Constructive Type Theory and Interactive Theorem Proving

Peter Dybjer
Chalmers Tekniska Högskola
Göteborg, Sweden

VERITE
Kanazawa, 22 September 2005

Interactive theorem provers - proof assistants

Examples:

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

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

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

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

What is constructive type theory? Some roots.

- Constructivism. Brouwer 1908.
- Type theory. Russell, Whitehead 1910. Church 1940
- Intuitionistic logic. BHK. Realizability interpretation, Kleene
- 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 adjoints, ...

Constructive mathematics and computer programming

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

Constructive mathematics = Computer programming

A quotation from "Constructive Mathematics and Computer Programming" (Martin-Löf 1979).

"the whole conceptual apparatus of programming mirrors that of modern mathematics (set theory, that is, not geometry) and yet is supposed to be different from it. How come? The reason for this curious situation is, I think, that *mathematical notions have gradually received an interpretation*, the interpretation which we refer to as classical, which makes them *unusable for programming*. Fortunately, I do not need to enter the philosophical debate as to whether the classical interpretation of the primitive logical and mathematical notions (proposition, truth, set, element, function etc.) is sufficiently clear, because this much is at least clear, that if a *function* is defined as a *binary relation satisfying the usual existence and unicity conditions*, whereby classical reasoning is allowed in the existence proof, or a set of ordered pairs satisfying the corresponding conditions, then *a function cannot be the same kind of thing as a program*. Similarly, if a *set* is understood in Zermelo's way as a member of the *cumulative hierarchy*, then a set cannot be the same thing as a *data type*."

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

...

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

What is constructive mathematics?

- Functions are computable
- Proofs of implications are computable functions ("methods")
- A proof of a disjunction is either a proof of left or of right disjunct
- A proof of existence gives a witness

Hence, not excluded middle, not double negation.

The Brouwer-Heyting-Kolmogorov interpretation

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

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

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 element a of A returns a proof of $B[x := a]$.

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

Propositions as types - towards constructive type theory

Curry 1957 observed the similarity between the types of the K and 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 rule of modus 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 type inference

In a functional language such as Haskell we may write functions

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

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

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

Typability and type inference in constructive type theory

Let Term be the set of terms of the lambda calculus, Type the set of types of the lambda calculus, and :: be the typing relation so that $M :: \sigma$ means that M has type σ.

Consider the following proposition in typed predicate logic

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

where

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

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

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

- 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:

$$0 \quad : \quad \mathrm{N}$$

$$\mathrm{Succ} \quad : \quad \mathrm{N} \to \mathrm{N}$$

Primitive recursion = mathematical induction

Elimination rule = rule for building proofs by mathematical induction = rule for typing functions from natural numbers where the target is a dependent type.:

$$\begin{aligned} \mathrm{R} \quad : \quad & (C : \mathrm{N} \to \mathrm{Set}) \to C\ 0 \to ((x : \mathrm{N}) \to C\ x \to C\ (\mathrm{Succ}\ x)) \to \\ & (n : \mathrm{N}) \to C\ n \end{aligned}$$

Computation rules:

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

Primitive recursive schema

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

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

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 the Ackermann function.

Arithmetic in $\mathbf{MLTT_U}$

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

What about division? It is primitive recursive, but the Euclidean algorithm can be implemented by using primitive recursion of higher type and a measure.

Equality of natural numbers

Define

$$\mathrm{eq_N} \quad : \quad \mathrm{N} \to \mathrm{N} \to \mathrm{Bool}$$

by pattern matching on constructors

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

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

Use the elimination rule for N and define it by primitive recursion of 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" and the step case is to define "to be equal to $m + 1$" in terms of "to be equal to m".

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). Can we define it in $\mathbf{MLTT_U}$? Note that we cannot use R directly. Why?

Inductive-recursive definition of the universe à la Tarski

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

$$
\begin{array}{rcll}
\hat{\mathrm{N}} & : & \mathrm{U} & \mathrm{T}\,\hat{\mathrm{N}} = \mathrm{N} \\
\hat{\mathbf{0}} & : & \mathrm{U} & \mathrm{T}\,\hat{\mathbf{0}} = \mathbf{0} \\
\hat{\mathbf{1}} & : & \mathrm{U} & \mathrm{T}\,\hat{\mathbf{1}} = \mathbf{1} \\
(\hat{+}) & : & \mathrm{U} \to \mathrm{U} \to \mathrm{U} & \mathrm{T}\,(a \hat{+} b) = \mathrm{T}\,a + \mathrm{T}\,b \\
(\hat{\times}) & : & \mathrm{U} \to \mathrm{U} \to \mathrm{U} & \mathrm{T}\,(a \hat{\times} b) = \mathrm{T}\,a \times \mathrm{T}\,b \\
\hat{\Sigma} & : & (a : \mathrm{U}) \to (\mathrm{T}\,a \to \mathrm{U}) \to \mathrm{U} & \mathrm{T}\,(\hat{\Sigma}\,a\,b) = \Sigma\,(\mathrm{T}\,a)\,(\lambda x.\mathrm{T}\,(bx)) \\
 & \vdots & & \vdots
\end{array}
$$

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/predicate
- Well-orderings W (Martin-Löf 1979) - a generalized inductive definition
- Hierarchy of universes $\mathrm{U}_0, \mathrm{U}_1, \mathrm{U}_2, \ldots$.

Well-orderings

A generalized inductive definition.

$$\mathrm{W} \quad : \quad (A : \mathrm{Set}) \to (A \to \mathrm{Set}) \to \mathrm{Set}$$

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

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 can for example represent the finite von Neumann ordinals:

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

(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, i e, finite sets all of whose elements are also hereditarily finite sets. However, when comparing two hereditarily finite sets for equality, order and repetition of elements do not matter. We define extensional equality as bisimilarity:

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

(Note: we have omitted the two parameter arguments of Sup.)

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 finite sets, e g:

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

Aczel's constructive cumulative hierarchy V

$\mathrm{V}_{\mathrm{fin}}$ only contains hereditarily finite iterative sets. In a similar way we 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 infinite) small set $\mathrm{T}\ a$. The definitions of extensional equality and extensional membership are analogous to those for $\mathrm{V}_{\mathrm{fin}}$, except that their values are in Set rather than in Bool.

Aczel gives axioms for a constructive version CZF of ZF set theory, where the axioms hold for V with extensional equality and extensional membership.

Constructive foundations

Predicative constructive systems:

Type theory. Martin-Löf type theory

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

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

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

Part II: Interactive theorem provers based on constructive type theory

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

Alf, Agda, Alfa. Chalmers, from early 1980s (Alf 1990, Agda 1996). Collaboration with AIST from 2004. Intensional Martin-Löf type theory.

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

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

From Martin-Löf type theory to Agda

- The implementation is based on a type-checking algorithm. Intensional constructive type theory has the strong normalization property and type-checking of normal terms is decidable!

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

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

- Recent trends: lighter notation by introducing "implicit" arguments, 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 data types").
- The typing relation $M :: \sigma$ between a term and a type is an *inductively defined relation*.

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

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

Inductive definitions and constructive foundations

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

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

Martin-Löf 1972: "The type N is just the prime example of a type introduced by an *ordinary inductive definition*. However, it seems preferable to treat this special case rather than to give a necessarily much more complicated general formulation which would include $(\Sigma \in A)B(x)$, $A + B$, N_n and N as special cases. See Martin-Löf 1971 for a general formulation of inductive definitions in the language of ordinary first order predicate logic."

Inductively defined relation = inductively defined family

```
HasType :: Term -> Type -> Set
= idata Ktype (A,B :: Type)   :: _ K (A => (B => A))
        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 relation

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

What is an inductive definition in general? Examples

- the rules for generating natural numbers by zero and successor
- the rules for generating well-formed formulas of a logic
- the axioms and inference rules generating theorems of the logic
- 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 datatypes

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

Reflexive and nested datatypes

Note that recursive datatypes in functional languages (e g Haskell) 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 Martin-Löf type theory.

Inductive definitions and constructive foundations

Classically, inductive definitions are understood as least fixed points of monotone operators (or least sets closed under a set of rules).

P. Aczel (An introduction to inductive definitions, Handbook of Mathematical Logic, 1976, pp 779 and 780.):

> An alternative approach is to take induction as a primitive notion, not needing justification in terms of other methods. ... It would be interesting to formulate a coherent conceptual framework that made induction the principal notion.

No universal principle. We may discover new stronger inductive generation principles.

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

Martin-Löf type theory is such a coherent conceptual framework.

> "(1) a set A is defined by prescribing how a canonical element of A is formed as well as how two equal canonical elements of A are formed."
>
> Per Martin-Löf (p8 in Intuitionistic Type Theory, Bibliopolis 1984)

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

Martin-Löf type theory and inductive definitions

- 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 need for them: lists, binary trees, the well-founded part of a relation, ...
- Exactly what is a good inductive definition? Schemata for inductive definitions, indexed inductive definitions, inductive-recursive definitions
- Generic formulation: universes for inductive definitions, indexed inductive definitions, inductive-recursive definitions

Inductive-recursive definitions

Recall the inductive-recursive definition of the universe á la Tarski. We only display one constructor to show the inductive-recursive nature of the 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 Martin-Löf style meaning explanations!

Inductive-recursive definition of ordered lists

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

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

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

Recursion schemata

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

In Agda one does not need to adhere to this principle strictly:

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

Examples of definitions accepted by Agda

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

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

Examples of definitions accepted by Agda - 2

Also recursive definitions of sets are accepted directly without reduction to universes:

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

Remark: Agda has a construct `case` for definition by case analysis.

Building proofs by pointing and clicking

The most recent interactive theorem prover for Martin-Löf type theory built at Chalmers, main implementor Catarina Coquand with extension by Makoto Takeyama (former Chalmers now at AIST).

The window interface Alfa written by Thomas Hallgren.

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

$$a : A$$

by step-wise constructing a and A. Either think of a as a term of type A or as a program with the specification A or as a proof of the proposition A.

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 constructive type theory it is unfeasible to directly write it down and type-check it.

Interactively refine typing with metavariables

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

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

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

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

Interactively refine typing with metavariables - 2

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

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

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

Etc. At each stage the type-checking algorithm maintains the consistency of the typing. Unlike Coq, Agda always shows the partial term/proof-term on the screen. Agda also has a command "suggest" suggesting possible refinements.

Proof construction

Proof construction is the same as term construction - you manipulate a proof term on the screen. (This is unlike most other systems, including Coq, where you do not see the proof terms directly, but instead give commands/tactics manipulating proofs, reducing goals to subgoals) Contrast with systems such as Coq, where you write the script "refine", "give", "auto", ...

Automation - three possibilities

Reflection. Write internal decision procedure:

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

Proof search by external tool producing proof object. Example: Agsy, the Agda Synthesizer. Proof-object checked by type-checker.

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

Combining tests and proofs

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

Example. The following type expresses the correctness of a sorting algorithm `sort`

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

Test it by randomly generating elements of `List N`, and check the RHS!

Cover project at Chalmers is about combining random testing with automatic and interactive proof.

Conclusion: intensional constructive type theory vs classical logic as basis for interactive theorem provers

Advantages:

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

Disadvantages:

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