Constructive Mathematics and Functional Programming

Thierry Coquand

Budapest, April 1, 2008

Constructive Mathematics

Connections between *reasoning* and *computations* in mathematics

1800 direct connection, algebra/analysis as describing algorithms (elimination, differentiation, integration); maybe not feasible but possible in theory

1820 Gauss, Abel, Galois, *irreducible* polynomials, rational functions (field)

1860 Dedekind, Kronecker, Hilbert, non constructive existence proof (Hilbert's basis theorem)

1905 Discussion on the Axiom of Choice, well-ordering of the reals

1908 Brouwer: the source of non constructivity is in the non restricted use of excluded-middle

Constructive Mathematics

1930 Heyting, Kolmogorov, explanation of intuitionistic logic

1958/1967 Curry, W. Howard, W. Tait, P. Martin-Löf propositions as types

1967 Bishop Foundations of constructive analysis

1979 P. Martin-Löf Constructive mathematics and computer programming

2001 B. Gregoire, X. Leroy A Compiled Implementation of Strong Reduction

2004 G. Gonthier, B. Werner Checking of the Four Color Theorem

Reasoning and computations

Excluded-Middle as the source of non effectivity (not the axiom of choice)

For functions: we have $\forall n. f(n) = 0$ or $\exists n. f(n) \neq 0$

But there is no algorithm, program of type $(\mathsf{nat} \rightarrow \mathsf{bool}) \rightarrow \mathsf{bool}$, which can decide if f if always 0 or not

$\exists n. \forall m. f(n) \leqslant f(m)$ (Hilbert's basis theorem), there is no program of type $(\mathsf{nat} \rightarrow \mathsf{nat}) \rightarrow \mathsf{nat}$ which can find this minimum value

If $f : [0,1] \rightarrow \mathbb{R}$ continuous function and $f(0) = -1,\ f(1) = 1$ then f has a zero. But there is no algorithm which can find this zero in general

Constructive Mathematics

Constructive mathematics best characterization (Richman, Bridges)

constructive mathematics is mathematics developed in intuitionistic logic

Notice that this characterization does not mention explicitly the notion of algorithms/recursive functions

Constructive Mathematics

Kolmogorov's suggestive explanation of intuitionistic logic: propositions as problems

To solve $A \wedge B$: solve A and solve B, $A \times B$

To solve $A \vee B$: solve A or solve B, $A + B$

To solve $A \rightarrow B$: reduce the problem B to the problem A, $A \rightarrow B$

To solve $\neg A$: show that the problem A has no solution, $A \rightarrow N_0$

Why $A \vee \neg A$ may not be valid: we should have a general method which given a problem A, either find a solution to A or show that A has no solution

Reunifying reasoning and computations

A (constructive) proof that a property P on a set A is decidable

$$\forall x : A.\ P(x) \vee \neg P(x)$$

can be seen as an algorithm deciding P

It is only possible to specify the algorithm in this way if we use intuitionistic logic

Reunifying reasoning and computations

Example: modal logic LTL, Büchi automata, we have properties of infinite paths that may not be, a priori, decidable

To represent the theory of Büchi automata in a constructive way is a(n interesting) challenge. Maybe one can develop the theory S1S in a purely syntactical way, following D. Siefkes book *Decidable Theories*, LNM 120

Example: domain theory. The *elements* of a domain are in general infinite objects, the ordering is not decidable in general. But the *finite elements* are expected to be *finitary objects* with a decidable ordering.

Reunifying reasoning and computations

The computations should be possible *in theory*

Possible to refine this with various notion of *feasible* computations

But a basic principle is that there is a fundamental difference between the situation where a computation is possible in theory and when where a computation is not possible at all

Reunifying reasoning and computations

Type theory (P. Martin-Löf, R. Constable, 1970s)

A formalism in which to formulate constructive mathematics

close to functional programming: *total* functional programming

programs (and proofs) are terminating functional programs

like a set theory where everything is computable

implementations: Coq (INRIA, B. Barras, H. Herbelin, J.C. Filliatre), Epigram (Nottingham, C. McBride), Agda (Chalmers, U. Norell)

Total functional programming

"The driving force of functional programming is to make programming more closely related to mathematics. A program in a functional language . . . consists of equations which are both computation rules and a basis for simple algebraic reasoning. The existing model of functional programming, although elegant and powerful, is compromised to a greater extent than is commonly recognized by the presence of partial functions. We consider a simple discipline of *total functional programming* designed to exclude the possibility of non termination."

D.A.Turner, 2004, J.U.C.S

Total functional programming

Functional programming allows elegant solutions for *some* class of problems

For most examples in this class, *total* functional programming is even nicer

There is a strong analogy with constructive logic: for the problems that do have a constructive solution it is in general not elegant to use excluded middle

Reunifying reasoning and computations

"relating constructive mathematics to computer programming seems to me to have a beneficial influence on both parties.

By choosing to program in a formal language for constructive mathematics, like the theory of types, one gets access to the whole conceptual apparatus of pure mathematics, neglecting those parts that depend critically on the law of excluded middle"

P. Martin-Löf, 1979

Real numbers and analysis

“Having formalized the construction of the real numbers (for example, as Cauchy sequences of rationals) in the theory of types, we can prove as a corollary to the normalization theorem that every individual real number which we can construct in the formal theory can be computed by a machine with any preassigned degree of precision.”

Also Bishop 1967

Real numbers and analysis

Turing *On computable numbers with an application to the Entscheidungsproblem*, 1936

Computable *real* numbers x: if we can compute the decimals by a finite machine

Problem: if x is very close to 1, maybe 0.99999999 or 1.000000001

A correction (pointed out by Bernays): Brouwer's definition (shrinking sequence of *overlapping* intervals), 1937, should allow negative decimals $0, 1, \overline{1} = -1$ so that 0.999999999 can be represented as $1.00000000\overline{1}$

Church *A note on the Entscheidungsproblem*, 1936

Real numbers and analysis

Work of Russell O'Connor (Nijmegen): real numbers as *functions*

Work of Yves Bertot (INRIA, Sophia-Antipolis): real numbers as *streams*

Real numbers and analysis

Metric Completion as a monad

$X \to C(X)$

$C(C(X)) \to C(X)$

$(X \to Y) \to (C(X) \to C(Y))$

Functions are uniformly continuous functions, $\mathbb{R} = C(\mathbb{Q})$

Works for *any* metric space; if we start with X metric space of finite set of rational points with Haussdorf metric, we get the set of compact subsets (e.g. fractals)

Real numbers and analysis

Evaluate within 10^{-20} (with correctness proofs)

$\sqrt{e/\pi}$ 1 sec

$sin((e+1)^3)$ 25 sec

$e^{e^{e^{1/2}}}$ 146 sec

All these computations are proved correct w.r.t. an existing library of constructive mathematics, developed by the Foundations Group, Radboud University Nijmegen

Real numbers as streams

Go back to Brouwer's original definition

Simplest case, *streams* of $-1, 0, 1$, as used in functional programming

Y. Bertot, N. Julien, computations of series and basic properties (for instance, computation of 1000 decimals of π in 4 sec.)

Used by R. Zuemkeller (INRIA) to check some parts of T. Hales proof of the Kepler conjecture

Planar graphs

Real numbers are complicated objects constructively. For instance $x < y$ not decidable, $x < y \vee y \leqslant x$ does not hold (and is replaced by $x < z \rightarrow \ x < y \vee y < z$)

By contrast, the notion of *finite graph* is combinatorial

More difficult: what is a planar graph (a graph drawn on the plane)?

Should we use topology, analysis, Jordan curve theorem?

Planar graphs

Search for a "computable" formalization (G. Gonthier, Microsoft-INRIA)

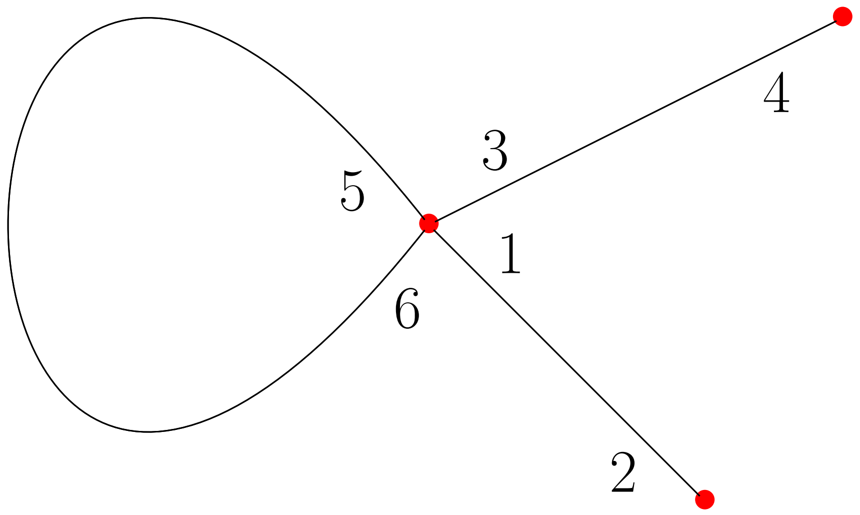
Should be easily formalisable in type theory

A structure where graph traversal is easy to program

G. Gonthier rediscovered in this way the structure of *hypermaps*: a finite set d with two maps $\alpha, \sigma : d \to d$ where α is a involution without fixed points and σ is a permutation

The elements of d are *darts* (oriented edges)

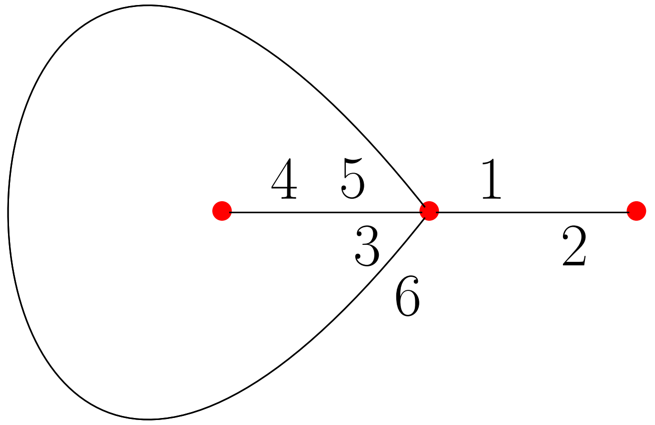
The *edges* are the cycles of α and the *nodes* are the cycles of σ



$$\alpha_1 = (1,2)(3,4)(5,6), \qquad \sigma_1 = (1,3,5,6)(2)(4)$$

The faces are the cycles of

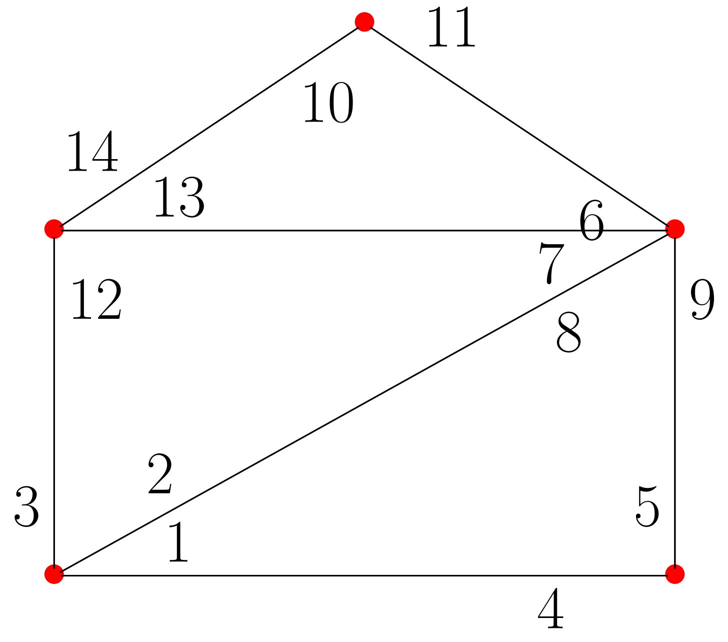
$$\phi_1 = \sigma_1^{-1}\alpha_1^{-1} = (1,2,6,3,4)(5)$$



$$\alpha_2 = (1,2)(3,4)(5,6), \qquad \sigma_2 = (1,5,3,6)(2)(4)$$

The faces are the cycles of

$$\phi_2 = \sigma_2^{-1}\alpha_2^{-1} = (1,2,6)(3,4,5)$$



$$\alpha = (1,4)(2,8)(3,12)(5,9)(6,11)(7,13)(10,14)$$

$$\sigma = (1,2,3)(4,5)(6,7,8,9)(10,11)(12,13,14)$$

$$\phi = (1,12,10,6,5)(2,4,9)(3,8,13)(7,11,14)$$

Planar maps

Further simplification (replacing reasoning by computation)

Structure d finite set with $\alpha, \sigma, \phi : d \rightarrow d$ such that $\sigma\alpha\phi = Id$

Cauchy: any substitution is a product of disjoint cycles

Symmetrical representation with respect to edges, nodes and faces.

Now used in the proof of Kepler conjecture

This represents a graph drawn on a *surface* (oriented). One needs an extra (combinatorial) condition to state that this can is drawn on the plane.

Graph theory is “reduced” to finite group theory

Hypermaps in type theory

J.F. Dufourd (U. Strasbourg) uses a similar notion of hypermap to formalize and synthetise programs in computational geometry

Formalized Euler's theorem for polyhedra in type theory $V - E + F = 2$

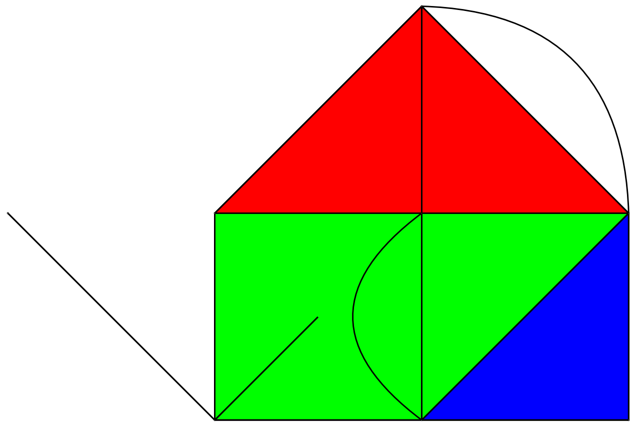
Cf. discussion in Lakatos *Proofs and Refutations*

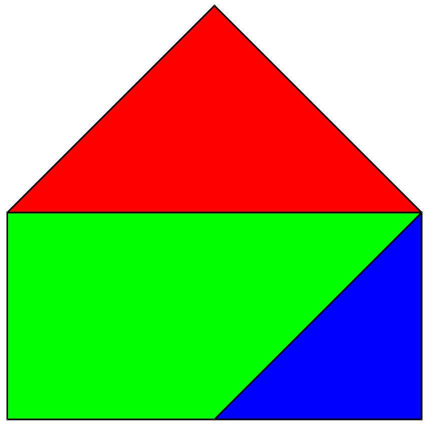
Image segmentation algorithm

How to specify and build a correct image segmentation algorithm

Need the mathematical theory for the specification and the proof of correctness

Direct to rewrite the algorithm in C (if desired)





Planar graphs

Results intuitive but not trivial to represent formally

Finite structures: the properties are computable

Excluded-middle holds for these structures and properties

Four Color Theorem

First proof by Kempe, published in 1879

Error in the proof that took ten years to spot (and 100 years to fix)

1976: Appel and Haken, required computer calculations too large to be checked by hand

1997: Simplification by Robertson, Sanders, Seymour, Thomas, reduce the number of cases, but still too large to be checked by hand

Four Color Theorem

"Though proof assistants have been around for some 30 years, we are the first to use them to prove a major result that absolutely requires the use of computers. The main technique we used to accomplish this, known as computation reflection, basically amounts to replacing mathematical proof with software debugging; this raises the perspective that proof assistant technology could be effectively transferred to the engineering of reliable software."

Started as a programming project for students in a basic computer science course

Four Color Theorem

(1) A purely combinatorial part:

find a set of configurations (633 configurations) such that two properties hold

Reducibility: 1.000.000.000 cases

Unavoidability: 10.000 cases

Use only intuitionistic logic, all properties are computable

(2) Verify that the combinatorial part fits the topology (analysis, use classical logic)

Four Color Theorem

Define a set config

Define cfreducible : config → prop

Define checkreducible : config → bool and prove

$$\forall x : \text{config}.\ \text{checkreducible}\ x \rightarrow \text{cfreducible}\ x$$

Apply this for each configuration (the actual computation for one configuration may involve up to 20.000.000 cases)

Functional Programming

Robertson, Sanders, Seymour, Thomas (1997) 35 pages + a C program

Example of a proof (Result 3.3) *This is a "folklore" theorem, and we omit its proof, which is straightforward*

35 pages formalized in type theory

no extra C program: purely functional, using more sophisticated algorithms, multiway decision diagrams, zipper (G. Huet), Davis-Putnam, abstract interpretation

Even with the added sophistication, the program verification part was the easiest, most straightforward part of this project

New mathematics

Gonthier found a new combinatorial statement of Jordan's curve Theorem

This gives a new combinatorial definition of planarity

Four Color Theorem

Some conclusions of this work:

Sophisticated decision procedures can be cast as functional programs

correctness proofs are typically easy

formally proving programs is easier than formally proving theorems

machine formalization can lead to new mathematical insights

Four Color Theorem

Although this work is purportedly about using computer programming to help doing mathematics, we expect that most of its fallout will be in the reverse direction - using mathematics to help programming computers. The approach that proved successful for this proof was to turn almost every mathematical concept into a data structure or a program, thereby converting the entire enterprise into one of program verification

In many respects, these proof scripts are closer to debugger or testing scripts than to to mathematical texts.

We believe it is quite significant that such a simple-minded strategy succeeded on a "higher mathematics" problem. Clearly, this is the most important conclusion one should draw from this work

Functional programming

The complete checking of the proof of the four color theorem was made possible by an efficient implementation of type checking ($=$ proof checking) by Benjamin Gregoire and Xavier Leroy (2001)

This implementation reduces strong evaluation of functional terms (evaluation under abstraction, partial evaluation) to weak evaluation, which is used in functional programming

Functional programming

Terms/programs

$$M \ ::= \ x \mid M \ M \mid \lambda x.M$$

Computation rule $(\lambda x.N)\ M = N[x/M]$ (β-reduction)

Values/results of computation

$$u \ ::= \ \lambda x.M$$

Ordinary functional programming only computes the *weak head-normal form*

Functional programming

An weak evaluator computes the value $\mathsf{U}(M)$ of a *closed* term

$$\frac{}{\mathsf{U}(\lambda x.M) = \lambda x.M}$$

$$\frac{\mathsf{U}(M_1) = \lambda x.N \qquad \mathsf{U}(M_2) = u_2 \qquad \mathsf{U}(N[x/u_2]) = v}{\mathsf{U}(M_1\ M_2) = v}$$

Functional programming

Uniform implementation: Landin SECD machine (call-by-value), Shmidt/Krivine's machine (call-by-name), CAM (call-by-value)

Efficient implementation: Leroy ZAM machine (call-by-value) virtual machine of caml

For dependent type theory we need to compute open terms

How to get Strong Reduction?

Solution: introduce new symbolic terms $X_0, X_1, \ldots$

First do weak normalization; second *read back* the resulting value as a normalized term, recursing over the bodies of the function if needed

How to get Strong Reduction?

$$u \ ::= \ \lambda x.M \mid k, \qquad k \ ::= \ X_i \mid k\ u$$

The full normal form of a closed term M is computed as $\mathsf{R}_0(\mathsf{U}(M))$ where R_i is the *readback* procedure

$$\mathsf{R}_i(\lambda x.M) = \lambda X_i.\mathsf{R}_{i+1}(\mathsf{U}(M[x/X_i]))$$

$$\mathsf{R}_i(X_l) = X_l, \qquad \mathsf{R}_i(k\ u) = \mathsf{R}_i(k)\ \mathsf{R}_i(u)$$

How to get Strong Reduction?

Example: M is $(\lambda x.x)(\lambda y.(\lambda z.z)\ y\ (\lambda t.t))$

$\mathsf{U}(M)$ is $u_0 = \lambda y.(\lambda z.z)\ y\ (\lambda t.t)$

$\mathsf{R}_0(u_0)$ is $\lambda X_0.\mathsf{R}_1(\mathsf{U}((\lambda z.z)\ X_0\ (\lambda t.t)))$

$\mathsf{U}((\lambda z.z)\ X_0\ (\lambda t.t))$ is $u_1 = X_0\ (\lambda t.t)$

$\mathsf{R}_1(u_1)$ is $X_0\ (\lambda X_1.\mathsf{R}_2(\mathsf{U}(X_1))) = X_0\ (\lambda X_1.X_1)$

Thus the (strong) normal form of M is $\lambda X_0.X_0\ (\lambda X_1.X_1)$

How to get Strong Reduction?

In this way we can use the ordinary evaluation machine of functional programs (in this case the virtual machine of CAML) to evaluate terms of type theory

This has been crucial for the actual checking of the four color theorem

In general $\mathsf{R}_0(\mathsf{U}(M))$ will compute the *Böhm tree* of M

Functional programming

For type theory/functional programming we need to add *constructor values* and functions defined by cases

$$M \ ::= \ x \mid M \ M \mid \lambda x.M \mid c \ \vec{M} \mid f(M_1, \ldots, M_l)$$

New computation rules of the form $f(x_1, \ldots, x_l)(c \ \vec{x}) = M$

Functional programming

For instance

$$f(a,b)(0) = a, \qquad f(a,b)(S\ x) = b\ x\ (f(a,b)(x))$$

higher-order primitive recursion, Hilbert 1925, Gödel 1941

The values are now

$$u\ ::=\ \lambda x.M \mid c\ \vec{u} \mid f(u_1,\dots,u_l)$$

Functional programming

R.M. Burstall *Proving properties of programs by structural induction*, Computer Journal 12 (1): 41-48 (1969)

The main aim of this paper is to suggest some syntactic devices for writing programs in a way which makes it easier to derive proofs by structural induction

Functional programming

$$\frac{}{\mathsf{U}(\lambda x.M) = \lambda x.M}$$

$$\frac{\mathsf{U}(M_1) = \lambda x.N \qquad \mathsf{U}(M_2) = u_2 \qquad \mathsf{U}(N[x/u_2]) = v}{\mathsf{U}(M_1\ M_2) = v}$$

Functional programming

$$\frac{\mathsf{U}(M_1) = u_1 \ \ \ldots \ \ \mathsf{U}(M_n) = u_n}{\mathsf{U}(c\ M_1 \ \ \ldots \ \ M_n) = c\ u_1 \ \ \ldots \ \ u_n}$$

$$\frac{\mathsf{U}(M_1) = u_1 \ \ \ldots \ \ \mathsf{U}(M_n) = u_n}{\mathsf{U}(f(M_1, \ldots, M_n)) = f(u_1, \ldots, u_n)}$$

$$\frac{\mathsf{U}(M_1) = f(\vec{u}) \qquad \mathsf{U}(M_2) = c\ \vec{v} \qquad \mathsf{U}(N[\vec{u}, \vec{v}]) = v}{\mathsf{U}(M_1\ M_2) = v}$$

where we have the computation $f(x_1, \ldots, x_l)(c\ \vec{x}) = N$

Strong reduction

$$u \;::=\; \lambda x.M \mid c\ \vec{u} \mid f(u_1,\ldots,u_l) \mid k$$

$$k \;::=\; X_i \mid k\ u \mid f(u_1,\ldots,u_l)(k)$$

Strong reduction

$$\mathsf{R}_i(\lambda x.M) = \lambda X_i.\mathsf{R}_{i+1}(\mathsf{U}(M[x/X_i]))$$

$$\mathsf{R}_i(f(u_1,\ldots,u_l)) = \lambda X_i.\mathsf{R}_{i+1}(f(u_1,\ldots,u_l)(X_i))$$

$$\mathsf{R}_i(c\ u_1\ \ \ldots\ \ u_n) = c\ (\mathsf{R}_i\ u_1)\ \ \ldots\ \ (\mathsf{R}_i\ u_n)$$

$$\mathsf{R}_i(X_l) = X_l, \qquad \mathsf{R}_i(k\ u) = \mathsf{R}_i(k)\ \mathsf{R}_i(u)$$

$$\mathsf{R}_i(f(u_1,\ldots,u_l)(k)) = f(\mathsf{R}_i(u_1),\ldots,\mathsf{R}_i(u_l))(\mathsf{R}_i(k))$$

Normalization and domain theory

Denotational semantics

$$\mathsf{V} \;=\; (\mathsf{V} \rightarrow_s \mathsf{V}) \oplus \Sigma c\; \mathsf{V}^{\otimes ar(c)}$$

We consider the *strict* semantics $[\![M]\!]_s \in \mathsf{V}$

Theorem: (U. Berger, 2005) *If* $[\![M]\!]_s \neq \perp$ *then* M *is strongly normalisable*

How to improve existing systems based on type theory

Type theory as *total* functional programming (cf. D. Turner, 2004) with dependent types

Denotational semantics

Better module systems (first step is denotational semantics)

More functional programming notations

For one step in this direction, see Agda (agda wiki), Epigram

How to improve existing systems based on type theory

```
filter : {A : Set}  -> (A -> Bool) -> List A -> List A
filter p [] = []
filter p (x :: xs) with p x
...                 | true = x : filter p xs
...                 | false = filter p xs

subset  : {A : Set}  -> (p : A -> Bool) ->
          (xs : List A) -> subseteq (filter p xs) xs
subset p [] = stop
subset p (x :: xs) with p x
...                 | true = keep (subset p xs)
...                 | false = drop (subset p xs)
```

Classical Logic

$$\frac{\text{computer programming}}{\text{functional programming}} \equiv \frac{\text{classical mathematics}}{\text{constructive mathematics}}$$

$EM : \neg\neg A \rightarrow A,\qquad C[EM\ u] = u\ (\lambda x^A.C[x])$ if $u : \neg\neg A$

Not possible with dependent types: $\lambda x^A.C[x]$ may not be well-typed for instance if $C[x]$ is $h\ x\ p$ with

$$p : P(EM\ u) \qquad h : \Pi x : A.P(x) \rightarrow \perp$$

$h\ (EM\ u)\ p$ is well-typed but not $h\ x\ p\ (x : A)$

Classical Logic

Open problem 1: to extend type theory with computation rules for EM

Open problem 2: to extend type theory with computation rules for extensional choice (which implies EM)

The fact that this should be possible is suggested by the proof of the elimination of the ϵ symbol (Hilbert, Bernays)

Computer Programming

Use of type theory to check imperative programs; X. Leroy (INRIA) Compcert: a compiler that generates PowerPC assembly code from a (large) subset of C. Its correctness is proved in type theory.

D. Pichardie: constructive abstract interpretation

Concurrent C Minor, joint Princeton INRIA project

Ynot project: extending type theory with imperative and concurrent features

Hoare Type Theory