheme for its greatest fixed point $\nu\Phi$:

$$\mathcal{P} \subseteq \Phi(\mathcal{P}) \to \mathcal{P} \subseteq \nu\Phi$$

The associated data type is the final coalgebra $\mathrm{Out}_\varphi : \nu\varphi \to \varphi(\mu\varphi)$.

The coinduction scheme is realized by the coiterator $\mathbf{Coit}_\varphi$ that coiterates any "step function" (i.e. φ-coalgebra) $f : \alpha \to \varphi(\alpha)$ to a coalgebra morphism $\mathbf{Coit}_\varphi(f) : \alpha \to \mu\varphi$ with computation rule (i.e. morphism equation)

$$\mathrm{Out}_\varphi(\mathbf{Coit}_\varphi(f)(a)) = \mathbf{map}_\varphi(\mathbf{Coit}_\varphi(f))(f(a))$$

Equivalently, using the fact that Out_φ has an inverse In_φ,

$$\mathbf{Coit}_\varphi(f)(a) = \mathrm{In}_\varphi(\mathbf{map}_\varphi(\mathbf{Coit}_\varphi(f))(f(a)))$$

Example: Signed digit representation

We are after a signed digit representation of real numbers x in the compact interval $\mathbb{I} := [-1, 1]$, i.e. we want

$$x = \sum_{n=0}^{\infty} d_n \cdot 2^{-(n+1)} \tag{1}$$

where $d_i \in \mathrm{SD} := \{-1, 0, 1\}$.

Example: Signed digit representation

We are after a signed digit representation of real numbers x in the compact interval $\mathbb{I} := [-1, 1]$, i.e. we want

$$x = \sum_{n=0}^{\infty} d_n \cdot 2^{-(n+1)} \tag{1}$$

where $d_i \in \mathrm{SD} := \{-1, 0, 1\}$.

(1) is equivalent to the fact that there are $x_0, x_1, \ldots \in \mathbb{I}$ such that $x = 1/2(d_0 + x_0) = 1/2(d_0 + 1/2(d_1 + x_1)) = \ldots$

Example: Signed digit representation

We are after a signed digit representation of real numbers x in the compact interval $\mathbb{I} := [-1, 1]$, i.e. we want

$$x = \sum_{n=0}^{\infty} d_n \cdot 2^{-(n+1)} \tag{1}$$

where $d_i \in \mathrm{SD} := \{-1, 0, 1\}$.

(1) is equivalent to the fact that there are $x_0, x_1, \ldots \in \mathbb{I}$ such that $x = 1/2(d_0 + x_0) = 1/2(d_0 + 1/2(d_1 + x_1)) = \ldots$

This suggests the following coinductive predicate on $\mathbb{I}$:

$$\mathrm{C}_0 = \nu X.\{x \mid \exists d \in \mathrm{SD}\, \exists x_0\, (x = \frac{d + x_0}{2} \wedge X(x_0)\}$$

The data type associated with C_0 is the type of infinite streams of signed digits. A stream $d_0, d_1, \ldots$ realizes $\mathrm{C}_0(x)$ precisely when (1) holds.

Extracting exact real number algorithms

Using coinduction one can prove, for example:

Theorem 1 $x \in \mathrm{C}_0$ iff $\forall n \in \mathbb{N}\, \exists q \in \mathbb{Q} \cap \mathbb{I}\, |x - q| \leq 2^{-n}$.

Theorem 2 If $x, y \in \mathrm{C}_0$ then $\frac{x+y}{2} \in \mathrm{C}_0$.

Theorem 3 If $x, y \in \mathrm{C}_0$ then $xy \in \mathrm{C}_0$.

Extracting exact real number algorithms

Using coinduction one can prove, for example:

Theorem 1 $x \in \mathrm{C}_0$ iff $\forall n \in \mathbb{N}\, \exists q \in \mathbb{Q} \cap \mathbb{I}\, |x - q| \leq 2^{-n}$.

Theorem 2 If $x, y \in \mathrm{C}_0$ then $\frac{x+y}{2} \in \mathrm{C}_0$.

Theorem 3 If $x, y \in \mathrm{C}_0$ then $xy \in \mathrm{C}_0$.

From the proofs of these theorems one extracts programs translating between the signed-digit- and the Cauchy-representation, as well as implementations of addition and multiplication w.r.t. the signed digit representation.

Extracting exact real number algorithms

Using coinduction one can prove, for example:

Theorem 1 $x \in \mathrm{C}_0$ iff $\forall n \in \mathbb{N}\, \exists q \in \mathbb{Q} \cap \mathbb{I}\, |x - q| \le 2^{-n}$.

Theorem 2 If $x, y \in \mathrm{C}_0$ then $\frac{x+y}{2} \in \mathrm{C}_0$.

Theorem 3 If $x, y \in \mathrm{C}_0$ then $xy \in \mathrm{C}_0$.

From the proofs of these theorems one extracts programs translating between the signed-digit- and the Cauchy-representation, as well as implementations of addition and multiplication w.r.t. the signed digit representation.

Similar implementations were studied by Edalat, Potts, Heckmann, Escardo, Marcial-Romero, Ciaffaglione, Gianantonio, ...

The difference is that we *extract* the programs, together with their correctness proofs.

Characterizing uniform continuity by induction/coinduction

Recall the coinductive definition of reals in $\mathbb{I}$ that have a signed digit representation:

$$\mathrm{C}_0 = \nu X.\{x \mid \exists d \in \mathrm{SD}\, \exists x_0\, (x = \mathrm{av}_d(x_0) \wedge X(x_0))\}$$

where $\mathrm{av}_d(x_0) := \frac{d+x_0}{2}$.

We generalize this to a characterization of (uniformly) continuous functions $f : \mathbb{I} \to \mathbb{I}$:

$$\begin{aligned}\mathrm{C}_1 \quad = \quad & \nu X.\mu Y.\{f \mid \exists d \in \mathrm{SD}\, \exists f_0\, (f = \mathrm{av}_d \circ f_0 \wedge X(f_0)) \\ & \qquad\qquad \vee\ \forall d \in \mathrm{SD}\, Y(f \circ \mathrm{av}_d)\}\end{aligned}$$

The left disjunct is analogous to C_0 and means that f *emits* a digit.

The right disjunct means that f *absorbs* a digit from the input.

Memo tries for continuous functions

Theorem 4 $f \in \mathbb{I}^{\mathbb{I}}$ is continuous iff $f \in \mathrm{C}_1$.

Memo tries for continuous functions

Theorem 4 $f \in \mathbb{I}^{\mathbb{I}}$ is continuous iff $f \in \mathrm{C}_1$.

From the proof of this theorem one extracts programs translating between realisers of "f is continuous" (where continuity has to be defined in a constructively meaningful way) and realisers of "$f \in \mathrm{C}_1$".

Memo tries for continuous functions

Theorem 4 $f \in \mathbb{I}^{\mathbb{I}}$ is continuous iff $f \in \mathrm{C}_1$.

From the proof of this theorem one extracts programs translating between realisers of "f is continuous" (where continuity has to be defined in a constructively meaningful way) and realisers of "$f \in \mathrm{C}_1$".

What is a realiser of "$f \in \mathrm{C}_1$"?

Memo tries for continuous functions

Theorem 4 $f \in \mathbb{I}^{\mathbb{I}}$ is continuous iff $f \in \mathrm{C}_1$.

From the proof of this theorem one extracts programs translating between realisers of "f is continuous" (where continuity has to be defined in a constructively meaningful way) and realisers of "$f \in \mathrm{C}_1$".

What is a realiser of "$f \in \mathrm{C}_1$"?

It is a finitely branching non-wellfounded tree describing when f emits and absorbs digits. I.p. it is a *data structure*, not a function.

Memo tries for continuous functions

Theorem 4 $f \in \mathbb{I}^{\mathbb{I}}$ is continuous iff $f \in \mathrm{C}_1$.

From the proof of this theorem one extracts programs translating between realisers of "f is continuous" (where continuity has to be defined in a constructively meaningful way) and realisers of "$f \in \mathrm{C}_1$".

What is a realiser of "$f \in \mathrm{C}_1$"?

It is a finitely branching non-wellfounded tree describing when f emits and absorbs digits. I.p. it is a *data structure*, not a function.

Similar trees have been studied by P. Hancock, D. Pattinson, N. Ghani.

Extracting memoized exact real arithmetic

Extracting memoized exact real arithmetic

The definition of $\mathrm{C}_1 \subseteq \mathbb{I}^{\mathbb{I}}$ can be generalised to $\mathrm{C}_n \subseteq \mathbb{I}^{(\mathbb{I}^n)}$.

Extracting memoized exact real arithmetic

The definition of $\mathrm{C}_1 \subseteq \mathbb{I}^{\mathbb{I}}$ can be generalised to $\mathrm{C}_n \subseteq \mathbb{I}^{(\mathbb{I}^n)}$.

Theorem 5 The average function lies in C_2.

Extracting memoized exact real arithmetic

The definition of $\mathrm{C}_1 \subseteq \mathbb{I}^{\mathbb{I}}$ can be generalised to $\mathrm{C}_n \subseteq \mathbb{I}^{(\mathbb{I}^n)}$.

Theorem 5 The average function lies in C_2.

Theorem 6 Multiplication lies in C_2.

Extracting memoized exact real arithmetic

The definition of $\mathrm{C}_1 \subseteq \mathbb{I}^{\mathbb{I}}$ can be generalised to $\mathrm{C}_n \subseteq \mathbb{I}^{(\mathbb{I}^n)}$.

Theorem 5 The average function lies in C_2.

Theorem 6 Multiplication lies in C_2.

From Theorems 5,6 one extracts implementations of addition and multiplication as memo-tries.

Extracting memoized exact real arithmetic

The definition of $\mathrm{C}_1 \subseteq \mathbb{I}^{\mathbb{I}}$ can be generalised to $\mathrm{C}_n \subseteq \mathbb{I}^{(\mathbb{I}^n)}$.

Theorem 5 The average function lies in C_2.

Theorem 6 Multiplication lies in C_2.

From Theorems 5,6 one extracts implementations of addition and multiplication as memo-tries.

Experiments show considerable speed-up when sampling "hard" functions (e.g. high iterations of the logistic map) on a very fine grid.

Extracting memoized exact real arithmetic

The definition of $\mathrm{C}_1 \subseteq \mathbb{I}^{\mathbb{I}}$ can be generalised to $\mathrm{C}_n \subseteq \mathbb{I}^{(\mathbb{I}^n)}$.

Theorem 5 The average function lies in C_2.

Theorem 6 Multiplication lies in C_2.

From Theorems 5,6 one extracts implementations of addition and multiplication as memo-tries.

Experiments show considerable speed-up when sampling "hard" functions (e.g. high iterations of the logistic map) on a very fine grid.

Theorem 7 If $f \in \mathrm{C}_1$, then $\int f \in \mathrm{C}_0$.

Extracting memoized exact real arithmetic

The definition of $\mathrm{C}_1 \subseteq \mathbb{I}^{\mathbb{I}}$ can be generalised to $\mathrm{C}_n \subseteq \mathbb{I}^{(\mathbb{I}^n)}$.

Theorem 5 The average function lies in C_2.

Theorem 6 Multiplication lies in C_2.

From Theorems 5,6 one extracts implementations of addition and multiplication as memo-tries.

Experiments show considerable speed-up when sampling "hard" functions (e.g. high iterations of the logistic map) on a very fine grid.

Theorem 7 If $f \in \mathrm{C}_1$, then $\int f \in \mathrm{C}_0$.

The extracted program program has some similarity with A. Simpson's, but is more efficient because the functions to be integrated are represented differently.

Conclusion (program extraction)

Conclusion (program extraction)

Program extraction replaces the traditional workflow

$$\text{specification} \to \text{coding} \to \text{verification}$$

Conclusion (program extraction)

Program extraction replaces the traditional workflow

$$\text{specification} \to \text{coding} \to \text{verification}$$

by

$$\text{specification} \to \text{proof} \to \text{program extraction}$$

Conclusion (program extraction)

Program extraction replaces the traditional workflow

$$\text{specification} \to \text{coding} \to \text{verification}$$

by

$$\text{specification} \to \text{proof} \to \text{program extraction}$$

The advantages are

- proofs can be checked automatically for correctness
- program extraction is automatic and includes a correctness proof
- Realizability extends the original specification of the main program to specifications of all sub-programs. This is important when sub-programs are optimized.

Conclusion (program extraction)

Program extraction replaces the traditional workflow

$$\text{specification} \to \text{coding} \to \text{verification}$$

by

$$\text{specification} \to \text{proof} \to \text{program extraction}$$

The advantages are

- proofs can be checked automatically for correctness
- program extraction is automatic and includes a correctness proof
- Realizability extends the original specification of the main program to specifications of all sub-programs. This is important when sub-programs are optimized.

Results, open questions

- Program extraction turns out to be very helpful (not a burden) in the example areas covered.
- New (correct!) programs have been extracted that would have been difficult to "guess".
- Using a fine tuning of realisability it is possible to do abstract mathematics as usual, and still get computational content.
- For example, there is no problem with using discontinuous and partial functions (sign function, least root of a polynomial).

Results, open questions

- Program extraction turns out to be very helpful (not a burden) in the example areas covered.
- New (correct!) programs have been extracted that would have been difficult to "guess".
- Using a fine tuning of realisability it is possible to do abstract mathematics as usual, and still get computational content.
- For example, there is no problem with using discontinuous and partial functions (sign function, least root of a polynomial).

- Can we apply program extraction to areas that are less mathematical in nature?
- Can we address resource issues?

Spin-off

The proof-as-programs paradigm is not only useful for program extraction, but also creates new ideas, methods and results.

For example:

- new methods and results in approximation- fixedpoint- and ergodic-theory
- Memoized computation in higher types
- New forms of bar recursion
- Selection functions
- New "computationally efficient" definitions of uniform continuity
- Uniform logical connectives
- A new "ultra memoized" model and implementation of computation in higher types

References

B.
From coinductive proofs to exact real arithmetic. CSL 2009. LNCS 5771, 132–146.

B.
Realisability for Induction and Coinduction with Applications to Constructive Analysis. Jour. Universal Comput. Sci. 16(18), 2535–2555, 2010.

M. Seisenberger and B.
Proofs, programs, processes. CiE 2010, LNCS 6158, 39–48.

References

T. Altenkirch.
Representations of first order function types as terminal coalgebras. TLCA 2001. LNCS 2044, 8–21, 2001.

Y. Bertot.
Coinduction in Coq. In Lecture Notes of TYPES Summer School 2005, August 15-26 2005, Sweden, vol. II (2005).

J. Blanck.
Efficient exact computation of iterated maps. *JLAP*, 64:41–59, 2005.

A. Ciaffaglione, P. Di Gianantonio, Di P.
A certified, corecursive implementation of exact real numbers. *TCS* 351:39–51, 2006.

References

A. Edalat, R. Heckmann.
Computing with real numbers - I. The LFT approach to real number computation - II. A domain framework for computational geometry. International summer school on applied semantics, Caminha, Portugal, Springer, 193–267, 2002.

A. Edalat, P.J. Potts, P. Sünderhauf.
Lazy computation with exact real numbers. Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, 185-194, 1998.

References

M.H. Escardó.
PCF extended with real numbers. *TCS* 162:79–115, 1996.

M.H. Escardó, A. Simpson.
A universal characterization of the closed Euclidean interval. *LICS 2001*, 115-125.

J. Raymundo Marcial–Romero, M.H. Escardó.
Semantics of a sequential language for exact real-number computation. *TCS* 379:120–141, 2007.

P. Hancock, D. Pattinson, N. Ghani.
Representation of stream processors using nested fixed points. *unpublished*. 2008.

References

R. Hinze.
Memo functions, polytypically!". Proceedings of the Second Workshop on Generic Programming, WGP 2000, Ponte de Lima, Portugal. 2000.

B. Jacobs, J. Rutten.
A Tutorial on (Co)Algebras and (Co)Induction. EATCS Bulletin 62, 222–259, 1997.

R. O'Connor.
Certified Exact Transcendental Real Number Computation in Coq. Unpublished. 2008

R. O'Connor, B. Spitters.
A computer verified monadic, functional implementation of the integral. Unpublished. 2008

References

M. Niqui.
Formalising exact arithmetic in type theory. CiE 2005: New Computational Paradigms. Amsterdam, LNCS **3526** (2005) 368–377.

M.H. Escardó, D. Pavlovic.
Calculus in coinductive form. School of Cognitive and Computing Sciences, University of Sussex. Technical Report 97:05, 1997.

D. Pavlovic, V. Pratt.
The continuum as a final coalgebra. *TCS* 280:105–122, 2002.

D. Plume.
A Calculator for Exact Real Number Computation. 4th year project. Departments of Computer Science and Artificial Intelligence, University of Edinburgh (1998).

References

P.J. Potts, A. Edalat and M.H. Escardó.
Semantics of exact real number arithmetic. *Lics 1997*.

M. Tatsuta.
Realizability of Monotone Coinductive Definitions and Its Application to Program Synthesis. *Proceedings of Fourth International Conference on Mathematics of Program Construction*, LNCS 1422 (1998) 338–364.