プログラムと論理

The only way to rectify our reasonings is to make them as tangible as those of the Mathematicians, so that we can find our error at a glance, and when there are disputes among persons, we can simply say: Let us calculate, without further ado, to see who is right.

-- Leibniz

Mathematics has been developed over two millennia as the best approach to rigorous human reasoning. A couple of decades of pseudo-programming language design poses no threat to its preeminence. The best way to reason mathematically is to use mathematics, not a pseudo-programming language.

-- Lamport

Parts of Leibniz's Dream became reality, and it is quite understandable that this happened mostly in Departments of Computing Science, rather than in Departments of Mathemetics. ...

....

In short, the world of computing became Leibniz's home; that it was my home as well was my luck.

-- Dijkstra

はじめに

- 小論は、Deep Specificationのような現代の「形式的手法」の理解を深めることを目的としている。
- 現代の「形式的手法」には、いずれも50年近く前の1970年代に遡る、二つの異なった「起源」がある。両者ともに「プログラムと論理」という問題意識は共通していたのだが、生まれた場所が異なっていた。一つは、論理学＝数学の領域で生まれ、もう一つは計算機科学の領域で生まれた。
- 前者の代表は、ハワードやマーチン・レフである。彼らは、「型の理論」を作り上げ、抽象的だが、プログラムの実行は、ある定理の証明に他ならないという認識にたどり着いた。
- 後者の代表は、ホーアやダイクストラである。彼らは、代入文やif文、シーケンシャルな実行や繰り返しのWhile文といった具体的なプログラムの構成と論理との関係を研究した。

はじめに

- ただ、20世紀の段階では、「プログラムと論理」という問題を扱うには、 論理学＝数学からのアプローチでは、現実のコンピュータ技術や具体的なプログラム言語との接点は明確ではなかった。計算機科学からのアプローチでは、論理の多様な性質への理解と、何よりも強力な「証明マシン」を欠いていた。
- 様々な紆余曲折を経て、両者の統一は、21世紀に持ち込まれることとなった。筆者は、こうした二つの流れが合流したところに、 Deep Specificationのような現代の「形式的手法」が生まれたと捉えている。
- 小論では、20世紀に計算機科学の中で追求された「形式的手法」とその「挫折」を、一つのトピックにしている。このテーマでは、歴史を振り返るのには意味があると考えている。なぜなら、現代の「形式的手法」は、かつて一度失敗したムーブメントの再復活という面を持つからである。

はじめに

- 現代の形式的手法の紹介は、前回のマルレクに続くものだが、今回は、出来るだけ具体的な実例の紹介を心がけた。
- コンピュータが論理的＝数学的推論能力を持つことの認識と強力な証明マシンの存在こそが、20世紀の形式的手法と現代の形式的手法を分かつ最大の違いだと筆者は考えている。
- と言っても、講演とスライドだけでは、「コンピュータによる証明」「コンピュータによる証明のチェック」の強力さ（スピードと規模感）が伝わらないと感じている。その技術は、現代の我々の手の届くところにあるのだ。ただ、多くのIT技術者は、そのことに気づいていない。
- 改めて、現代の形式的手法の基礎であり、現代のソフトウェア科学の Lingua Franca であるCoqの利用者を増やすことの重要性を強く感じている。

Agenda
プログラムと論理

- はじめに
  -- 形式的手法の二つの起源
- 第一部:　計算と論理 -- 20世紀
- 第二部:　プログラムと論理 -- 20世紀
- 第三部:　21世紀の形式的手法
- おわりに
  -- ダイクストラのメッセージを考える
- Appendix
  --「ライプニッツの夢」

Agenda
第一部:　計算と論理 -- 20世紀

- 型付きラムダ計算と論理
- Dependent Type と論理
- 証明とは何か？
- “Propositions as Types”
“Proofs as Terms”
- 証明支援システム Coq

Agenda
第二部: プログラムと論理 -- 20世紀

- 形式手法
  - Hoare – Hoare Logic
  - Dijkstra -- Nondeterminacy
  - Lamport – Temporal Logic
- 証明の性質をめぐる論争
- 形式的手法の後退
  - Hoare
  - Lamport

Agenda
第三部:　21世紀の形式的手法

- Deep Specificationと証明支援システムCoq
  -- 形式手法のLingua Franca
- 形式的推論・証明サンプル
  -- Hoare Logicの三つ組の証明
- Deep Specificationにおける仕様と実装例
  -- Kami プロジェクトを例に
- Software Foundation

第一部

計算と論理 -- 20世紀

Agenda
第一部:　計算と論理 -- 20世紀

- 型付きラムダ計算と論理
- Dependent Type と論理
- 証明とは何か？
- “Propositions as Types”
  “Proofs as Terms”
- 証明支援システム Coq

型付きラムダ計算と論理

Curry-Howard対応

- 1940年代のChurchの仕事から、ラムダ計算に対する関心が、ふたたび活発化するのは、20年近くたった1960年代からである。
- 理由ははっきりしている。コンピュータが現実に動き出したからである。
- この時期の代表的な成果は、HowardのCurry-Howard対応の研究である。

Curryの発見
型付きラムダ計算と直観主義論理との対応

- 既に1934年に、Curryは、型付きラムダ計算と直観主義論理とのあいだに、対応関係があることを発見していた。

- 型を持つラムダ計算の関数の型を示す矢印 → と、論理式“A → B” の中に出てくる含意を意味する矢印 → との間に、対応関係があるというのだ。

Curry, Haskell (1934), "Functionality in Combinatory Logic”
http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1076489/pdf/pnas01751-0022.pdf

Curry

型付きラムダ計算と論理との関係

ラムダ計算への型の導入
型σから型τへの関数は、型 $\sigma \to \tau$ を持つ。

- 型σから型τへの関数は、型 $\sigma \to \tau$ を持つ。
- あるλ式 e が型 τ を持つことを、$e : \tau$ と表す。
- アルファ変換、ベータ還元、エータ変換のα, β, ηの変換ルールは型のないラムダ計算と同じものとする。

関数の「型」に、矢印 “→” を使うことは、Haskell, ML 等の関数型言語では、広く使われていることを知っている人も多いと思う。Curryの発見は、この関数の型を表す矢印 “→” が、論理式の含意 A → B (A ならば B) を表す論理記号の矢印 “→”と対応があるということだった。

型付きラムダ式の型の定義

型付きλ式の型の定義を次のように行う。

1. 単純な変数 v_i は型を持つ。 $v_i : \tau$
2. $x : \sigma$ で $e : \tau$ なら、$(\lambda x_\sigma . e) : (\sigma \rightarrow \tau)$
3. $e_1 : (\sigma \rightarrow \tau)$で、$e_2 : \sigma$ なら、$(e_1\ e_2) : \tau$

そのことを見るには、上の型付きラムダ式の三番目の定義に注目すればいい。この定義は、ある関数の引数に値を与えて、関数の値を計算することに対応する操作、「適用 apply」という操作である。次に、この型付きラムダ計算での、適用の操作を少し詳しく見ることにしよう。

この例は、次のように考えるとわかりやすい

「適用」は型を持つ

$e_1 : (\sigma \to \tau)$で、$e_2 : \sigma$ なら、

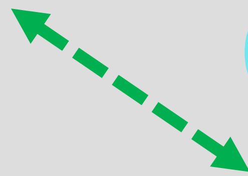
$(e_1\ e_2) : \tau$

この例は、次のように考えるとわかりやすい

「適用」は型を持つ

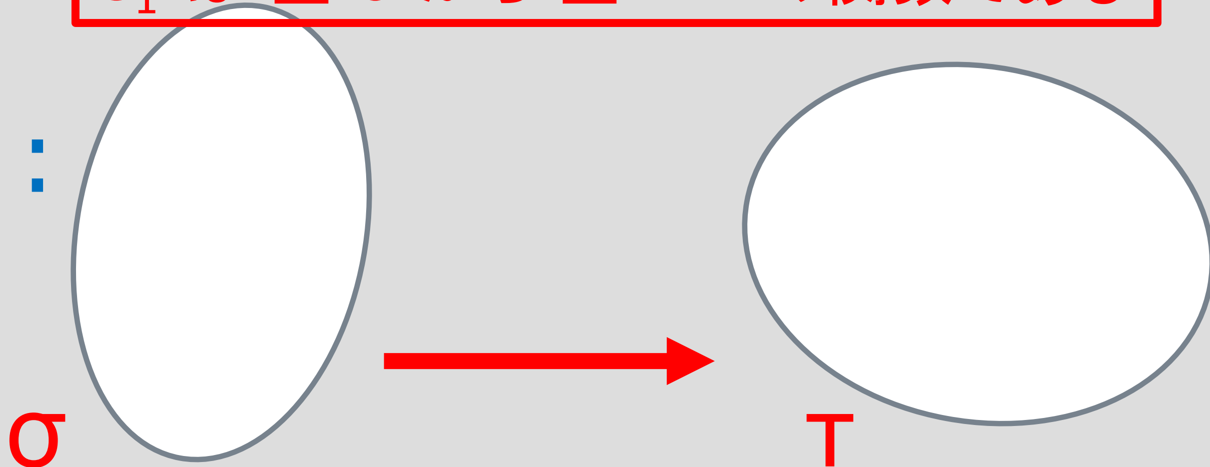
$e_1 : (\sigma \to \tau)$で、$e_2 : \sigma$ なら、

$(e_1\ e_2) : \tau$



e_1 は 型 σ から 型 τ への関数である

関数 e_1 :



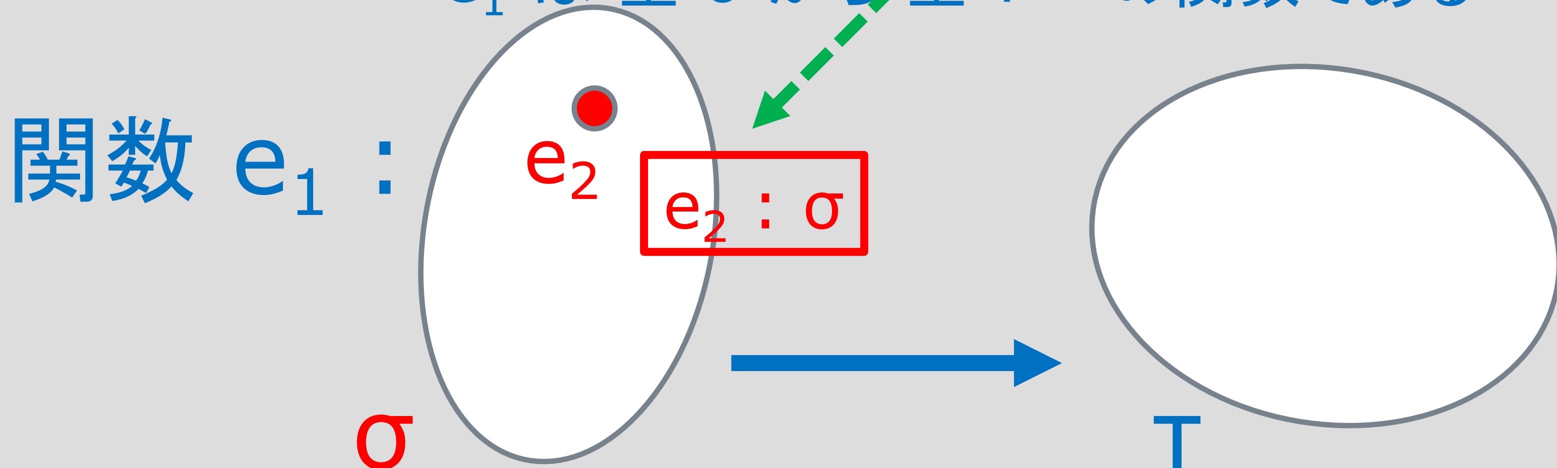
この例は、次のように考えるとわかりやすい

「適用」は型を持つ

$e_1 : (\sigma \rightarrow \tau)$で、$e_2 : \sigma$ なら、

$(e_1\ e_2) : \tau$

e_1 は 型 σ から 型 τ への関数である

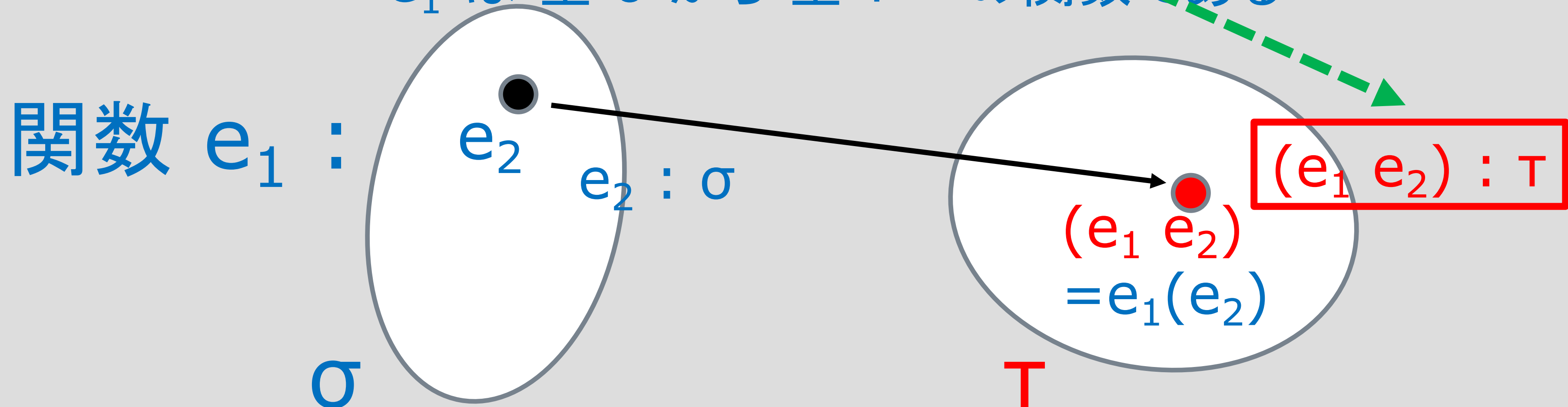


この例は、次のようにに考えるとわかりやすい

「適用」は型を持つ

e_1 : (σ → τ)で、e_2 : σ なら、

$(e_1\ e_2)$: τ



型付きラムダ式の型の定義と論理式

型付きλ式の型の定義。

1. 単純な変数 v_i は型を持つ。 $v_i : \tau$
2. $x : \sigma$ で $e : \tau$ なら、$(\lambda x_\sigma.e) : (\sigma \to \tau)$
3. $e_1 : (\sigma \to \tau)$で、$e_2 : \sigma$ なら、$(e_1\ e_2) : \tau$

この型付きラムダ計算は、
「A → B で A なら Bである」（三段論法）
という論理式と同じ構造をしている。

Howardによる発展

Howardによる発展
論理式はラムダ計算の型と見なすことが出来る

- Howardは、このCurryの発見を、さらに深く考えた。
- 1969年の彼の論文のタイトルが示すように、論理式は型付きラムダ計算の型と見なすことが出来るのである。
- ただし、彼のこの論文が公開されたのは、1980年になってからのことらしい。

Howard,Williams (1980)
"The formulae-as-types notion of construction”
http://www.cs.cmu.edu/~crary/819-f09/Howard80.pdf

「型」と「命題」、「要素」と「証明」との対応

- Curry-Howard対応の、中心的な概念は、「型」と「命題」、「要素」と「証明」との対応である。

- 「*p* が命題 *P* の証明である」ことを、***p* : *P*** と表すことができる。
- 「*p* は、型 *P* の要素である」ことも、***p* : *P*** と表すことができる。

「型」と「命題」、「要素」と「証明」との対応

- Curry-Howard対応の、中心的な概念は、「型」と「命題」、「要素」と 「証明」との対応である。

- 「*p* が命題 *P* の証明である」ことを、***p* : *P***
  と表すことができる。
- 「*p* は、型 *P* の要素である」ことも、***p* : *P***
  と表すことができる。

「*p* が命題 *P* の証明である」という判断を表す ***p* : *P*** は、「*p* は、型 *P* の要素である」という判断を表す ***p* : *P*** と見なすことが出来るし、また、逆の見方も出来るのである。

Dependent Type と論理

Dependent Type Theory

- 現在の「型の理論」の基本が出来上がるのは、1980年代になってからである。その中心人物は、Martin-Löfである。
- 現在のCoq, Agdaという証明支援システムは、彼のDependent Typeの理論に基礎付けられている。

命題への型の拡張

- Churchの単純な型を持つラムダ計算の体系は、
基本的には、型A, 型Bに対して、型 A → Bで表現される関数型の型しか持たない。

- それに対して、Martin-Löfは、論理的な命題にも、自然なスタイルで型を定義した。例えば、Aが型であり、Bが型であれば、A∧BもA→BもAvBも型であるというように。もちろん、それぞれは異なる型である。

あるaが型Aを持つ時、***a : A*** で表す。

Dependent Type

- これまで、A∧B, A → B, A∨B といった、元になる型 A, B を指定すると新しい型が決まるといったスタイルで型を導入してきた。
- これとは異なる次のようなスタイル、型Aそのものではなく型Aに属する要素 a : Aに応じて新しい型 B(a)が変化するような型の導入が可能である。
- 例えば、実数R上のn次元のベクトル Vec(R,n)とn+1次元のベクトルVec(R,n+1)は、別の型を持つのだが、これは nに応じて型が変わると考えることが出来る。

Dependent Type

- こうした型をDependent Typeと呼び、次のように表す。

  $$\Pi(x:A).B(x)$$

- Dependent Typeは、ある型Aの値aにディペンドして変化する型である。
- 先の例のベクトルの型は、次のように表される。

  $$\Pi(x:N).Vec(R,x)$$

  これは、全てのnについてVec(R,n)を考えることに対応している。
- 型の理論では、全称記号は、Dependent Typeとして導入される。

Dependent TypeとPolymorphism

- Dependent Typeの例を、もう一つあげよう。
  n次元のベクトルは、自然数 N、整数 Z、実数 R、複素数 C上でも定義出来る。
  { N, Z, R, C } : T とする時、次のように定義される Dependent Typeを考えよう。
  $\Pi(x : T)Vec(x,n)$
- これは、次元nは同じだが、定義された領域が異なる別の型 Vec(N,n),Vec(Z,n),Vec(R,n), Vec(C,n)を考えることに相当する。
- こうして、Polymorphicな関数は、Dependent Typeとして表現されることが分かる。

Inductive Type

- 型の理論では、基本的な定数と関数から、新しい型を帰納的に定義することが出来る。こうした型をInductive Type（帰納的型）と呼ぶ。
- 次は、そうした帰納的な定義の例である。
  ここでは、自然数の型natが、ゼロを意味する定数Oと「その次の数」を表す関数Sで、帰納的に定義されている。

Coqでの「自然数」 nat の定義

```
Inductive nat : Type :=
  | O : nat
  | S : nat -> nat.
```

Sは、natからnatへの関数である
直感で気には、S(n)=n+1 である

Inductive Type nat (自然数)

0. O
1. S(O)
2. S(S(O))
3. S(S(S(O)))
4. S(S(S(S(O))))
5. S(S(S(S(S(O)))))
6. S(S(S(S(S(S(O))))))

   ……

   ……

n. S(S(S(S(S(S(S(…S(O)…)))))))

n個のS

数学的帰納法

- ある命題Pが全ての自然数で成り立つことを示すのに次の「数学的帰納法」を用いることができる。
  1. Pはゼロで成り立つ。
     P(0)
  2. もし、Pがnで成り立つなら、Pはn+1でも成り立つ。
     P(n) --> P(n+1)
  3. この時、全ての自然数nについてPは成り立つ。
     ∀n P(n)
- 数学的帰納法は、その名前に反して、数学的演繹の強力な手段である。

同一性への型の付与

- Martin-Löf は、式 a = b にも型を与える。
  $a =_A b : Id(A\ a\ b)$
  型 Id(A a b) を持つ式は、a と b は等しいという意味を持つ。
- a $=_A$ b のAは、型Aの中での同一性であることを表している。すなわち、a : A で b : Aで、かつa = bの時にはじめて、a $=_A$ b は型Id(A a b)
  を持つ。
- ここでは式の同一性について述べたが、式の同一性と型の同一性は、異なるものである。

型の理論の記述

- Martin-Löfの型の理論は、次のことを示す、四つの形式で記述される。

1. ある対象aが、型であること

   $a : Type$

2. ある表現式 aが、型 αの要素であること

   $a : \alpha$

3. 二つの表現式が、同じ型の内部で、等しいこと

   $a = b : \alpha$

4. 二つの型が、等しいこと

   $\alpha = \beta \ : Type$

証明とは何か？

Kolmogorov
a → b (aならばb)の証明の解釈

- Kolmogorovは、命題 a → b (aならばb)の証明に、「命題 aの証明を命題 bの証明に変換する方法を構築すること」という解釈を与えた。
- このことは、a → bの証明を、aの証明からbの証明への関数と見ることが出来るということを意味する。
- 同様に、次に見るように、命題の意味を、その証明と関連づけることができる。
- Kolmogorovのこの考え(構成主義という)は、 Martin-Löfの型の理論に深い影響を与えた。

Kolmogorovの解釈の下では、命題の証明は、何から構成されるか？

□ **⊥（偽）**	証明なし
□ **A ∧ B**	Aの証明とBの証明の両方
□ **A ∨ B**	Aの証明、あるいは、Bの証明
□ **A → B**	Aの証明からBの証明を導く方法
□ **(∀x)B(x)**	任意のaに対してB(a)の証明を与える方法
□ **(∃x)B(x)**	あるaに対するB(a)の証明

Kolmogorovの解釈の下では、命題の証明は、何から構成されるか？形式的に

□ ⊥（偽）	none
□ $A \wedge B$	Aの証明であるaと、Bの証明であるbのペア**(a,b)**
□ $A \vee B$	Aの証明である**i(a)**、あるいは、Bの証明である**j(b)**
□ $A \rightarrow B$	Aの証明であるaに対して、Bの証明b(a)を与える**(λx)b(x)**
□ $(\forall x)B(x)$	任意のaに対してBの証明b(a)を与える**(λx)b(x)**
□ $(\exists x)B(x)$	あるaと、B(a)の証明であるbのペア**(a,b)**

$a : A$ の解釈

- 論理＝数学的には、 $a : A$ には、次のような様々な解釈がある。

1. 集合論: Russellの立場
   Aは集合であり、aはその要素である。 $a \in A$
2. 構成主義: Kolmogorovの立場
   Aは問題であり、aはその解決である
3. PAT: Curry & Howardの立場
   Aは命題であり、aはその証明である。 $a : A$
4. 型の理論: Martin-Löf
   Aは型であり、aはその項である。 $a : A$
5. HoTT: Voevodskyの立場
   Aは空間であり、aはその点である。 $a : A$

a : *A* の解釈

- 論理＝数学的には、**a : A** には、次のような様々な解釈がある。

1. 集合論: Russellの立場
   Aは集合であり、aはその要素である。 $a \in A$
2. 構成主義: Kolmogorovの立場
   Aは問題であり、aはその解決である
3. PAT: Curry & Howardの立場
   Aは命題であり、aはその証明である。 $a : A$
4. 型の理論: Martin-Löf
   Aは型であり、aはその項である。 $a : A$
5. HoTT: Voevodskyの立場
   Aは空間であり、aはその点である。 $a : A$

$a : A$ の解釈

- 論理＝数学的には、$a : A$ には、次のような様々な解釈がある。

1. 集合論: Russellの立場
   Aは集合であり、aはその要素である。$a \in A$
2. 構成主義: Kolmogorovの立場
   Aは問題であり、aはその解決である
3. PAT: Curry & Howardの立場
   Aは命題であり、aはその証明である。 $a : A$
4. 型の理論：Martin-Löf
   Aは型であり、aはその項である。 $a : A$
5. HoTT: Voevodskyの立場
   Aは空間であり、aはその点である。 $a : A$

Curry-Howard対応

“Propositions as Types”
“Proofs as Terms”

“Propositions as Types”
“Proofs as Terms”

- Howardの洞察は、”Propositions as Types”, “Proofs as Terms” (これを、PATという)として、Martin-Löfの型の理論に大きな影響を与えた。
- 同時に、Curry-Howard対応は、型付きラムダ計算をプログラムと見なせば、”Proof as Program”としても解釈出来る。

こうした観点は、Coq等の証明支援言語の理論的基礎になっているばかりではなく、”Proof as Program” は、Deep Specification等の現代の「形式的手法」の中心的なコンセプトである。

Curry-Howard対応
「型」と「命題」、「要素」と「証明」との対応

ここでは、「型」と「命題」、「要素」と「証明」との対応という Curry-Haward対応をわかりやすく図解してみよう。
光子が、「波」という性質と「粒子」という性質を同時に持つことを、「双対性」というのだが、同じように、「要素：型」の世界と「証明：命題」の世界は、双対であると考えることができる。

Curry-Howard対応

「型」と「命題」、「要素」と「証明」との対応

「a は、命題 P の証明である」

証明　　　　　命題

$$\boldsymbol{a : P}$$

Curry-Howard対応

「型」と「命題」、「要素」と「証明」との対応

$$a : P$$

要素 型

「a は、型 P の要素である」

Curry-Howard対応

「型」と「命題」、「要素」と「証明」との対応

「a は、命題 P の証明である」

証明　　命題

$$\boldsymbol{a : P}$$

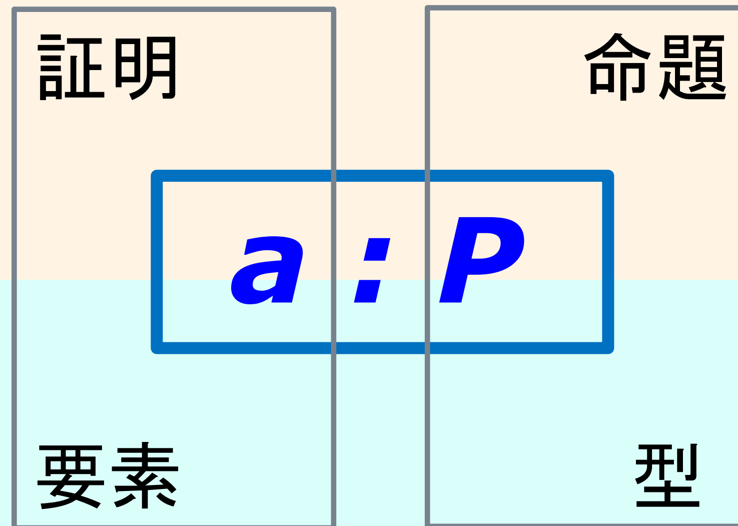
要素　　型

「a は、型 P の要素である」

Curry-Howard対応

「型」と「命題」、「要素」と「証明」との対応

「*a* は、命題 *P* の証明である」

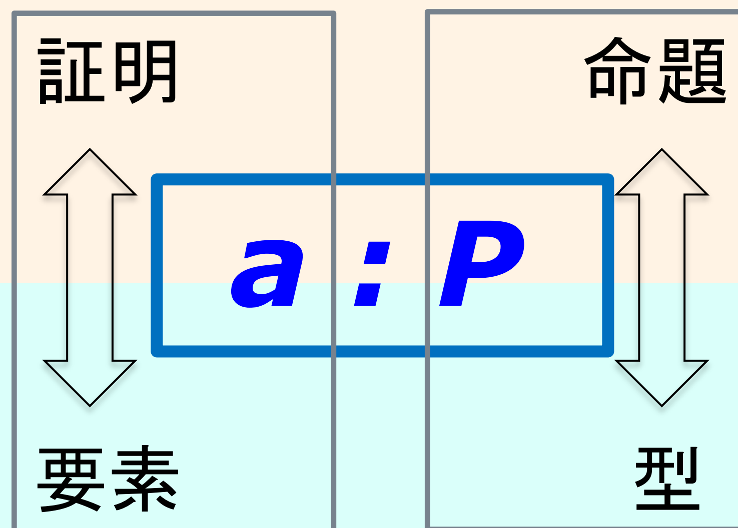


「*a* は、型 *P* の要素である」

Curry-Howard対応

「型」と「命題」、「要素」と「証明」との対応

「*a* は、命題 *P* の証明である」

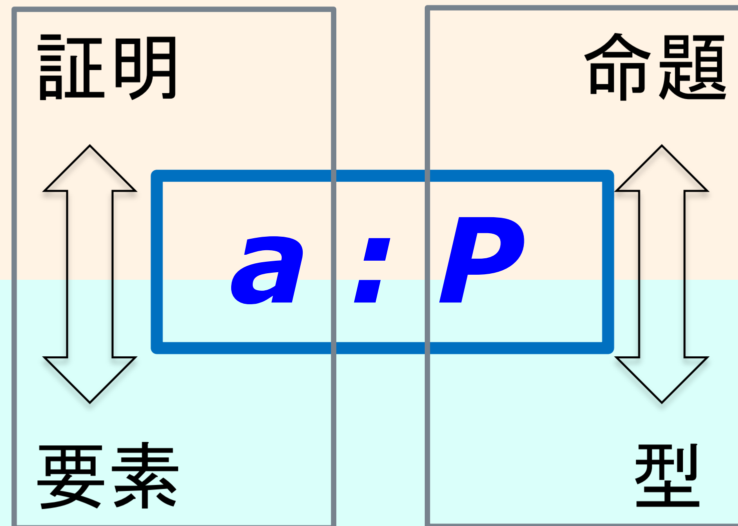


「*a* は、型 *P* の要素である」

Curry-Howard対応

「型」と「命題」、「要素」と「証明」との対応

「*a* は、命題 *P* の証明である」



証明は要素である

命題は型である

「*a* は、型 *P* の要素である」

証明支援システム Coq

Coqの開発

- Coqの開発は、Thierry Coquand とGérard Huetによって1984年から、INRIA（Institut National de Recherche en Informatique et en Automatique フランス国立情報学自動制御研究所）で開始された。その後、Christine Paulin-Mohringが加わり、40人以上の協力者がいる。
- 最初の公式のリリースは、拡張された素朴な帰納タイプのCoC 4.10で1989年。1991年には、Coqと改名された。
- それ以来、ユーザーのコミュニティは増大を続けており、豊かな型付き言語として、また、インタラクティブな定理証明システムとして、Coqのコンセプトのオリジナリティとその様々な特徴に魅了されている。

- 2013年には、ACMのSIGPLANのProgramming Languages Software Awardを授賞した。
- Coqは、数学・セマンティックス・プログラムの検証の、形式的な正確性の保証の新しい時代への移行の中で、本質的に重要な役割を果たしているのである。

現在では、Coqは、ソフトウェアの基礎科学の「普遍言語 Lingua Franca」と見なされている。

Coq コード サンプル

COQでは、全ての型は、関数の型かInductive Typeとして定義される。こうしたアプローチを CIC (Calculus of Inductive Constructions)と呼ぶ。

‘偽’ の定義

```
Inductive False : Prop := .
```

□ ⊥（偽）	証明なし
□ A ∧ B	Aの証明とBの証明の両方
□ A ∨ B	Aの証明、あるいは、Bの証明
□ A → B	Aの証明からBの証明を導く方法
□ (∀x)B(x)	任意のaに対してB(a)の証明を与える方法
□ (∃x)B(x)	あるaに対するB(a)の証明

```
Inductive True : Prop :=
   | I : True.
```

論理演算の定義

```
Definition not (A : Prop) := A -> False.
Notation "~ x" := (not x) : type_scope.
```

```
Inductive or (A B : Prop) : Prop :=
  | or_introl : A -> A ∨ B
  | or_intror : B -> A ∨ B
where "A ∨ B" := (or A B) : type_scope.
```

```
Inductive and (A B : Prop) : Prop :=
   conj : A -> B -> A ∧ B
where "A ∧ B" := (and A B) : type_scope.
```

証明される命題の例

- 先の定義から、例えば、次のような命題が証明できる。

- A -> A
- A -> (A ->B) -> B
- A -> A ∨ B
- B -> A ∨ B
- A ∨ B -> B ∨ A
- A ∧ B -> A
- A ∧ B -> B
- A ∧ B -> B ∧ A
- …

自然数と和･積の定義

```
Inductive nat : Type :=
  | 0 : nat
  | S : nat -> nat.
```

```
Fixpoint plus (n m:nat) : nat :=
  match n with
    | O => m
    | S p => S ((plus p m))
  end.
```

0 + m = m
(p+1) +m = p + m + 1

```
Fixpoint mult (n m:nat) : nat :=
  match n with
    | O => 0
    | S p => plus(m (mult p m))
  end.
```

0 * m = 0
(p+1) * m = p*m + m

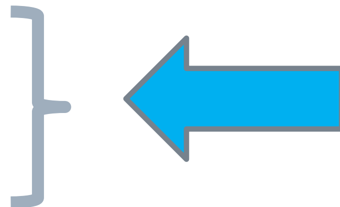
証明される命題の例

- 先の定義から、例えば、自然数について、次のような命題が証明できる。

- 0 ＋ n ＝ n
- n + 0 = n
- n + m = m + n
- 0 x n = 0
- n x 0 = 0
- n x m = m x n
- ...

COQとInductive Type

- 先のnatのInductiveとされる定義が、recursiveな定義であることはすぐわかるのだが、それと数学的帰納法のInductiveとの関係は、自明ではない。
- 実は、COQでは、Inductive Type **x** が定義されると、次の名前を持つ三つの型が、内部で自動的に定義される。
  - **x_ind**
  - **x_rect**
  - **x_rec**
- Inductive Type natの場合は、次のような三つの型が定義されることになる。
  - **nat_ind**
  - **nat_rect**
  - **nat_rec**



```
Inductive nat : Type :=
  | 0 : nat
  | S : nat -> nat.
```

nat_rect はどういう型か？

- nat_rect はどういう型か、COQのコマンド Checkで見てみよう。

```
Check nat_rect.
```

nat_rect の型をチェックする

nat_rect

$$: \forall\ P : \textbf{nat} \rightarrow \mathtt{Type},$$

$$P\ \mathsf{O} \rightarrow (\forall\ n : \textbf{nat},\ P\ n \rightarrow P\ (\mathsf{S}\ n)) \rightarrow \forall\ n : \textbf{nat},\ P\ n$$

nat_rect はどういう型か？

- nat_rect はどういう型か、COQのコマンド Checkで見てみよう。

```
Check nat_rect.
```

nat_rect の型をチェックする

nat_rect

$$: \forall P : \mathbf{nat} \to \mathrm{Type}, \quad P\ \mathsf{O} \to (\forall n : \mathbf{nat}, P\ n \to P\ (\mathsf{S}\ n)) \to \forall n : \mathbf{nat}, P\ n$$

nat上で定義されTypeに型を持つ全てのPについて

nat_rect はどういう型か？

- nat_rect はどういう型か、COQのコマンド Checkで見てみよう。

```
Check nat_rect.
```

nat_rect の型をチェックする

nat_rect

$$: \forall\ P : \mathbf{nat} \to \mathrm{Type},$$
$$\boxed{P\ \mathsf{O}} \to (\forall\ n : \mathbf{nat},\ P\ n \to P\ (\mathsf{S}\ n)) \to \forall\ n : \mathbf{nat},\ P\ n$$

nat上で定義されTypeに型を持つ全てのPについて
P O が成り立ち

nat_rect はどういう型か？

- nat_rect はどういう型か、COQのコマンド Checkで見てみよう。

```
Check nat_rect.
```

nat_rect の型をチェックする

nat_rect

$$: \forall\ P : \mathbf{nat} \to \mathrm{Type},$$
$$P\ \mathsf{O} \to \boxed{(\forall\ n : \mathbf{nat},\ P\ n \to P\ (\mathsf{S}\ n))} \to \forall\ n : \mathbf{nat},\ P\ n$$

nat上で定義されTypeに型を持つ全てのPについて
P O が成り立ち
natに属する全てのnについて、P n ならば P (S n)) が成り立つなら

nat_rect はどういう型か？

- nat_rect はどういう型か、COQのコマンド Checkで見てみよう。

```
Check nat_rect.
```

nat_rect の型をチェックする

nat_rect

$$: \forall\ P : \textbf{nat} \to \mathrm{Type},\ P\ \mathsf{O} \to (\forall\ n : \textbf{nat},\ P\ n \to P\ (\mathsf{S}\ n)) \to \forall\ n : \textbf{nat},\ P\ n$$

nat上で定義されTypeに型を持つ全てのPについて
P O が成り立ち
natに属する全てのnについて、P n ならば P (S n)) が成り立つなら
natに属する全てのnについて、P nが成り立つ、

nat_rect はどういう型か？

- nat_rect はどういう型か、COQのコマンド Checkで見てみよう。

```
Check nat_rect.
```

nat_rect の型をチェックする

nat_rect

$$: \forall\ P : \mathbf{nat} \to \mathrm{Type},$$
$$P\ \mathsf{O} \to (\forall\ n : \mathbf{nat},\ P\ n \to P\ (\mathsf{S}\ n)) \to \forall\ n : \mathbf{nat},\ P\ n$$

nat上で定義されTypeに型を持つ全てのPについて
P O が成り立ち
natに属する全てのnについて、P n ならば P (S n)) が成り立つなら
natに属する全てのnについて、P nが成り立つ、

これは、数学的帰納法に他ならない！

関連リンク

- **The Coq Proof Assistant**
  https://coq.inria.fr/
- **Coq Github**
  https://github.com/coq/coq/
- **Coq Reference Manual**
  https://coq.inria.fr/distrib/current/refman/

- **“Certified Programming with Dependent Types”**
  http://adam.chlipala.net/cpdt/
- **“Formal Reasoning About Programs”**
  http://adam.chlipala.net/frap/

第二部

プログラムと論理 – 20世紀

Agenda
第二部:　プログラムと論理 -- 20世紀

- 形式手法
  - Hoare – Hoare Logic
  - Dijkstra -- Nondeterminacy
  - Lamport – Temporal Logic
- 証明の性質をめぐる論争
- 形式的手法の後退
  - Hoare
  - Lamport

形式手法の登場

An axiomatic basis for computer programming

C.A.R Hoare

1969年

https://www.cs.cmu.edu/~crary/819-f09/Hoare69.pdf

- In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantage, both theoretical and practical, may follow from a pursuance of these topics.

An Axiomatic Basis for Computer Programming

Tony Hoare, 1969

Presented by Alexa VanHattum, Great Works in PL Spring 2019
Mentor Jonathan DiLorenzo

software bugs are bad

manual testing is not enough

formal reasoning is better

Motivation

"Computer programming is an exact science in that all the **properties of a program** and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of **purely deductive reasoning**."

http://bit.ly/2mTNDLg

Historical Context

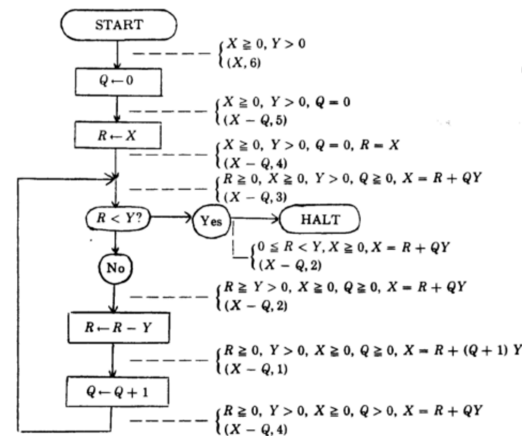


FIGURE 5. Algorithm to compute quotient Q and remainder R of $X \div Y$, for integers $X \geq 0$, $Y > 0$

"Assigning Meaning to Programs"
Robert Floyd, 1967

動機

「コンピュータ・プログラミングは、正確な科学である。その中では、プログラムの全ての性質と、どんな環境の下でそれが実行されたとしても、その実行の結果は、原理的には、プログラム自身のテキストから、純粋に演繹的推論の手段で見つけ出すことができる。」

Motivation

"Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning."

Historical Context

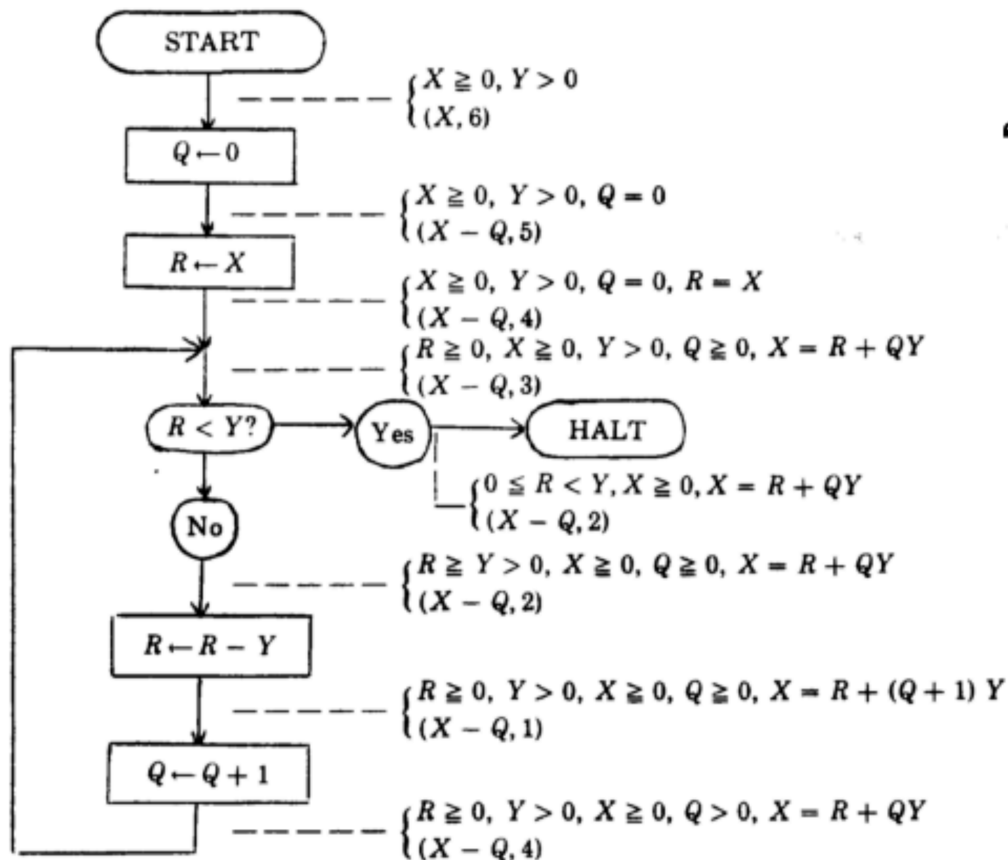


FIGURE 5. Algorithm to compute quotient Q and remainder R of $X \div Y$, for integers $X \geqq 0$, $Y > 0$

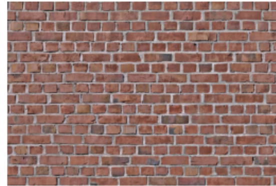
"Assigning Meaning to Programs"
Robert Floyd, 1967

"If the initial values of the program variables satisfy the relation R_1, the final values on completion will satisfy the relation R_2"

The Strategy

bra.org

economictimes.indiatimes.com


rebelwalls.com

Axioms Deductive Rules Theorems

Hoare's contribution

$$P\{Q\}R$$

Precondition Program Postcondition

If **P** holds and **Q** executes and terminates, then **R** holds

Valid Hoare Triples?

```
true {x := 1} x = 1  ✅
x = 0 {x := x + 1} x = 1  ✅
x = n {x := x * 2} x = 2n  ✅
false {x := 1} x = 0  ✅
x > 0 {while x > 1 do x := x + 1} x < 1  ❌
x > 0 {while x > 1 do x := x + 1} x = 1  ✅
```

Hoare triple

- **Empty statement axiom schema**

$$\frac{}{\{P\}\mathbf{skip}\{P\}}$$

- **Assignment axiom schema**

$$\frac{}{\{P[E/x]\}x := E\{P\}}$$

- **Rule of composition**

$$\frac{\{P\}S\{Q\} \quad , \quad \{Q\}T\{R\}}{\{P\}S;T\{R\}}$$

Hoare triple

- **Conditional rule**

$$\frac{\{B \land P\}S\{Q\} \quad , \quad \{\neg B \land P\}T\{Q\}}{\{P\}\textbf{if } B \textbf{ then } S \textbf{ else } T \textbf{ endif}\{Q\}}$$

- **Consequence rule**

$$\frac{P_1 \to P_2 \quad , \quad \{P_2\}S\{Q_2\} \quad , \quad Q_2 \to Q_1}{\{P_1\}S\{Q_1\}}$$

- **While rule**

$$\frac{\{P \land B\}S\{P\}}{\{P\}\textbf{while } B \textbf{ do } S \textbf{ done}\{\neg B \land P\}}$$

Iteration

$$\text{If } \vdash P \wedge B\{S\}P,$$
$$\text{then } \vdash P\{\text{while } B \text{ do } S\}\neg B \wedge P$$

Iteration

$$\text{If } \vdash P \wedge B\{S\}P,$$
$$\text{then } \vdash P\{\text{while } B \text{ do } S\}\neg B \wedge P$$

```
x > 0 {while x > 1 do x := x + 1} x = 1
```

Iteration

$$\text{If } \vdash P \wedge B\{S\}P,$$
$$\text{then } \vdash P\{\text{while } B \text{ do } S\}\neg B \wedge P$$

consequence rule

```
¬(x > 1) ∧ x > 0 ⇒ x = 1
x > 0 {while x > 1 do x := x + 1} ¬(x > 1) ∧ x > 0
x > 0 {while x > 1 do x := x + 1} x = 1
```

Iteration

$$\text{If } \vdash P \wedge B\{S\}P,$$
$$\text{then } \vdash P\{\text{while } B \text{ do } S\}\neg B \wedge P$$

iteration rule

```
x > 0 ∧ x > 1 {x := x + 1} x > 0
x > 0 {while x > 1 do x := x + 1} ¬(x > 1) ∧ x > 0
x > 0 {while x > 1 do x := x + 1} x = 1
```

Iteration

$$\text{If } \vdash P \wedge B\{S\}P,$$
$$\text{then } \vdash P\{\text{while } B \text{ do } S\}\neg B \wedge P$$

consequence rule

$$\dfrac{\dfrac{\texttt{x > 0} \wedge \texttt{x > 1} \Rightarrow \texttt{x + 1 > 0} \quad \dfrac{}{\texttt{x + 1 > 0 \{x := x + 1\} x > 0}}}{\texttt{x > 0} \wedge \texttt{x > 1 \{x := x + 1\} x > 0}}}{\dfrac{\texttt{x > 0 \{while x > 1 do x := x + 1\}} \neg\texttt{(x > 1)} \wedge \texttt{x > 0}}{\texttt{x > 0 \{while x > 1 do x := x + 1\} x = 1}}}$$

Iteration

$$\text{If } \vdash P \wedge B\{S\}P,$$
$$\text{then } \vdash P\{\text{while } B \text{ do } S\}\neg B \wedge P$$

assignment rule

$$\dfrac{\dfrac{\dfrac{}{\texttt{x + 1 > 0 \{x := x + 1\} x > 0}}}{\texttt{x > 0} \wedge \texttt{x > 1 \{x := x + 1\} x > 0}}}{\dfrac{\texttt{x > 0 \{while x > 1 do x := x + 1\}} \neg\texttt{(x > 1)} \wedge \texttt{x > 0}}{\texttt{x > 0 \{while x > 1 do x := x + 1\} x = 1}}}$$

Iteration

$$\text{If } \vdash P \wedge B\{S\}P,$$
$$\text{then } \vdash P\{\text{while } B \text{ do } S\}\neg B \wedge P$$

How do we find P?
Can we automate it?

Extension to Hoare Logic: Separation Logic

- Extends Hoare logic to include reasoning over shared data
- Separation conjunction $*$: P $*$ Q asserts P and Q hold for separate regions of memory

$$\frac{\{p\}\ c\ \{q\}}{\{p * r\}\ c\ \{q * r\}}$$

The frame rule (when c does not modify the free variables of r)

Conclusion

- Relate deductive reasoning to programs via Hoare triples
- Formalize/automate axiomatic reasoning via rules
- Enable pen-and-paper proofs and automated reasoning tools
- Axioms can leave aspects of the language undefined

$$P\{Q\}R$$

結論

「単純ではないプログラムに証明を与えようとする実践は、かなり強力な証明技術が利用可能になるまでは、広く広がることはないだろう。そうなったとしても、それは簡単ではないだろう。しかし、プログラム証明の実践的な利点は、プログラミングのエラーによるコストの増大という視点から見て、結局のところ、これらの困難を圧倒するだろう。」

Conclusion

"The practice of supplying proofs for nontrivial programs will not become widespread until considerably more powerful proof techniques become available, and even then will not be easy. But the practical advantages of program proving will eventually outweigh the difficulties, in view of the increasing costs of programming errors."

Guarded Commands, Nondeterminacy and Formal Derivation of Programs

Edsger W. Dijkstra
1975年

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.97&rep=rep1&type=pdf

guarded commands

- So-called "guarded commands" are introduced as a building block for alternative and repetitive constructs that allow nondeterministic program components for which at least the activity evokt, but possibly even the final state, is not necessarily uniqilely determined by the initial state. For the formal derivation of programs expressed in terms of these constructs, a calculus will be be shown.

Weakest Pre-conditions

The way in which we use predicates (as a tool for defining sets of initial or final states) for the definition of the semantics of programming language constructs has been directly inspired by Hoare [1], the main difference being that we have tightened things up a bit: while Hoare introduces sufficient pre-conditions such that the mechanisms will not produce the wrong result (but may fail to terminate), we shall introduce necessary and sufficient--i.e, so-called "weakest"--pre-conditions such that the mechanisms are guaranteed to produce the right result.

Predicate transformer

More specifically: we shall use the notation **wp(S, R),** where S denotes a statement list and R some condition on the state of the system, to denote the weakest precondition for the initial state of the system such that activation of S is guaranteed to lead to a properly terminating activity leaving the system in a final state satisfying the post-condition R. Such a wp--which is called "a predicate transformer" because it associates a pre-condition to any post-condition R

⟨guarded command⟩ ::= ⟨guard⟩ → ⟨guarded list⟩

⟨guard⟩ ::= ⟨boolean expression⟩

⟨guarded list⟩ ::= ⟨statement⟩ {; ⟨statement⟩}

⟨guarded command set⟩ ::= ⟨guarded command⟩
{□ ⟨guarded command⟩}

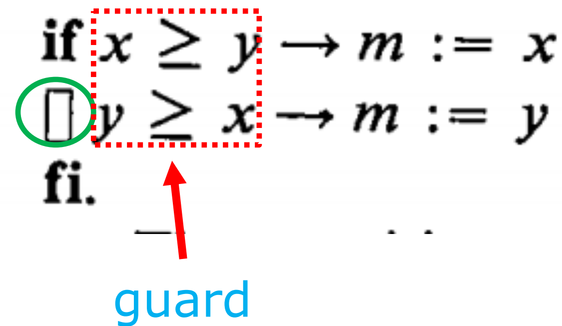
⟨alternative construct⟩ ::= **if** ⟨guarded command set⟩ **fi**

⟨repetitive construct⟩ ::= **do** ⟨guarded command set⟩ **od**

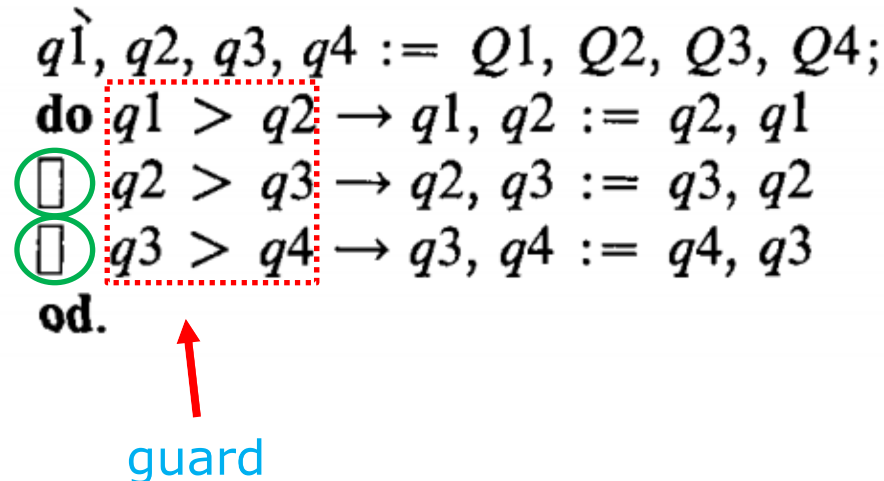
⟨statement⟩ ::= ⟨alternative construct⟩ |
⟨repetitive construct⟩ | "other statements"

If the guarded command set consists of more than one guarded command, they are mutually separated by the separator □ ; our text is then an arbitrarily ordered enumeration of an unordered set; i.e. the order in which the guarded commands of a set appear in our text is semantically irrelevant.

x,y の最大値を求める。



q1, q2 q3, q4 を小さい順に並べ替える



wp = Weakest Pre-conditions

- Skip

$$wp(\mathbf{skip}, R) \;=\; R$$

- Abort

$$wp(\mathbf{abort}, R) \;=\; \mathbf{false}$$

- Assignment

$$wp(x := E, R) \;=\; R[x \leftarrow E]$$

- Sequence

$$wp(S_1; S_2, R) \;=\; wp(S_1, wp(S_2, R))$$

wp = Weakest Pre-conditions

- Conditional

$$wp(\textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}, R)$$
$$= (E \Rightarrow wp(S_1, R)) \wedge (\neg E \Rightarrow wp(S_2, R))$$

- While loop

$$wlp(\textbf{while } E \textbf{ do } S \textbf{ done}, R)$$
$$\begin{aligned} & I \\ = & \wedge \forall y, ((E \wedge I) \Rightarrow wlp(S, I))[x \leftarrow y] \\ & \wedge \forall y, ((\neg E \wedge I) \Rightarrow R)[x \leftarrow y] \end{aligned}$$

Finally, I would like to add a word or two about the potential nondeterminacy. Having worked mainly with hardly self-checking hardware, with which nonreproducing behavior of user programs is a very strong indication of a machine malfunctioning, I had to overcome a considerable mental resistance before I found myself willing to consider nondeterministic programs seriously. It is, however, fair to say that I could never have discovered the calculus before having taken that hurdle: the simplicity and elegance of the above would have been destroyed by requiring the derivation of deterministic programs only. Whether nondeterminacy is eventually removed mechanically--in order not to mislead the maintenance engineer--or (perhaps only partly) by the programmer himself because, at second thought, he does care--e.g, for reasons of efficiency-- which alternative is chosen is something I leave entirely to the circumstances. In any case we can appreciate the nondeterministic program as a helpful stepping stone..

A Temporal Logic of Actions

Leslie Lamport
1990年

http://lamport.azurewebsites.net/pubs/old-tla-src.pdf

A Perspective by Kevin D.

Perspective by Kevin D. Jones

It is generally accepted by the software engineering community that some means of formally specifying software is an important tool in increasing confidence in its correctness, as this allows the possibility of formally proving important properties of the system with respect to the semantics of the specification.

Much work has been done in this area, with the usual route being, unsurprisingly, a passing of the torch from theoreticians to engineers.

The current state of the art for sequential systems illustrates this well. From the early “theoretical” work done in the late 60s by Floyd and Hoare, formal specification and verification of sequential programs has matured into tools like VDM and Z that are now being used in industry. The most successful of these are based on the concept of modelling state transformations in some suitable logic, usually first-order predicate calculus (or a variant thereof). This approach is both sufficiently expressive and of manageable complexity.

Concurrent programs have posed more of a challenge, and specifying concurrent systems is not yet practical. One approach has been to extend the state-based methods mentioned above by replacing simple predicate calculus with a temporal logic. Early attempts suffered from a lack of expressiveness. The common approach to this problem has been to increase the power of the temporal operators. Whilst this gives the required expressiveness, it is at the price of increased logical complexity. This raises the sophistication of the reasoning involved in verification and has resulted in these techniques being difficult to translate into practice.

In this work, the author has taken a different direction. Rather than extending the logical connectives, he has extended the base terms to include predicates on pairs of states (actions). Much of the complexity in verification now involves reasoning about actions, rather than about the temporal system. The logic has been shown to be applicable to practical problems in the verification of concurrent algorithms, and the author is continuing to work on such examples.

His approach is more in the spirit of what has been shown to be successful in the sequential world. As stated in the report, the current area of application is the verification of concurrent algorithms, rather than complete programs. This allows the possibility of machine-checked verification of important parts of any concurrent system. Eventually, one hopes that this method may be extended to permit the practical application of formal specification and verification to complete concurrent systems.

A Temporal Logic of Actions

Leslie Lamport
1990年

http://lamport.azurewebsites.net/pubs/old-tla-src.pdf

Author’s Abstract

In 1977, Pnueli introduced to computer scientists a temporal logic for reasoning about concurrent programs. His logic was simple and elegant, based on the single temporal modality “forever”, but it was not expressive enough to completely describe programs. Since then, a plethora of more expressive logics have been proposed, all with additional temporal modalities such as “next”, “until”, and “since”. Here, a temporal logic is introduced based only on Pnueli’s original modality “forever”, but with predicates (assertions about a single state) generalized to actions—assertions about pairs of states.

This logic has all the expressive power needed to describe and reason about concurrent programs. Much of the temporal reasoning required with other logics is replaced by nontemporal reasoning about actions.

States, Actions, and Temporal Formulas States and State Functions

- We assume a **set V of values**.
- **State variables** are primitive terms in the logic. They represent quantities that can change during execution of a program. The meaning [[x]] of a state variable x is a function from S to V. Intuitively, [[x]](s) is the value of x when the computing device is in state s.
- **A state function** is an expression made from state variables. Its meaning is a function from S to V. For example, if u and v are state variables, then 2u − v is the state function whose meaning [[2u − v]] is the function defined by [[2u − v]](s) $\Delta =$ 2[[u]](s) − [[v]](s), for any state s in S.

- **A state predicate** is a boolean-valued state function. A state predicate P is valid, written $\models P$, iff P is true in all states.
  Formally, $[[\models P]]$ equals $\forall s \in S : [[P]](s)$.

Actions
Actions and their Meaning

- **An action** is a boolean-valued expression containing primed and unprimed state variables. Its meaning is a boolean-valued function on $S \times S$, where unprimed state variables are applied to the first component and primed variables to the second component.
  For example, the meaning of the action $y' - x > 1$ is defined by
  $[[y' - x > 1]](s, t) \equiv [[y]](t) - [[x]](s) > 1.$

- We think of an action as specifying a set of allowed state transitions. Action A allows the transition $s \rightarrow t$ from state s to state t iff [[A]](s, t) equals true. A state transition allowed by A is called an **A transition.**

```
var  integer  x, y        = 0;
     semaphore  sem       = 1;

cobegin  loop  α1: ⟨P(sem)⟩;
               β1: ⟨x := x + 1⟩;
               γ1: ⟨V(sem)⟩
         end loop
   □
         loop  α2: ⟨P(sem)⟩;
               β2: ⟨y := y + 1⟩;
               γ2: ⟨V(sem)⟩
         end loop
coend
```

Action

$$
\begin{array}{lll}
\textbf{var} & \textbf{integer} \; x, y & = 0; \\
 & \textbf{semaphore} \; sem & = 1; \\
\textbf{cobegin} & \textbf{loop} \; \alpha_1: & \langle \mathbf{P}(sem) \rangle; \\
 & \quad \beta_1: & \langle x := x + 1 \rangle; \\
 & \quad \gamma_1: & \langle \mathbf{V}(sem) \rangle \\
 & \textbf{end loop} & \\
\Box & & \\
 & \textbf{loop} \; \alpha_2: & \langle \mathbf{P}(sem) \rangle; \\
 & \quad \beta_2: & \langle y := y + 1 \rangle; \\
 & \quad \gamma_2: & \langle \mathbf{V}(sem) \rangle \\
 & \textbf{end loop} & \\
\textbf{coend} & &
\end{array}
$$

Action

$$
\beta_2 \triangleq \begin{array}{lll}
pc_1' = pc_1 & \wedge \quad x' = x & \wedge \\
pc_2 = \beta & \wedge \quad y' = y + 1 & \wedge \\
pc_2' = \gamma & \wedge \quad sem' = sem &
\end{array}
$$

$$
\beta_2 \triangleq \begin{array}{lll}
pc_2 = \beta & \wedge \quad y' = y + 1 & \wedge \\
pc_2' = \gamma & \wedge \quad \textbf{unchanged} \; \{x, \, sem, \, pc_1\} &
\end{array}
$$

プログラムのアトミックな操作に対応した actionの例

$$\beta_2 \triangleq \begin{array}{lll} pc_2 = \beta & \wedge & y' = y + 1 \quad \wedge \\ pc_2' = \gamma & \wedge & \textbf{unchanged}\ \{x,\ sem,\ pc_1\} \end{array}$$

$$\gamma_2 \triangleq \begin{array}{lll} pc_2 = \gamma & \wedge & sem' = sem + 1 \quad \wedge \\ pc_2' = \alpha & \wedge & \textbf{unchanged}\ \{x,\ y,\ pc_1\} \end{array}$$

$$\alpha_1 \triangleq \begin{array}{lll} pc_1 = \alpha & \wedge & sem > 0 \quad \wedge \\ pc_1' = \beta & \wedge & sem' = sem - 1 \quad \wedge \\ \multicolumn{3}{l}{\textbf{unchanged}\ \{x,\ y,\ pc_2\}} \end{array}$$

Enabled Predicate

$$
\begin{array}{lcl}
\beta_2 & \triangleq & pc_2 = \beta \;\wedge\; y' = y + 1 \;\wedge \\
 & & pc_2' = \gamma \;\wedge\; \textbf{unchanged}\ \{x,\ sem,\ pc_1\}
\end{array}
$$

$$
\begin{array}{lcl}
\gamma_2 & \triangleq & pc_2 = \gamma \;\wedge\; sem' = sem + 1 \;\wedge \\
 & & pc_2' = \alpha \;\wedge\; \textbf{unchanged}\ \{x,\ y,\ pc_1\}
\end{array}
$$

$$
\begin{array}{lcl}
\alpha_1 & \triangleq & pc_1 = \alpha \;\wedge\; sem > 0 \;\wedge \\
 & & pc_1' = \beta \;\wedge\; sem' = sem - 1 \;\wedge \\
 & & \textbf{unchanged}\ \{x,\ y,\ pc_2\}
\end{array}
$$

$$
\begin{array}{lcl}
Enabled(\beta_2) & = & pc_2 = \beta \\
Enabled(\gamma_2) & = & pc_2 = \gamma \\
Enabled(\alpha_1) & = & pc_1 = \alpha \;\wedge\; sem > 0
\end{array}
$$

behavior

A *behavior* is an infinite sequence of states; the set of all behaviors is denoted by $\mathbf{S}^\omega$. If σ is the behavior $s_0, s_1, \ldots,$ then σ_i denotes the ith state s_i. The ith *step* of σ is the state transition $\sigma_{i-1} \rightarrow \sigma_i$. It is called a *stuttering step* iff states σ_{i-1} and σ_i are equal.

$$\begin{aligned} [\![\neg F]\!](\sigma) &\triangleq \neg [\![F]\!](\sigma) \\ [\![F \wedge G]\!](\sigma) &\triangleq [\![F]\!](\sigma) \wedge [\![G]\!](\sigma) \end{aligned}$$

A formula F is valid iff it is true for all behaviors:

$$[\![\models F]\!] \triangleq \forall \sigma \in \mathbf{S}^\omega : [\![F]\!](\sigma)$$

operator $\Box$ (usually read "always")

operator $\Diamond$ (read "eventually")

The temporal logic operator $\Box$ (usually read "always") is defined as follows. If σ is a behavior, let σ^{+i} denote the behavior $\sigma_i, \sigma_{i+1}, \ldots$ obtained by cutting off the first i states in the sequence σ. For any formula F and behavior σ,

$$[\![\Box F]\!](\sigma) \;\triangleq\; \forall i \geq 0 : [\![F]\!](\sigma^{+i}) \tag{5}$$

Intuitively, a temporal formula F holds at a certain time iff F holds for the infinite behavior starting at that time. The formula $\Box F$ asserts that F holds at all times—now and in the future.

The operator $\Diamond$ (read "eventually") is defined by $\Diamond F \triangleq \neg\Box\neg F$. Intuitively, $\Diamond F$ asserts that F holds now or at some time in the future.

operator $\leadsto$ (read "leads to"),

The operators $\Box$ and $\Diamond$ can be nested and combined with logical operators to provide more complicated temporal modalities. For example, $\Box\Diamond F$ asserts that at all times, F must be true then or at some future time. In other words, $\Box\Diamond F$ as operator $\leadsto$ (read "leads to"), often. Of particularly interest is the operator $\leadsto$ (read "leads to"), where $F \leadsto G \triangleq \Box(F \Rightarrow \Diamond G)$. Intuitively, $F \leadsto G$ asserts that whenever F is true, G is true then or at some later time.

Temporal Reasoning

Invariance Rule

$$\frac{\{P\}\mathcal{A}\{P\}}{\Box[\mathcal{A}] \;\Rightarrow\; (P \Rightarrow \Box P)}$$

- The hypothesis asserts that any A transition with P true in the starting state has P true in the ending state. The conclusion asserts that if every nonstuttering step is an A transition, then P true initially implies that it remains true forever. Observe that the hypothesis is an action, while the conclusion is a temporal formula.

Expressing Programs as Temporal Formulas

A program Π is described by four things:

1. A collection of **state variables**.
2. A **state predicate** $Init_\Pi$ specifying the initial state.
3. An **action** N_Π specifying the state transitions allowed by the program.
4. A **temporal formula** L_Π specifying the program's progress condition.

The program itself is the temporal logic formula Π defined by

$$\Pi \triangleq Init_\Pi \wedge \Box[\mathcal{N}_\Pi] \wedge L_\Pi$$

Reasoning with Fairness

WF Rule

$$\frac{\begin{array}{l}\{P\}\ \mathcal{A}\ \{Q\} \\ \{P\}\ \mathcal{N} \land \neg\mathcal{A}\ \{P \lor Q\} \\ P \Rightarrow \mathit{Enabled}(\mathcal{A})\end{array}}{\Box[\mathcal{N}] \land \mathrm{WF}(\mathcal{A}) \ \Rightarrow\ (P \leadsto Q)}$$

SF Rule

$$\frac{\begin{array}{l}\{P\}\ \mathcal{A}\ \{Q\} \\ \{P\}\ \mathcal{N} \land \neg\mathcal{A}\ \{P \lor Q\} \\ \Box F \land \Box[\mathcal{N} \land \neg\mathcal{A}] \Rightarrow (P \land \neg\mathit{Enabled}(\mathcal{A}) \leadsto \mathit{Enabled}(\mathcal{A}))\end{array}}{\Box F \land \Box[\mathcal{N}] \land \mathrm{SF}(\mathcal{A}) \ \Rightarrow\ (P \leadsto Q)}$$

Reasoning About Programs

- Safety Properties: type-correct

The first safety property one usually proves about a program is that it is *type-correct*, meaning that the values of all variables are of the expected "type". Type-correctness for the program of Figure 1 is expressed by the formula $\Pi \Rightarrow \Box T$, where

$$\begin{array}{rcl} T & \triangleq & pc_1 \in \{\alpha, \beta, \gamma\} \quad \wedge \quad x \in \mathbf{Int} \quad \wedge \quad sem \in \mathbf{Nat} \quad \wedge \\ & & pc_2 \in \{\alpha, \beta, \gamma\} \quad \wedge \quad y \in \mathbf{Int} \end{array} \tag{9}$$

- Liveness Properties

we prove

$$\Pi \ \Rightarrow \ (x = n \leadsto x = n + 1)$$

Liveness Properties

A. Control in process 1 is either at α_1, β_1, or γ_1.

B. If control is at γ_1 with x equal to n, then eventually (after executing γ_1) control will be at α_1 with x equal to n.

C. If control is at α_1 with x equal to n, then eventually (after executing α_1) control will be at β_1 with x equal to n.

D. If control is at β_1, then eventually (after executing β_1) x will equal $n + 1$.

Liveness Properties

A. $\Pi \;\Rightarrow\; (x = n) \leadsto$
$((pc_1 = \gamma \wedge x = n) \vee (pc_1 = \alpha \wedge x = n) \vee (pc_1 = \beta \wedge x = n))$

B. $\Pi \;\Rightarrow\; (pc_1 = \gamma \wedge x = n) \leadsto (pc_1 = \alpha \wedge x = n)$

C. $\Pi \;\Rightarrow\; (pc_1 = \alpha \wedge x = n) \leadsto (pc_1 = \beta \wedge x = n)$

D. $\Pi \;\Rightarrow\; (pc_1 = \beta \wedge x = n) \leadsto (x = n + 1)$

General Logic

$$
\begin{aligned}
Init_\Phi &\triangleq x = 0 \land y = 0 \\
\mathcal{N}_\Phi^1 &\triangleq x' = x + 1 \land y' = y \\
\mathcal{N}_\Phi^2 &\triangleq y' = y + 1 \land x' = x \\
\mathcal{N}_\Phi &\triangleq \mathcal{N}_\Phi^1 \lor \mathcal{N}_\Phi^2 \\
\Phi &\triangleq Init_\Phi \land \Box[\mathcal{N}_\Phi] \land \mathrm{WF}(\mathcal{N}_\Phi^1) \land \mathrm{WF}(\mathcal{N}_\Phi^2)
\end{aligned}
$$

$$
\begin{aligned}
\Pi &\triangleq Init_\Pi \land \Box[\mathcal{N}_\Pi]_{\mathbf{w}} \land \mathrm{SF}_{\mathbf{w}}(\mathcal{N}_\Pi^1) \land \mathrm{SF}_{\mathbf{w}}(\mathcal{N}_\Pi^2) \\
\Phi &\triangleq Init_\Phi \land \Box[\mathcal{N}_\Phi]_{\{x,y\}} \land \mathrm{WF}_{\{x,y\}}(\mathcal{N}_\Phi^1) \land \mathrm{WF}_{\{x,y\}}(\mathcal{N}_\Phi^2)
\end{aligned}
$$

With these definitions, the formula $\Pi \Rightarrow \Phi$ is valid.

証明の性質をめぐる論争

Social Processes and Proofs of Theorems and Programs

Richard A. De Millo et al.

1979年

https://www.cs.umd.edu/~gasarch/BLOGPAPERS/social.pdf

形式的手法をめぐる「論争」

- 70年代のホーアやダイクストラのコンピュータ・サイエンスに「形式的手法」を導入しようという試みは、必ずしも広く受け入れられた訳ではなかった。
- 1979年、ACMは、その論文誌CACMに次のような内容の論文を掲載する。
- 「この論文では、プログラムの形式的検証は、たとえそれが得られたとしても、計算機科学とソフトウェア・エンジニアリングにおいては、数学での証明と同じ役割は果たす事はないと議論される。更に言えば、連続性の欠如、変化の不可避性、決定的に多くの現実のプログラムの仕様の複雑さは、形式的検証の過程を、正当化することも管理することも困難にする形式的検証の使いやすさは、プログラム言語の設計に大きな影響力を持つことはないように感じられる。」

https://www.cs.umd.edu/~gasarch/BLOGPAPERS/social.pdf

「証明」の役割について

- ホーア、ダイクストラ、ランポートらの「形式的手法」に反対する先の論文の著者たちは、「証明」について、次のような考えを持っていた。
- 「数学では、証明の目的は、ある定理の正しさに対する人の確信を増大させることである。確かに、数学者がこの目的のための道具の一つとして、理論的に利用することができるのは、形式論理の長い連鎖である。
しかし、事実はそうではない。彼らが使う証明は、全く別の生き物なのだ。証明は、問題を解決さえしない。
証明は、その名が示唆するものとは逆に、確信に向かう一つのステップにすぎないのだ。結局のところ、数学者がある定理に確信を持つか否かを決定するのは、社会的プロセスなのだと我々は信じている。」

Social Process ?

In mathematics, the aim is to increase one's confidence in the correctness of a theorem, and it's true that one of the devices mathematicians could in theory use to achieve this goal is a long chain of formal logic. But in fact they don't. What they use is a proof, a very different animal. Nor does the proof settle the matter; contrary to what its name suggests, a proof is only one step in the direction of confidence. We believe that, in the end, it is a social process that determines whether mathematicians feel confident about a theorem --- and we believe that, because no comparable social process can take place among program verifiers, program verification is bound to fail. We can't see how it's going to be able to affect anyone's confidence about programs.

https://www.cs.umd.edu/~gasarch/BLOGPAPERS/social.pdf

「社会的なプロセス」

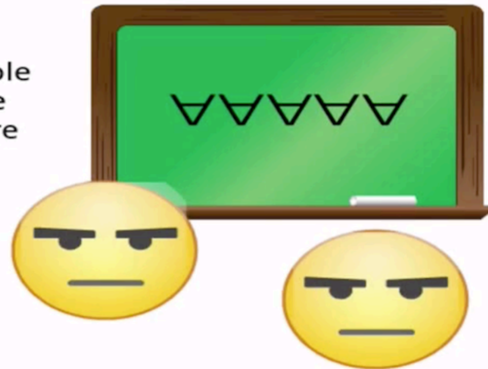
- 彼らの論文のタイトルに、「社会的なプロセス」とあったのは、こういう意味だったのだ。
- 「プログラムの検証者たちの間に、数学と比較できるような社会的プロセスが存在しないのだから、プログラムの検証は、失敗するように定められていると、我々は信じている。プログラムについての誰かの確信に、証明がどのように影響を与えるのか、我々は見ることができないのだ。」
- （僕の理解では、たいていの数学者にとって、「定理に対する確信」は、まず、最初に直観的に生まれるものだ。だから、それを証明しようとする。それに、数学者は、IT業界の人のように、コミュニティや学会に集まって「社会的プロセス」での合意をつくるのが好きなようには見えないのだが。）

- 数学論的には全くの愚論である。ACMは、計算機科学ではもっとも「権威ある」学会の一つなのだが、そこでも、一時期、こうした議論が行われていたことには注意が必要だと思う。
- なぜなら、それは現実の多数の開発者の心情を代弁している可能性があるからである。
- 一方、「形式的手法」の側は、ランポートだけでなく、あとで紹介するダイクストラを含めて、この論文が論文として学会誌に受容されたことにショックを受けたように思う。
- 逆の意味で、40年近く経った今も、この論文は「有名」なのである。写真は、この論文を「紹介」する、チッパラ。

Q: Aren't These Proofs Too Boring for Mortals?

It is argued that formal verifications of programs, no matter how obtained, will not play the same key role in the development of computer science and software engineering as proofs do in mathematics. Furthermore the absence of continuity, the inevitability of change, and the complexity of specification of significantly many real programs make the formal verification process difficult to justify and manage.

– De Millo, Lipton, and Perlis, "Social Processes and Proofs of Theorems and Programs," CACM, 1979



12

34C3: tuwat!

ランポートの抗議

- この論文、“Social Processes and Proofs of Theorems and Programs” De Millo, Lipton and Perils https://www.cs.umd.edu/~gasarch/BLOGPAPERS/social.pdf は、「プログラムの検証なんて、必ず失敗する。"program verification is bound to fail. "」と断じたのだ。

- ランポートは、直ちにACMの編集部に、抗議のメールを送る。"Letter to the Editor" https://www.microsoft.com/.../publicati.../letter-to-the-editor/

ランポートの抗議
Letter to the Editor -- Leslie Lamport

- In the May, 1979 CACM, De Millo, Lipton, and Perlis published an influential paper titled Social Process and Proofs of Theorems and Programs. This paper made some excellent observations. However, by throwing in a few red herrings, they came to some wrong conclusions about program verification. More insidiously, they framed the debate as one between a reasonable engineering approach that completely ignores verification and a completely unrealistic view of verification advocated only by its most naive proponents. (There were, unfortunately, quite a few such proponents.)

https://www.microsoft.com/en-us/research/publication/letter-to-the-editor/

- The purpose of my letter was to express my dismay. I ironically suggested that they had succumbed to the arguments of De Millo, Lipton, and Perlis in their policy. As a result, my letter was published as a rebuttal to the De Millo, Lipton, and Perlis paper. No one seems to have taken it for what it was–a plea to alter the ACM algorithms policy to require that there be some argument to indicate that an algorithm worked.

https://www.microsoft.com/en-us/research/publication/letter-to-the-editor/

Dijkstraの評価(2001年）

Hilbert's revolution was in any case to redefine "proof" to become a completely rigorous notion, totally different from the psycho/sociological "A proof is something that convinces other mathematicians.". A major shortcoming of the latter view is that it gives no technical guidance for proof design and makes it very difficult to teach that kind of mathematics. Yet, or perhaps for this reason, many of the more conservative mathematicians still cling to this form of consensus mathematics; they will even vigorously defend their informality.

形式主義の後退

Hoare論文から30年後のHoare

- 昨日紹介したホーアのスライドが面白いのは、スライドの末尾に、論文から30年たった時点での形式的手法への、彼の評価が追加されていることだ。

30 years later …

“Researchers into formal methods […] predicted that the programming world would embrace with gratitude every assistance promised by formalization to solve the problems of reliability that arise when programs get large and more safety-critical […]

It has turned out that the world just does not suffer significantly from the kind of problem that our research was originally intended to solve.”

Tony Hoare, 1996

Hoare論文から30年後のHoare

- 「形式手法に入った研究者たちは、プログラムが大規模になりもっと安全性が重要になった時に生まれる信頼性の問題を解決するという、形式化によるすべての支援の約束を、プログラミングの世界は、感謝をこめて抱きしめるだろうと予言していた。」
- 「しかし、世界は、我々の研究がもともと解決しようと意図した類の問題には、深刻には苦しめられていないことが明らかになったのである。」
- 彼のこうした立場は、次の1996年の論文では、さらに明確に述べられる。

"How did software get so reliable without proof ? "

C.A.R Hoare
1996年

https://www.gwern.net/docs/math/1996-hoare.pdf

- 「最近のマッキンゼーの分析は、コンピュータに依存したことに起因する可能性もあると、これまで考えられていた数千人の死亡事故のうち、ソフトウェアのエラーで説明できる死者は、10人程度にすぎないことを明らかにした。」
- その上で、ホーアは、ソフトウェアが「こんなにも信頼できる」ようになった要因を列挙する。
  - 管理手法の改革
  - テスト
  - デバッグ
  - 多くの技術の投入
  - プログラミングの方法論

- それぞれの展開は、説得力もあり、なかなか興味ふかいものだ。
- ただ、それは現在のソフトウェア・エンジニアリングのスタイルの基本が、こうして30年近く前には、形成されていたことを示すものだ。
- 30年も経てば、いろんなものが変わっていく。
- 現在の我々は、再び、「ソフトウェアは安全なのか？」と言う問いかけに、あるいは、チッパラの言う「開発者が、テスト・デバッグ・コードレビューを繰り返すことで、バグのないプログラムが作れるのか？」という問いかけの前に立たされているのだ。

Management

- The most dramatic advances in the timely delivery of dependable software are directly attributed to a wider recognition of the fact that the process of program development can be predicted, planned, managed and controlled in the same way as in any other branch of engineering. The eventual workings of the program itself are internal to a computer and invisible to the naked eye; but that is no longer any excuse for keeping the design process out of the view of management; and the visibility should preferably extend to all management levels up to the most senior.

Success in the use of mathematics for specification, design and code reviews does not require strict formalisation of all the proofs. Informal reasoning among those who are fluent in the idioms of mathematics is extremely efficient, and remarkably reliable. It is not immune from failure; for example simple misprints can be surprisingly hard to detect by eye. Fortunately, these are exactly the kind of error that can be removed by early tests. More formal calculation can be reserved for the most crucial issues, such as interrupts and recovery procedures, where bugs would be most dangerous, expensive, and most difficult to diagnose by tests.

A facility in formalisation and effective reasoning is only one of the talents that can help in a successful review. There are many other less formal talents which are essential. They include a wide understanding of the application area and the marketplace, an intuitive sympathy with the culture and concerns of the customer, a knowledge of the structure and style of existing legacy code, acquaintance and professional rapport with the most authoritative company experts on each relevant topic, a sixth sense for the eventual operational consequences of early design decisions, and above all, a deep sense of personal commitment to quality, and the patience to survive long periods of intellectual drudgery needed to achieve a thoroughly professional result.

Testing

- Thorough testing is the touchstone of reliability in quality assurance and control of modern production engineering. Tests are applied as early as possible at all stations in the production line. They are designed rigorously to maximise the likelihood of failure, and so detect a fault as soon as possible. For example, if parameters of a production process vary continuously, they are tested at the extreme of their intended operating range. Satisfaction of all tests in the factory affords considerably increased confidence, on the part of the designer, the manufacturer, and the general public, that the product will continue to work without fail throughout its service lifetime. And the confidence is justified: modern consumer durables are far more durable than they were only twenty years ago.

- But computing scientists and philosophers remain skeptical. E.W. Dijkstra has pointed out that program testing can reveal only the presence of bugs, never their absence. Philosophers of science have pointed out that no series of experiments, however long and however favourable can ever prove a theory correct; but even only a single contrary experiment will certainly falsify it. And it is a basic slogan of quality assurance that "you cannot test quality into a product". How then can testing contribute to reliability of programs, theories and products? Is the confidence it gives illusory?

Debugging

- The secret of the success of testing is that it checks the quality of the process and methods by which the code has been produced. These must be subjected to continued improvement, until it is normal to expect that every test will be passed first time, every time. Any residual lapse from this ideal must be tracked to its source, and lead to lasting and widely propagated improvements in practice. Expensive it may be, but that too is part of the cure. In all branches of commerce and industry, history shows dramatic reduction in the error rates when their cost is brought back from the customer to the perpetrator.

Over-engineering

- The first benefit of a superabundance of resource is to make possible a decision to avoid any kind of sophistication or optimisation in the design of algorithms or data structures.
- Profligacy of resources can bring benefits in other ways. When considering a possible exceptional case, the programmer may be quite confident that it has already been discriminated and dealt with elsewhere in some other piece of code; as a result in fact the exception can never arise at this point. Nevertheless, for safety, it is better to discriminate again, and write further code to deal with it.

- Another profligate use of resources is by cloning of code. A new feature to be added to a large program can often be cheaply implemented by making a number of small changes to some piece of code that is already there. But this is felt to be risky: the existing code is perhaps used in ways that are not at all obvious by just looking at it, and any of these ways might be disrupted by the proposed change.

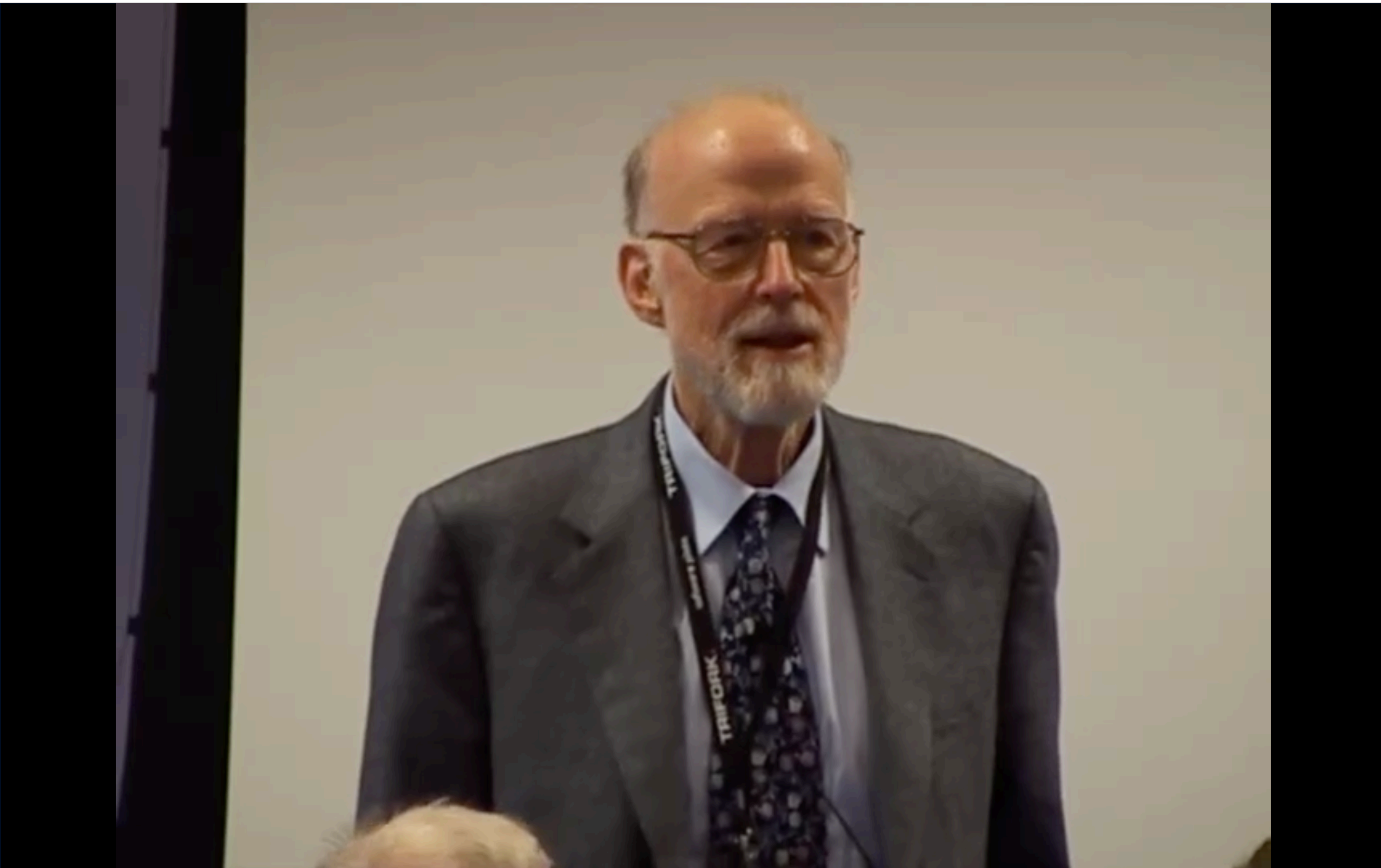
Programming Methodology

- Most of the measures described so far for achieving reliability of programs are the same as those which have proved to be equally effective in all engineering and industrial enterprises, from space travel to highway maintenance, from electronics to the brewing of beer. But the best general techniques of management, quality control, and safety engineering would be totally useless, by themselves; they are only effective when there is a general understanding of the specific field of endeavour, and a common conceptual framework and terminology for discussion of the relationship between cause and effect, between action and consequence in that field.

- Perhaps initially, the understanding is based just on experience and intuition; but the goal of engineering research is to complement and sometimes replace these informal judgements by more systematic methods of calculation and optimisation based on scientific theory.
- Research into programming methodology has a similar goal, to establish a conceptual framework and a theoretical basis to assist in systematic derivation and justification of every design decision by a rational and explicable train of reasoning.

- The primary method of research is to evaluate proposed reasoning methods by their formalisation as a collection of proof rules in some completely formal system. This permits definitive answers to the vital questions: is the reasoning valid? is it adequate to prove everything that is needed? and is it simpler than other equally valid and adequate alternatives? It is the provably positive answer to these simple questions that gives the essential scientific basis for a sound methodological reeornmendation- certainly an improvement on mere rhetoric, speculation, fashion, salesmanship, charlatanism or worse.

Null References: The Billion Dollar Mistake



http://bit.ly/2oXpgNz

The Billion Dollar Mistake
25 Aug 2009 InfoQ

- Null references have historically been a bad idea
- Early compilers provided opt-out switches for run-time checks, at the expense of correctness
- Programming language designers should be responsible for the errors in programs written in that language
- Customer requests and markets may not ask for what's good for them; they may need regulation to build the market
- If the billion dollar mistake was the null pointer, the C gets function is a multi-billion dollar mistake that created the opportunity for malware and viruses to thrive

"Formalism and method"

Egidio Astesiano, Gianna Reggio

Theoretical Computer Science 236 (2000) 3–34

https://core.ac.uk/download/pdf/82747498.pdf

- プログラミングが論理＝数学に基づく「正確な科学」だとしても、ソフトウェア・エンジニアリング全体が、そのまま数理科学に還元されるわけではない。
- それは、橋を作るのに力学の知識は必須だが、土木工学＝力学となるわけではないのと同じである。土木工学もソフトウェア・エンジニアリングも、論理＝数学や力学という科学の領域ではなく、技術の領域に属する。
- 科学と技術の関係をどう捉えるかは重要な問題である。もっというと、科学と技術とビジネスの関係は、現代では、さらに重要である。ここでは、こうした問題については一般的には触れない。

- 具体的に「プログラミングの科学」と「ソフトウェア・エンジニアリング」について考えよう。
- 重要なことは、土木工学の基礎に力学があるように、ソフトウェア・エンジニアリングの基礎には、「プログラミングの科学」があると言うことだと思う。
- この間、形式手法をめぐる、20~30年前の論争をチェックしているのだが、次の論文は興味深かった。
- 形式手法が、ソフトウェア開発で経験的に蓄積された様々のノウハウ（先の投稿でホーアがまとめたような）やソフトウェア開発のコンテキストを無視するのは、よくないという意見への反論だと思う。

- 形式的仕様を与えるにあたって、コンテキストを考えるのは当然で、その上で形式的モデルと形式的仕様を考えるとしている。

ACTIVITY

To give a formal specification

CONTEXT

END PRODUCTS $\mathscr{EP}$	the kind of the end-products of the development process
LOCATION	qualification and location of the activity in the development process

FORMALISM

FORMAL MODELS $\mathscr{M}$	mathematical strucutres representing the end-products
SPECIFICATIONS $\mathscr{SPEC}$, $[\![_]\!]$	specifications as artifacts

- さらに、形式的モデルがいかに最終製品をモデル化したのかを示す「モデリング」、形式的仕様作成のための「ガイドライン」、形式的仕様を人間に説明するための「プレゼンテーション」、作業のまとめとしての「ドキュメント」が必要だとしている。

FORMALISM

FORMAL MODELS	$\mathscr{M}$	mathematical strucutres representing the end-products
SPECIFICATIONS	$\mathscr{SPEC}$, $[\![_]\!]$	specifications as artifacts

IMPACT ON METHOD

PRAGMATICS

MODELLING	$\leftrightharpoons$	how the formal models model the end-products
GUIDELINES		guidelines for the specification task
PRESENTATION		presentation of the specifications for humans
DOCUMENTATION		documenting the performed task

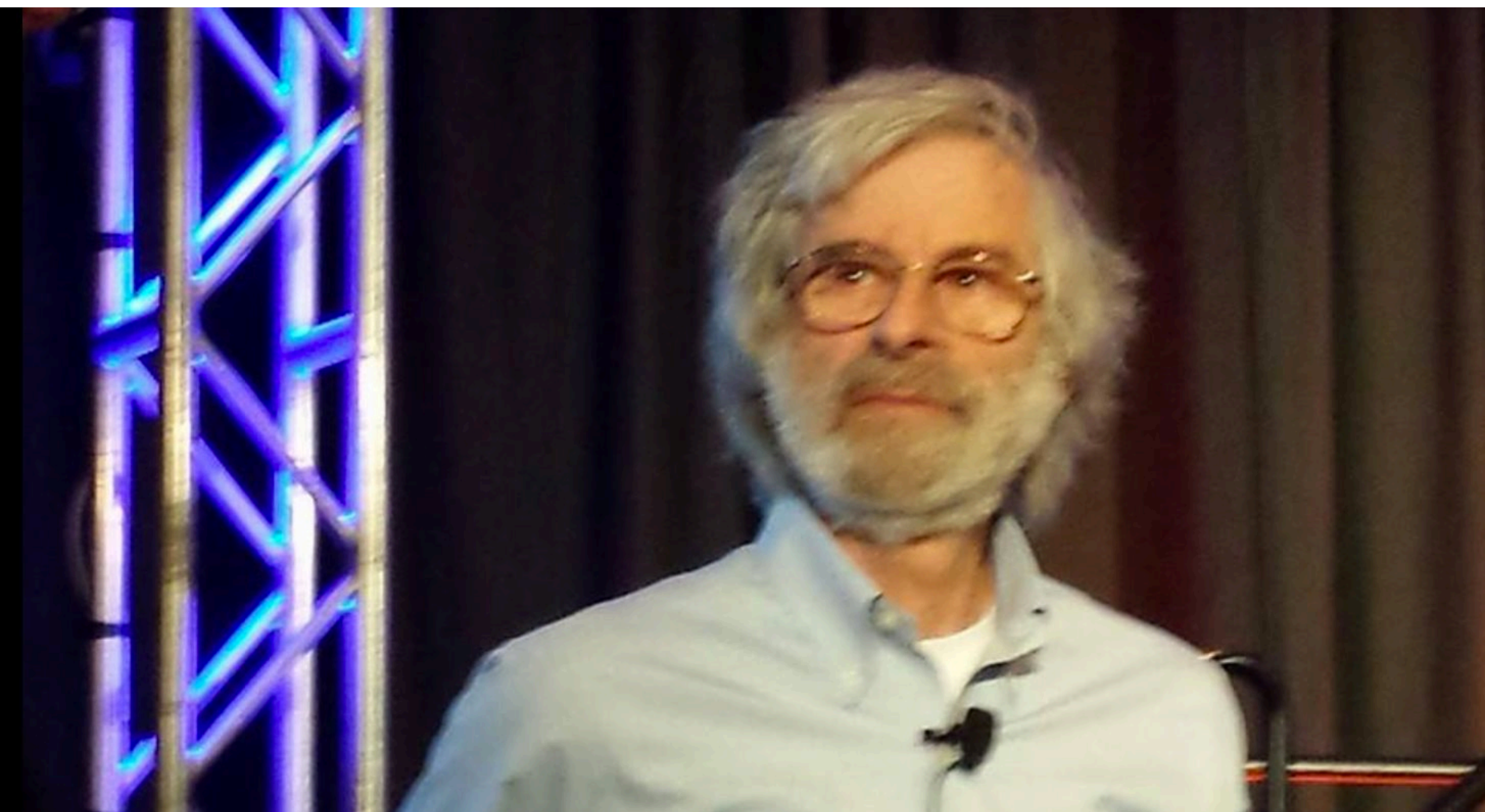
- 僕が興味を持ったのは、こうした「モデリング」「ガイドライン」「プレゼンテーション」「ドキュメント」を、基礎としての「形式主義」との対比で「プラグマティックス」と呼んでいることである。
- 「形式主義」と「プラグマチックス」の二つ合わせたものが、形式手法のソフトウェア・エンジニアリングを構成するという考えなのだと思う。
- 経験から得られる「プラグマティックス」だけでは、ソフトウェア・エンジニアリングは完結しない。「プラグマティックス」だけでは、安全で堅牢なソフトウェアを作る保証にはならない。
- それは、「流行の」「優れたデザインで」「人気の」橋を作っても、ましてや「設計の経験がない人でも」「安くできる」橋を作っても、その橋が安全な橋になる保証がないのと同じである。

Composition: A Way to Make Proofs Harder

Leslie Lamport

24 December 1997

https://lamport.azurewebsites.net/pubs/lamport-composition.pdf



Lamport

- 80年代の「形式的手法」のムーブメントはやがて衰退する。皮肉なことに、それに終止符を打った一人は、かつてはホーア、ダイクストラらとともに形式的手法を推進したランポートだった。
- 1997年の論文で、彼は次のように語る。

"Composition: A Way to Make Proofs Harder"

- 「現実のシステムの設計にエラーを見つけるという問題に直面した時、我々は何をなすべきか？　完全な設計というものは、形式的的手法で扱うには、ほとんど全ての場合、あまりに複雑すぎる。」
- 「1997年の時点では、不幸な現実は、技術者が自分が構築するシステムについて、形式的に仕様を決めたり推論することがほとんどないということである。オープン・システムの仕様の組み合わせによる構成について推論することが、実践的な関心にのぼることは、次の15年の間には、ありそうもない。」

Lamportのスタンスと問題提起

- ただ、Hoareの立論が、理論的というより「プラグマティック」なもので、形式手法に反対して、当時確立しつつあったソフトウェア・エンジニアリングの手法（それは、多少の形を変えながら現在に続いている）を楽観的に擁護するものだったのに対して、Lamportの問題意識は違っているように見える。
- 彼は、当時の形式手法が抱えていた問題を、理論的に整理している。論文を見てみよう。

Abstract

- Compositional reasoning about a system means writing its specification as the parallel composition of components and reasoning separately about each component. When distracting language issues are removed and the underlying mathematics is revealed, compositional reasoning is seen to be of little use.

6.3 Why Bother?
擬似プログラミング言語の使用への批判

- Many computer scientists believe that their favorite pseudo-programming language is better than mathematics because it provides wonderful abstractions such as message passing, or synchronous communication, or objects, or some other popular fad. For centuries, bridge builders, rocket scientists, nuclear physicists, and number theorists have used their own abstractions. They have all expressed those abstractions directly in mathematics, and have reasoned "at the semantic level". Only computer scientists have felt the need to invent new languages for reasoning about the objects they study

実装の正しさを証明しようとすると、証明が非常に複雑になるという指摘

- Two empirical laws seem to govern the difficulty of proving the correctness of an implementation, and no pseudo-programming language is likely to circumvent them: (1) the length of a proof is proportional to the product of the length of the low-level specification and the length of the invariant, and (2) the length of the invariant is proportional to the length of the low-level specification. Thus, the length of the proof is quadratic in the length of the low-level specification.

証明を複雑にしないために、高い抽象度で仕様を書けば、現実のシステムの問題を検出できない

- The most effective way to reduce the length of an implementation proof is to reduce the length of the low-level specification. A specification is a mathematical abstraction of a real system. When writing the specification, we must choose the level of abstraction. A higher-level abstraction yields a shorter specification. But a higher-level abstraction leaves out details of the real system, and a proof cannot detect errors in omitted details. Verifying a real system involves a tradeoff between the level of detail and the size (and hence difficulty) of the proof.

Lamportのこの時点での結論

- この時点で、Lamportはどのような結論を出したのだろうか？1997年の論文で、彼は次のように結論づける。

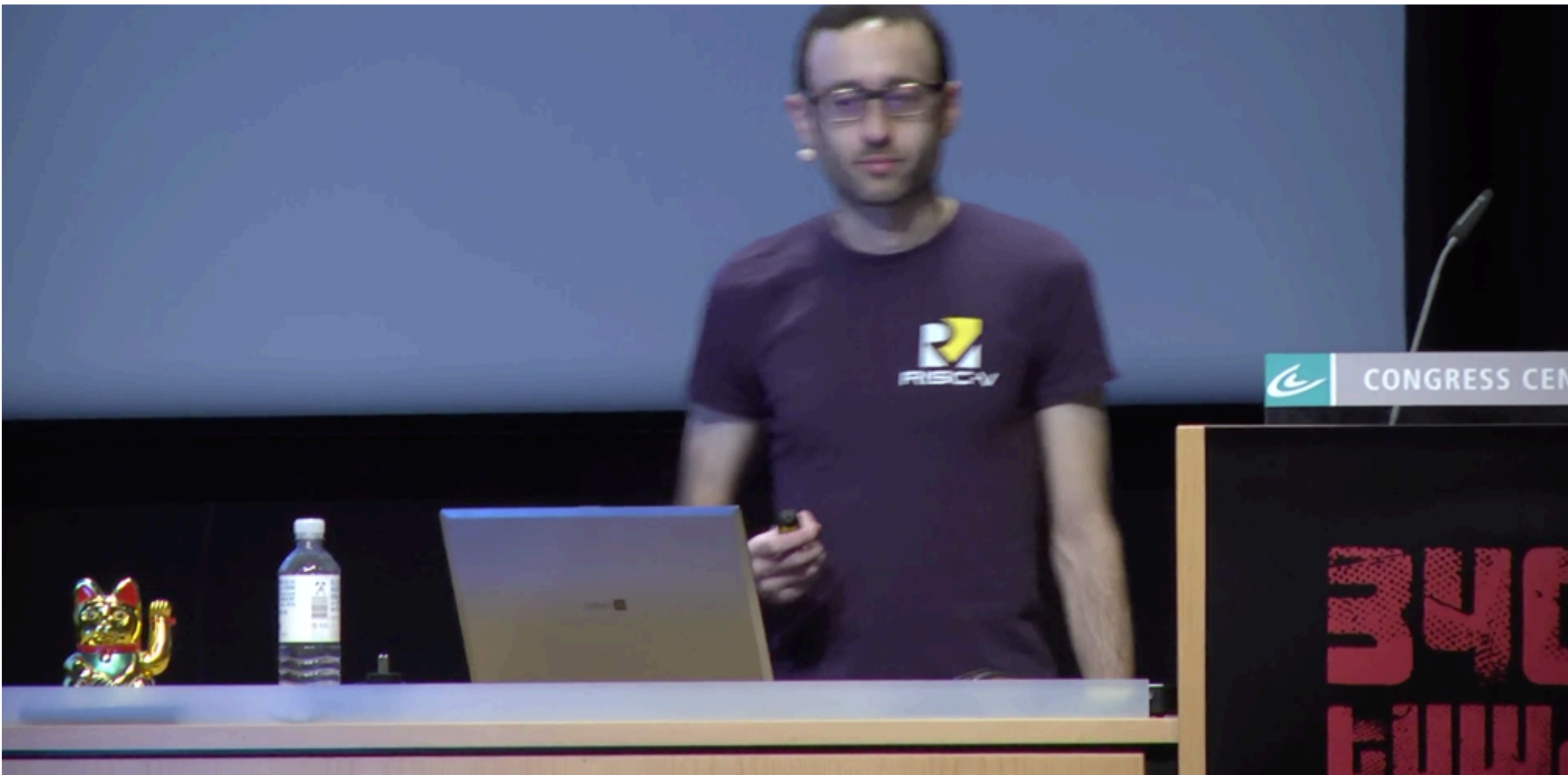
Conclusion

現実のシステムの設計にエラーを見つけるという問題に直面した時、我々は何をなすべきか？　完全な設計というものは、形式的的手法で扱うには、ほとんど全ての場合、あまりに複雑すぎる。我々は、与えられた限られた時間と利用可能なマンパワーのもとで、設計の出来るだけ多くを表現する抽象について推論しなければならない。理想的なアプローチは、コンピュータに検証をさせることだ。それはモデル・チェックを意味する。モデル・チェッカーは、仕様の限られたクラスを扱うことができるだけだ。

これらの仕様は、一般的には小さく単純なので、どの言語を使って書かれていても大きな差はない。伝統的な数学でも擬似プログラム言語でもうまくいくだろう。多くのシステムではモデル・チェックに使いやすい抽象は、設計の重要な側面について、大量の出力を行う。その時、人間の推論、すなわち数学的証明が必要となる。うまくいけば、この推論を、問題をモデルチェックに適した部分問題に分解して、仕様を複数のプロセスの構成として書き換えることに制限することができる。多くの場合にはこうした問題の分解は現実的ではない。この時には、数学的推論が唯一の選択肢となる。

- 数学では、どんな証明も数学的である。望む結果をより単純なサブゴールに階層的に分解していく。証明を書く賢明なやり方は、証明の長さと厳密さのトレードオフを認めながら、この階層的分解を明確なものにしていくだろう。数学は、別々の分離したコンポーネントの並行した構成として仕様を書くよりも、もっと一般的でもっと強力な証明の分解の方法を提供する。仕様でのような分解の形式に人気がある唯一の理由は、それがコンピュータ・サイエンティストが好む擬似プログラム言語の言葉で表現できるからに他ならない。
- 数学は、2000年以上にわたって、人間の厳密な推論への最良のアプローチであった。擬似プログラム言語による設計の20年が、数学の優位性への脅威になることはない数学的に推論する最良のやり方は、数学を使うことであって擬似プログラム言語を使うことではない。

“Coming Soon: Machine-Checked Mathematical Proofs in Everyday Software and Hardware Developme”



Adam Chlipala http://bit.ly/2GppRxe

第三部

21世紀の形式的手法

Agenda
第三部:　21世紀の形式的手法

- Deep Specificationと証明支援システムCoq
  -- 形式手法のLingua Franca
- 形式的推論・証明サンプル
  -- Hoare Logicの三つ組の証明
- Deep Specificationにおける仕様と実装例
  -- Kami プロジェクトを例に
- Software Foundation

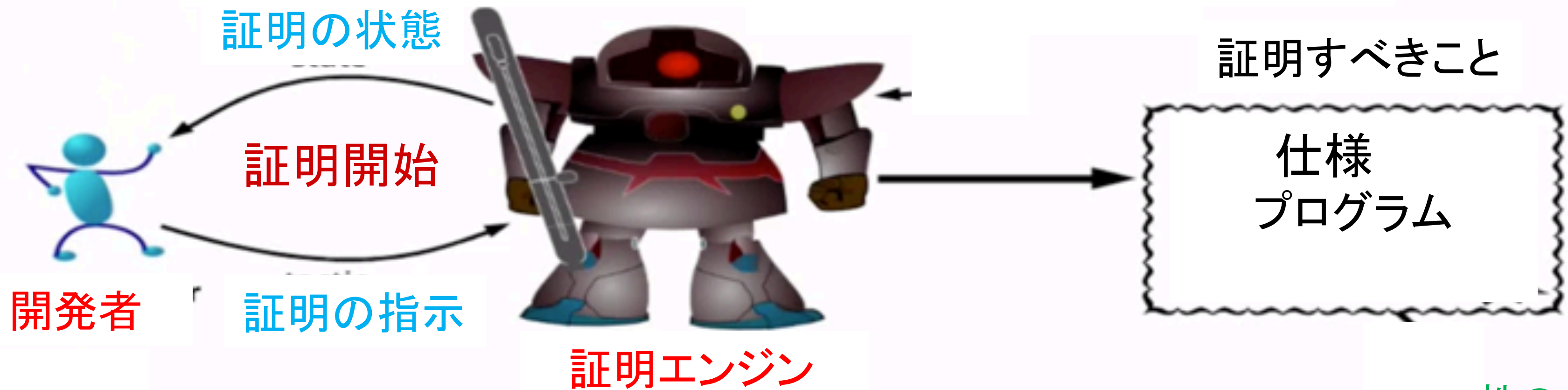
Deep Specificationと
証明支援システム Coq
-- 形式的手法の Lingua Franca

- 1980年代の形式手法も現代の形式手法も、プログラムの正しさを形式的＝数学的に証明しようという点では、共通の立場に立つ。当然である。ただ、両者の大きな違いの一つは、「形式的証明」の進め方にある。
- 80年代には、コンピュータによる証明支援システムは、まだ成熟していなかった。形式手法においても、コンピュータが行なっていたのはモデル・チェッキングのみで、「証明」は基本的には人間が行なっていた。

- しかし、今日の形式手法では、「証明」は、人間とコンピュータが協力して行い、「証明」が正しいか否かのチェックはコンピュータが行う。
- だから「証明支援システム」という。人間の証明をコンピュータが「支援」するのだ。別の見方をすれば、コンピュータの証明を人間が支援しているとも言える。これは、とても面白い問題だ。
- 「証明支援システム」の登場は、形式手法の「実効性」「実現可能性」を大きく変えたのだ。

近未来の開発者

「テスト･デバッグ･コードレビュー」に代わるもの



近未来の開発者

「テスト・デバッグ・コードレビュー」に代わるもの

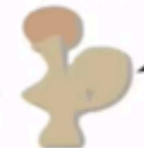
証明の新しいノウハウ

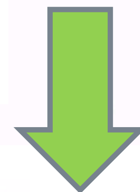
証明成功！

開発者

証明エンジン

証明
チェッカー

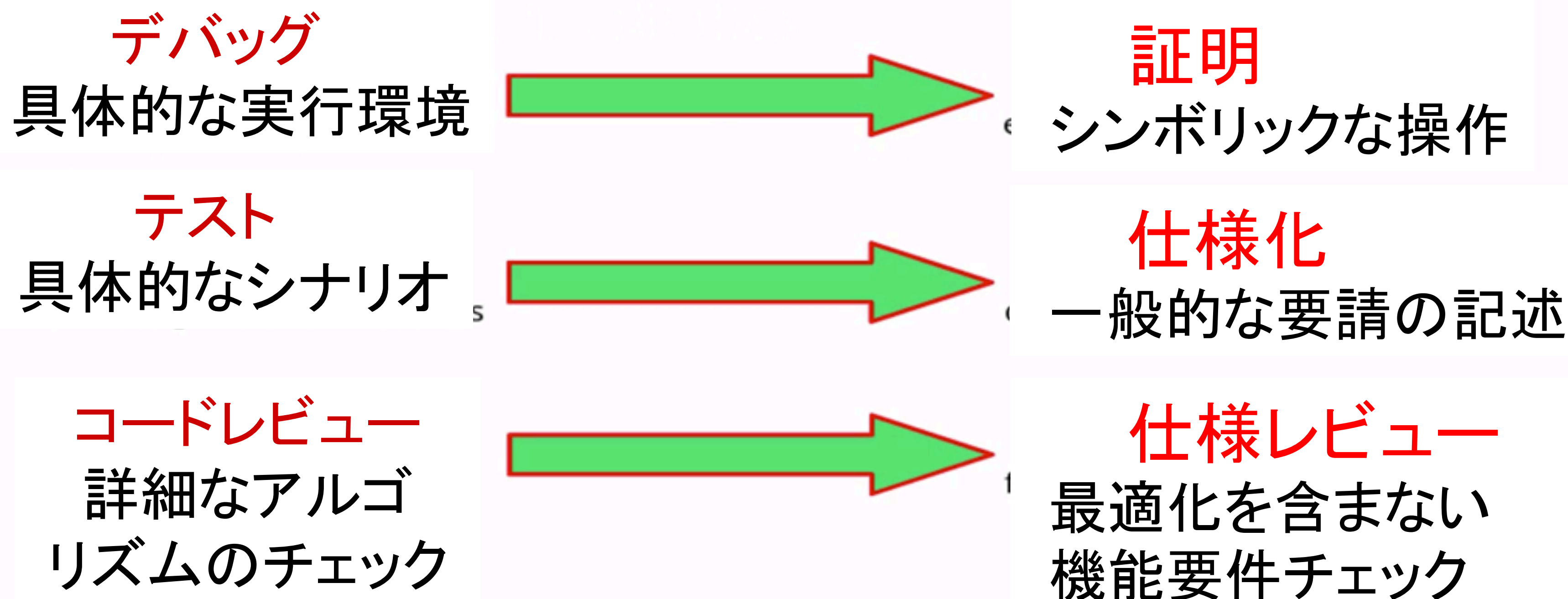




信頼性の
獲得

検証された
仕様
プログラム

「デバッグ・テスト・コードレビュー」の本質的な置き換えが起きる。



Deep Specificationとは何か？

Deep Specificationとは何か？

我々は、他の多くのプロジェクトでも使われている小規模な検証済みコンポーネントから、大規模な検証済みシステムが常に構築されている世界を期待している。

そのために、全産業規模でのソフトウェアとハードウェア・コンポーネントの正式仕様の重要な基礎を明らかにしようとするプロジェクト「Deep Specificationの科学」での我々の取り組みを紹介する。

我々は、異なるチームの作業をまたいで、コンポーネントの正しさを示す強い定理を結びつける、重要な仕様のクラスを同定してきた。 それらは、すでにいくつかの実験で使われている。

こうしたスタイルのユニークな利点を一般に広めるために、我々はそれを「深い仕様 Deep Specification」 と呼んでいる。

https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.2016.0331

The Science of Deep Specification

このスタイルのユニークな利点を皆に知ってもらうのを助けるために、我々はそれに、deep specificationという名を授けた。

我々は、それは、豊かで(rich)、二面を持った(two-sided)、形式的で(formal)、コンピュータで証明可能な(live)、諸仕様を包含するものだと考えている。

我々の中核チームは、(Coq証明支援システムをベースにして)このコンセプトの実証システムの実験を展開している。その仕様と検証作業は、我々の四つの組織を横断して、大きく分離されたサブ・チームに分割されている。それは、分野横断の原則と効率的な仕様開発のツールにそって、ハードウェア・マイクロアーキテクチャー、コンパイラー、オペレーティング・システム、アプリケーションを包含している。我々は、基礎分野の研究者だけでなく、産業界のユーザーにも、このアプローチへの関心を、拡大しようと考えている。

Deep Specificationが満たすべき条件

- **rich** (複雑なコンポーネントの振る舞いを詳細に記述する)
- **two-sided** (実装とクライアントの双方に接続する)
- **formal** (明確なsemanticsを持った数学的な表記で記述されている。型チェッカーや分析・テストツール、自動化されたあるいは機械による証明システム、高度なIDEをサポートしている)
- **live** (コンピュータでチェック可能な証明を通じて、実装とクライアントのコードと結びついている)

人工知能技術の本流

- コンピュータには、論理的＝数学的推論を、正確に実行する能力があるのだ。我々は、コンピュータの、そうした力をコントロールする方法を手に入れつつある。それは巨大な進歩である。
- 残念なことに、というか皮肉なことに、「人工知能ブーム」の中では、こうした達成は、十分な注目を集めていないように見える。僕は、こうした技術が「人工知能」の「本流」と考えているのだが。
- 立ち上がり始めた量子コンピュータ技術が、最初の応用を量子コンピュータ自身の「量子の世界」のシミレーションに見つけたように、立ち上がり始めた新しい人工知能技術も、コンピュータ自身のプログラムに、応用分野を見つけたと思えばいい。

形式的推論･証明サンプル

Hoare Logicの三つ組の証明

”Formal Reasoning about Programs”

Adam Chlipala
2018年

http://adam.chlipala.net/frap/frap_book.pdf

”Formal Reasoning about Programs”

Chapter 1. Why Prove the Correctness of Programs?

Chapter 2. Formalizing Program Syntax

Chapter 3. Data Abstraction

Chapter 4. Semantics via Interpreters

Chapter 5. Transition Systems and Invariants

Chapter 6. Model Checking

Chapter 7. Operational Semantics

Chapter 8. Abstract Interpretation and Dataflow Analysis

Chapter 9. Compiler Correctness via Simulation Arguments

Chapter 10. Lambda Calculus and Simple Type Safety

Chapter 10. Lambda Calculus and Simple Type Safety

Chapter 11. Types and Mutation

Chapter 12. Hoare Logic: Verifying Imperative Programs

Chapter 13. Deep Embeddings, Shallow Embeddings, and Options in Between

Chapter 14. Separation Logic

Chapter 15. Connecting to Real-World Programming Languages

Chapter 16. Deriving Programs from Specifications

Chapter 17. Introduction to Reasoning About Shared-Memory Concurrency

Chapter 18. Concurrent Separation Logic

Chapter 19. Process Algebra and Refinement

Chapter 20. Session Types

Appendix A. The Coq Proof Assistant

Index

CHAPTER 12

Hoare Logic: Verifying Imperative Programs

We now take a step away from the last chapters in two dimensions: we switch back from functional to imperative programs, and we return to proofs of deep correctness properties, rather than mere absence of type-related crashes. Nonetheless, the essential proof structure winds up being the same, as we once again prove invariants of transition systems!

12.1. An Imperative Language with Memory

To provide us with an interesting enough playground for program verification, let's begin by defining an imperative language with an infinite mutable heap. For reasons that will become clear shortly, we do a strange bit of mixing of syntax and semantics. In certain parts of the syntax, we include *assertions* a, which are arbitrary mathematical predicates over program state, split between heaps h and variable valuations v.

12.2. Hoare Triples

Much as we did with type systems, we define a syntactic predicate and prove it sound once and for all. Afterward, we can automatically show that particular programs and their specifications inhabit the predicate. This time, predicate instances will be written like $\{P\}c\{Q\}$, with c the command being verified, P its *precondition* (assumption about the program state before we start running c), and Q its *postcondition* (obligation about the program state after c finishes). We call any such fact a *Hoare triple*, and the overall predicate is an instance of *Hoare logic*.

A first rule for skip is easy: anything that was true before is also true after.

$$\frac{}{\{P\}\mathsf{skip}\{P\}}$$

A rule for assignment is slightly more involved: to state what we know is true after, we recall that there existed a prestate satisfying the precondition, which then evolved into the poststate in the expected way.

$$\frac{}{\{P\}x \leftarrow e\{\lambda(h, v).\ \exists v'.\ P(h, v') \land v = v'[x \mapsto [\![e]\!](h, v')]\}}$$

The memory-write command is treated symmetrically.

$$\frac{}{\{P\}*[e_1] \leftarrow e_2\{\lambda(h, v).\ \exists h'.\ P(h', v) \land h = h'[[\![e_1]\!](h', v) \mapsto [\![e_2]\!](h', v)]\}}$$

To model sequencing, we thread predicates through in an intuitive way.

$$\frac{\{P\}c_1\{Q\} \quad \{Q\}c_2\{R\}}{\{P\}c_1; c_2\{R\}}$$

the end of the command after running either subcommand, we take the disjunction of their postconditions.

$$\frac{\{\lambda s.\ P(s) \wedge [\![b]\!](s)\}c_1\{Q_1\} \quad \{\lambda s.\ P(s) \wedge \neg [\![b]\!](s)\}c_2\{Q_2\}}{\{P\}\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\{\lambda s.\ Q_1(s) \vee Q_2(s)\}}$$

Coming to loops, we at last have a purpose for the assertion annotated on each one. We call those assertions *loop invariants*; one of these is meant to be true every time a loop iteration begins. We will try to avoid confusion with the more fundamental concept of invariant for transition systems, though in fact the two are closely related formally, which we will see in the last section of this chapter. Essentially, the loop invariant gives the *induction hypothesis* that makes the program correctness proof go through. We encapsulate the induction reasoning once and for all, in the proof of soundness for Hoare triples. To verify an individual program, it is only necessary to prove the premises of the rule, which we give now.

$$\frac{(\forall s.\ P(s) \Rightarrow I(s)) \quad \{\lambda s.\ I(s) \wedge [\![b]\!](s)\}c\{I\}}{\{P\}\{I\}\mathsf{while}\ b\ \mathsf{do}\ c\{\lambda s.\ I(s) \wedge \neg [\![b]\!](s)\}}$$

```
299  (** * An alternative correctness theorem for Hoare logic, with small-step semantics *)
300
301  Inductive step : heap * valuation * cmd -> heap * valuation * cmd -> Prop :=
302  | StAssign : forall h v x e,
303    step (h, v, Assign x e) (h, v $+ (x, eval e h v), Skip)
304  | StWrite : forall h v e1 e2,
305    step (h, v, Write e1 e2) (h $+ (eval e1 h v, eval e2 h v), v, Skip)
306  | StStepSkip : forall h v c,
307    step (h, v, Seq Skip c) (h, v, c)
308  | StStepRec : forall h1 v1 c1 h2 v2 c1' c2,
309    step (h1, v1, c1) (h2, v2, c1')
310    -> step (h1, v1, Seq c1 c2) (h2, v2, Seq c1' c2)
311  | StIfTrue : forall h v b c1 c2,
312    beval b h v = true
313    -> step (h, v, If_ b c1 c2) (h, v, c1)
314  | StIfFalse : forall h v b c1 c2,
315    beval b h v = false
316    -> step (h, v, If_ b c1 c2) (h, v, c2)
317  | StWhileFalse : forall I h v b c,
318    beval b h v = false
319    -> step (h, v, While_ I b c) (h, v, Skip)
320  | StWhileTrue : forall I h v b c,
321    beval b h v = true
322    -> step (h, v, While_ I b c) (h, v, Seq c (While_ I b c))
323  | StAssert : forall h v (a : assertion),
324    a h v
325    -> step (h, v, Assert a) (h, v, Skip).
```

https://github.com/achlipala/frap/blob/master/HoareLogic_template.v

```coq
334 Definition unstuck (st : heap * valuation * cmd) :=
335   snd st = Skip
336   \/ exists st', step st st'.
337
338 Lemma hoare_triple_unstuck : forall P c Q,
339   {{P}} c {{Q}}
340   -> forall h v, P h v
341                -> unstuck (h, v, c).
342 Proof.
343   induct 1; unfold unstuck; simplify; propositional; eauto.
344
345   apply IHhoare_triple1 in H1.
346   unfold unstuck in H1; simplify; first_order; subst; eauto.
347   cases x.
348   cases p.
349   eauto.
350
351   cases (beval b h v); eauto.
352
353   cases (beval b h v); eauto.
354
355   apply H0 in H2.
356   apply IHhoare_triple in H2.
357   unfold unstuck in H2; simplify; first_order.
358 Qed.
```

https://github.com/achlipala/frap/blob/master/HoareLogic_template.v

```
367  Lemma hoare_triple_step : forall P c Q,
368    {{P}} c {{Q}}
369    -> forall h v h' v' c',
370        step (h, v, c) (h', v', c')
371        -> P h v
372        -> {{h''&v'' ~> h'' = h' /\ v'' = v'}} c' {{Q}}.
```

```
417  Theorem hoare_triple_invariant : forall P c Q h v,
418    {{P}} c {{Q}}
419    -> P h v
420    -> invariantFor (trsys_of (h, v, c)) unstuck.
```

https://github.com/achlipala/frap/blob/master/HoareLogic_template.v

```
450  (* A very simple example, just to show all this in action *)
451  Definition forever := (
452    "i" <- 1;;
453    "n" <- 1;;
454    {{h&v ~> v $! "i" > 0}}
455    while 0 < "i" loop
456      "i" <- "i" * 2;;
457      "n" <- "n" + "i";;
458      assert {{h&v ~> v $! "n" >= 1}}
459    done;;
460
461    assert {{_&_ ~> False}}
462    (* Note that this last assertion implies that the program never terminates! *)
463  )%cmd.
464
465  Theorem forever_ok : {{_&_ ~> True}} forever {{_&_ ~> False}}.
466  Proof.
467    ht.
468  Qed.
469
470  Theorem forever_invariant : invariantFor (trsys_of ($0, $0, forever)) unstuck.
471  Proof.
472    eapply hoare_triple_invariant.
473    apply forever_ok.
474    simplify; trivial.
475  Qed.
```

https://github.com/achlipala/frap/blob/master/HoareLogic_template.v

In [81]:

```
Lemma hoare_triple_unstuck : forall P c Q,
  {{P}} c {{Q}}
  -> forall h v, P h v
             -> unstuck (h, v, c).
```

Out[82]:

```
Proving: hoare_triple_unstuck

1 subgoal

1/1 -----------
forall (P : assertion) (c : cmd) (Q : assertion),
{{P}} c {{Q}} ->
forall (h : heap) (v : valuation), P h v -> unstuck (h, v, c)
```

In [82]:
```
Proof.
  induct 1; unfold unstuck; simplify; propositional; eauto.
```

Out[84]:
```
Proving: hoare_triple_unstuck

4 subgoals

P, Q, R : assertion
c1, c2 : cmd
H : {{P}} c1 {{Q}}
H0 : {{Q}} c2 {{R}}
IHhoare_triple1 : forall (h : heap) (v : valuation),
                  P h v -> unstuck (h, v, c1)
IHhoare_triple2 : forall (h : heap) (v : valuation),
                  Q h v -> unstuck (h, v, c2)
h : heap
v : valuation
H1 : P h v
```

```
1/4 ----------
((c1;; c2)%cmd = Skip)%reset \/
(exists st' : heap * valuation * cmd, step (h, v, (c1;; c2)%cmd) st')
2/4 ----------
((when b then c1 else c2 done) = Skip)%reset \/
(exists st' : heap * valuation * cmd,
  step (h, v, when b then c1 else c2 done) st')
3/4 ----------
(({{I}}while b loop c done) = Skip)%reset \/
(exists st' : heap * valuation * cmd,
  step (h, v, {{I}}while b loop c done) st')
4/4 ----------
(c = Skip)%reset \/ (exists st' : heap * valuation * cmd, step (h, v, c) st')
```

```
In [84]: apply IHhoare_triple1 in H1.

Out[85]: Proving: hoare_triple_unstuck

4 subgoals

P, Q, R : assertion
c1, c2 : cmd
H : {{P}} c1 {{Q}}
H0 : {{Q}} c2 {{R}}
IHhoare_triple1 : forall (h : heap) (v : valuation),
                  P h v -> unstuck (h, v, c1)
IHhoare_triple2 : forall (h : heap) (v : valuation),
                  Q h v -> unstuck (h, v, c2)
h : heap
v : valuation
H1 : unstuck (h, v, c1)
```

中略

In [93]:
```
apply IHhoare_triple in H2.
```

Out[94]:
```
Proving: hoare_triple_unstuck

1 subgoal

P, Q, P', Q' : assertion
c : cmd
H : {{P}} c {{Q}}
H0 : forall (h : heap) (v : valuation), P' h v -> P h v
H1 : forall (h : heap) (v : valuation), Q h v -> Q' h v
IHhoare_triple : forall (h : heap) (v : valuation),
                 P h v -> unstuck (h, v, c)
h : heap
v : valuation
H2 : unstuck (h, v, c)

1/1 -----------
(c = Skip)%reset \/ (exists st' : heap * valuation * cmd, step (h, v, c) st')
```

In [94]:
```
unfold unstuck in H2; simplify; first_order.
```

Out[95]:
```
Proving: hoare_triple_unstuck

No more subgoals
```

In [95]:
```
Qed.
```

A very simple example, just to show all this in action

In [155]:
```
Definition forever := (
  "i" <- 1;;
  "n" <- 1;;
  {{h&v ~> v $! "i" > 0}}
  while 0 < "i" loop
   "i" <- "i" * 2;;
   "n" <- "n" + "i";;
   assert {{h&v ~> v $! "n" >= 1}}
  done;;

  assert {{_&_ ~> False}}
  (* Note that this last assertion implies that the program never terminates! *)
)%cmd.

Theorem forever_ok : {{_&_ ~> True}} forever {{_&_ ~> False}}.
Proof.
  ht.
Qed.
```

Out[160]:

In [160]:
```
Theorem forever_invariant : invariantFor (trsys_of ($0, $0, forever)) unstuck.
Proof.
  eapply hoare_triple_invariant.
  apply forever_ok.
  simplify; trivial.
Qed.
```

CHAPTER 14

Separation Logic

In our Hoare-logic examples so far, we have intentionally tread lightly when it comes to the potential aliasing of pointer variables in a program. Generally, we have only worked with, for instance, a single array at a time. Reasoning about multi-array programs usually depends on the fact that the arrays don't overlap in memory at all. Things are even more complicated with linked data structures, like linked lists and trees, which we haven't even attempted up to now. However, by using *separation logic*, a popular variant of Hoare logic, we will find it quite pleasant to prove programs that used linked structures, with no need for explicit reasoning about aliasing, assuming that we keep all of our data structures disjoint from each other through simple coding patterns.

14.1. An Object Language with Dynamic Memory Allocation

Before we get into proofs, let's fix a mixed-embedding object language.

$$\begin{array}{lrcl} \text{Commands} & c & ::= & \mathsf{Return}\ v \mid x \leftarrow c; c \mid \mathsf{Loop}\ i\ f \mid \mathsf{Fail} \\ & & & \mid \mathsf{Read}\ n \mid \mathsf{Write}\ n\ n \mid \mathsf{Alloc}\ n \mid \mathsf{Free}\ n\ n \end{array}$$

We can also define natural comparison operators between assertions, overloading the usual notations for equivalence and implication of propositions.

$$\begin{array}{rcl} P \Leftrightarrow Q & = & \forall h.\ h \in P \Leftrightarrow h \in Q \\ P \Rightarrow Q & = & \forall h.\ h \in P \Rightarrow h \in Q \end{array}$$

The core connectives satisfy a number of handy algebraic laws. Here is a sampling.

$$\frac{\phi \to (P \Rightarrow Q)}{P * [\phi] \Rightarrow Q} \quad \frac{\phi \quad P \Rightarrow Q}{P \Rightarrow Q * [\phi]} \quad \frac{\phi}{P \Leftrightarrow [\phi] * P}$$

$$\frac{}{P * Q \Leftrightarrow Q * P} \quad \frac{}{P * (Q * R) \Leftrightarrow (P * Q) * R} \quad \frac{P_1 \Rightarrow P_2 \quad Q_1 \Rightarrow Q_2}{P_1 * Q_1 \Rightarrow P_2 * Q_2}$$

$$\frac{}{(P * \exists x.\ Q(x)) \Leftrightarrow \exists x.\ P * Q(x)} \quad \frac{\forall x.\ P(x) \Rightarrow Q}{(\exists x.\ P(x)) \Rightarrow Q} \quad \frac{P \Rightarrow Q(v)}{P \Rightarrow \exists x.\ Q(x)}$$

Deep Specificationにおける仕様と実装の例

仕様と実装

- Deep Specificationを何か形式的な仕様から何か具体的なプログラムの実装を自動的に生成する技術だと考えていると、なかなかイメージが掴めなくなる。
- Deep Specificationの仕様は、動かない抽象的なものではなく、それ自身プログラムとして動作するものだ。Deep Specificationでは、仕様と実装の差は、そのプログラムが一般的か特殊的かの差であって、動くか動かないかに違いがあるわけでは無い。
- ある問題をコンピュータで解こうとする時、我々は様々なデータ構造やアルゴリズムを使ってプログラムを書く。Deep Specificationの「仕様」もこれと全く同じように書かれる。それは「実装」と言ってもいい。

仕様と実装

- ただ、「実装」する言語が違うのだ。その言語は、プログラム自身についての形式的推論を可能にする能力を持つもの、例えば、Coqを用いる。こうした能力は、プログラムの検証で大きな役割を果たす。それについては、あとで説明する。
- 様々なデータ構造やアルゴリズムを使って、ある問題を解くプログラムをCoqで「実装」してみる。それが、Deep Specificationの「仕様」の第一歩だと考えていい。
- ただ、プログラムで「実装」された「仕様」（それは、文字で書かれた仕様書より「形式的」なものと考えられる）ができれば、それで終わりなのでは無い。
- この「仕様＝実装」から、実行用の高速なコード（OcamlやCやVerlogのコード）が生成されることになる。こちらを、形式的「仕様」から導出された「実装」と考えることもできる。

” Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification”

JOONWON CHOI , MURALIDARAN VIJAYARAGHAVAN , BENJAMIN SHERMAN, ADAM CHLIPALA, and ARVIND

2017年

http://adam.chlipala.net/papers/KamiICFP17/KamiICFP17.pdf

https://github.com/sifive/Kami

Verify software programs with proof assistants

(1) **Implement the program** in the functional programming language built into the proof assistant.

(2) In a rich higher-order logic, **state the most natural correctness theorem** for the program.

(3) **Prove the theorem** using scripts of tactics for proof steps at different levels of granularity, saving the user from tedious details while still giving an opportunity to spell out the key insights manually.

(4) **Use extraction** to translate the program to a language like OCaml automatically, and **from here use standard development tools** to compile and run it.

What is Kami?

https://github.com/sifive/Kami/

What is Kami

Kami is an umbrella term used to denote the following: .
A Coq-based DSL for writing hardware designs .

A compiler for translating said hardware designs into Verilog .

A simulator for said hardware designs, by generating an executable in Haskell, using user-defined functions to drive inputs and examine outputs for the hardware design .

A formal definition of the semantics of the DSL in Coq, including a definition of whether one design *implements* another simpler design, i.e. whether an *implementation adheres to its specification* .

A set of theorems or properties about said semantics, formally proven in Coq . A set of tactics for formally proving that an implementation adhere to its specification

What is Kami

In Kami, one can write generators, i.e. functions that generate hardware when its parameters are specified, and can prove that the generators are correct with respect to their specification.

Unlike traditional model-checking based approaches, the ability to prove theorems involving higher-order logic in Coq enables one to easily prove equivalence between a generator and its specification.

The semantics of Kami was inspired by Bluespec SystemVerilog. The original version of Kami was developed in MIT. Based on the experience of developing and using Kami at MIT, it was rewritten at SiFive to make it practical to build provably correct chips.

Semantics of Kami: an informal overview
Module

Any hardware block or *module* is written as a set of registers representing the state of the block, and a set of *rules*.

The behavior of the module is represented by a sequence of execution of rules. Rules execute by reading and writing the state *atomically*, i.e. when one rule is executing, no other rule executes. During its execution, a rule can also interact with the external world by calling methods, to which the rule supplies arguments (an output from the module), and takes back the result returned by the external world (an input to the module). Once a rule finishes execution, another rule is picked non-deterministically and is executed, and so on.

Semantics of Kami: an informal overview **implementationとspecification**

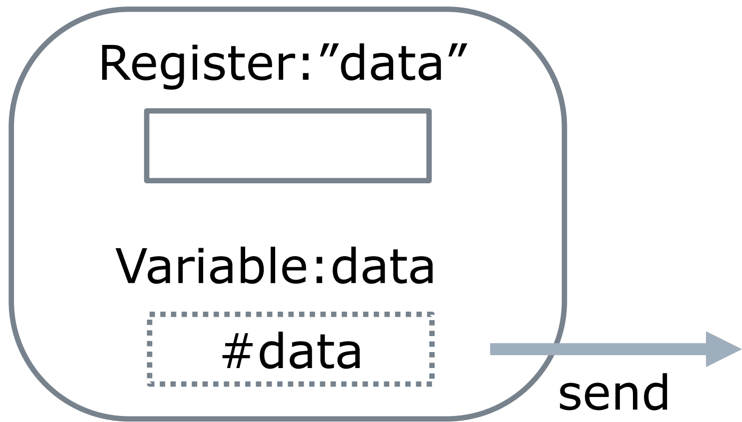
A module *A* is said to implement a specification module *B* if, during every rule execution in *A*, if the rule calls any methods, then these methods (along with their arguments and return values) are the same as those called by some rule execution in *B*, and this property holds for every sequence of rule executions in *A*. Note that the return values are functions of the external world; we assume that the same value can be returned by the external world if the same method is called with the same argument in both *A* and *B*. The methods along with their arguments and return values that are called in a rule's execution are called a label, and the sequence of labels corresponding to the sequence of rule execution is called a trace. The above definition of *A* implementing *B* can be rephrased as follows: any trace that can be produced by *A* can also be produced by *B*. We call this property Trace Inclusion.

Kami：実行サンプル

https://github.com/sifive/Kami/tutorial.v

Producer モジュールの実装

Module: producer

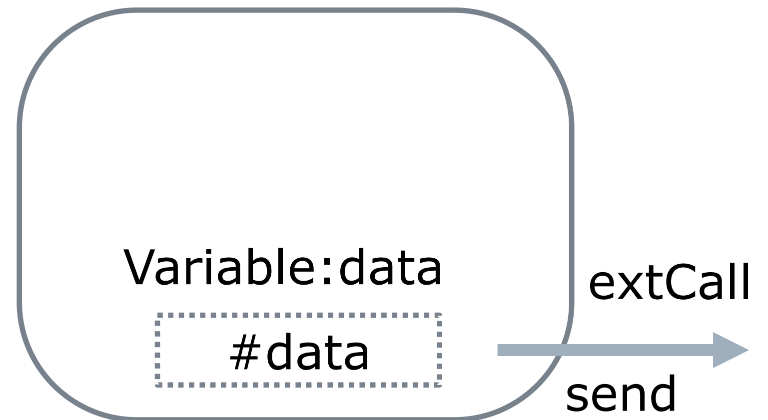


send後
Write "data" <- #data + $1

Producer モジュールの実装

```
Definition producer :=
  MODULE {
    Register "data" : Bit 32 (* type *) <- Default (* initial value *)

    with Rule "produce" :=
      Read data <- "data";
        (* Explicit actions are required to read values of registers into local variables. *)
      Call (MethodSig "send" (Bit 32 (* parameter type *)): Void (* return type*)) (#data) (* using [#] to read from variables *);
        (* Note embedding of a type assumption for the function being called, not just its name. *)
      Write "data" <- #data + $1 (* using [$] for a literal expression *);
      Retv
  }.
```

consumer モジュールの実装

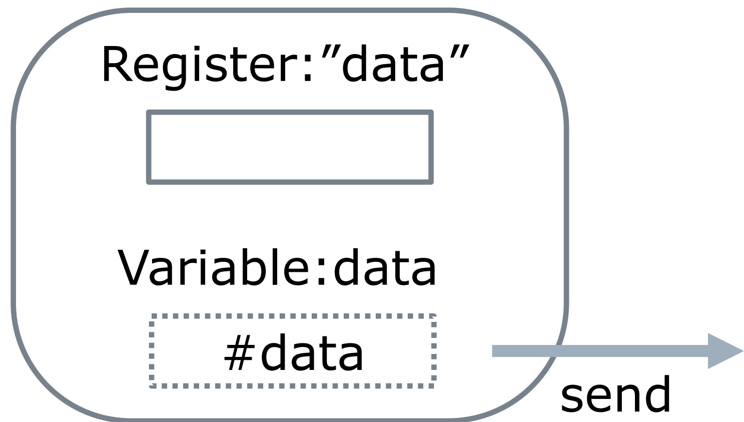


consumer モジュールの実装

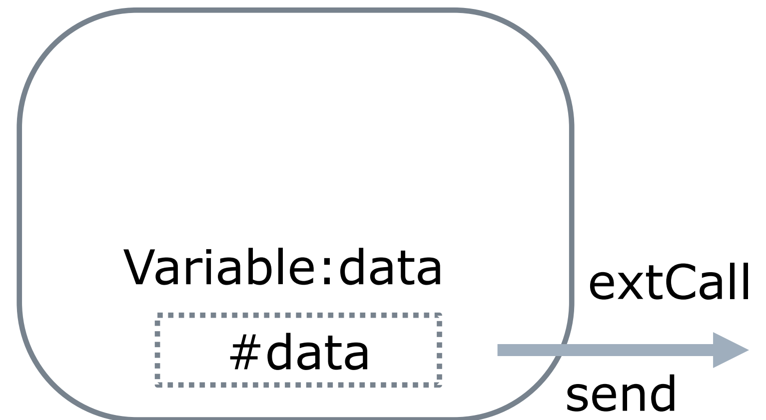
```
(** For proof automation, it is recommended to register module
definitions to "ModuleDefs". *)
Hint Unfold producer : ModuleDefs.

(** Consumer only has one method, which takes the data sent by
Producer and calls an external function with the data. *)
Definition consumer :=
  MODULE {
    Method "send" (data: Bit 32): Void :=
      Call (MethodSig "extCall" (Bit 32): Void) (#data);
      Retv
  }.
```

producerモジュールとconsumer モジュール

Module: producer



Module: consumer

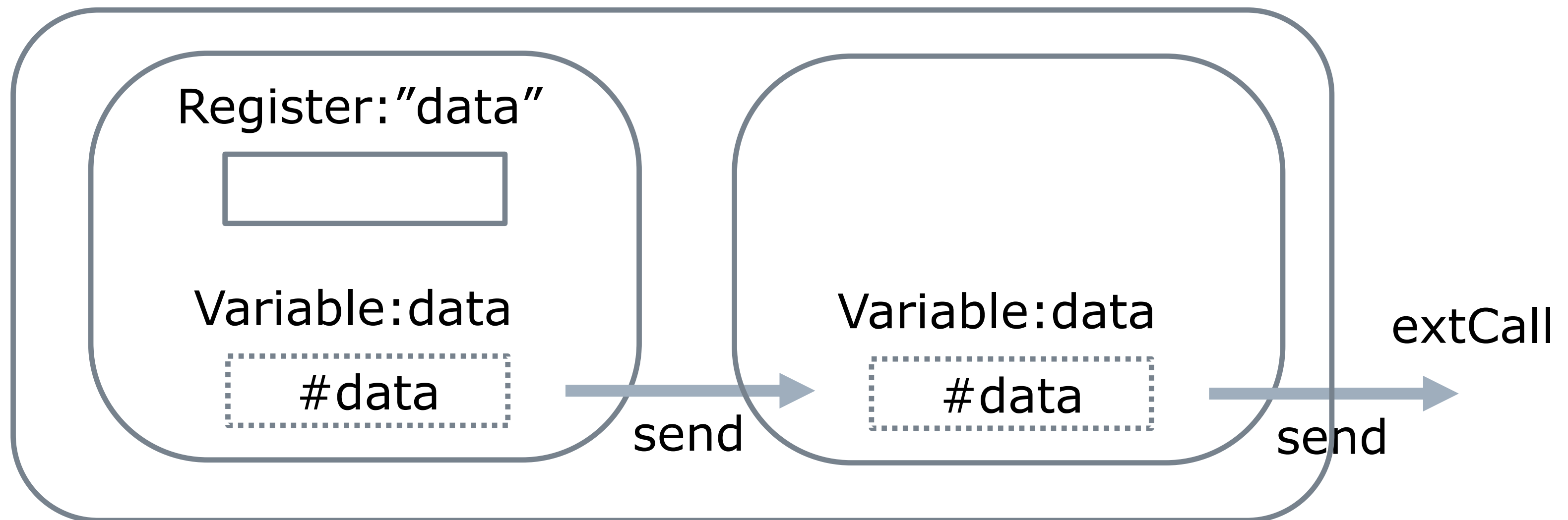


send後
Write "data" <- #data + $1

producerモジュールとconsumer モジュールの実装を組み合わせてproducerConsumerImplを作る

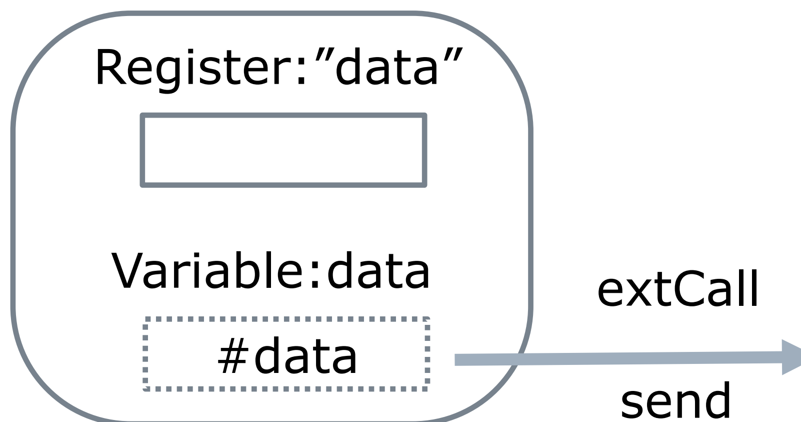
Definition producerConsumerImpl := (producer ++ consumer)%kami.

Module: producerconsumerImpl



send後
Write "data" <- #data + $1

producerConsumerImplの仕様 producerConsumerSpecを書く



send後
Write "data" <- #data + $1

producerConsumerImplの仕様producerConsumerSpecを書く

```
Definition producerConsumerSpec :=
  MODULE {
    Register "data" : Bit 32 <- Default

    with Rule "produce_consume" :=
      Read data <- "data";
      Call (MethodSig "extCall" (Bit 32): Void) (#data);
      Write "data" <- #data + $1;
      Retv
  }.
```

実装producerConsumerImplは、
仕様producerConsumerSpecを
実装したものかをチェックする

次の定理を証明する。

```
Theorem producer_consumer_refinement:
  producerConsumerImpl <<== producerConsumerSpec.

Proof.
   (次のページ)
Qed.
```

きちんと証明できる。

Theorem producer_consumer_refinement:

```
Theorem producer_consumer_refinement:
  producerConsumerImpl <<== producerConsumerSpec.
Proof.
  kinline_left implInlined.
  (* Inlining: replace internal function calls in [impl]. *)
  kdecompose_nodefs producer_consumer_regMap
producer_consumer_ruleMap.
  (* Decomposition: consider all steps [impl] could take, requiring
that each be matched appropriately in [spec]. *)
  kinvert.
  (* Inversion on the took-a-step hypothesis, to produce one new
subgoal per [impl] rule, etc. *)
  kinv_magic_light.
  (* We have only one case for this example (for the one rule), and
it's easy. *)
Qed.
```

Software Foundations

https://deepspec.org/page/SF/

Volume 1

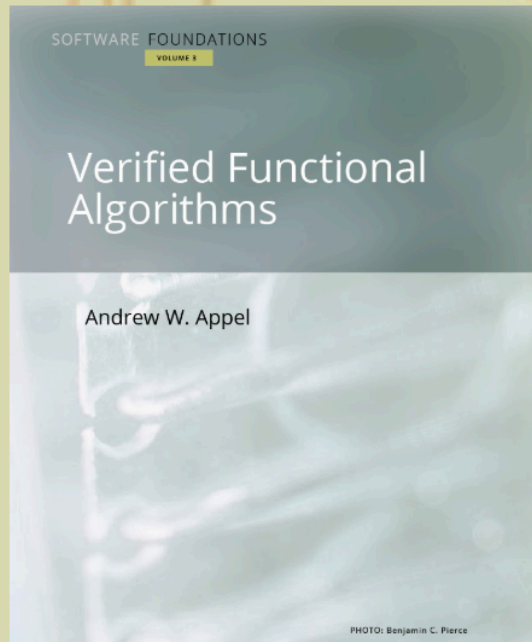
Logical Foundations **is the entry-point to the series. It covers functional programming, basic concepts of logic, computer-assisted theorem proving, and Coq.**

Volume 2

Programming Language Foundations **surveys the theory of programming languages, including operational semantics, Hoare logic, and static type systems.**

Volume 3

Verified Functional Algorithms shows how a variety of fundamental data structures can be specified and mechanically verified.



Volume 4

QuickChick: Property-Based Testing in Coq introduces tools and techniques for combining randomized property-based testing with formal specification and proof in the Coq ecosystem.

Software Foundations

- **Volume 1: Logical Foundations**
  https://softwarefoundations.cis.upenn.edu/lf-current/toc.html
  PDF版
  https://softwarefoundations.cis.upenn.edu/lf-current/lf.tgz
- **Volume 2: Programming Language Foundations**
  https://softwarefoundations.cis.upenn.edu/plf-current/toc.html
  PDF版
  https://softwarefoundations.cis.upenn.edu/plf-current/plf.tgz

Software Foundations

- **Volume 3: Verified Functional Algorithms**
  https://softwarefoundations.cis.upenn.edu/vfa-current/toc.html
  PDF版
  https://softwarefoundations.cis.upenn.edu/vfa-current/vfa.tgz
- **Volume 4:QuickChick: Property-Based Testing in Coq**
  https://softwarefoundations.cis.upenn.edu/qc-current/toc.html
  PDF版
  https://softwarefoundations.cis.upenn.edu/qc-current/qc.tgz

おわりに
-- ダイクストラのメッセージを考える

Under the spell of Leibniz‵s Dream

Edsger W. Dijkstra

2001年

https://www.cs.utexas.edu/users/EWD/transcriptions/EWD12xx/EWD1298.html

ダイクストラの最後のメッセージ

- ダイクストラは、21世紀の初めまで生き、2002年になくなる。最晩年の2001年に彼があらわした「ライプニッツの夢の魔力の下で」”Under the spell of Leibniz�.s Dream” は、自身の生涯を振り返りつつ、コンピュータ・サイエンスについての彼の立場を語った、いわば彼の「遺言」ともいえる文書である。https://www.cs.utexas.edu/…/transcript…/EWD12xx/EWD1298.html

最後まで形式的手法の側に立つ

- Hoare や Lamport らが、形式的手法の陣営から脱落し、Hoareのように公然と形式的手法に反対する中、彼は最後まで形式的手法の立場に立っていた。
- ここでは、まず、この文書で「形式的手法」について、彼が述べていることを紹介したい。いずれも大事な指摘である。
- ほとんど20年前の文書だが、これらの指摘は、現代の「形式的手法」が進もうとしている道を、的確に指し示している。

「信頼は証明の上にのみ基礎付けられる」

1. 全ての場合を尽くすテストが不可能な時には、すなわち、ほとんどの場合はそうなのだが、我々の信頼は、(それが機械的なものであろうとなかろうと)証明の上にのみ基礎付けられる。
2. 我々が、それを信ずべき明確な方法のないプログラムは、疑わしい価値しか持たない。
3. プログラムは、その正しさについての議論が分かりやすく、かつ、不必要に複雑にならないように構造化されるべきである。
4. 証明が与えられている時、その証明によって正当化されるプログラムを導き出す方が、プログラムが与えられて、それを正当化する証明を構成するより、はるかに簡単である。

1. When exhaustive testing is impossible —i.e., almost always— our trust can only be based on proof (be it mechanized or not).
2. A program for which it is not clear why we should trust it, is of dubious value.
3. A program should be structured in such a way that the argument for its correctness is feasible and not unnecessarily laborious.
4. Given the proof, deriving a program justified by it, is much easier than, given the program, constructing a proof justifying it.

ダイクストラのメッセージ

ダイクストラの最後のメッセージ

- ダイクストラの論点は多岐にわたるが、いずれも刺激的である。
  - コンピュータ・サイエンスでの「理論」と「実践」の関係
  - コンピュータ・サイエンスとコンピュータ企業の関係
  - 大学の役割とその変化
  - プログラミングは「誰でもできる」のか？
  - ……
- この文書が「刺激的」なのは、我々多数の「通念」とは、ほとんど逆のことを彼が述べているからである。おそらく反発する人も多いだろう。

ダイクストラの最後のメッセージ

- こうした主張がコンピュータの世界に存在していることを知ることは、「未来の変化」を展望した時に、大きな意味があると僕は考えている。なぜなら、未来は現在の連続的延長上にあるとは限らず、変化は、現在の「常識」のいくつかを覆すだろうから。
- 文章は平易である。論旨も明確である。興味ある人は、是非、目を通していただければと思う。

「理論」と「実践」の関係

- if the University is not careful, external forces, which do make the distinction, will drive a wedge between "theory" and "practice" and may try to banish the "theorists" to a ghetto of separate departments and separate buildings.
- A simple extrapolation will tell us that in due time the isolated practitioners will have little to apply; this is well-known, but has never prevented the financial mind from killing the goose that lays the golden eggs. The worst thing with institutes explicitly devoted to applied science is that they tend to become institutes of second-rate theory.
- I grew up with an advice of my grandfather's: "Devote yourself to your most genuine interests, for then you will become a capable man and a place will be found for you.". We were free not to try to become rich (which for mathematicians was impossible anyhow).

コンピュータ・サイエンスとコンピュータ企業の関係

- About 10 years ago I tried to make up my mind on the question whether Computing Science could save the computer industry, and my conclusion was negative; since then I felt it my duty to try to prevent the computer industry from killing Computing Science, but I doubt that I have been successful....
- The ALGOL implementation was one of my proudest achievements. Its major significance was that it was done in 8 months at the investment of less than 3 man-years. The rumour was that IBM's FORTRAN implementation had taken 300 man-years and thus had been a project in size way beyond what a university could ever hope to undertake.

- The advent of more powerful and —what people sometimes forget— much more reliable computers invited many ambitious projects at all sorts of places. Some were successful, some were overambitious, all of them drove home the message that such systems could easily get so complicated as to become intellectually unmanageable and, hence, no longer trustworthy. These were serious concerns, which led to a number of observations and conclusions. I mention a few.

プログラミングは「誰でもできる」のか

- Considerations like the above gave rise to my "Notes on Structured Programming (from 1969) and C.A.R.Hoare's "Notes on Data Structuring" (from 1970). It is hard to quantify influence, but I think they have been seminal papers. People don't refer to them anymore, many haven't even read them, but the message has not been forgotten, it has been absorbed. [For your information, the book containing both texts is still in print; for 1999, our worldwide sales were 4 copies.]
- In short, programming was viewed as a perhaps painful but simple, low-level task. Needless to say, this confusion between the score and the composition led to an underestimation of the intellectual challenges programming presents.

- Secondly, there was the widely spread belief that, once we had the right, sufficiently powerful programming language, programming would be so easy that everybody could do it.
- The forces to play down the difficulty of programming were just too strong. I mentioned the sense of "popular justice" of the democratization movement that felt that everybody should be able to program. Then there was the religion at Xerox PARC that it should all be so natural that toddlers would love to do it. Then there was COBOL's creed that if your programming language was similar enough to English, even officers would be able to program.

- A major force was IBM, which wanted to sell hardware and hence was eager to present its machines as solutions rather than as the source of new problems. Finally, the prejudices of management did not allow industrial programming to be viewed as difficult: in the first half of the century, it had become a management goal to organize for the sake of stability the industrial enterprise in such a fashion that it would be as independent as possible of the competence of individual employees, with the result that in the second half of the century the mere suggestion that even industrial programming could require brains was blasphemy. For a while, the "deskilling of the programmer" was a hot topic.

- The prevailing attitude was reflected in the creation of two literary figures —admittedly of rather poor literature,but nevertheless of great paralyzing power—, viz. "the average programmer" and "the casual user". Up to these days, academic research in programming methodology has been supposed to respect the severe intellectual limitations of these fictitious morons; consequently any proposal that required any further education of the programming person was out. Academia has suffered, for programming methodology could not flourish as a research topic in a world in which it was considered irrelevant. For many a university, this has been a great loss; here and there the situation might be improving, but it is a slow process.

Leibniz

- I stayed with Burroughs Corporation for 11 wonderful and very productive years, my primary charter being to do my own thing.
- It was too good to last, and it did not. After the company had changed its CEO, it rapidly lost interest in science and technology, and the groups with which I had built up my relations disbanded the one after the other.
- As far as I know, Gottfried Wilhelm Leibniz, who lived from 1646 to 1716, has been the first to tackle effective reasoning as a technical problem. As a youngster of 20 years of age he conceived, possibly inspired by the work of Descartes, a vision of reasoning as applying a calculus.

- Like modern computing scientists, he invented impressive names for what had still to be invented, and, for good reasons not overly modest, he called his system no more and no less than "Characteristica Universalis". And again like modern computing scientists, he grossly underestimated the time the project would take: he confidently prophesied that a few well-chosen men could do the job in five years, but the whole undertaking was at the time of such a radical novelty that even the genius of Leibniz did not suffice for its realization, and it was only after another two centuries that George Boole began to realize something similar to the subsystem that Leibniz had called the "calculus raticinator".

- Let me quote from E.T.Bell young Leibniz's description of what he was aiming at:
  - "a general method in which all truths of the reason would be reduced to a kind of calculation. At the same time this would be a sort of universal language or script, but infinitely different from all those projected hitherto; for the symbols and even the words in it would direct reason; and errors, except those of fact, would be mere mistakes in calculation."
- I think it absolutely astounding that he foresaw how "the symbols would direct the reasoning", for how strongly they would do so was one of the most delightful discoveries of my professional life.

- Hilbert's revolution was in any case to redefine "proof" to become a completely rigorous notion, totally different from the psycho/sociological "A proof is something that convinces other mathematicians.". A major shortcoming of the latter view is that it gives no technical guidance for proof design and makes it very difficult to teach that kind of mathematics. Yet, or perhaps for this reason, many of the more conservative mathematicians still cling to this form of consensus mathematics; they will even vigorously defend their informality.

- During Hilbert's life, Leibniz's Dream, by and large, just stayed a dream. People viewed formal proofs as an interesting theoretical possibility or an unrealistic idealization, and they would regard their own proof as a usually sufficient sketch of a formal argument. They would even assure you that, if you insisted, they could formalize their informal argument, but how often that claim was valid is anybody's guess.

- Parts of Leibniz's Dream became reality, and it is quite understandable that this happened mostly in Departments of Computing Science, rather than in Departments of Mathemetics. Firstly, the computing scientists were in more urgent need of such calculational techniques because, by virtue of its mechanical interpretability, each programming language is eo ipso a formal system to start with. Secondly, for the manipulation of uninterpreted formulae, the world of computing provided a most sympathetic environment because we are so used to it: it is what compilers and theorem provers do all the time! And, finally, when the symbol manipulation would become too labour-intensive, computing science could provide the tools for mechanical assistance. In short, the world of computing became Leibniz's home; that it was my home as well was my luck.

Appendix

「ライプニッツの夢」

Symbolic thought

Leibniz believed that much of human reasoning could be reduced to calculations of a sort, and that such calculations could resolve many differences of opinion:

The only way to rectify our reasonings is to make them as tangible as those of the Mathematicians, so that we can find our error at a glance, and when there are disputes among persons, we can simply say: Let us calculate [calculemus], without further ado, to see who is right.[79]

https://en.wikipedia.org/wiki/Gottfried_Wilhelm_Leibniz

Leibniz's calculus ratiocinator, which resembles symbolic logic, can be viewed as a way of making such calculations feasible. Leibniz wrote memoranda[80] that can now be read as groping attempts to get symbolic logic—and thus his *calculus*—off the ground. These writings remained unpublished until the appearance of a selection edited by C.I. Gerhardt (1859). L. Couturat published a selection in 1901; by this time the main developments of modern logic had been created by Charles Sanders Peirce and by Gottlob Frege.

Leibniz thought symbols were important for human understanding. He attached so much importance to the development of good notations that he attributed all his discoveries in mathematics to this. His notation for calculus is an example of his skill in this regard. Peirce, a 19th-century pioneer of semiotics, shared Leibniz's passion for symbols and notation, and his belief that these are essential to a well-running logic and mathematics.

But Leibniz took his speculations much further. Defining a character as any written sign, he then defined a "real" character as one that represents an idea directly and not simply as the word embodying the idea. Some real characters, such as the notation of logic, serve only to facilitate reasoning. Many characters well known in his day, including Egyptian hieroglyphics, Chinese characters, and the symbols of astronomy and chemistry, he deemed not real.[81] Instead, he proposed the creation of a *characteristica universalis* or "universal characteristic", built on an alphabet of human thought in which each fundamental concept would be represented by a unique "real" character:

It is obvious that if we could find characters or signs suited for expressing all our thoughts as clearly and as exactly as arithmetic expresses numbers or geometry expresses lines, we could do in all matters insofar as they are subject to reasoning all that we can do in arithmetic and geometry. For all investigations which depend on reasoning would be carried out by transposing these characters and by a species of calculus.[82]

Complex thoughts would be represented by combining characters for simpler thoughts. Leibniz saw that the uniqueness of prime factorization suggests a central role for prime numbers in the universal characteristic, a striking anticipation of Gödel numbering. Granted, there is no intuitive or mnemonic way to number any set of elementary concepts using the prime numbers.

Because Leibniz was a mathematical novice when he first wrote about the *characteristic*, at first he did not conceive it as an algebra but rather as a universal language or script. Only in 1676 did he conceive of a kind of "algebra of thought", modeled on and including conventional algebra and its notation. The resulting *characteristic* included a logical calculus, some combinatorics, algebra, his *analysis situs* (geometry of situation), a universal concept language, and more.

What Leibniz actually intended by his *characteristica universalis* and calculus ratiocinator, and the extent to which modern formal logic does justice to calculus, may never be established.[83] Leibniz's idea of reasoning through a universal language of symbols and calculations remarkably foreshadows great 20th-century developments in formal systems, such as Turing completeness, where computation was used to define equivalent universal languages (see Turing degree).

Calculus ratiocinator

- The ***Calculus ratiocinator*** is a theoretical universal logical calculation framework, a concept described in the writings of Gottfried Leibniz, usually paired with his more frequently mentioned *characteristica universalis*, a universal conceptual language.

https://en.wikipedia.org/wiki/Calculus_ratiocinator

The analytic view

The received point of view in analytic philosophy and formal logic, is that the *calculus ratiocinator* anticipates mathematical logic—an "algebra of logic".[1] The analytic point of view understands that the *calculus ratiocinator* is a formal inference engine or computer program, which can be designed so as to grant primacy to calculations. That logic began with Frege's 1879 *Begriffsschrift* and C.S. Peirce's writings on logic in the 1880s. Frege intended his "concept script" to be a *calculus ratiocinator* as well as a *lingua characteristica*. That part of formal logic relevant to the calculus comes under the heading of proof theory. From this perspective the *calculus ratiocinator* is only a part (or a subset) of the *universal characteristic*, and a complete *universal characteristic* includes a "logical calculus".

The synthetic view

The synthetic view understands the *calculus ratiocinator* as referring to a "calculating machine". The cybernetician Norbert Wiener considered Leibniz's *calculus ratiocinator* a forerunner to the modern day digital computer:

> The history of the modern computing machine goes back to Leibniz and Pascal. Indeed, the general idea of a computing machine is nothing but a mechanization of Leibniz's calculus ratiocinator. (Wiener 1948: 214)

...like his predecessor Pascal, [Leibniz] was interested in the construction of computing machines in the Metal. ... just as the calculus of arithmetic lends itself to a mechanization progressing through the abacus and the desk computing machine to the utra-rapid computing machines of the present day, so the calculus ratiocinator of Leibniz contains the germs of the machina ratiocinatrix, the reasoning machine (Wiener 1965: 12)