Extracting concurrent programs from proofs

Ulrich Berger
Swansea University

j.w.w.

Hideki Tsuiki
Kyoto University

Workshop in Honour of Thierry Coquand's 60th Birthday
Gothenburg, Sweden, 24th - 26th August 2022

In this talk I will report about a recent branch of the Minlog project which tries to capture concurrency through program extraction.

In this talk I will report about a recent branch of the Minlog project which tries to capture concurrency through program extraction.

Thierry Coquand's work had a big influence. I will highlight this in the talk.

Overview

1. Background: Program extraction from proofs
2. Intuitionistic Fixed Point Logic
3. Programs and their semantics
4. McCarthy's Amb
5. Concurrent Fixed Point Logic
6. Applications: Infinite Gray-code and Gaussian elimination
7. Implementation
8. Conclusion

Program extraction from proofs via realizability

Kleene (1945) defined the relation

$$e \,\mathsf{r}\, A \qquad (\text{`}e \text{ realizes } A\text{'})$$

where e is a natural number (a code of a Turing machine) and A is an arithmetical formula. He showed:

Program extraction from proofs via realizability

Kleene (1945) defined the relation

$$e \,\mathsf{r}\, A \qquad (\text{`}e \text{ realizes } A\text{'})$$

where e is a natural number (a code of a Turing machine) and A is an arithmetical formula. He showed:

Soundness Theorem. From an intuitionistic proof of A one can extract a realizer of A.

Program extraction from proofs via realizability

Kleene (1945) defined the relation

$$e \,\mathsf{r}\, A \qquad (\text{`}e \text{ realizes } A\text{'})$$

where e is a natural number (a code of a Turing machine) and A is an arithmetical formula. He showed:

Soundness Theorem. From an intuitionistic proof of A one can extract a realizer of A.

Kreisel (1959) extended this to analysis (quantification over functions) and used total continuous functionals of higher types as realizers (instead of numbers).

Program extraction from proofs via realizability

Kleene (1945) defined the relation

$$e \,\mathsf{r}\, A \qquad (\text{`}e \text{ realizes } A\text{'})$$

where e is a natural number (a code of a Turing machine) and A is an arithmetical formula. He showed:

Soundness Theorem. From an intuitionistic proof of A one can extract a realizer of A.

Kreisel (1959) extended this to analysis (quantification over functions) and used total continuous functionals of higher types as realizers (instead of numbers).

To every formula A a type $\tau(A)$ of 'potential realizers' is assigned, and the Soundness Theorem becomes:

Program Extraction Theorem. From an intuitionistic proofs of a formula A one can extract a program that denotes a functional of type $\tau(A)$ realizing A.

Program extraction in computer science

Also known as ‘Curry-Howard correspondence’.

Implemented in Nuprl, Coq, Isabelle/HOL, Agda, Minlog.

Applies to constructive systems with inductive and coinductive definitions (least and greatest fixed points).

Program extraction in computer science

Also known as 'Curry-Howard correspondence'.

Implemented in Nuprl, Coq, Isabelle/HOL, Agda, Minlog.

Applies to constructive systems with inductive and coinductive definitions (least and greatest fixed points).

Some case studies in Minlog:

- Dijkstra's algorithm (Benl, Schwichtenberg 1997)
- Higman's Lemma (Seisenberger 2008)
- Normalization by evaluation for the simply typed λ-calculus (Schwichtenberg/B 2006)
  [Also Berghofer (Isabelle), Letouzey (Coq)]
- In-place Quicksort (Seisenberger, Woods, B)
- DPLL SAT-solver (Forsberg, Lawrence, Seisenberger, 2015)
- Computable analysis (Koepp, Schwichtenberg, 2022)

Strengths and weaknesses of program extraction

Strengths and weaknesses of program extraction

+

Strengths and weaknesses of program extraction

+

- Specification and proof take place in an abstract logical system which is easy to understand and technically easy to handle.

Strengths and weaknesses of program extraction

+

- Specification and proof take place in an abstract logical system which is easy to understand and technically easy to handle.
- No formalization of or reasoning about programs required.

Strengths and weaknesses of program extraction

+

- Specification and proof take place in an abstract logical system which is easy to understand and technically easy to handle.
- No formalization of or reasoning about programs required.
- Program, data, and correctness proof generated automatically.

Strengths and weaknesses of program extraction

\+

- Specification and proof take place in an abstract logical system which is easy to understand and technically easy to handle.
- No formalization of or reasoning about programs required.
- Program, data, and correctness proof generated automatically.

\−

Strengths and weaknesses of program extraction

+

- Specification and proof take place in an abstract logical system which is easy to understand and technically easy to handle.
- No formalization of or reasoning about programs required.
- Program, data, and correctness proof generated automatically.

−

- Extracted programs are necessarily purely functional (essentially core Haskell).

logic	typed lambda-calculus
induction	recursion

Strengths and weaknesses of program extraction

+

- Specification and proof take place in an abstract logical system which is easy to understand and technically easy to handle.
- No formalization of or reasoning about programs required.
- Program, data, and correctness proof generated automatically.

−

- Extracted programs are necessarily purely functional (essentially core Haskell).

logic	typed lambda-calculus
induction	recursion

- Other programming paradigms are not covered.

Strengths and weaknesses of program extraction

\+

- Specification and proof take place in an abstract logical system which is easy to understand and technically easy to handle.
- No formalization of or reasoning about programs required.
- Program, data, and correctness proof generated automatically.

\−

- Extracted programs are necessarily purely functional (essentially core Haskell).

logic	typed lambda-calculus
induction	recursion

- Other programming paradigms are not covered.

?	imperative programming
?	nondeterminism/concurrency
?	memory management

Intuitionistic Fixed Point logic (IFP)

B, Tsuiki (2021), Annals of Pure and Applied Logic 172*(3)*

`https://arxiv.org/abs/2002.00188`

Intuitionistic Fixed Point logic (IFP)

B, Tsuiki (2021), Annals of Pure and Applied Logic 172*(3)*
`https://arxiv.org/abs/2002.00188`

- Intuitionistic first-order logic with equality.
- Extra constants, function symbols and atomic predicates (not necessarily decidable), depending on applications.
- Free predicate variables $X, Y, \ldots$.
- Inductive and coinductive definitions as least and largest fixed points of strictly positive (s.p.) operators.
- Extra axioms consisting of *non-computational (nc)*, that is, disjunction-free, formulas (depending on applications).
- For the classically minded user it suffices for these extra axioms to be classically true in the intended model.

Intuitionistic Fixed Point logic (IFP)

B, Tsuiki (2021), Annals of Pure and Applied Logic 172*(3)*
`https://arxiv.org/abs/2002.00188`

- Intuitionistic first-order logic with equality.
- Extra constants, function symbols and atomic predicates (not necessarily decidable), depending on applications.
- Free predicate variables $X, Y, \ldots$.
- Inductive and coinductive definitions as least and largest fixed points of strictly positive (s.p.) operators.
- Extra axioms consisting of *non-computational (nc)*, that is, disjunction-free, formulas (depending on applications).
- For the classically minded user it suffices for these extra axioms to be classically true in the intended model.

The rational for this system is to stay as close as possible to the axiomatic style common in mathematics while still being able to extract useful computational content from proofs.

Example: Real and natural numbers

- Variables $x, y, \ldots$ are intended to range over abstract real numbers
- Constants and function symbols: $0, 1, +, -, *, /, |\cdot|, \ldots$.
- Atomic predicates: $<, \leq, \ldots$.
- Non computational ($\vee$-free) axioms: $\forall x \,.\, x + 0 = x$, $\ldots$.

Example: Real and natural numbers

- Variables $x, y, \ldots$ are intended to range over abstract real numbers
- Constants and function symbols: $0, 1, +, -, *, /, |\cdot|, \ldots$.
- Atomic predicates: $<, \leq, \ldots$.
- Non computational ($\vee$-free) axioms: $\forall x \,.\, x + 0 = x$, $\ldots$.
- Inductive predicate defining the natural numbers as a subset of the reals numbers: $\mathsf{N} \stackrel{\text{Def}}{=} \mu\, \Phi$, where $\Phi = \lambda X\, \lambda x \,.\, x = 0 \vee X(x - 1)$.
  We write this more intuitively as $\mathsf{N}(x) \stackrel{\mu}{=} x = 0 \vee \mathsf{N}(x - 1)$.

Example: Real and natural numbers

- Variables $x, y, \ldots$ are intended to range over abstract real numbers
- Constants and function symbols: $0, 1, +, -, *, /, |\cdot|, \ldots$.
- Atomic predicates: $<, \leq, \ldots$.
- Non computational ($\vee$-free) axioms: $\forall x \,.\, x + 0 = x$, $\ldots$.
- Inductive predicate defining the natural numbers as a subset of the reals numbers: $\mathsf{N} \stackrel{\text{Def}}{=} \mu\, \Phi$, where $\Phi = \lambda X\, \lambda x \,.\, x = 0 \vee X(x - 1)$.
  We write this more intuitively as $\mathsf{N}(x) \stackrel{\mu}{=} x = 0 \vee \mathsf{N}(x - 1)$.
- $\mathsf{Q}(x) \stackrel{\text{Def}}{=} \exists n, m, k \in \mathsf{N} \,.\, k > 0 \wedge x = (n - m)/k$

Example: Real and natural numbers

- Variables $x, y, \ldots$ are intended to range over abstract real numbers
- Constants and function symbols: $0, 1, +, -, *, /, |\cdot|, \ldots$.
- Atomic predicates: $<, \leq, \ldots$.
- Non computational ($\vee$-free) axioms: $\forall x \,.\, x + 0 = x$, $\ldots$.
- Inductive predicate defining the natural numbers as a subset of the reals numbers: $\mathsf{N} \stackrel{\text{Def}}{=} \mu\, \Phi$, where $\Phi = \lambda X\, \lambda x \,.\, x = 0 \vee X(x-1)$.
  We write this more intuitively as $\mathsf{N}(x) \stackrel{\mu}{=} x = 0 \vee \mathsf{N}(x-1)$.
- $\mathsf{Q}(x) \stackrel{\text{Def}}{=} \exists n, m, k \in \mathsf{N} \,.\, k > 0 \wedge x = (n-m)/k$
- Coinductive predicate defining those real numbers that can be approximated by rationals (i.e. the closure of Q): $\mathsf{A} \stackrel{\text{Def}}{=} \nu\, \Psi$, where $\Psi = \lambda X\, \lambda x \,.\, \exists q \in \mathsf{Q}\, |x - q| \leq 1 \wedge X(2x)$.
  Intuitive notation $\mathsf{A}(x) \stackrel{\nu}{=} \exists q \in \mathsf{Q}\, |x - q| \leq 1 \wedge \mathsf{A}(2x)$.

Example: Real and natural numbers

- Variables $x, y, \ldots$ are intended to range over abstract real numbers
- Constants and function symbols: $0, 1, +, -, *, /, |\cdot|, \ldots$.
- Atomic predicates: $<, \leq, \ldots$.
- Non computational ($\vee$-free) axioms: $\forall x \,.\, x + 0 = x$, $\ldots$.
- Inductive predicate defining the natural numbers as a subset of the reals numbers: $\mathsf{N} \stackrel{\text{Def}}{=} \mu\, \Phi$, where $\Phi = \lambda X\, \lambda x \,.\, x = 0 \vee X(x-1)$.
  We write this more intuitively as $\mathsf{N}(x) \stackrel{\mu}{=} x = 0 \vee \mathsf{N}(x-1)$.
- $\mathsf{Q}(x) \stackrel{\text{Def}}{=} \exists n, m, k \in \mathsf{N} \,.\, k > 0 \wedge x = (n-m)/k$
- Coinductive predicate defining those real numbers that can be approximated by rationals (i.e. the closure of Q): $\mathsf{A} \stackrel{\text{Def}}{=} \nu\, \Psi$, where $\Psi = \lambda X\, \lambda x \,.\, \exists q \in \mathsf{Q}\, |x - q| \leq 1 \wedge X(2x)$.
  Intuitive notation $\mathsf{A}(x) \stackrel{\nu}{=} \exists q \in \mathsf{Q}\, |x - q| \leq 1 \wedge \mathsf{A}(2x)$.

One can prove $\mathsf{A}(x) \leftrightarrow \forall k \in \mathsf{N}\, \exists q \in \mathsf{Q}\, |x - q| \leq 2^{-k}$.

Realizability

The realizability interpretation of a formula A is a unary predicate $\mathrm{R}(A)$ whose argument ranges over a Scott domain D defined by a recursive domain equation.

Realizability

The realizability interpretation of a formula A is a unary predicate $\mathrm{R}(A)$ whose argument ranges over a Scott domain D defined by a recursive domain equation.

$\mathrm{R}(A)(a)$ is usually written $a \,\mathsf{r}\, A$ ('a realizes A').

Realizability

The realizability interpretation of a formula A is a unary predicate $\mathrm{R}(A)$ whose argument ranges over a Scott domain D defined by a recursive domain equation.

$\mathrm{R}(A)(a)$ is usually written $a \,\mathsf{r}\, A$ ('a realizes A').

The definition of $a \,\mathsf{r}\, A$ is largely as expected, e.g.,

$$\begin{aligned} c \,\mathsf{r}\, (A \vee B) &= \exists a\, (c = \mathrm{L}(a) \wedge a \,\mathsf{r}\, A) \vee \exists b\, (c = \mathrm{R}(b) \wedge b \,\mathsf{r}\, B) \\ f \,\mathsf{r}\, (A \to B) &= \forall a\, (a \,\mathsf{r}\, A \to (f\ a) \,\mathsf{r}\, B) \end{aligned}$$

Realizability

The realizability interpretation of a formula A is a unary predicate $\mathrm{R}(A)$ whose argument ranges over a Scott domain D defined by a recursive domain equation.

$\mathrm{R}(A)(a)$ is usually written $a \,\mathsf{r}\, A$ ('a realizes A').

The definition of $a \,\mathsf{r}\, A$ is largely as expected, e.g.,

$$\begin{array}{rcl} c \,\mathsf{r}\, (A \vee B) & = & \exists a\,(c = \mathrm{L}(a) \wedge a \,\mathsf{r}\, A) \vee \exists b\,(c = \mathrm{R}(b) \wedge b \,\mathsf{r}\, B) \\ f \,\mathsf{r}\, (A \to B) & = & \forall a\,(a \,\mathsf{r}\, A \to (f\ a) \,\mathsf{r}\, B) \end{array}$$

But:

$$b \,\mathsf{r}\, (A \to B) \quad = \quad A \to b \,\mathsf{r}\, B \quad \text{if } A \text{ nc, i.e. } \vee\text{-free}$$

Realizability

The realizability interpretation of a formula A is a unary predicate $\mathrm{R}(A)$ whose argument ranges over a Scott domain D defined by a recursive domain equation.

$\mathrm{R}(A)(a)$ is usually written $a \,\mathsf{r}\, A$ ('a realizes A').

The definition of $a \,\mathsf{r}\, A$ is largely as expected, e.g.,

$$\begin{aligned} c \,\mathsf{r}\, (A \vee B) &= \exists a\,(c = \mathrm{L}(a) \wedge a \,\mathsf{r}\, A) \vee \exists b\,(c = \mathrm{R}(b) \wedge b \,\mathsf{r}\, B) \\ f \,\mathsf{r}\, (A \to B) &= \forall a\,(a \,\mathsf{r}\, A \to (f\ a) \,\mathsf{r}\, B) \end{aligned}$$

But:

$$\begin{aligned} b \,\mathsf{r}\, (A \to B) &= A \to b \,\mathsf{r}\, B \quad \text{if } A \text{ nc, i.e. } \vee\text{-free} \\ a \,\mathsf{r}\, \forall x\, A &= \forall x\,(a \,\mathsf{r}\, A) \\ a \,\mathsf{r}\, \exists x\, A &= \exists x\,(a \,\mathsf{r}\, A) \end{aligned}$$

Realizability

The realizability interpretation of a formula A is a unary predicate $\mathrm{R}(A)$ whose argument ranges over a Scott domain D defined by a recursive domain equation.

$\mathrm{R}(A)(a)$ is usually written $a \,\mathsf{r}\, A$ ('a realizes A').

The definition of $a \,\mathsf{r}\, A$ is largely as expected, e.g.,

$$\begin{aligned} c \,\mathsf{r}\, (A \vee B) &= \exists a\,(c = \mathrm{L}(a) \wedge a \,\mathsf{r}\, A) \vee \exists b\,(c = \mathrm{R}(b) \wedge b \,\mathsf{r}\, B) \\ f \,\mathsf{r}\, (A \to B) &= \forall a\,(a \,\mathsf{r}\, A \to (f\ a) \,\mathsf{r}\, B) \end{aligned}$$

But:

$$\begin{aligned} b \,\mathsf{r}\, (A \to B) &= A \to b \,\mathsf{r}\, B \quad \text{if } A \text{ nc, i.e. } \vee\text{-free} \\ a \,\mathsf{r}\, \forall x\, A &= \forall x\,(a \,\mathsf{r}\, A) \\ a \,\mathsf{r}\, \exists x\, A &= \exists x\,(a \,\mathsf{r}\, A) \end{aligned}$$

The 'uniform' interpretation of quantifiers allows us to interpret this system in abstract (not necessarily constructively given) mathematical structures.

Programs and their semantics

Programs and their semantics

Elements of D are denoted by programs, that is, terms of a type-free lambda calculus with constructors, case-terms, and recursion (essentially, a fragment of Haskell).

Programs and their semantics

Elements of D are denoted by programs, that is, terms of a type-free lambda calculus with constructors, case-terms, and recursion (essentially, a fragment of Haskell).

Programs, M, have a

- *denotational* semantics, $[\![M]\!] \in D$, and a
- *big-step operational* semantics, $M \Downarrow M'$.

which are linked through a *Computational Adequacy Theorem*:

Programs and their semantics

Elements of D are denoted by programs, that is, terms of a type-free lambda calculus with constructors, case-terms, and recursion (essentially, a fragment of Haskell).

Programs, M, have a

- *denotational* semantics, $[\![M]\!] \in D$, and a
- *big-step operational* semantics, $M \Downarrow M'$.

which are linked through a *Computational Adequacy Theorem*:

If $[\![M]\!] = C(-)$, then $M \Downarrow C(-)$.

Programs and their semantics

Elements of D are denoted by programs, that is, terms of a type-free lambda calculus with constructors, case-terms, and recursion (essentially, a fragment of Haskell).

Programs, M, have a

- *denotational* semantics, $[\![M]\!] \in D$, and a
- *big-step operational* semantics, $M \Downarrow M'$.

which are linked through a *Computational Adequacy Theorem*:

If $[\![M]\!] = C(-)$, then $M \Downarrow C(-)$.

Proof: Define for each compact domain element a, a set of programs $\mathrm{pr}(a)$, by recursion on the rank of a (in the style of logical relations).

Show that $a \sqsubseteq [\![M]\!]$ implies $[\![M]\!] \in \mathrm{pr}(a)$, by structural induction on M.

By the definition of $\mathrm{pr}(a)$, the result follows.

Tait's strong normalization proof for Scott domains

The proof of the Adequacy Theorem can be viewed as a domain-theoretic version of Tait's proof of SN for the simply typed lambda-calculus:

Tait's strong normalization proof for Scott domains

The proof of the Adequacy Theorem can be viewed as a domain-theoretic version of Tait's proof of SN for the simply typed lambda-calculus:

$$'M \in \mathrm{pr}(a)'$$

corresponds to Tait's

$$'M \text{ is strongly computable of type } a'$$

where in Tait's proof, a ranges over simple types.

Tait's strong normalization proof for Scott domains

The proof of the Adequacy Theorem can be viewed as a domain-theoretic version of Tait's proof of SN for the simply typed lambda-calculus:

$$'M \in \mathrm{pr}(a)'$$

corresponds to Tait's

$$'M \text{ is strongly computable of type } a'$$

where in Tait's proof, a ranges over simple types.

The idea, to replace simple types by compact domain elements, is taken from the paper

T. Coquand, A. Spiwack (2006)
A proof of strong normalisation using domain theory
LICS'06, 307-316.

which in turn is built on Martin-Loef's domain interpretation of type theory.

McCarthy's Amb (1963)

McCarthy's Amb (1963)

Amb(M, N) terminates iff M or N terminates, and in that case the value of one of the terminating programs is returned (angelic nondeterminism).

McCarthy's Amb (1963)

Amb(M, N) terminates iff M or N terminates, and in that case the value of one of the terminating programs is returned (angelic nondeterminism).

Let A be a (typically undecidable) nc condition and M, N programs such that:

- If A holds, then M terminates and realizes B.
- If $\neg A$ holds, then N terminates and realizes B

McCarthy's Amb (1963)

Amb(M, N) terminates iff M or N terminates, and in that case the value of one of the terminating programs is returned (angelic nondeterminism).

Let A be a (typically undecidable) nc condition and M, N programs such that:

- If A holds, then M terminates and realizes B.
- If $\neg A$ holds, then N terminates and realizes B

With classical logic, it should follow that Amb(M, N) realizes B. In other words, Amb should realize the rule

$$\frac{A \to B \quad \neg A \to B}{\Downarrow\!\Downarrow(B)}$$

where $\Downarrow\!\Downarrow(B)$ indicates that B may be realized by a potentially nondeterministic concurrent program.

McCarthy's Amb (1963)

Amb(M, N) terminates iff M or N terminates, and in that case the value of one of the terminating programs is returned (angelic nondeterminism).

Let A be a (typically undecidable) nc condition and M, N programs such that:

- If A holds, then M terminates and realizes B.
- If $\neg A$ holds, then N terminates and realizes B

With classical logic, it should follow that Amb(M, N) realizes B.
In other words, Amb should realize the rule

$$\frac{A \to B \quad \neg A \to B}{\Downarrow\!\Downarrow(B)}$$

where $\Downarrow\!\Downarrow(B)$ indicates that B may be realized by a potentially nondeterministic concurrent program.

Unfortunately not: M may terminate even if A does not hold.
In that case, Amb(M, N) may deliver an incorrect result.

Concurrent Fixed Point logic (CFP)

Concurrent Fixed Point logic (CFP)

B, Tsuiki (2022): Extracting total Amb programs from proofs
ESOP'22, LNCS 13240, 85-113
`https://arxiv.org/abs/2104.14669`

Concurrent Fixed Point logic (CFP)

B, Tsuiki (2022): Extracting total Amb programs from proofs
ESOP'22, LNCS 13240, 85-113
`https://arxiv.org/abs/2104.14669`

We extend IFP by a unary propositional operator $\downdownarrows$, and define its realizablity by

$$\begin{aligned}\mathsf{Amb}(a,b)\,\mathsf{r}\downdownarrows(B) \quad = \quad & (\mathsf{Def}(a)\vee\mathsf{Def}(b)) \ \wedge \\ & (\mathsf{Def}(a)\to a\,\mathsf{r}\,B)\wedge(\mathsf{Def}(b)\to b\,\mathsf{r}\,B)\end{aligned}$$

where $\mathsf{Def}(a)$ means $a\neq\bot$.

Restriction $B|_A$

To repair the failure of the rule

$$\frac{A \to B \quad \neg A \to B}{\Downarrow\!(B)}\ (A \text{ nc})$$

we introduce a variant, $B|_A$, of the implication, $A \to B$,
where A is required to be nc, and B is required to be *strict*, that is,
B must not be realized by $\bot$.

Restriction $B|_A$

To repair the failure of the rule

$$\frac{A \to B \quad \neg A \to B}{\Downarrow\!(B)}\ (A \text{ nc})$$

we introduce a variant, $B|_A$, of the implication, $A \to B$,
where A is required to be nc, and B is required to be *strict*, that is,
B must not be realized by $\bot$.

We realize restriction as

$$a\,\mathsf{r}\,(B|_A) \quad \stackrel{\text{Def}}{=} \quad (A \to \mathrm{Def}(a)) \wedge (\mathrm{Def}(a) \to a\,\mathsf{r}\,B)$$

Restriction $B|_A$

To repair the failure of the rule

$$\frac{A \to B \quad \neg A \to B}{\Downarrow\!(B)}\ (A \text{ nc})$$

we introduce a variant, $B|_A$, of the implication, $A \to B$,
where A is required to be nc, and B is required to be *strict*, that is,
B must not be realized by $\bot$.

We realize restriction as

$$a\,\mathsf{r}\,(B|_A) \stackrel{\text{Def}}{=} (A \to \mathsf{Def}(a)) \wedge (\mathsf{Def}(a) \to a\,\mathsf{r}\,B)$$

$$a\,\mathsf{r}\,(A \to B) = (A \to a\,\mathsf{r}\,B)$$

Restriction $B|_A$

To repair the failure of the rule

$$\frac{A \to B \quad \neg A \to B}{\Downarrow\!(B)}\ (A \text{ nc})$$

we introduce a variant, $B|_A$, of the implication, $A \to B$, where A is required to be nc, and B is required to be *strict*, that is, B must not be realized by $\bot$.

We realize restriction as

$$a\,\mathsf{r}\,(B|_A) \stackrel{\text{Def}}{=} (A \to \mathrm{Def}(a)) \wedge (\mathrm{Def}(a) \to a\,\mathsf{r}\,B)$$

$$a\,\mathsf{r}\,(A \to B) = (A \to a\,\mathsf{r}\,B)$$

We replace the rule above by the following concurrent law of excluded middle

$$\frac{B|_A \quad B|_{\neg A}}{\Downarrow\!(B)}\ \mathit{CLEM}$$

Realizing CLEM

Realizing CLEM

We show that Amb realizes CLEM.

Realizing CLEM

We show that Amb realizes CLEM. Hence assume

$$a \,\mathsf{r}\, B|_A = (A \to \mathrm{Def}(a)) \wedge (\mathrm{Def}(a) \to a \,\mathsf{r}\, B)$$

$$b \,\mathsf{r}\, B|_{\neg A} = (\neg A \to \mathrm{Def}(b)) \wedge (\mathrm{Def}(b) \to a \,\mathsf{r}\, B)$$

Realizing CLEM

We show that Amb realizes CLEM. Hence assume

$$a \,\mathsf{r}\, B|_A = (A \to \mathrm{Def}(a)) \wedge (\mathrm{Def}(a) \to a \,\mathsf{r}\, B)$$

$$b \,\mathsf{r}\, B|_{\neg A} = (\neg A \to \mathrm{Def}(b)) \wedge (\mathrm{Def}(b) \to a \,\mathsf{r}\, B)$$

We have to show $\mathrm{Amb}(a, b) \,\mathsf{r}\, \Downarrow\!\Downarrow(B)$, that is,

$1 : \mathrm{Def}(a) \vee \mathrm{Def}(b)$

$2 : \mathrm{Def}(a) \to a \,\mathsf{r}\, B$

$3 : \mathrm{Def}(b) \to b \,\mathsf{r}\, B$

Realizing CLEM

We show that Amb realizes CLEM. Hence assume

$$a \,\mathsf{r}\, B|_A = (A \to \mathrm{Def}(a)) \wedge (\mathrm{Def}(a) \to a \,\mathsf{r}\, B)$$

$$b \,\mathsf{r}\, B|_{\neg A} = (\neg A \to \mathrm{Def}(b)) \wedge (\mathrm{Def}(b) \to a \,\mathsf{r}\, B)$$

We have to show $\mathrm{Amb}(a, b) \,\mathsf{r}\, \Downarrow\!\Downarrow(B)$, that is,

$1 : \mathrm{Def}(a) \vee \mathrm{Def}(b)$

$2 : \mathrm{Def}(a) \to a \,\mathsf{r}\, B$

$3 : \mathrm{Def}(b) \to b \,\mathsf{r}\, B$

Only 1 is critical: The assumptions clearly imply

$$\neg\neg(\mathrm{Def}(a) \vee \mathrm{Def}(b))$$

Realizing CLEM

We show that Amb realizes CLEM. Hence assume

$$a \,\mathsf{r}\, B|_A = (A \to \mathrm{Def}(a)) \wedge (\mathrm{Def}(a) \to a \,\mathsf{r}\, B)$$

$$b \,\mathsf{r}\, B|_{\neg A} = (\neg A \to \mathrm{Def}(b)) \wedge (\mathrm{Def}(b) \to a \,\mathsf{r}\, B)$$

We have to show $\mathrm{Amb}(a, b) \,\mathsf{r}\, \Downarrow\!\Downarrow(B)$, that is,

$$1 : \mathrm{Def}(a) \vee \mathrm{Def}(b)$$

$$2 : \mathrm{Def}(a) \to a \,\mathsf{r}\, B$$

$$3 : \mathrm{Def}(b) \to b \,\mathsf{r}\, B$$

Only 1 is critical: The assumptions clearly imply

$$\neg\neg(\mathrm{Def}(a) \vee \mathrm{Def}(b))$$

If a and b are given by programs, then $\mathrm{Def}(a) \vee \mathrm{Def}(b)$ is a Σ^0_1 formula (via Computational Adequacy). Hence Markov's principle suffices to prove 1.

Realizing CLEM

We show that Amb realizes CLEM. Hence assume

$$a \,\mathsf{r}\, B|_A = (A \to \mathrm{Def}(a)) \wedge (\mathrm{Def}(a) \to a \,\mathsf{r}\, B)$$

$$b \,\mathsf{r}\, B|_{\neg A} = (\neg A \to \mathrm{Def}(b)) \wedge (\mathrm{Def}(b) \to a \,\mathsf{r}\, B)$$

We have to show $\mathrm{Amb}(a, b) \,\mathsf{r}\, \Downarrow\!\Downarrow(B)$, that is,

$$1 : \mathrm{Def}(a) \vee \mathrm{Def}(b)$$

$$2 : \mathrm{Def}(a) \to a \,\mathsf{r}\, B$$

$$3 : \mathrm{Def}(b) \to b \,\mathsf{r}\, B$$

Only 1 is critical: The assumptions clearly imply

$$\neg\neg(\mathrm{Def}(a) \vee \mathrm{Def}(b))$$

If a and b are given by programs, then $\mathrm{Def}(a) \vee \mathrm{Def}(b)$ is a Σ^0_1 formula (via Computational Adequacy). Hence Markov's principle suffices to prove 1.

Note that, in the proof, A can be arbitrary. In particular, A need not be decidable.

The difference between $A \to B$ and $B|_A$

The difference between $A \to B$ and $B|_A$

Let $I = [0, 1] = \{x \mid 0 \leq x \leq 1\}$.

The difference between $A \to B$ and $B|_A$

Let $I = [0, 1] = \{x \mid 0 \leq x \leq 1\}$.

We have $x \in I \to (x \leq 0 \vee x \geq 0)$

The difference between $A \to B$ and $B|_A$

Let $I = [0, 1] = \{x \mid 0 \leq x \leq 1\}$.

We have $x \in I \to (x \leq 0 \vee x \geq 0)$
(Right is a realizer, independently of x)

The difference between $A \to B$ and $B|_A$

Let $I = [0, 1] = \{x \mid 0 \leq x \leq 1\}$.

We have $x \in I \to (x \leq 0 \vee x \geq 0)$
(Right is a realizer, independently of x)

But we do not have $(x \leq 0 \vee x \geq 0)|_{x \in I}$

The difference between $A \to B$ and $B|_A$

Let $I = [0,1] = \{x \mid 0 \leq x \leq 1\}$.

We have $x \in I \to (x \leq 0 \vee x \geq 0)$
(Right is a realizer, independently of x)

But we do not have $(x \leq 0 \vee x \geq 0)|_{x \in I}$
(a realizer a would have to satisfy
$(x \in I \to \mathrm{Def}(a)) \wedge (\mathrm{Def}(a) \to a\,\mathsf{r}\,(x \leq 0 \vee x \geq 0))$
and hence would have to depend on x)

The difference between $A \to B$ and $B|_A$

Let $I = [0,1] = \{x \mid 0 \leq x \leq 1\}$.

We have $x \in I \to (x \leq 0 \vee x \geq 0)$
(Right is a realizer, independently of x)

But we do not have $(x \leq 0 \vee x \geq 0)|_{x \in I}$
(a realizer a would have to satisfy
$(x \in I \to \mathrm{Def}(a)) \wedge (\mathrm{Def}(a) \to a\,\mathsf{r}\,(x \leq 0 \vee x \geq 0))$
and hence would have to depend on x)

However we have $x \in \mathbb{Q} \to (x \leq 0 \vee x \geq 0)|_{x \in I}$

The difference between $A \to B$ and $B|_A$

Let $I = [0,1] = \{x \mid 0 \leq x \leq 1\}$.

We have $x \in I \to (x \leq 0 \vee x \geq 0)$
(Right is a realizer, independently of x)

But we do not have $(x \leq 0 \vee x \geq 0)|_{x \in I}$
(a realizer a would have to satisfy
$(x \in I \to \mathrm{Def}(a)) \wedge (\mathrm{Def}(a) \to a\,\mathsf{r}\,(x \leq 0 \vee x \geq 0))$
and hence would have to depend on x)

However we have $x \in \mathbb{Q} \to (x \leq 0 \vee x \geq 0)|_{x \in I}$
(realized by a function that uses the decidability of $\leq$ on $\mathbb{Q}$)

The difference between $A \to B$ and $B|_A$

Let $I = [0,1] = \{x \mid 0 \leq x \leq 1\}$.

We have $x \in I \to (x \leq 0 \vee x \geq 0)$
(Right is a realizer, independently of x)

But we do not have $(x \leq 0 \vee x \geq 0)|_{x \in I}$
(a realizer a would have to satisfy
$(x \in I \to \mathrm{Def}(a)) \wedge (\mathrm{Def}(a) \to a\,\mathsf{r}\,(x \leq 0 \vee x \geq 0))$
and hence would have to depend on x)

However we have $x \in \mathbb{Q} \to (x \leq 0 \vee x \geq 0)|_{x \in I}$
(realized by a function that uses the decidability of $\leq$ on $\mathbb{Q}$)

In general

$$\begin{array}{ccccc} (\neg A \vee B) & \to & (B|_A) & \to & (A \to B) \\ & \not\leftarrow & & \not\leftarrow & \end{array}$$

Realizable proof rules of CFP

(A nc, B strict)

$$\frac{B|_A \quad B|_{\neg A}}{\Downarrow\!(B)}\ CLEM$$

$$\frac{B}{\Downarrow\!(B)} \qquad \frac{\Downarrow\!(B) \quad B \to B'}{\Downarrow\!(B')}$$

$$\frac{B}{B|_A} \qquad \frac{B|_A \quad B \to B'|_A}{B'|_A}$$

Realizable proof rules of CFP

(A nc, B strict)

$$\frac{B|_A \quad B|_{\neg A}}{\downdownarrows(B)}\ CLEM$$

$$\frac{B}{\downdownarrows(B)} \qquad \frac{\downdownarrows(B) \quad B \to B'}{\downdownarrows(B')}$$

$$\frac{B}{B|_A} \qquad \frac{B|_A \quad B \to B'|_A}{B'|_A}$$

$$\frac{A \to (B_0 \vee B_1) \quad \neg A \to (B_0 \wedge B_1)}{(B_0 \vee B_1)|_A} \qquad B_0, B_1 \text{ nc}$$

Realizable proof rules of CFP

(A nc, B strict)

$$\frac{B|_A \quad B|_{\neg A}}{\Downarrow\!(B)}\ \mathit{CLEM}$$

$$\frac{B}{\Downarrow\!(B)} \qquad \frac{\Downarrow\!(B) \quad B \to B'}{\Downarrow\!(B')}$$

$$\frac{B}{B|_A} \qquad \frac{B|_A \quad B \to B'|_A}{B'|_A}$$

$$\frac{A \to (B_0 \vee B_1) \quad \neg A \to (B_0 \wedge B_1)}{(B_0 \vee B_1)|_A} \qquad B_0, B_1 \text{ nc}$$

$$\frac{\forall x \in \mathbb{N}(P(x) \vee \neg P(x))}{(\exists x \in \mathbb{N}\, P(x))|_{\neg\neg\exists x \in \mathbb{N}\, P(x)}}$$

Program Extraction

Soundness Theorem. From a CFP proof of A one can extract a potentially nondeterministic concurrent program realizing A.

Program Extraction

Soundness Theorem. From a CFP proof of A one can extract a potentially nondeterministic concurrent program realizing A.

This refers to the denotational semantics of nondeterministic programs which may contain unresolved choices (Amb).

Program Extraction

Soundness Theorem. From a CFP proof of A one can extract a potentially nondeterministic concurrent program realizing A.

This refers to the denotational semantics of nondeterministic programs which may contain unresolved choices (Amb).

In applications one wants information about data computed in concrete situations. These situations are described by *data formulas* which, roughly, correspond to Σ^0_1-formulas A that do not contain restriction (restriction usually comes in only in subproofs of intermediately results).

Program Extraction

Soundness Theorem. From a CFP proof of A one can extract a potentially nondeterministic concurrent program realizing A.

This refers to the denotational semantics of nondeterministic programs which may contain unresolved choices (Amb).

In applications one wants information about data computed in concrete situations. These situations are described by *data formulas* which, roughly, correspond to Σ^0_1-formulas A that do not contain restriction (restriction usually comes in only in subproofs of intermediately results).

Using the Computational Adequacy Theorem, one can show that any data computed from the extracted program satisfies the IFP formula A^- obtained by removing all occurrences of $\downarrow\!\downarrow$:

Program Extraction

Soundness Theorem. From a CFP proof of A one can extract a potentially nondeterministic concurrent program realizing A.

This refers to the denotational semantics of nondeterministic programs which may contain unresolved choices (Amb).

In applications one wants information about data computed in concrete situations. These situations are described by *data formulas* which, roughly, correspond to Σ^0_1-formulas A that do not contain restriction (restriction usually comes in only in subproofs of intermediately results).

Using the Computational Adequacy Theorem, one can show that any data computed from the extracted program satisfies the IFP formula A^- obtained by removing all occurrences of $\downdownarrows$:

Program Extraction Theorem

From a proof of a data formula A one can extract a terminating program M such that whenever M reduces to a data d, then d realizes A^-.

Infinite Gray code for exact real numbers

Gray code for real numbers was introduced by Hideki Tsuiki in

Real number computation through Gray code embedding. Theoretical Computer Science, 284:467–485, 2002.

Infinite Gray code for exact real numbers

Gray code for real numbers was introduced by Hideki Tsuiki in

> *Real number computation through Gray code embedding. Theoretical Computer Science, 284:467–485, 2002.*

Pure Gray code represents a real number in $[-1, 1]$ by its itinerary of the *tent map*

$$\mathrm{tent}(x) = 1 - 2|x|$$

Infinite Gray code for exact real numbers

Gray code for real numbers was introduced by Hideki Tsuiki in

> *Real number computation through Gray code embedding. Theoretical Computer Science, 284:467–485, 2002.*

Pure Gray code represents a real number in $[-1, 1]$ by its itinerary of the *tent map*

$$\mathrm{tent}(x) = 1 - 2|x|$$

That is, $x \in [-1, 1]$ is represented by the stream $d_0 : d_1 : \ldots$ where

$$d_n = \begin{cases} 1 & \text{if } \mathrm{tent}^n(x) > 0 \\ \perp & \text{if } \mathrm{tent}^n(x) = 0 \\ -1 & \text{if } \mathrm{tent}^n(x) < 0 \end{cases}$$

Infinite Gray code for exact real numbers

Gray code for real numbers was introduced by Hideki Tsuiki in

> *Real number computation through Gray code embedding. Theoretical Computer Science, 284:467–485, 2002.*

Pure Gray code represents a real number in $[-1, 1]$ by its itinerary of the *tent map*

$$\mathrm{tent}(x) = 1 - 2|x|$$

That is, $x \in [-1, 1]$ is represented by the stream $d_0 : d_1 : \ldots$ where

$$d_n = \begin{cases} 1 & \text{if } \mathrm{tent}^n(x) > 0 \\ \perp & \text{if } \mathrm{tent}^n(x) = 0 \\ -1 & \text{if } \mathrm{tent}^n(x) < 0 \end{cases}$$

Note that $\mathrm{tent}^n(x) = 0$ can happen for at most one n.

Gray code requires partiality and non-determinism

By definition, (pure) Gray code is *partial*.

Moreover, as shown by Tsuiki, computation with Gray code requires *non-determinism*.

Gray code requires partiality and non-determinism

By definition, (pure) Gray code is *partial*.

Moreover, as shown by Tsuiki, computation with Gray code requires *non-determinism*.

The intuitive reason is as follows:

Gray code requires partiality and non-determinism

By definition, (pure) Gray code is *partial*.

Moreover, as shown by Tsuiki, computation with Gray code requires *non-determinism*.

The intuitive reason is as follows:

- Because one digit of of Gray code may be undefined, a (Turing) machine reading or writing Gray code must have two heads running concurrently, since one head might get stuck at an undefined digit.

Gray code requires partiality and non-determinism

By definition, (pure) Gray code is *partial*.

Moreover, as shown by Tsuiki, computation with Gray code requires *non-determinism*.

The intuitive reason is as follows:

- Because one digit of of Gray code may be undefined, a (Turing) machine reading or writing Gray code must have two heads running concurrently, since one head might get stuck at an undefined digit.
- Since the two heads act independently the machine's behaviour is non-deterministic.

From Gray code to signed digits

Gray code has the remarkable property that each real number $x \in [-1, 1]$ has exactly one representation.

From Gray code to signed digits

Gray code has the remarkable property that each real number $x \in [-1, 1]$ has exactly one representation.

In contrast, the well-known *signed representation*, which represents a real number $x \in [-1, 1]$ by an infinite stream of digits $d_i \in \mathrm{SD} \stackrel{\mathrm{Def}}{=} \{-1, 0, 1\}$ such that

$$x = \sum_{i \in \mathbb{N}} d_i 2^{-(i+1)},$$

is redundant (as are all other admissible total representations of the reals).

From Gray code to signed digits

Gray code has the remarkable property that each real number $x \in [-1, 1]$ has exactly one representation.

In contrast, the well-known *signed representation*, which represents a real number $x \in [-1, 1]$ by an infinite stream of digits $d_i \in \mathrm{SD} \stackrel{\mathrm{Def}}{=} \{-1, 0, 1\}$ such that

$$x = \sum_{i \in \mathbb{N}} d_i 2^{-(i+1)},$$

is redundant (as are all other admissible total representations of the reals).

We sketch how to extract a concurrent program that translates infinite Gray code into signed representation.

Formalization of Gray code and signed digit representation

Formalization of Gray code and signed digit representation

$$\begin{aligned} \mathrm{SD} &\overset{\mathrm{Def}}{=} \{-1, 0, 1\} \\ \mathbb{I}_d &\overset{\mathrm{Def}}{=} [d/2 - 1/2, d/2 + 1/2] \end{aligned}$$

Formalization of Gray code and signed digit representation

$$\begin{aligned} \mathrm{SD} &\overset{\mathrm{Def}}{=} \{-1, 0, 1\} \\ \mathbb{I}_d &\overset{\mathrm{Def}}{=} [d/2 - 1/2, d/2 + 1/2] \end{aligned}$$

$$\mathrm{C}(x) \overset{\nu}{=} \exists d \in \mathrm{SD}\,(x \in \mathbb{I}_d \wedge \mathrm{C}(2x - d))$$

Formalization of Gray code and signed digit representation

$$\begin{aligned} \mathrm{SD} &\overset{\mathrm{Def}}{=} \{-1, 0, 1\} \\ \mathbb{I}_d &\overset{\mathrm{Def}}{=} [d/2 - 1/2, d/2 + 1/2] \end{aligned}$$

$$\begin{aligned} \mathrm{C}(x) &\overset{\nu}{=} \exists d \in \mathrm{SD}\,(x \in \mathbb{I}_d \wedge \mathrm{C}(2x - d)) \\ \mathrm{C}_2(x) &\overset{\nu}{=} \Downarrow(\exists d \in \mathrm{SD}\,(x \in \mathbb{I}_d \wedge \mathrm{C}_2(2x - d))) \end{aligned}$$

Formalization of Gray code and signed digit representation

$$
\begin{aligned}
\mathrm{SD} &\overset{\mathrm{Def}}{=} \{-1, 0, 1\} \\
\mathbb{I}_d &\overset{\mathrm{Def}}{=} [d/2 - 1/2, d/2 + 1/2]
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{C}(x) &\overset{\nu}{=} \exists d \in \mathrm{SD}\,(x \in \mathbb{I}_d \wedge \mathrm{C}(2x - d)) \\
\mathrm{C}_2(x) &\overset{\nu}{=} \Downarrow(\exists d \in \mathrm{SD}\,(x \in \mathbb{I}_d \wedge \mathrm{C}_2(2x - d)))
\end{aligned}
$$

$$
\mathrm{G}(x) \overset{\nu}{=} (x \neq 0 \to x \leq 0 \vee x \geq 0) \wedge \mathrm{G}(\mathsf{tent}(x))
$$

Formalization of Gray code and signed digit representation

$$
\begin{aligned}
\mathrm{SD} &\overset{\mathrm{Def}}{=} \{-1, 0, 1\} \\
\mathbb{I}_d &\overset{\mathrm{Def}}{=} [d/2 - 1/2, d/2 + 1/2]
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{C}(x) &\overset{\nu}{=} \exists d \in \mathrm{SD}\,(x \in \mathbb{I}_d \wedge \mathrm{C}(2x - d)) \\
\mathrm{C}_2(x) &\overset{\nu}{=} \Downarrow(\exists d \in \mathrm{SD}\,(x \in \mathbb{I}_d \wedge \mathrm{C}_2(2x - d)))
\end{aligned}
$$

$$
\mathrm{G}(x) \overset{\nu}{=} (x \neq 0 \to x \leq 0 \vee x \geq 0) \wedge \mathrm{G}(\mathsf{tent}(x))
$$

$s \,\mathsf{r}\, \mathrm{C}$ iff s is a signed digit representation of x.

$s \,\mathsf{r}\, \mathrm{G}$ iff s is an infinite Gray code of x.

$s \,\mathsf{r}\, \mathrm{C}_2$ iff s is a non-deterministic signed digit rep. of x.

Formalization of Gray code and signed digit representation

$$\begin{aligned} \mathrm{SD} &\overset{\mathrm{Def}}{=} \{-1, 0, 1\} \\ \mathbb{I}_d &\overset{\mathrm{Def}}{=} [d/2 - 1/2, d/2 + 1/2] \end{aligned}$$

$$\begin{aligned} \mathrm{C}(x) &\overset{\nu}{=} \exists d \in \mathrm{SD}\,(x \in \mathbb{I}_d \wedge \mathrm{C}(2x - d)) \\ \mathrm{C}_2(x) &\overset{\nu}{=} \Downarrow(\exists d \in \mathrm{SD}\,(x \in \mathbb{I}_d \wedge \mathrm{C}_2(2x - d))) \end{aligned}$$

$$\mathrm{G}(x) \overset{\nu}{=} (x \neq 0 \to x \leq 0 \vee x \geq 0) \wedge \mathrm{G}(\mathsf{tent}(x))$$

$s \,\mathsf{r}\, \mathrm{C}$ iff s is a signed digit representation of x.

$s \,\mathsf{r}\, \mathrm{G}$ iff s is an infinite Gray code of x.

$s \,\mathsf{r}\, \mathrm{C}_2$ iff s is a non-deterministic signed digit rep. of x.

$\mathrm{C} \subseteq \mathrm{G}$ is easy. Our main goal is to show $\mathrm{G} \subseteq \mathrm{C}_2$.

$\mathrm{G} \subseteq \mathrm{C}_2$ (main step)

$\mathrm{G} \subseteq \mathrm{C}_2$ (main step)

Assume $\mathrm{G}(x)$.

$\mathrm{G} \subseteq \mathrm{C}_2$ (main step)

Assume $\mathrm{G}(x)$.

$A(x) \stackrel{\mathrm{Def}}{=} \exists d \in \mathrm{SD}\, x \in \mathbb{I}_d$

$\mathrm{G} \subseteq \mathrm{C}_2$ (main step)

Assume $\mathrm{G}(x)$.

$A(x) \stackrel{\mathrm{Def}}{=} \exists d \in \mathrm{SD}\, x \in \mathbb{I}_d$

We show $\downdownarrows(A(x))$, i.e. the first digit of the signed digit representation exists, nondeterministically.

$\mathrm{G} \subseteq \mathrm{C}_2$ (main step)

Assume $\mathrm{G}(x)$.

$A(x) \stackrel{\mathrm{Def}}{=} \exists d \in \mathrm{SD}\, x \in \mathbb{I}_d$

We show $\downdownarrows(A(x))$, i.e. the first digit of the signed digit representation exists, nondeterministically.

Recall $\mathrm{G}(x) \stackrel{\nu}{=} (x \neq 0 \to x \leq 0 \vee x \geq 0) \wedge \mathrm{G}(\mathsf{tent}(x))$.

$\mathrm{G} \subseteq \mathrm{C}_2$ (main step)

Assume $\mathrm{G}(x)$.

$A(x) \stackrel{\mathrm{Def}}{=} \exists d \in \mathrm{SD}\, x \in \mathbb{I}_d$

We show $\Downarrow\!\!\downarrow(A(x))$, i.e. the first digit of the signed digit representation exists, nondeterministically.

Recall $\mathrm{G}(x) \stackrel{\nu}{=} (x \neq 0 \to x \leq 0 \vee x \geq 0) \wedge \mathrm{G}(\mathrm{tent}(x))$.

$$\dfrac{\dfrac{\dfrac{\mathrm{G}(x)}{x \neq 0 \to (x \leq 0 \vee x \geq 0)} \qquad x = 0 \to (x \leq 0 \wedge x \geq 0)}{x \leq 0 \vee x \geq 0|_{x \neq 0}} \qquad \dfrac{\dfrac{\mathrm{G}(x)}{\mathrm{G}(\mathrm{tent}(x))}}{\vdots}}{A(x)|_{x \neq 0}} \qquad \dfrac{}{A(x)|_{x=0}}}{\Downarrow\!\!\downarrow(A(x))}$$

Pivoting and Gaussian elimination

Given a non-singular real $n \times n$ matrix, its inverse can be computed using Gaussian elimination:

Pick a column of the matrix (for example, the last one) and find a *pivot element*, that is, a non-zero entry (which must exist due to non-singularity).

Perform elementary matrix operations.

Repeat . . . .

Pivoting and Gaussian elimination

Given a non-singular real $n \times n$ matrix, its inverse can be computed using Gaussian elimination:

Pick a column of the matrix (for example, the last one) and find a *pivot element*, that is, a non-zero entry (which must exist due to non-singularity).

Perform elementary matrix operations.

Repeat . . . .

If the entries of the matrix are exact real numbers, pivoting is problematic from a constructive point of view.

Pivoting

Pivot Lemma

$\forall x_0, \ldots, x_n \in \mathrm{A} \Downarrow_n ((\exists i \leq n\, x_i \# 0)|_{\neg \forall i \leq n\, x_i = 0})$

where $x \# 0 \stackrel{\mathrm{Def}}{=} \exists k \in \mathbb{N}\, |x| \geq 2^{-k}$
and $\Downarrow_n$ is a generalization of $\Downarrow$ to n concurrent threads.

Pivoting

Pivot Lemma

$\forall x_0, \ldots, x_n \in \mathrm{A}\ \Downarrow_n((\exists i \leq n\, x_i \ \# \ 0)|_{\neg \forall i \leq n\, x_i = 0})$

where $x \ \# \ 0 \stackrel{\mathrm{Def}}{=} \exists k \in \mathbb{N}\, |x| \geq 2^{-k}$
and $\Downarrow_n$ is a generalization of $\Downarrow$ to n concurrent threads.

Proof

Using a generalized concurrent law of the excluded middle and the

Search Lemma $\forall x \in \mathrm{A}\ x \ \# \ 0 \parallel x \neq 0.$

Pivoting

Pivot Lemma

$\forall x_0, \ldots, x_n \in \mathsf{A} \Downarrow_n ((\exists i \leq n\, x_i \# 0)|_{\neg \forall i \leq n\, x_i = 0})$

where $x \# 0 \stackrel{\text{Def}}{=} \exists k \in \mathbb{N}\, |x| \geq 2^{-k}$
and $\Downarrow_n$ is a generalization of $\Downarrow$ to n concurrent threads.

Proof

Using a generalized concurrent law of the excluded middle and the

Search Lemma $\forall x \in \mathsf{A}\; x \# 0 \parallel x \neq 0.$

The Search Lemma is realized by

$$\varphi(a : as) \quad = \quad \text{if } |a| > 2 \text{ then } 0 \text{ else } 1 + \varphi(2 * as)$$

the Pivot Lemma by $\lambda(as_0, \ldots, as_n)\,.\,\mathsf{Amb}(\varphi_0(as_0), \ldots, \varphi_n(as_n))$
with $\varphi_i(as) \stackrel{\text{Def}}{=} (i, \varphi(as))$.

Implementation

Realizability and program extraction (and more) is implemented in the interactive proof system *Minlog* developed by Schwichtenberg in Munich.

`http://www.mathematik.uni-muenchen.de/~logik/minlog/`

Implementation

Realizability and program extraction (and more) is implemented in the interactive proof system *Minlog* developed by Schwichtenberg in Munich.

`http://www.mathematik.uni-muenchen.de/~logik/minlog/`

The concurrent extension of the language of realizers is implemented in Haskell using the package `Control.Concurrent`.

`https://github.com/ujberger/cfp`

Towards a type-theoretic interpretation of IFP and CFP (j.w.w. Sewon Park, Holger Thies, and Hideki Tsuiki)

Shallow embedding of IFP into Coq

- Sorts are constants in `Prop`
- Inductive and coinductive definitions can be partly interpreted as Coq's
- Coq's program extraction can be partly used though unwanted elements remain.

Towards a type-theoretic interpretation of IFP and CFP (j.w.w. Sewon Park, Holger Thies, and Hideki Tsuiki)

Shallow embedding of IFP into Coq

- Sorts are constants in `Prop`
- Inductive and coinductive definitions can be partly interpreted as Coq's
- Coq's program extraction can be partly used though unwanted elements remain.

We are trying to overcome the shortcomings of the shallow embedding by combining it with a deep embedding using MetaCoq. This will allow us to define our own realizability interpretation and more general forms of induction and coinduction.

Conclusion

Conclusion

- IFP/CFP permits program extraction from proofs in abstract mathematics.

Conclusion

- IFP/CFP permits program extraction from proofs in abstract mathematics.
- Not only programs are extracted from proofs but also the data they are operating on.

Conclusion

- $\mathrm{IFP}/\mathrm{CFP}$ permits program extraction from proofs in abstract mathematics.
- Not only programs are extracted from proofs but also the data they are operating on.
- Classical logic is permitted for nc-formulas. CFP permits in addition a concurrent form of the Law of Excluded Middle.

Conclusion

- $\mathrm{IFP}/\mathrm{CFP}$ permits program extraction from proofs in abstract mathematics.
- Not only programs are extracted from proofs but also the data they are operating on.
- Classical logic is permitted for nc-formulas. CFP permits in addition a concurrent form of the Law of Excluded Middle.
- The operational semantics of CFP implements *globally angelic nondeterminism*

Conclusion

- $\mathrm{IFP}/\mathrm{CFP}$ permits program extraction from proofs in abstract mathematics.
- Not only programs are extracted from proofs but also the data they are operating on.
- Classical logic is permitted for nc-formulas. CFP permits in addition a concurrent form of the Law of Excluded Middle.
- The operational semantics of CFP implements *globally angelic nondeterminism*
- *Locally angelic nondeterminism* can be modelled as well.

Conclusion

- $\mathrm{IFP}/\mathrm{CFP}$ permits program extraction from proofs in abstract mathematics.
- Not only programs are extracted from proofs but also the data they are operating on.
- Classical logic is permitted for nc-formulas. CFP permits in addition a concurrent form of the Law of Excluded Middle.
- The operational semantics of CFP implements *globally angelic nondeterminism*
- *Locally angelic nondeterminism* can be modelled as well.
- So far, concurrency is limited to potentially nonterminating processes running in parallel.

Conclusion

- $\mathrm{IFP}/\mathrm{CFP}$ permits program extraction from proofs in abstract mathematics.
- Not only programs are extracted from proofs but also the data they are operating on.
- Classical logic is permitted for nc-formulas. CFP permits in addition a concurrent form of the Law of Excluded Middle.
- The operational semantics of CFP implements *globally angelic nondeterminism*
- *Locally angelic nondeterminism* can be modelled as well.
- So far, concurrency is limited to potentially nonterminating processes running in parallel.
- Applications in computable analysis

Conclusion

- $\mathrm{IFP}/\mathrm{CFP}$ permits program extraction from proofs in abstract mathematics.
- Not only programs are extracted from proofs but also the data they are operating on.
- Classical logic is permitted for nc-formulas. CFP permits in addition a concurrent form of the Law of Excluded Middle.
- The operational semantics of CFP implements *globally angelic nondeterminism*
- *Locally angelic nondeterminism* can be modelled as well.
- So far, concurrency is limited to potentially nonterminating processes running in parallel.
- Applications in computable analysis
- Further applications: concurrent computation of compact sets (joint work with Dieter Spreen).

Conclusion

- $\mathrm{IFP}/\mathrm{CFP}$ permits program extraction from proofs in abstract mathematics.
- Not only programs are extracted from proofs but also the data they are operating on.
- Classical logic is permitted for nc-formulas. CFP permits in addition a concurrent form of the Law of Excluded Middle.
- The operational semantics of CFP implements *globally angelic nondeterminism*
- *Locally angelic nondeterminism* can be modelled as well.
- So far, concurrency is limited to potentially nonterminating processes running in parallel.
- Applications in computable analysis
- Further applications: concurrent computation of compact sets (joint work with Dieter Spreen).
- Further work: Classical choice principles, Raoult's general form open induction.

References

Hideki Tsuiki.
Real Number Computation through Gray Code Embedding.
Theor. Comput. Sci., *284(2):467–485, 2002.*

B., Kenji Miyamoto, Helmut Schwichtenberg, Hideki Tsuiki.
Logic for Gray-code computation.
In: Concepts of Proof in Mathematics, Philosophy, and Computer Science, de Gruyter, 2016.

B, Olga Petrovska, Hideki Tsuiki.
Prawf: An interactive proof system for program extraction.
CiE 2020. LNCS 12098.

Dieter Spreen.
Computing with continuous objects: a uniform co-inductive approach.
Mathematical Structures in Computer Science 31*(2), 144–192 (2021).*

B, Monika Seisenberger, Dieter Spreen, Hideki Tsuiki.
Concurrent Gaussian elimination.
To appear, 2022.