Constructive Type Theory
and
Interactive Theorem Proving

Peter Dybjer
Chalmers Tekniska Högskola
Göteborg, Sweden

JSSST
Sendai, 15 September 2005

Interactive theorem provers - proof assis

Examples:

Classical set theory, Zermelo 1908: Mizar (1973-)

Classical type theory, Church 1940: HOL (early 1980s),
(PVS)

Constructive type theory, Scott 1970, Martin-Löf 1972:
1980s), Coq (1990-), Agda, ...

(Early systems: Automath, LCF, ...)

What is constructive type theory? Some

- Constructivism. Brouwer 1908.
- Type theory. Russell, Whitehead 1910. Church 1940
- Intuitionistic logic. BHK. Realizability interpretation, Klee
- Propositions as types, Curry-Howard 1957, 1969.
- Foundations of constructive analysis. Bishop 1967
- Constructive type theory. Scott 1970, Martin-Löf 1972

Also: primitive recursion, Gödel's T, Lawvere's quantifiers as

Constructive mathematics and computer prog

Constructive type theory = Functional programming dependent types where all programs terminate

Constructive mathematics = Computer programming

A quotation from "Constructive Mathematics and Comp ming" (Martin-Löf 1979).

“the whole conceptual apparatus of programming mir
modern mathematics (set theory, that is, not geometry)
supposed to be different from it. How come? The reas
curious situation is, I think, that *mathematical notions hav*
received an interpretation, the interpretation which we
classical, which makes them *unusable for programming*. Fo
do not need to enter the philosophical debate as to wheth
sical interpretation of the primitive logical and mathemat
(proposition, truth, set, element, function etc.) is suffic
because this much is at least clear, that if a *function* is c
binary relation satisfying the usual existence and unicity
whereby classical reasoning is allowed in the existence
set of ordered pairs satisfying the corresponding conditi
function cannot be the same kind of thing as a program.
a *set* is understood in Zermelo’s way as a member of the
hierarchy, then a set cannot be the same thing as a *data*

Now it is the contention of the intuitionists (or the con
I shall use these terms synonymously) that the basic m
notions, above all the notion of function, ought to be
in such a way that *the cleavage between mathematic*
mathematics, that is, and programming that we are w
present disappears.

...

What I have just said about the close connection be
structive mathematics and programming explains why the *i*
type theory ..., which I began to develop solely with the
ical motive of *clarifying the syntax and semantics of i*
mathematics, may equally well be viewed as a *programmin*

What is constructive mathematics?

- Functions are computable
- Proofs of implications are computable functions ("methoc
- A proof of a disjunction is either a proof of left or of righ
- A proof of existence gives a witness

Hence, not excluded middle, not double negation.

The Brouwer-Heyting-Kolmogorov interpr

A proof of $A \supset B$ is a method which transforms a proof o of B.

A proof of $A \wedge B$ is a pair consisting of a proof of A and

A proof of $A \vee B$ is either a proof of A or a proof of B.

A proof of $\forall x : A. B$ is a method which for an arbitrary e returns a proof of $B[x := a]$.

A proof of $\exists x : A. B$ is a pair consisting of a element witness) and a proof of $B[x := a]$.

Propositions as types - towards constructive t

Curry 1957 observed the similarity between the types
S-combinators

$$\begin{aligned} \mathrm{K} &: A \to B \to A \\ \mathrm{S} &: (A \to B \to C) \to (A \to B) \to A \to C \end{aligned}$$

and two Hilbert-style axioms for implication

$$A \supset B \supset A$$
$$(A \supset B \supset C) \supset (A \supset B) \supset A \supset C$$

Moreover, the typing rule for application corresponds to the
ponens!

The Curry-Howard identification

$$\begin{aligned}
A \supset B &= A \to B \\
A \wedge B &= A \times B \\
A \vee B &= A + B \\
\forall x : A.B &= \Pi x : A.B \\
\exists x : A.B &= \Sigma x : A.B \\
\top &= \mathbf{1} \\
\bot &= \mathbf{0} \\
\neg A &= A \to \mathbf{0}
\end{aligned}$$

An example: Hindley-Milner typability and typ

In a functional language such as Haskell we may write fu

```
(i)   has_type :: Term -> Bool
(ii)  type_of  :: Term -> Maybe Type
```

which test (i) whether a term is typable (ii) in case it is retu
it. Here

```
data Maybe a = Nothing | Just a
```

Typability and type inference in constructive t

Let Term be the set of terms of the lambda calculus, T
types of the lambda calculus, and $::$ be the typing relation s
means that M has type σ.

Consider the following proposition in typed predicate logi

$$\mathrm{dec\_type} \ : \ \forall M : \mathrm{Term}.(\mathrm{Typable}\ M) \vee \neg\ (\mathrm{Typabl}$$

where

$$\mathrm{Typable}\ M = \exists \sigma : \mathrm{Type}.M :: \sigma$$

Classical proof is trivial! Constructive proof is a decision alg
inference algorithm, which computes its own correctness witn

Original Martin-Löf type theory with one u
$(\mathbf{MLTT_U})$

- Set formers for predicate logic: $\mathbf{0}, \mathbf{1}, +, \times, \to, \Sigma, \Pi$.
- Natural numbers N.
- Universe of small sets U.

All these were introduced in Martin-Löf 1972.

Rules for natural numbers

Formation rule:

$$\mathrm{N} \quad : \quad \mathrm{Set}$$

Introduction rules:

$$\begin{aligned} 0 \quad &: \quad \mathrm{N} \\ \mathrm{Succ} \quad &: \quad \mathrm{N} \rightarrow \mathrm{N} \end{aligned}$$

Primitive recursion = mathematical indu

Elimination rule = rule for building proofs by mathema = rule for typing functions from natural numbers where t dependent type.:

$$\begin{array}{lll}\mathrm{R} & : & (C : \mathrm{N} \to \mathrm{Set}) \to C\ 0 \to ((x : \mathrm{N}) \to C\ x \to C\ (\mathrm{S} \\ & & (n : \mathrm{N}) \to C\ n\end{array}$$

Computation rules:

$$\begin{array}{rcl}\mathrm{R}\, C\ d\ e\, 0 & = & d : C\ 0 \\ \mathrm{R}\, C\ d\ e\, (\mathrm{Succ}\, n) & = & e\ n\, (\mathrm{R}\, C\ d\ e\ n) : C\ (\mathrm{Succ}\end{array}$$

Primitive recursive schema

If $C : \mathrm{N} \to \mathrm{Set}, d : C\ 0, e : (x : \mathrm{N}) \to C\ x \to C\ (\mathrm{Succ}\ x)$,

$$\begin{aligned} f\ 0 &= d \\ f\ (\mathrm{Succ}\ n) &= e\ n\ (f\ n) \end{aligned}$$

then we can define

$$f = \mathrm{R}\ C\ d\ e : (n : \mathrm{N}) \to C\ n$$

Observe, that $C\ n$ can be a function type; we can program t
function.

Arithmetic in $\mathrm{MLTT_U}$

$$
\begin{aligned}
\mathrm{pred}\; n &= \mathrm{R}\;(\lambda x.\mathrm{N})\;0\;(\lambda x, y.\;x)\;n \\
m + n &= \mathrm{R}\;(\lambda x.\mathrm{N})\;m\;(\lambda x, y.\;\mathrm{Succ}\;y)\;n \\
m \dot{-} n &= \mathrm{R}\;(\lambda x.\mathrm{N})\;m\;(\lambda x, y.\;\mathrm{pred}\;y)\;n \\
m * n &= \mathrm{R}\;(\lambda x.\mathrm{N})\;0\;(\lambda x, y.\;y + m)\;n
\end{aligned}
$$

What about division? It is primitive recursive, but the Euclic
can be implemented by using primitive recursion of highe
measure.

Equality of natural numbers

Define

$$\mathrm{eq}_{\mathrm{N}} \quad : \quad \mathrm{N} \to \mathrm{N} \to \mathrm{Bool}$$

by pattern matching on constructors

$$\begin{aligned}
\mathrm{eq}_{\mathrm{N}}\ 0\ 0 &= \mathrm{True} \\
\mathrm{eq}_{\mathrm{N}}\ 0\ (\mathrm{Succ}\ n) &= \mathrm{False} \\
\mathrm{eq}_{\mathrm{N}}\ (\mathrm{Succ}\ m)\ 0 &= \mathrm{False} \\
\mathrm{eq}_{\mathrm{N}}\ (\mathrm{Succ}\ m)\ (\mathrm{Succ}\ n) &= \mathrm{eq}_{\mathrm{N}}\ m\ n
\end{aligned}$$

Equality of natural numbers in $\mathbf{MLTT}$

Use the elimination rule for N and define it by primitiv
higher type (primitive recursive functional) as follows. Define

$$\mathrm{eq}_{\mathrm{N}}\ m : \mathrm{N} \to \mathrm{Bool}$$

by induction on $m : \mathrm{N}$. The base case is "to be equal to zero
case is to define "to be equal to $m + 1$" in terms of "to be e

Note that in $\mathbf{MLTT_U}$ we define $\mathrm{Bool} = \mathbf{1} + \mathbf{1}$.

How to define dependent types

Recursively, define a family of types (a dependent type):

$$\mathrm{Vect} \quad : \quad \mathrm{Set} \to \mathrm{N} \to \mathrm{Set}$$

abbreviated $A^n = \mathrm{Vect}\ A\ n$

$$\begin{aligned} A^0 &= \mathbf{1} \\ A^{\mathrm{Succ}\ n} &= A \times A^n \end{aligned}$$

This definition is directly accepted by Agda (using case). Ca
in $\mathbf{MLTT_U}$? Note that we cannot use R directly. Why?

Inductive-recursive definition of the universe a

The universe $\mathrm{U} : \mathrm{Set}$ of small sets is inductively generate
time as its decoding $\mathrm{T} : \mathrm{U} \to \mathrm{Set}$ is defined recursively:

$$
\begin{array}{rcl@{\quad}rcl}
\hat{\mathrm{N}} & : & \mathrm{U} & \mathrm{T}\ \hat{\mathrm{N}} & = & \mathrm{N} \\
\hat{\mathbf{0}} & : & \mathrm{U} & \mathrm{T}\ \hat{\mathbf{0}} & = & \mathbf{0} \\
\hat{\mathbf{1}} & : & \mathrm{U} & \mathrm{T}\ \hat{\mathbf{1}} & = & \mathbf{1} \\
(\hat{+}) & : & \mathrm{U} \to \mathrm{U} \to \mathrm{U} & \mathrm{T}\ (a \hat{+} b) & = & \mathrm{T}\ a\ \text{-} \\
(\hat{\times}) & : & \mathrm{U} \to \mathrm{U} \to \mathrm{U} & \mathrm{T}\ (a \hat{\times} b) & = & \mathrm{T}\ a\ \times \\
\hat{\Sigma} & : & (a : \mathrm{U}) \to (\mathrm{T}\ a \to \mathrm{U}) \to \mathrm{U} & \mathrm{T}\ (\hat{\Sigma}\ a\ b) & = & \Sigma\ (\mathrm{T} \\
 & \vdots & & & \vdots &
\end{array}
$$

Note that U is not a small set.

The universe at work

Now we can define

$$A^n = \mathrm{T}\ (\mathrm{R}\ (\lambda x.\mathrm{U})\ \hat{\mathbf{1}}\ (\lambda x, X.A \hat{\times} X)\ n)$$

for $A : \mathrm{U}$. (Note that we only define A^n for small A!)

The universe can also be used to define a family

$$\mathrm{Fin} : \mathrm{N} \to \mathrm{Set}$$

by

$$\begin{aligned} \mathrm{Fin}\ 0 &= \mathbf{0} \\ \mathrm{Fin}\ (\mathrm{Succ}\ n) &= \mathbf{1} + \mathrm{Fin}\ n \end{aligned}$$

More set formers

- Identity I (Martin-Löf 1973) - an inductive family/predica
- Well-orderings W (Martin-Löf 1979) - a generalized induc
- Hierarchy of universes $\mathrm{U}_0, \mathrm{U}_1, \mathrm{U}_2, \ldots$.

Well-orderings

A generalized inductive definition.

$$
\begin{aligned}
\mathrm{W} \quad &: \quad (A : \mathrm{Set}) \to (A \to \mathrm{Set}) \to \mathrm{Set} \\[1em]
\mathrm{Sup} \quad &: \quad (A : \mathrm{Set}) \to \\
& \quad\ (B : A \to \mathrm{Set}) \to \\
& \quad\ (a : A) \to \\
& \quad\ (B\ a \to \mathrm{W}\ A\ B) \to \\
& \quad\ \mathrm{W}\ A\ B
\end{aligned}
$$

The set of finitely branching trees

A special case of W:

$$\mathrm{V_{fin}} = \mathrm{W\ N\ Fin}$$

Finite trees will represent hereditarily finite sets. We ca
represent the finite von Neumann ordinals:

$$\begin{array}{rcl} \emptyset & = & \mathrm{Sup}\ 0\ \mathrm{case}_0 \\ \{\emptyset\} & = & \mathrm{Sup}\ 1\ b_1 \text{ where } b_1\ 0 = \emptyset \\ \{\emptyset, \{\emptyset\}\} & = & \mathrm{Sup}\ 2\ b_2 \text{ where } b_2\ 0 = \emptyset, b_2\ 1 = \{ \end{array}$$

(using $0 : \mathrm{Fin}\ 1$ and $0, 1 : \mathrm{Fin}\ 2$)

Hereditarily finite iterative sets

The elements of $\mathrm{V}_{\mathrm{fin}}$ can represent the hereditarily finite
sets all of whose elements are also hereditarily finite sets. H
comparing two hereditarily finite sets for equality, order an
elements do not matter. We define extensional equality as bi

$$\begin{aligned}\mathrm{Sup}\; n\; b =_{\mathrm{ext}} \mathrm{Sup}\; n'\; b' \quad &= \quad \forall i : \mathrm{Fin}\; n.\; \exists i' : \mathrm{Fin}\; n'.\; b\; i = \\ & \quad\;\; \forall i' : \mathrm{Fin}\; n'.\; \exists i : \mathrm{Fin}\; n.\; b'\; i'\end{aligned}$$

(Note: we have omitted the two parameter arguments of Su

Extensional membership is defined by

$$a \in_{\mathrm{ext}} \mathrm{Sup}\; n\; b \quad = \quad \exists i : \mathrm{Fin}\; n. a =_{\mathrm{ext}} b\; i$$

Operations on hereditarily finite sets

We can now define computable operations on herediarily

- $\cap, \cup : \mathrm{V}_{\mathrm{fin}} \to \mathrm{V}_{\mathrm{fin}} \to \mathrm{V}_{\mathrm{fin}}$
- $\bigcup . \mathcal{P} : \mathrm{V}_{\mathrm{fin}} \to \mathrm{V}_{\mathrm{fin}}$

Aczel's constructive cumulative hierarch

$\mathrm{V}_{\mathrm{fin}}$ only contains hereditarily finite iterative sets. In a s can define Aczel's set V of iterative sets by

$$\mathrm{V} = \mathrm{W}\ \mathrm{U}\ \mathrm{T}$$

The branching can now be indexed by an arbitrary (possibly set $\mathrm{T}\ a$. The definitions of extensional equality and extension are analogous to those for $\mathrm{V}_{\mathrm{fin}}$, except that their values are than in Bool.

Aczel gives axioms for a constructive version CZF of where the axioms hold for V with extensional equality a membership.

Constructive foundations

Predicative constructive systems:

Type theory. Martin-Löf type theory

Lambda calculus (untyped). Aczel's first order theory o (logical theory of constructions etc.). Use intuitionistic and inductive predicates on domain of lambda expressions. explicit mathematics.

Set theory. Myhill-Aczel's Constructive ZF - use axioms for

Category theory. Moerdijk - Palmgren's predicative topos - category of setoids in Martin-Löf type theory

Part II: Interactive theorem provers base
constructive type theory

NuPRL. Cornell, from early 1980s. Extensional Martin-Löf t

Alf, Agda, Alfa. Chalmers, from early 1980s (Alf 1990, Ag
laboration with AIST from 2004. Intensional Martin-Löf t

Coq. INRIA, from 1984 (Coq 1990). The Calculus of Induc
tions (intensional impredicative type theory).

Cf Japanese tradition - program extraction from constructive
Hayashi (PX), Sato, etc).

From Martin-Löf type theory to Agd

- The implementation is based on a type-checking algorith constructive type theory has the strong normalization prop checking of normal terms is decidable!

- $\mathbf{MLTT_U}$ (+W, etc) is an inconvenient language for prog general inductive definitions, general recursive schemata wi checker, records, and modules.

- Proof by pointing and clicking! Interactively refine typin with metavariables.

- Recent trends: lighter notation by introducing "implici plugins of tools for proof search and random testing.

Inductive definitions

Consider again the problem of ML-style type inference.

- Type and Term are *inductively defined sets* ("recursive da

- The typing relation $M :: \sigma$ between a term and a type is *defined relation*.

It is possible to code these definitions in $\mathbf{MLTT_U}$, but in taken as primitives. There is a construct `data` which makes declare new inductively defined sets much like one declares a type in a functional language, e g the terms of combinatory

```
Term :: Set = data K | S | App (f :: Term)
```

Inductive definitions and constructive foun

Each inductive definition comes with its own formation elimination, and computation rules, which can be systematic from the definition.

Martin-Löf 1984: "We can follow the same pattern u natural numbers to introduce other inductively defined sets the example of lists".

Martin-Löf 1972: "The type N is just the prime exam introduced by an *ordinary inductive definition*. However, it se to treat this special case rather than to give a necessari complicated general formulation which would include $(\Sigma \in A$ N_n and N as special cases. See Martin-Löf 1971 for a general inductive definitions in the language of ordinary first order pr

Inductively defined relation = inductively defi

```
HasType :: Term -> Type -> Set
= idata Ktype (A,B :: Type)   :: _ K (A => (B =>
        Stype (A,B,C :: Type) :: ...
        Apptype (A,B :: Type)
                (f,a :: Term)
                (d :: HasType f (A => B))
                (e :: HasType a A) ::
                _ (App f a) B
```

is Agda's representation of the definition of the typing relatio

$$\mathrm{K} : A \Rightarrow B \Rightarrow A \qquad \mathrm{S} : \cdots \qquad \frac{f : A \Rightarrow B}{f\ a : B}$$

What is an inductive definition in general? I

- the rules for generating natural numbers by zero and succ
- the rules for generating well-formed formulas of a logic
- the axioms and inference rules generating theorems of the
- the productions of a context-free grammar
- the computation rules for a programming language
- the reflexive-transitive closure of a relation

Inductive definitions and recursive datat

- lists generated by `Nil` and `Cons`
- binary trees generated by `EmptyTree` and `MkTree`
- algebraic types in general: parameterized, many sorted ter
- infinitely branching trees; Brouwer ordinals; etc.
- inductive dependent types (vectors of a certain length, tre height, balanced trees, etc)
- inductive-recursive definitions (sorted lists, freshlists, etc)

Reflexive and nested datatypes

Note that recursive datatypes in functional languages
include reflexive datatypes

```
data Lambda = Nil | Lambda (Lambda -> Lambda
```

and nested datatypes

```
data Nest a = Nil | Cons a (Nest (a,a))
data Bush a = Nil | Cons a (Bush (Bush a))
```

Neither is accepted verbatim as an inductive definition in M
theory.

Inductive definitions and constructive foun

Classically, inductive definitions are understood as least f
monotone operators (or least sets closed under a set of rules

P. Aczel (An introduction to inductive definitions, Handb
matical Logic, 1976, pp 779 and 780.):

> An alternative approach is to take induction as a primi
> not needing justification in terms of other methods. ...
> interesting to formulate a coherent conceptual framework
> induction the principal notion.

No universal principle. We may discover new stronger induct
principles.

Inductive definitions and the notion of
in Martin-Löf type theory

Martin-Löf type theory is such a coherent conceptual fran

"(1) a set A is defined by prescribing how a canonical
A is formed as well as how two equal canonical elemen
formed."

Per Martin-Löf (p8 in Intuitionistic Type Theory, Biblic

This is the same as saying that a set is defined by its introc
e, the rules for inductively generating its members.

Martin-Löf type theory and inductive defi

- Basic set formers: $\Pi, \Sigma, +, \mathrm{I}, \mathrm{N}, \mathrm{N}_n, \mathrm{W}, \mathrm{U}_n$
- Adding new set formers with their rules when there is a n
lists, binary trees, the well-founded part of a relation, ...
- Exactly what is a good inductive definition? Schemata
definitions, indexed inductive definitions, inductive-recursiv
- Generic formulation: universes for inductive definitions, inc
definitions, inductive-recursive definitions

Inductive-recursive definitions

Recall the inductive-recursive definition of the universe á
only display one constructor to show the inductive-recursive
definition:

$$
\begin{array}{rcl}
\mathrm{U} & : & \mathrm{Set} \\
\mathrm{T} & : & \mathrm{U} \to \mathrm{Set}
\end{array}
$$

$$
\begin{array}{rcl}
\hat{\Sigma} & : & (a : \mathrm{U}) \to (\mathrm{T}a \to \mathrm{U}) \to \mathrm{U} \\
\mathrm{T}\ (\hat{\Sigma}\ a\ b) & = & \Sigma x : \mathrm{T}\ a . \mathrm{T}\ (b\ x)
\end{array}
$$

Why is such a strange definition constructively valid? Use M
meaning explanations!

Inductive-recursive definition of ordered

$$
\begin{aligned}
\mathrm{OrdList} &: \mathrm{Set} \\
\mathrm{lb} &: \mathrm{N} \to \mathrm{OrdList} \to \mathrm{Bool}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Nil} &: \mathrm{OrdList} \\
\mathrm{Cons} &: (x : \mathrm{N}) \to (xsp : \mathrm{OrdList}) \to \mathrm{T}\ (\mathrm{l}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{lb}\ x\ \mathrm{Nil} &= \mathrm{True} \\
\mathrm{lb}\ x\ (\mathrm{Cons}\ y\ xsp\ q) &= x \leq y
\end{aligned}
$$

Recursion schemata

In $\mathbf{MLTT_U}$ all recursion must be expressed using the rec
nators (elimination rule), that is, programming must be dor
(or structural) recursion. This is inconvenient in practice.

In Agda one does not need to adhere to this principle stri

- Functions can be defined by case analysis
- Recursive calls are checked by separate termination checker
is that recursive calls are on *structurally smaller* terms.

Examples of definitions accepted by Ag

$$
\begin{aligned}
\text{half } 0 &= 0 \\
\text{half } (\text{Succ } 0) &= 0 \\
\text{half } (\text{Succ } (\text{Succ } n)) &= \text{Succ } (\text{half } n)
\end{aligned}
$$

$$
\begin{aligned}
\text{eq}_{\mathrm{N}}\ 0\ 0 &= \text{True} \\
\text{eq}_{\mathrm{N}}\ 0\ (\text{Succ } n) &= \text{False} \\
\text{eq}_{\mathrm{N}}\ (\text{Succ } m)\ 0 &= \text{False} \\
\text{eq}_{\mathrm{N}}\ (\text{Succ } m)\ (\text{Succ } n) &= \text{eq}_{\mathrm{N}}\ m\ n
\end{aligned}
$$

Examples of definitions accepted by Agd

Also recursive definitions of sets are accepted directly wit
to universes:

$$\begin{array}{rcl} A^0 & = & \mathbf{1} \\ A^{\mathrm{Succ}\ n} & = & A \times A^n \end{array}$$

Remark: Agda has a construct `case` for definition by case ar

Building proofs by pointing and clicki

The most recent interactive theorem prover for Martin-L built at Chalmers, main implementor Catarina Coquand wit Makoto Takeyama (former Chalmers now at AIST).

The window interface Alfa written by Thomas Hallgren.

Alf. Main idea. "Do proof by pointing and clicking". Bui

$$a : A$$

by step-wise constructing a and A. Either think of a as a terr as a program with the specification A or as a proof of the pr

An example

Build the polymorphic identity function.

$$\lambda A.\lambda x.x : (A : \mathrm{Set}) \to A \to A$$

Write this in Agda syntax, and let Agda type-check it!

```
id :: (A :: Set) -> A -> A
id = \A -> \x -> x
```

However, for complex dependent programs and proofs in con
theory it is unfeasible to directly write it down and type-chec

Interactively refine typing with metavari

First, give the function a name, eg "id", with an unkn
unknown definition:

```
id :: ?0
id = ?1
```

You can now stepwise instantiate the type $?1$ and term $?2$.
type. It is a dependent function type. Place the cursor on $?$
template for dependent function space. `(A :: ?) -> ?` anc
command "refine"! Agda checks that it is a correct partial ty
Your screen is

```
id :: (A :: ?2) -> ?3
id = ?1
```

Interactively refine typing with metavariab

```
id :: (A :: ?2) -> ?3
id = ?1
```

You can now refine either $?1, ?2$, or 3. If we refine $?1$ we c
command "abstract" after typing a variable name e g A in th
for $?1$. We get

```
id :: (A :: ?2) -> ?3
id = \(A :: ?4) -> ?5
```

Etc. At each stage the type-checking algorithm maintains t
of the typing. Unlike Coq, Agda always shows the partial te
on the screen. Agda also has a command "suggest" sugg
refinements.

Proof construction

Proof construction is the same as term construction - y
a proof term on the screen. (This is unlike most other syst
Coq, where you do not see the proof terms directly, but inst
mands/tactics manipulating proofs, reducing goals to subg
with systems such as Coq, where you write the script "re
"auto", ...

Automation - three possibilities

Reflection. Write internal decision procedure:

```
decide :: Sublogic -> Bool
[[-]]  :: Sublogic -> Set
sound  :: (phi :: Sublogic) -> decide phi = Tr
```

Proof search by external tool producing proof object. E
the Agda Synthesizer. Proof-object checked by type-check

Proof search by external tool producing no proof object
FOL-plugins of AgdaLight and Agda.

Combining tests and proofs

Some of Agda's propositions (types) are testable in a sim QuickCheck tool of Claessen and Hughes. Cf Hayashi's use connection with PX.

Example. The following type expresses the correctnes algorithm sort

```
(xs :: List N) ->
  (ordered (sort xs) && permutation xs (sort xs)
```

Test it by randomly generating elements of `List N`, and che

Cover project at Chalmers is about combining randon automatic and interactive proof.

Conclusion: intensional constructive type th
classical logic as basis for interactive theorem

Advantages:

- "Native" functional programming language with powerful
- Normalization during type-checking. Reflection.

Disadvantages:

- Intensionality?
- Automatic techniques for classical logic more well-develop