Tutorial: computer verified implementation of analysis

Russell O’Connor    Bas Spitters

Radboud University Nijmegen

18th August 2009
Meeting in Nijmegen

5th October: PhD-defense Russell O’Connor
6th October: workshop
Speakers include:
John Harrison
Norbert Mueller
Helmut Schichtenberg
Integration

Outline

1. Introduction: computer mathematics, type theory
2. Implementation of the reals in Coq
3. Plots and integration
Two motivations

- Provably correct implementation of computations with continuous structures
- Computer ‘referees’ for computer proofs

From theory to practice
Computing with real numbers

- Floating points, fixed precision
- Multi precision floats (MPFR)
- Multi precision intervals (MPFI)

Multi precision floats $p \cdot 2^n$, $n \in \mathbb{Z}$. Numbers are bounded only by the memory size.
Computing with real numbers

- Floating points, fixed precision
- Multi precision floats (MPFR)
- Multi precision intervals (MPFI)

Multi precision floats $p \cdot 2^n, n \in \mathbb{Z}$.
Numbers are bounded only by the memory size.
To avoid rounding errors, compute with intervals (Moore).
Computing with real numbers

- Floating points, fixed precision
- Multi precision floats (MPFR)
- Multi precision intervals (MPFI)
- Exact real arithmetic (IRAMM, RealLib)
Computing with real numbers

- Floating points, fixed precision
- Multi precision floats (MPFR)
- Multi precision intervals (MPFI)
- Exact real arithmetic (IRAMM, RealLib)

Using intervals the user specifies the input precision. Instead, for exact real numbers one specifies the output precision.
Computing with real numbers

- Floating points, fixed precision
- Multi precision floats (MPFR)
- Multi precision intervals (MPFI)
- Exact real arithmetic (IRAMM, RealLib)

Using intervals the user specifies the input precision. Instead, for exact real numbers one specifies the output precision. An exact real number is presented by a Cauchy sequence. An exact real number is a computer program which on input \( n \) returns an output with precision \( n \).

RealLib is only \( 3 \times \) slower than floats.
Computing with real numbers

- Floating points, fixed precision
- Multi precision floats (MPFR)
- Multi precision intervals (MPFI)
- Exact real arithmetic (IRAMM, RealLib)

Using intervals the user specifies the input precision. Instead, for exact real numbers one specifies the output precision. An exact real number is presented by a Cauchy sequence. An exact real number is a computer program which on input $n$ returns an output with precision $n$. RealLib is only $3 \times$ slower than floats.
This method can be extended to complete separable metric spaces:
- The dense set can be represented in the computer.
- The limits are computer programs for Cauchy sequences.

Examples:
- $C([0, 1], \mathbb{R})$
- $L_1$ (integrable functions)
- Hilbert spaces
- Banach spaces
- Metric space of all compact sets with Hausdorff metric (plots)

...
We use informal constructive mathematics. Cf. TTE via realizability. Do not (yet) distinguish between Turing ($K_1$) and Type 2 ($K_2$) computability.
Motivation

Why use exact analysis?

1. Convenience of programming (at a small cost)
   - datatypes: let H be a computable Sobolev space...
   - maintainability
   - parallel computation !?
Motivation

Why use exact analysis?

1. Convenience of programming (at a small cost)
   - datatypes: let $H$ be a computable Sobolev space...
   - maintainability
   - parallel computation !?

2. Emerging standard of rigor for computer proofs
Motivation

Why use exact analysis?

1. Convenience of programming (at a small cost)
2. Emerging standard of rigor for computer proofs

Proposal: use constructive analysis based on type theory for computer verified implementation of numerics and as a basis for computer proofs.
Motivation

...Emerging standard of rigor for computer proofs
Computers are used to prove theorems
(4-color, Kepler conjecture, existence of chaos, ...)
Typical approaches:

- Computer checks a large finite number of cases
- Computer finds bounds using interval computations
- ...
Motivation

...Emerging standard of rigor for computer proofs
Computers are used to prove theorems
(4-color, Kepler conjecture, existence of chaos, ...)
Typical approaches:

- Computer checks a large finite number of cases
- Computer finds bounds using interval computations
- ...

Recent development:
Use computer verified proofs of the correctness of these computations
Slogan: Computer proofs deserve computer ‘referees’.
Proof assistant

Russell O'Connor, Bas Spitters

Tutorial: computer verified implementation of analysis
Design of a proof assistant:
‘Small’ trusted core: set theory, type theory
This can be inspected by hand, tested, proved almost correct in itself (Gödel)
See Hales Notices article
On top of this we develop a library of mathematics
The computer checks the reduction to the core
... Computer verified proofs.
Usual proofs serve two purposes:

- certifying correctness
- conveying understanding

Computers can help in the first purpose (cf. spell checker).
Proof assistants

Constructing proofs is difficult.
Checking formal proofs is easy.
Proof assistants

Constructing proofs is difficult. Checking formal proofs is easy. A formal proof is written in the language of logic and set theory. To check such a proof the computer just checks whether one of a limited number of axioms and rules is applied (e.g. If $A$ and $B$, then $A.$)
Proof assistants

Some milestones of computer verified proofs:

- 4-color theorem (Gonthier)
- Prime number theorem (Avigad)
- Constructive Fundamental theorem of algebra (Geuvers...)
- Gödel incompleteness (Shankar)
- Feit-Thompson, classification of finite simple groups (Gonthier)
- Kepler conjecture (Hales)

The 4-color theorem and the Kepler conjecture are famous computer proofs, there is no standard proof. The computer checks a huge number of configurations. Is this a **proof**? How do we certify its correctness?
Proof assistants

Some milestones of computer verified proofs:

- 4-color theorem (Gonthier)
- Prime number theorem (Avigad)
- Constructive Fundamental theorem of algebra (Geuvers…)
- Gödel incompleteness (Shankar)
- Feit-Thompson, classification of finite simple groups (Gonthier)
- Kepler conjecture (Hales)

The 4-color theorem and the Kepler conjecture are famous computer proofs, there is no standard proof. The computer checks a huge number of configurations. Is this a **proof**? How do we certify its correctness?

**Computer referees!** Gonthier completely formalized the proof including the correctness proof of the implementation of the program. The computer checks that this proof is indeed correct.
Proof assistants

Constructing proofs is difficult...
However, proof assistants help to construct formal proofs by automating triviality:

- Logical tautologies
- Computations $e \leq 3$
- decision procedures for special theories, e.g.
  natural numbers with $+, -, \times, <, \leq$
  linear inequalities on real numbers
- induction proofs
- rewrite databases ring theory
Motivation

Why use exact analysis?

1. Convenience of programming (at a small cost)
2. Emerging standard of rigor for computer proofs

Proposal: use constructive analysis based on type theory for computer verified implementation of numerics and as a basis for computer proofs.
Types in computer programming:
3:int and [1.0,2.0]:array float.
3+[1.0,2.0] gives a type error.
Types in computer programming:
3:int and [1.0,2.0]:array float.
3+[1.0,2.0] gives a type error.

Similar support for the computer formalization of mathematics.
This is absent from the set theoretic foundations.
The question: ‘Is the set $\pi$ an element of the set sin?’ is valid.
It’s truth depends on the arbitrary encoding of mathematics in sets.
In practice, one uses a type theory (possibly on top of sets).
Types

Inductive types: Natural numbers $N$, lists, trees, ...
Type formers: product of types, sum of types, quotients, ...
Examples:

- Rationals are a quotient of the product type $N \times N$
- Exceptions: $N + \bot$ (finite sum)
- Function types
- Arbitrary precision floats: $\prod_{n \in N} \text{float}(n)$ (dependent product/function)
- A finite group: $\Sigma_{n \in N} \text{Grp}(n)$ (dependent sum type)
Non-dependent type constructions are present in modern programming languages. (Some) dependent types are present in the Axiom computer algebra system.

Type theory can take the role of set theory as a foundation for mathematics. A type system for all of mathematics. Type theory as

- a foundation for mathematics and
- a programming language.
Curry-Howard-deBruijn isomorphism

The functional interpretation of constructive deductions is given by the Curry-Howard-deBruijn isomorphism. This isomorphism associates formulas with dependent types, and proofs of formulas with functional programs of the associated dependent types. For example, the identity function $\lambda x : A. x$ of type $A \Rightarrow A$ represents a proof of the tautology $A \Rightarrow A$.

<table>
<thead>
<tr>
<th>Logical Connective</th>
<th>Type Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>implication: $\Rightarrow$</td>
<td>function type: $\Rightarrow$</td>
</tr>
<tr>
<td>conjunction: $\wedge$</td>
<td>product type: $\times$</td>
</tr>
<tr>
<td>disjunction: $\vee$</td>
<td>disjoint union type: $+$</td>
</tr>
<tr>
<td>true: $\top$</td>
<td>unit type: ()</td>
</tr>
<tr>
<td>false: $\bot$</td>
<td>void type: $\emptyset$</td>
</tr>
<tr>
<td>for all: $\forall x. Px$</td>
<td>dependent function type: $\Pi x. Px$</td>
</tr>
<tr>
<td>exists: $\exists x. Px$</td>
<td>dependent pair type: $\Sigma x. Px$</td>
</tr>
</tbody>
</table>
The modest sets in a realizability model support a dependent type theory with (co)inductive types.
The semantics of NuPrl is defined by a realizability for extensional Martin-Löf type theory.
MetaPrl should allow a similar implementation for TTE, as $K_2$-realizability.
Reals are the completion of the rationals

$$\Sigma f : \mathbb{N} \Rightarrow \mathbb{Q}. \forall nm. |f(n) - f(m)| \leq 2^{-n} + 2^{-m}.$$ 

A real number is a pair $\langle f, p \rangle$, where $f : \mathbb{N} \Rightarrow \mathbb{Q}$ and $p$ is a proof that it is Cauchy

$$\forall nm. |f(n) - f(m)| \leq 2^{-n} + 2^{-m}.$$ 

This encodes the real number $\lim_{n \to \infty} f(n)$.

$f$ is the program and $p$ the proof that it meets its specification.

These can be conveniently combined in a dependent type.
Constructive logic

For exact real numbers there is no zero-test: Is $0.0000000\ldots = 0$ or not??
However, we can decide $|x| \leq \epsilon$ or $|x| \geq \frac{\epsilon}{2}$. 

Moral: when programming with exact reals avoid the effective decision $P$ or not $P$.

Only constructive logic can be used:
- is valid under realizability
- holds under the Curry-Howard isomorphism
For exact real numbers there is no zero-test: Is 0.0000000... = 0 or not?? However, we can decide \( |x| \leq \epsilon \) or \( |x| \geq \frac{\epsilon}{2} \).
Moral: when programming with exact reals avoid the effective decision
\( P \) or not \( P \).
Only constructive logic can be used:
- is valid under realizability
- holds under the Curry-Howard isomorphism
It is convenient to use maps $\mathbb{Q}^+ \Rightarrow \mathbb{Q}$ instead of sequences. Reals are the completion of the rationals

$$\Sigma f : \mathbb{Q}^+ \Rightarrow \mathbb{Q} . \forall \varepsilon_1 \varepsilon_2 . |f \varepsilon_1 - f \varepsilon_2| \leq \varepsilon_1 + \varepsilon_2.$$  

(note the dependent types)
It is convenient to use maps $\mathbb{Q}^+ \Rightarrow \mathbb{Q}$ instead of sequences. Reals are the completion of the rationals

$$\Sigma f : \mathbb{Q}^+ \Rightarrow \mathbb{Q}. \forall \epsilon_1 \epsilon_2. |f \epsilon_1 - f \epsilon_2| \leq \epsilon_1 + \epsilon_2.$$  

(note the dependent types)

This completion can be defined for general metric spaces:

$$\mathcal{C}X := \exists f : \mathbb{Q}^+ \Rightarrow X. \forall \epsilon_1 \epsilon_2. B^{X}_{\epsilon_1 + \epsilon_2}(f \epsilon_1)(f \epsilon_2).$$
Reals

It is convenient to use maps $\mathbb{Q}^+ \Rightarrow \mathbb{Q}$ instead of sequences. Reals are the completion of the rationals

$$\sum f : \mathbb{Q}^+ \Rightarrow \mathbb{Q}. \forall \varepsilon_1 \varepsilon_2. |f\varepsilon_1 - f\varepsilon_2| \leq \varepsilon_1 + \varepsilon_2.$$  

(note the dependent types)

This completion can be defined for general metric spaces:

$$\mathcal{C}X := \exists f : \mathbb{Q}^+ \Rightarrow X. \forall \varepsilon_1 \varepsilon_2. B_{\varepsilon_1 + \varepsilon_2}^X (f\varepsilon_1)(f\varepsilon_2).$$

Completion is a monad!
Functional programming

Imperative programs while/for-loops ... use a state and mutable data. They are difficult to reason about.

```plaintext
if n=0 then s:=1
else s:=1; for i=1..n do
    s:=s*i;
    od; fi
```

Instead, functional programs have no side effects, and are easier to reason about.

Fixpoint fact (n:nat) : nat :=
match n with
| O => 1
| S n => S n * fact n
end.

We use effects in a controlled way: enter monads.
Functional programming

Imperative programs while/for-loops ... use a state and mutable data. They are difficult to reason about.

```plaintext
if n=0 then s:=1
else s:=1; for i=1..n do
    s:=s*i;
    od; fi
```

Instead, functional programs have no side effects, and are easier to reason about.

```plaintext
Fixpoint fact (n:nat) : nat :=
    match n with
        | 0 => 1
        | S n => S n * fact n
    end.
```
Functional programming

Imperative programs while/for-loops ... use a state and mutable data. They are difficult to reason about.

if n=0 then s:=1
  else s:=1; for i=1..n do
    s:=s*i;
  od; fi

Instead, functional programs have no side effects, and are easier to reason about.

Fixpoint fact (n:nat) : nat :=
  match n with
  | O => 1
  | S n => S n * fact n
  end.

We use effects in a controlled way: enter monads.
Monads

Consider the algebras of signature $\tau$. (Example: Monoids)
This gives a category $\text{Alg}_\tau$.
The forgetful functor $G : \text{Alg}_\tau \to \text{Set}$ has a left adjoint $F$ which assigns to each set $S$ the free $\tau$-algebra over $S$.
The composition $T = GF$ defines a ‘monad’ on $\text{Set}$.
A monad is an endofunctor on $\mathcal{C}$ with nat trans $\eta : c \to Tc$ and $\mu : TTc \to Tc$ (+laws)
Monads

Consider the algebras of signature $\tau$. (Example: Monoids)
This gives a category $\text{Alg}_\tau$.
The forgetful functor $G : \text{Alg}_\tau \to \text{Set}$ has a left adjoint $F$ which
assigns to each set $S$ the free $\tau$-algebra over $S$.
The composition $T = GF$ defines a ‘monad’ on $\text{Set}$.
A monad is an endofunctor on $\mathbf{C}$ with nat trans $\eta : c \to Tc$ and
$\mu : TTc \to Tc$ (+laws)
All $\tau$-algebras can be refound as $T$-algebras! $\alpha : TS \to S$
satisfying certain laws.
Example: A monoid on $X$ is a map from the free monoid $\text{list}(X)$
to $X$ (+laws).
Monads

Consider the algebras of signature $\tau$. (Example: Monoids)
This gives a category $\text{Alg}_\tau$.
The forgetful functor $G : \text{Alg}_\tau \rightarrow \text{Set}$ has a left adjoint $F$ which assigns to each set $S$ the free $\tau$-algebra over $S$.
The composition $T = GF$ defines a ‘monad’ on $\text{Set}$.
A monad is an endofunctor on $\mathbf{C}$ with nat trans $\eta : c \rightarrow Tc$ and $\mu : TTc \rightarrow Tc$ (+laws)
All $\tau$-algebras can be refound as $T$-algebras! $\alpha : TS \rightarrow S$
satisfying certain laws.
Example: A monoid on $X$ is a map from the free monoid $\text{list}(X)$ to $X$ (+laws).
In computer science one uses ‘free’ $T$-algebras.
This category is equivalent to the Kleisli category of $T$ on $\mathbf{C}$:
Objects: objects of $\mathbf{C}$.
Arrow from $X$ to $Y$: $f : X \rightarrow TY$. 
Monads in Haskell

Monads are ubiquitous in Haskell style functional programming.
Monads are ubiquitous in Haskell style functional programming. A program with input $X$ and output $Y$ with access to a mutable state $S$ can be modeled by:

$$f : X \times S \Rightarrow Y \times S$$
Monads are ubiquitous in Haskell style functional programming. A program with input $X$ and output $Y$ with access to a mutable state $S$ can be modeled by:

$$f : X \times S \Rightarrow Y \times S \equiv X \Rightarrow (Y \times S)^S$$
Monads are ubiquitous in Haskell style functional programming. A program with input $X$ and output $Y$ with access to a mutable state $S$ can be modeled by:

$$f : X \times S \Rightarrow Y \times S \equiv X \Rightarrow (Y \times S)^S$$

The type constructor $MY := (Y \times S)^S$ is a monad.
Monads in Haskell

Monads are ubiquitous in Haskell style functional programming. A program with input $X$ and output $Y$ with access to a mutable state $S$ can be modeled by:

$$f : X \times S \Rightarrow Y \times S \equiv X \Rightarrow (Y \times S)^S$$

The type constructor $MY := (Y \times S)^S$ is a monad.
The monad $MY := Y + \bot$ gives partial functions $X \Rightarrow Y + \bot$. 
Monads are ubiquitous in Haskell style functional programming. A program with input $X$ and output $Y$ with access to a mutable state $S$ can be modeled by:

$$f : X \times S \Rightarrow Y \times S \equiv X \Rightarrow (Y \times S)^S$$

The type constructor $MY := (Y \times S)^S$ is a monad. The monad $MY := Y + \bot$ gives partial functions $X \Rightarrow Y + \bot$. The reader monad $MY := Y^E$ is used for input.
Formally, a monad is a triple $(M, \text{return}, \text{bind})$

- $\text{return} : \mathit{X} \Rightarrow MX$
- $\text{bind} : MX \Rightarrow (\mathit{X} \Rightarrow MY) \Rightarrow MY$

These two operations must satisfy some laws. Alternative formulation in terms of unit($\eta$), map, join($\mu$).
Monads

Formally, a monad is a triple \((M, \text{return}, \text{bind})\)

\[
\text{return} : X \Rightarrow MX \\
\text{bind} : MX \Rightarrow (X \Rightarrow MY) \Rightarrow MY
\]

These two operations must satisfy some laws.

Alternative formulation in terms of unit\((\eta)\), map, join\((\mu)\).

Functions \(f : X \Rightarrow MY\) and \(g : Y \Rightarrow MZ\)
can be composed using bind \(g \circ f : X \Rightarrow MZ\). Define the composition:

\[
f \gg g \equiv (\lambda x.fx \gg g)
\]

of type \(X \Rightarrow MZ\)
Completion is a monad:

- return : \(X \rightarrow C X\) is the embedding
- a uniformly continuous function \(f : X \rightarrow Y\) can be lifted to operate on complete metric spaces: map \(f : C X \rightarrow C Y\)
- join : \(C C X \rightarrow C X\)
- It suffices to define a function on a dense set: bind : \(C X \rightarrow (X \rightarrow C Y) \rightarrow C Y\)
O’Connor completion monad

Completion is a monad:

- \textbf{return} : \(X \rightarrow CX\) is the embedding
- \textbf{a uniformly continuous function} \(f : X \rightarrow Y\) \textbf{can be lifted to}\n  \textbf{operate on complete metric spaces: map} \(f : CX \rightarrow CY\)
- \textbf{join} : \(CCX \rightarrow CX\)
- \textbf{It suffices to define a function on a dense set:} \n  \textbf{bind} : \(CX \rightarrow (X \rightarrow CY) \rightarrow CY\)

\textbf{Structured} way of working only with finite objects, which are \textbf{directly representable}.
Motivation

Bishop’s program: use constructive mathematics as a programming language for exact analysis.

As written, this book is person-oriented rather than computer-oriented. It would be of great interest to have a computer-oriented version. Without such a version, it is hard to predict with any confidence what form computer-oriented abstract analysis will eventually assume. A thoughtful computer-oriented presentation should uncover many interesting phenomena.

Our aim is to provide such a presentation.
Motivation

Bishop’s program: use constructive mathematics as a programming language for exact analysis.

As written, this book is person-oriented rather than computer-oriented. It would be of great interest to have a computer-oriented version. Without such a version, it is hard to predict with any confidence what form computer-oriented abstract analysis will eventually assume. A thoughtful computer-oriented presentation should uncover many interesting phenomena.

Our aim is to provide such a presentation.
Moreover, we provide implementations in dependent type theory:
- a foundation for constructive mathematics and
- an efficient programming language with a very expressive type system.
Brief history of Bishop program in type theory

- Bishop (67) Foundations of Constructive Analysis
- de Bruijn (67) the Automath proof assistant
- Martin-Löf (70,82) Type theory as foundation for Bishop
- Implementations of TT: NuPrl, Coq, Agda, Epigram, ...
- CoRN (00-04): formalization of math in type theory is feasible (FTA, FTC)
- Executing formal [proofs/programs] is difficult (Cruz-Filipe/S.03)
- O’Connor implementation of constructive analysis in type theory
  - Real numbers, trigonometric functions (part II)
  - Plots (Part III)
  - Integrals [O’Connor/S] (Part III)

Beginning of the realization of Bishop’s program in type theory!?
We write new programs which are easy to prove correct. Our (functional) program meets its specification. Leroy: verified compiler from Cminor to assembly. Future: extend this to Coq Compiler.
Real world programs

We write new programs which are easy to prove correct.
Our (functional) program meets its specification.
Leroy: verified compiler from Cminor to assembly.
Future: extend this to Coq Compiler.
Alternative: verify existing programs:
Real world programs

We write new programs which are easy to prove correct. Our (functional) program meets its specification. Leroy: verified compiler from C minor to assembly. Future: extend this to Coq Compiler. Alternative: verify existing programs: Verification of the seL4 kernel(!) 10,000 theorems, 200,000 lines of formal proof Uses functional prototype as low-level specification.
References

- O’Connor, Spitters - A computer verified, monadic, functional implementation of the integral
- O’Connor - A Monadic, Functional Implementation of Real Numbers
- O’Connor - Certified Exact Transcendental Real Number Computation in Coq
- O’Connor - A Computer Verified Theory of Compact Sets
- Geuvers, Niqui, Spitters and Wiedijk - Constructive analysis, types and exact real numbers
- Cruz-Filipe, Spitters - Program extraction from large proof developments
We present definite integration in informal constructive mathematics. This presentation can be directly expressed in type theory.
Constructive mathematics

We present definite integration in informal constructive mathematics.
This presentation can be directly expressed in type theory.
We have a fully verified implementation in type theory (Coq).
We present definite integration in informal constructive mathematics. This presentation can be directly expressed in type theory. We have a fully verified implementation in type theory (Coq).

When programming/proving in type theory we often find interesting new mathematical structure.

DEMO
Programming Riemann integration:

1. Define step functions;
2. Introduce applicative functors and show that step functions form an applicative functor;
3. Show that the step functions from a metric space under both the $L^1$ and $L^\infty$ norms;
4. Define integrable functions as the completion of the step functions under the $L^1$ norm;
5. Define integration, first on step functions, then lift it to operate on integrable functions;
6. Define an injection from the continuous functions to the integrable functions in order to integrate them.
The type $SX$ of step functions on $X$ is defined inductively:

- $\hat{x}$, a constant function
- $f \triangleright o \blacktriangleleft g$ is $f$ squeezed into $[0, o]$ and $g$ squeezed into $[o, 1]$
Step Functions are inductive

\[
\begin{align*}
\text{fold} & : (X \Rightarrow Y) \Rightarrow ([0,1]_{\mathbb{Q}} \Rightarrow Y \Rightarrow Y \Rightarrow Y) \Rightarrow SX \Rightarrow Y \\
\text{fold } \varphi \psi \hat{x} & := \varphi x \\
\text{fold } \varphi \psi (f \triangleright o \triangleleft g) & := \psi o(\text{fold } \varphi \psi f)(\text{fold } \varphi \psi g)
\end{align*}
\]

Compare: \( \sum l = (\text{fold } 0 \text{ plus}) l \)
Step Functions form a monad

$S$ is a monad similar to the reader monad $\lambda X. X^{[0,1]}$

$[\lambda x. fx \equiv x \mapsto fx]$
Step Functions form a monad

$S$ is a monad similar to the reader monad $\lambda X. X^{[0,1]}$

$[\lambda x. fx \equiv x \mapsto fx]$

unit is the constant function

map is defined in the obvious way

join from $S(SX)$ to $SX$ is the formal variant of the join function from the reader monad: $\text{join } fz := fzz$
Step Functions form a monad

$S$ is a monad similar to the reader monad $\lambda X. X^{[0,1]}$
$[\lambda x. fx \equiv x \mapsto fx]$

unit is the constant function
map is defined in the obvious way
join from $S(SX)$ to $SX$ is the formal variant of the join function from the reader monad: $\text{join } f z := f z z$

We use the applicative functor interface to Step Functions
Every monad defines an applicative functor
Applicative functors (McBride/Paterson)

Have: \( \text{map} : X \rightarrow Y \rightarrow TX \rightarrow TY \).

Also want \( \text{map}_n \) which takes \( n \) arguments

Enter: applicative functors:

\[ \widehat{-} : X \Rightarrow TX \]
\[ @ : T(X \Rightarrow Y) \Rightarrow TX \Rightarrow TY \]

(+ laws)

Lifting functions:

An infix operator \( \otimes \) of type \( X \Rightarrow Y \Rightarrow Z \) can be lifted to an operator \( \langle \otimes \rangle \) of type \( TX \Rightarrow TY \Rightarrow TZ \), then we define

\[ f \langle \otimes \rangle g := (\lambda x y. x \otimes y)f@g, \]

So, \( f \langle - \rangle g \) is the pointwise difference.
Given a relation: $\lambda xy. x \propto y : X \Rightarrow Y \Rightarrow \text{prop}$, we apply map2 and get: $\lambda fg. f \langle \propto \rangle g : SX \Rightarrow SY \Rightarrow S\text{prop}$. (Compare truth values in $\text{Sh}([0,1])$).
Lifting relations

Given a relation: \( \lambda xy. x \propto y : X \Rightarrow Y \Rightarrow \text{prop} \),
we apply map2 and get: \( \lambda fg.f \langle \propto \rangle g : S_X \Rightarrow S_Y \Rightarrow S_{\text{prop}} \).
(Compare truth values in \( \text{Sh}([0,1]) \)).
We obtain a real proposition:

\[
f \{ \propto \} g := \text{fold}_{\text{prop}} (f \langle \propto \rangle g)
\]

where

\[
\text{fold}_{\text{prop}} := \text{fold } I (\lambda opq.p \land q)
\]

\[
\text{fold}_{\text{prop}} : S_{\text{prop}} \rightarrow \text{prop} \text{ is } \land
\]
Lifting relations

Given a relation: $\lambda xy. x \propto y : X \Rightarrow Y \Rightarrow \text{prop}$, we apply map2 and get: $\lambda fg. f \langle \propto \rangle g : SX \Rightarrow SY \Rightarrow S\text{prop}$.
(Compare truth values in $\text{Sh}([0,1])$).

We obtain a real proposition:

$$f \{\propto\} g := \text{fold}_{\text{prop}}(f \langle \propto \rangle g)$$

where

$$\text{fold}_{\text{prop}} := \text{fold} I(\lambda opq. p \land q)$$

$\text{fold}_{\text{prop}} : S\text{prop} \rightarrow \text{prop}$ is $\land$

Pointwise equality:

$$f \propto_{\mathcal{S}X} g := f \{\propto_{\mathcal{S}}\} g$$

Pointwise inequality:

$$f \leq_{\mathcal{S}Q} g := f \{\leq_{\mathcal{S}}\} g$$
Two metric spaces of Step functions

We define two norms:

\[ \|f\|_\infty := \text{fold}_{\sup}(\text{abs } f) \]
\[ \|f\|_1 := \text{fold}_{\text{affine}}(\text{abs } f) \]

where

\[ \text{fold}_{\sup} := \text{fold } \lambda o x y. \sup x y \]
\[ \text{fold}_{\text{affine}} := \text{fold } \lambda o x y. o x + (1 - o) y \]
Two metric spaces of Step functions

The metrics:

\[ d^\infty(f, g) := \| f \langle - \rangle g \|_{\infty} \]
\[ d^1(f, g) := \| f \langle - \rangle g \|_{1} \]
Two metric spaces of Step functions

The metrics:

\[ d^\infty(f, g) := \|f \langle - \rangle g\|_\infty \]
\[ d^1(f, g) := \|f \langle - \rangle g\|_1 \]

The monad \( S^\infty \) is a submonad of \( S^1 \):

\[
B^{S^\infty X}_\varepsilon(f, g) := \text{fold}_\star(B^{X}_\varepsilon f @ g)
\]
\[
B^{S^1 X}_\varepsilon(f, g) := \exists h : S^Q. \text{fold}_{\text{prop}}(B^X h @ f @ g) \land \|h\| \leq \varepsilon
\]
Bounded Functions and Integrable Functions

\[ \mathcal{B}_\mathbb{Q} := C(S^\infty \mathbb{Q}) \]
\[ \mathcal{I}_\mathbb{Q} := C(S^1 \mathbb{Q}) \]

And the functions:

\[ \sup f := \text{map}_C \text{fold}_{\sup} f : C(S^\infty \mathbb{Q}) \to C\mathbb{Q} \]
\[ \int f := \text{map}_C \text{fold}_{\text{affine}} f : C(S^1 \mathbb{Q}) \to C\mathbb{Q} \]
Defining integration

Continuous function $\rightarrow$ Bounded function $\rightarrow$ integrable function

First step:

Given a uniformly continuous function $f$ and a step function $s_4$ that approximates the identity function, the step function map $fs_4$ (or $fs_4$) approximates $f$ in the familiar Riemann way.
Defining integration

Given: \( f : \mathbb{Q} \to \mathbb{Q} \). Define: \( \text{map } f : S^\infty \mathbb{Q} \to S^\infty \mathbb{Q} \).
Lift it again, \( \text{map}_C(\text{map}_{S^\infty} f) : B\mathbb{Q} \to B\mathbb{Q} \).
Applying this to \( I_{[0,1]} \) yields \( f|_{[0,1]} \) as a bounded function.
Defining integration

Given: \( f : \mathbb{Q} \rightarrow \mathbb{Q} \). Define: map \( f : S^\infty \mathbb{Q} \rightarrow S^\infty \mathbb{Q} \).
Lift it again, map\(_C\) (map\(_{S^\infty}\) \( f \)) : \mathcal{B}\mathbb{Q} \rightarrow \mathcal{B}\mathbb{Q}.
Applying this to \( I_{[0,1]} \) yields \( f\bigr|_{[0,1]} \) as a bounded function.
We have: \( \iota : S^\infty \mathbb{Q} \rightarrow S^1 \mathbb{Q} \), so map\(_C\) \( \iota : \mathcal{B}\mathbb{Q} \rightarrow \mathcal{I}\mathbb{Q} \).
Defining integration

Given: \( f : \mathbb{Q} \to \mathbb{Q} \). Define: map \( f : S^\infty \mathbb{Q} \to S^\infty \mathbb{Q} \).
Lift it again, \( \text{map}_C(\text{map}_{S^\infty} f) : \mathcal{B}\mathbb{Q} \to \mathcal{B}\mathbb{Q} \).
Applying this to \( I_{[0,1]} \) yields \( f|_{[0,1]} \) as a bounded function.
We have: \( \iota : S^\infty \mathbb{Q} \to S^1 \mathbb{Q} \), so \( \text{map}_C \iota : \mathcal{B}\mathbb{Q} \to \mathcal{I}\mathbb{Q} \).
We obtain an integrable function:

\[
(\text{map}_C \iota (\text{map}_C (\text{map}_{S^\infty} f) I_{[0,1]}))
\]

Which can be integrated:

\[
\int_{[0,1]} f := \int (\text{map}_C \iota (\text{map}_C (\text{map}_{S^\infty} f) I_{[0,1]}))
\]
Defining integration

Given: \( f : \mathbb{Q} \to \mathbb{Q} \). Define: map \( f : S^\infty \mathbb{Q} \to S^\infty \mathbb{Q} \).

Lift it again, map\(_C (\text{map}_{S^\infty} f) : \mathcal{B} \mathbb{Q} \to \mathcal{B} \mathbb{Q} \).

Applying this to \( I_{[0,1]} \) yields \( f|_{[0,1]} \) as a bounded function.

We have: \( \iota : S^\infty \mathbb{Q} \to S^1 \mathbb{Q} \), so map\(_C \iota : \mathcal{B} \mathbb{Q} \to \mathcal{I} \mathbb{Q} \).

We obtain an integrable function:

\[
(\text{map}_C \iota (\text{map}_C (\text{map}_{S^\infty} f) I_{[0,1]}))
\]

Which can be integrated:

\[
\int_{[0,1]} f := \int (\text{map}_C \iota (\text{map}_C (\text{map}_{S^\infty} f) I_{[0,1]}))
\]

For the general case we need a distributive law for monads:

\[
dist : SC\mathbb{Q} \Rightarrow CS\mathbb{Q}
\]
Defining integration

Given: \( f : \mathbb{Q} \to \mathbb{Q} \). Define: \( \text{map} \, f : S^\infty \mathbb{Q} \to S^\infty \mathbb{Q} \).
Lift it again, \( \text{map}_C(\text{map}_{S^\infty} f) : \mathcal{B} \mathbb{Q} \to \mathcal{B} \mathbb{Q} \).
Applying this to \( I_{[0,1]} \) yields \( f|_{[0,1]} \) as a bounded function.
We have: \( \iota : S^\infty \mathbb{Q} \to S^1 \mathbb{Q} \), so \( \text{map}_C \, \iota : \mathcal{B} \mathbb{Q} \to \mathcal{I} \mathbb{Q} \).
We obtain an integrable function:

\[
\left( \text{map}_C \, \iota(\text{map}_C(\text{map}_{S^\infty} f)I_{[0,1]}) \right)
\]

Which can be integrated:

\[
\int_{[0,1]} f := \int (\text{map}_C \, \iota(\text{map}_C(\text{map}_{S^\infty} f)I_{[0,1]})
\]

For the general case we need a distributive law for monads:
\( \text{dist} : SC\mathbb{Q} \Rightarrow CS\mathbb{Q} \equiv \mathcal{S} \mathcal{R} \Rightarrow \mathcal{B} \mathbb{Q} \)
Defining integration

Given: \( f : \mathbb{Q} \to \mathbb{Q} \). Define: map \( f : S^\infty \mathbb{Q} \to S^\infty \mathbb{Q} \).
Lift it again, map_\mathcal{C}(\text{map}_{S^\infty} f) : \mathcal{B} \mathbb{Q} \to \mathcal{B} \mathbb{Q}.
Applying this to \( I_{[0,1]} \) yields \( f|_{[0,1]} \) as a bounded function.
We have: \( \iota : S^\infty \mathbb{Q} \to S^1 \mathbb{Q} \), so \( \text{map}_\mathcal{C} \iota : \mathcal{B} \mathbb{Q} \to \mathcal{I} \mathbb{Q} \).
We obtain an integrable function:

\[
(\text{map}_\mathcal{C} \iota(\text{map}_\mathcal{C}(\text{map}_{S^\infty} f)I_{[0,1]}))
\]

Which can be integrated:

\[
\int_{[0,1]} f := \int (\text{map}_\mathcal{C} \iota(\text{map}_\mathcal{C}(\text{map}_{S^\infty} f)I_{[0,1]}))
\]

For the general case we need a distributive law for monads:
\( \text{dist} : S \mathcal{C} \mathbb{Q} \Rightarrow \mathcal{C} S \mathbb{Q} \equiv S \mathbb{R} \Rightarrow \mathcal{B} \mathbb{Q} \)
Surprise: directly generalizes to Stieltjes integral by varying \( I_{[0,1]} \).
Correctness

The integral is equivalent to the one in the CoRN library, a formalization of Bishop’s real analysis in the Coq type theory.
Correctness

The integral is equivalent to the one in the CoRN library, a formalization of Bishop’s real analysis in the Coq type theory.

DEMO

Four man months of development.

The program/proof consists of:

- 1155 lines of specifications
- 3380 lines of proof
- 170,137 total characters
The integral is equivalent to the one in the CoRN library, a formalization of Bishop’s real analysis in the Coq type theory.

DEMO

Four man months of development.

The program/proof consists of:

- 1155 lines of specifications
- 3380 lines of proof
- 170,137 total characters

Very general, but slow algorithm.

Straightforward speedups:

machine integers, binary rationals (Floats in Coq)
Correctness

The integral is equivalent to the one in the CoRN library, a formalization of Bishop’s real analysis in the Coq type theory.

DEMO

Four man months of development.

The program/proof consists of:

- 1155 lines of specifications
- 3380 lines of proof
- 170,137 total characters

Very general, but slow algorithm.

Straightforward speedups:

- machine integers, binary rationals (Floats in Coq)
- Parallelizing a single function gives a $3 \times$ speed up on a 4 proc. machine (Google:MapReduce)
Implementation issues:
Defining SplitL and SplitR such that:
SplitLfa ▷ a ◁ SplitRfa is equivalent to f.
Proving that all operations (Map, etc) respect equivalence on step functions (difficult!)
Implementation issues:
Defining $\text{SplitL}$ and $\text{SplitR}$ such that:
$\text{SplitL} f a \triangleright a \triangleleft \text{SplitR} f a$ is equivalent to $f$.
Proving that all operations (Map, etc) respect equivalence on step functions (difficult!)

Developing a ‘view’, a double induction principle, which allows to work as if two step functions have a common partition.
Reasoning about binders (e.g. $\lambda x.fx$) is difficult. Avoid this using combinator presentation of the lambda calculus.

- $B f g x := f (g x)$ (compose)
- $C f x y := f y x$ (interchange)
- $I x := x$ (identity)
- $K x y := x$ (discard)
- $W f x := f x x$ (duplicate)
Reasoning about binders (e.g. \( \lambda x. fx \)) is difficult.
Avoid this using combinator presentation of the lambda calculus.

- \( B f g x := f(gx) \) (compose)
- \( C f x y := fyx \) (interchange)
- \( I x := x \) (identity)
- \( K x y := x \) (discard)
- \( W f x := fxx \) (duplicate)

The combinators \( B \) and \( I \) are preserved by every applicative functor. For the applicative functor \( S \), all of the combinators are preserved. (This means that we can lift any function definable with the \( \lambda \)-calculus to step functions.)

\[
\begin{align*}
C f @ x @ y & \approx_{S X} f @ y @ x \\
K x @ y & \approx_{S X} x \\
W f @ x & \approx_{S X} f @ x @ x
\end{align*}
\]
Lifting theorems

We want: \( \forall fgh : S \mathbb{Q}. f \{ \leq \mathbb{Q} \} g \Rightarrow g \{ \leq \mathbb{Q} \} h \Rightarrow f \{ \leq \mathbb{Q} \} h \)

We have: \( \forall xyz : \mathbb{Q}. x \leq \mathbb{Q} y \Rightarrow y \leq \mathbb{Q} z \Rightarrow x \leq \mathbb{Q} z \)
Lifting theorems

We want: \( \forall fgh : S \mathbb{Q}. f \{\leq \mathbb{Q}\} g \Rightarrow g \{\leq \mathbb{Q}\} h \Rightarrow f \{\leq \mathbb{Q}\} h \)

We have: \( \forall xyz : \mathbb{Q}. x \leq \mathbb{Q} y \Rightarrow y \leq \mathbb{Q} z \Rightarrow x \leq \mathbb{Q} z \)

We apply:

\[
(\forall (x : X) (y : Y) (z : Z). Rxyz) \Rightarrow \forall (f : SX) (g : SY) (h : SZ). \text{fold}_{\text{prop}}(Rf @ g @ h)
\]

To get:

\( \forall fgh : S \mathbb{Q}. \text{fold}_{\text{prop}}( (\lambda xyz. x \leq \mathbb{Q} y \Rightarrow y \leq \mathbb{Q} z \Rightarrow x \leq \mathbb{Q} z) f @ g @ h ). \)
Lifting theorems

We want: \( \forall fgh : S \mathbb{Q}. f \{ \leq \mathbb{Q} \} g \Rightarrow g \{ \leq \mathbb{Q} \} h \Rightarrow f \{ \leq \mathbb{Q} \} h \)

We have: \( \forall xyz : \mathbb{Q}. x \leq \mathbb{Q} y \Rightarrow y \leq \mathbb{Q} z \Rightarrow x \leq \mathbb{Q} z \)

We apply:

\[
(\forall (x : X)(y : Y)(z : Z).R_{xyz}) \Rightarrow \forall (f : SX)(g : SY)(h : SZ). \text{fold}_\text{prop}(Rf@g@h)
\]

To get:

\( \forall fgh : S \mathbb{Q}. \text{fold}_\text{prop}((\lambda xyz. x \leq \mathbb{Q} y \Rightarrow y \leq \mathbb{Q} z \Rightarrow x \leq \mathbb{Q} z)f@g@h). \)

In combinator form:

\[
S(BS(B(B(B(\Rightarrow))(\leq \mathbb{Q}))))(B(C(BS(B(B(\Rightarrow))(\leq \mathbb{Q}))))(\leq \mathbb{Q}))f@g@h
\]
Lifting theorems

We want: \( \forall fgh : S \subseteq Q. f \{ \leq Q \} g \Rightarrow g \{ \leq Q \} h \Rightarrow f \{ \leq Q \} h \)

We have: \( \forall xyz : Q. x \leq Q y \Rightarrow y \leq Q z \Rightarrow x \leq Q z \)

We apply:

\[
(\forall (x : X)(y : Y)(z : Z).Rxyz) \Rightarrow \forall (f : SX)(g : SY)(h : SZ). foldprop(\lambda f @ g @ h).
\]

To get:

\( \forall fgh : S \subseteq Q. foldprop((\lambda xyz. x \leq Q y \Rightarrow y \leq Q z \Rightarrow x \leq Q z)f @ g @ h). \)

In combinator form:

\[
S(BS(B(B(\Rightarrow)))(\leq Q)))(B(C(BS(B(B(\Rightarrow)))(\leq Q)))))(\leq Q)f @ g @ h
\]

Evaluation gives:

\( \forall fgh : S \subseteq Q. foldprop(f \{ \leq Q \} g \{ \Rightarrow \} g \{ \leq Q \} h \{ \Rightarrow \} f \{ \leq Q \} h). \)
Lifting theorems

We want: \( \forall fgh : S \mathcal{Q}. f \{ \leq \mathcal{Q} \} g \Rightarrow g \{ \leq \mathcal{Q} \} h \Rightarrow f \{ \leq \mathcal{Q} \} h \)

We have: \( \forall xyz : \mathcal{Q}. x \leq \mathcal{Q} y \Rightarrow y \leq \mathcal{Q} z \Rightarrow x \leq \mathcal{Q} z \)

We apply:

\[
(\forall (x : X)(y : Y)(z : Z). Rxyz) \Rightarrow \forall (f : SX)(g : SY)(h : SZ). \text{fold}_\text{prop}(Rf@g@h)
\]

To get:

\( \forall fgh : S \mathcal{Q}. \text{fold}_\text{prop}((\lambda xyz. x \leq \mathcal{Q} y \Rightarrow y \leq \mathcal{Q} z \Rightarrow x \leq \mathcal{Q} z) f@g@h). \)

In combinator form:

\[
S(BS(B(B(B(B(\Rightarrow))(\leq \mathcal{Q})))))(B(C(BS(B(B(\Rightarrow))(\leq \mathcal{Q})))))(\leq \mathcal{Q}))(\leq \mathcal{Q}))f@g@h
\]

Evaluation gives:

\[
\forall fgh : S \mathcal{Q}. \text{fold}_\text{prop}(f \langle \leq \mathcal{Q} \rangle g \langle \Rightarrow \rangle g \langle \leq \mathcal{Q} \rangle h \langle \Rightarrow \rangle f \langle \leq \mathcal{Q} \rangle h).
\]

We obtain our goal by repeatedly using:

\[
\forall PQ : S \text{prop}.(\text{fold}_\text{prop}(P \langle \Rightarrow \rangle Q)) \Rightarrow \text{fold}_\text{prop} P \Rightarrow \text{fold}_\text{prop} Q
\]
Reflection on Bishop’s program

Bishop ’67:

As written, this book is person-oriented rather than computer-oriented. It would be of great interest to have a computer-oriented version...

Here we made some first steps
Reflection on Bishop’s program

Bishop '67:

As written, this book is person-oriented rather than computer-oriented. It would be of great interest to have a computer-oriented version...

Here we made some first steps

‘It is clear that many results in this book could be programmed for a computer...[set theory] resist computerization’ But: setoids!
Reflection on Bishop’s program

Bishop ’67:

*As written, this book is person-oriented rather than computer-oriented. It would be of great interest to have a computer-oriented version...*

Here we made some first steps

‘It is clear that many results in this book could be programmed for a computer...[set theory] resist computerization’ But: setoids!

‘Other parts [soft analysis] need to be rewritten.’

Coquand/S made some first steps:

application of formal topology/Coquand-Lombardi Hilbert program to functional analysis: Gelfand duality, measure theory, Fourier analysis
Reflection on Bishop’s program

Bishop ’67:

As written, this book is person-oriented rather than computer-oriented. It would be of great interest to have a computer-oriented version...

Here we made some first steps
‘It is clear that many results in this book could be programmed for a computer...[set theory] resist computerization’ But: setoids!
‘Other parts [soft analysis] need to be rewritten.’

Coquand/S made some first steps:
application of formal topology/Coquand-Lombardi Hilbert program to functional analysis: Gelfand duality, measure theory, Fourier analysis

Accident: useful in mathematical physics!
Reflection on Bishop’s program

Bishop ’67:

As written, this book is person-oriented rather than computer-oriented. It would be of great interest to have a computer-oriented version...

Here we made some first steps
‘It is clear that many results in this book could be programmed for a computer...[set theory] resist computerization’ But: setoids!
‘Other parts [soft analysis] need to be rewritten.’

Coquand/S made some first steps:
application of formal topology/Coquand-Lombardi Hilbert program to functional analysis: Gelfand duality, measure theory, Fourier analysis

Accident: useful in mathematical physics!
Also: Bauer/Taylor programming in ASD
Conclusion

Use Bishop’s program in type theory to give a provably correct implementation of exact analysis for

- Provably correct implementation of computations with continuous structures (future of numerics!??)
- Computer ‘referees’ for computer proofs
References

- O’Connor, Spitters - A computer verified, monadic, functional implementation of the integral
- O’Connor - A Monadic, Functional Implementation of Real Numbers
- O’Connor - Certified Exact Transcendental Real Number Computation in Coq
- O’Connor - A Computer Verified Theory of Compact Sets
- Geuvers, Niqui, Spitters and Wiedijk - Constructive analysis, types and exact real numbers
- Cruz-Filipe, Spitters - Program extraction from large proof developments
Tutorial: Computer Verified Implementation of Analysis

III : Compact Sets

Bas Spitters & Russell O’Connor
Radboud University Nijmegen

Computability and Complexity in Analysis
Ljubljana, Slovenia.
2009-08-18
A Metric on List of Points
Find Closest Green Points
Pick the Longest One
This is not Symmetric
Take the Longest One
Metric Space of Finite Sets

\[ H_\varepsilon S T := \forall x \in S. \exists \varepsilon x \in T. B_\varepsilon x y \]

\[ B_\varepsilon S T := H_\varepsilon S T \land H_\varepsilon T S \]

- The classical exists is essential for proving the closedness property.
  - (closed) \( \forall \varepsilon : \mathbb{Q}^+. \forall x y : X. (\forall \delta : \mathbb{Q}^+. B_{\delta + \varepsilon} x y) \Rightarrow B_\varepsilon x y \)
Compact Sets

- Completing finite sets of $X$ yields the compact sets of the $C_X$
  - Example: $K(\mathbb{Q} \times \mathbb{Q})$ are the compact sets of the plane, $C(\mathbb{Q} \times \mathbb{Q})$ (aka $C\mathbb{Q} \times C\mathbb{Q}$)
  - The graph of uniformly continuous functions over compact intervals are compact
  - Finite approximations can be adjusted and plotted on a raster
Provably Correct Graph

ball (m:=Compact stableQ2) (324 # 2592)

graphCR (boundBelow 0 ◦ boundAbove 3 ◦ exp_bound_uc 1) [- (6) .. 1]
(Cunit

(- (6), 3)ᶠ

(1, 0))
Riemann Hypothesis

- Is there a program that
  - outputs 0 if the Riemann Hypothesis is True
  - outputs 1 if the Riemann Hypothesis is False
We do not know yet how to constructively prove

RH ∨ ¬RH
Filled Julia Sets

- Is every quadratic filled Julia set computable?
Yes, but there exists quadratic filled Julia sets that we do not know how to draw.
If Braverman had used constructive logic he would have only claimed:

- A quadratic filled julia set is compact when
  - The polynomial has an attracting orbit or
  - The polynomial has a parabolic orbit or
  - The polynomial has a Siegel orbit or
  - The polynomial has a Cremer orbit or
  - All the orbits of the polynomial are repelling
Further Reading

- Certified Exact Transcendental Real Number Computation in Coq.
  - TPHOLs 2008 proceedings
  - Online (right now, CoRR abs/0805.2438)

- A Computer Verified Theory of Compact Sets
  - SCSS 2008 proceedings
  - Online (CoRR abs/0806.3209)