Inductive and Recursive Definitions in Constructive Type Theory

Peter Dybjer
Chalmers Tekniska Högskola

TYPES summer school
Göteborg
August 2005

Some questions

What is an inductive definition of a set? What is a recursive definition of a function? Classically? Constructively?

What are the differences and similarities wrt recursive data types in functional languages?

What is the role of inductive definitions in the foundations of constructive mathematics? What is Martin-Löf type theory? What is the role of inductive definitions in Martin-Löf type theory? What other foundational systems for constructive mathematics are there? What is the role of inductive definitions for them?

What is the nature of Martin-Löf's meaning explanations? What is the syntactico-semantical approach to constructive foundations?

More questions

What inductive definitions are constructively acceptable? The versatility of the constructive notion of an inductive definition. What are inductive families (indexed inductive definitions)? What are generalized inductive definitions? What are inductive-recursive definitions?

What recursive definitions are constructively acceptable? What are the differences and similarities wrt recursive function definitions in functional languages? What is the role of pattern matching? What is the role of well-founded recursion vs structural recursion? What is the relationship between Martin-Löf type theory and Agda? How do you program with inductive definitions?

How can you axiomatize a general theory of inductive and recursive definitions in Martin-Löf type theory with a minimum of coding?

Plan

1. Martin-Löf type theory with one universe ($\mathbf{MLTT_U}$). Rules for natural numbers. Large elimination.

2. What is an inductive definition?

   (a) Examples
   (b) Classical definition. Rule sets. Monotone operators.

3. What is the language of constructive mathematics? What is the data? What is the role of inductive definitions?

4. Ordinary inductive definitions.

   (a) Inductive sets. Lists, binary trees, propositional formulas. General schema.
   (b) Inductive families. Family of theorems. General schema.

5. Generalized inductive definitions.

   (a) Brouwer ordinals.
   (b) Well-orderings. Hereditarily finite sets. Aczel's V and CZF.
   (c) Well-founded part of a relation. Termination of programs.

6. Induction-recursion. More about meaning explanations.

7. (Finite axiomatization of inductive and inductive-recursive definitions, if time permits)

Terminology

later Martin-Löf	lecture notes	Bishop	early Martin-Löf	other
type	sort		category	kind
set	type	preset	type	
extensional set	set	set		setoid, E-set
function	operation	operation	function	
extensional function	function	function		setoid map,

We here follow later Martin-Löf. The "lecture notes" above refer to "Type-theoretic Foundations of Constructive Mathematics" by Coquand, Dybjer, Palmgren, and Setzer.

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.

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 $\mathrm{U}_0, \mathrm{U}_1, \mathrm{U}_2, \ldots$.
- Universe à la Tarski (Martin-Löf 1984) U, T - an inductive-recursive definition

Rules for natural numbers

Formation rule:

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

Introduction rules:

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

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

Elimination and equality rules for natural numbers

Elimination rule:

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

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

Equality 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

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

Exercise: define some functions in $\mathbf{MLTT_U}$

1. addition, subtraction, and multiplication of natural numbers

2. the half function:

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

3. division of natural numbers

Equality of natural numbers

Define

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

by pattern matching on constructors

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

Exercise: define equality of natural numbers in $\mathbf{MLTT_U}$

Hint. 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 $\mathrm{Bool} = \mathbf{1} + \mathbf{1}$.

Recursive function definitions in Agda

The Alf/Agda philosophy: we do not limit ourselves to the primitive recursive schema formalized by $\mathbb{N}$-elimination, but allow more general recursion patterns. There is a termination checker which checks that the recursive calls refer to "structurally smaller" arguments.

For example, the above definition of equality is accepted as a good definition (syntax may use case analysis) since it passes the termination checker. There is ongoing research on extending the termination checker.

Recursive definitions of sets

Define

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

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?

Large elimination

If we modify R, so that the result type is Set instead of *a set* $C\ n$, then we get a *large* elimination rule

$$\mathrm{R}^{\mathrm{large}} \quad : \quad \mathrm{Set} \to (\mathrm{N} \to \mathrm{Set} \to \mathrm{Set}) \to \mathrm{N} \to \mathrm{Set}$$

Now we can define

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

The universe of small sets

Large elimination rules are not part of $\mathbf{MLTT_U}$. Instead we show how to use the universe U to approximate the effect of large elimination. We here choose the formulation à la Tarski (Aczel 1974, Martin-Löf 1984), where we have a set U of codes for small sets, and a decoding function T:

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

Remark: earlier versions of Martin-Löf type theory used universes à la Russell, where $a : \mathrm{Set}$ if $a : \mathrm{U}$.

Inductive-recursive definition of the universe à la Tarski

We have one introduction rule for U and one equality rule for T for each small set former:

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

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!)

Exercise: Define a family

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

so that $\mathrm{Fin}\ n$ is a set with n elements.

The equality proposition

We would like to have

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

so that $\mathrm{Eq_N}\ m\ n$ is inhabited iff $\mathrm{eq_N}\ m\ n = \mathrm{True}$. How is this defined in $\mathbf{MLTT_U}$? In Agda we can define directly (using case)

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

Exercise: some uses large elimination for truth values

Define the following functions in $\mathbf{MLTT_U}$:

1. the following function which converts a truth value to a proposition:

$$\begin{aligned} \mathrm{T}_{\mathrm{Bool}} &: \mathrm{Bool} \to \mathrm{Set} \\ \mathrm{T}_{\mathrm{Bool}}\,\mathrm{True} &= \mathbf{1} \\ \mathrm{T}_{\mathrm{Bool}}\,\mathrm{False} &= \mathbf{0} \end{aligned}$$

2. $\mathrm{Eq}_{\mathrm{N}} : \mathrm{N} \to \mathrm{N} \to \mathrm{Set}$.

Lists and other inductive definitions

List is not a primitive set former in $\mathbf{MLTT}_{\mathrm{U}}$. Can we encode it?

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". Exercise: write down the rules for list (formation, introduction, elimination, and equality rules).

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."

Inductive definitions – 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.

What is an inductive definition in general, classically?

Two equivalent notions of inductive definition of *subset* of a set V via

- rule sets on V
- monotone operators on subsets of V

See Aczel 1977: "An Introduction to Inductive Definitions" in Handbook of Mathematical Logic.

Sets inductively generated by rule sets

A *rule* on a base set V in Aczel's sense is a pair (X, x) (also written $\frac{X}{x}$), such that $X \subseteq V$ and $x \in V$.

Let Φ be a set of rules on V. A set Y is Φ-*closed* if for all $\frac{X}{x} \in \Phi$

$$X \subseteq Y \supset x \in Y$$

The set *inductively generated* by Φ is defined to be the least Φ-closed set

$$\mathcal{I}(\Phi) = \bigcap \{Y \subseteq V | Y \ \Phi\text{-}closed\},$$

The induction principle for $\mathcal{I}(\Phi)$ is "if Y is Φ-closed, then $\mathcal{I}(\Phi) \subseteq Y$". The introduction rules are "$\mathcal{I}(\Phi)$ is Φ-closed, that is, if $X \subseteq \mathcal{I}(\Phi)$ then $x \in \mathcal{I}(\Phi)$".

Example: reflexive-transitive closure of a relation

Rules for inductively generating $R^* \subseteq A \times A$ from $R \subseteq A \times A$:

$$\frac{}{xR^*x} \qquad \frac{xR^*y \qquad yR^*z}{xR^*z} \qquad \frac{xRy}{xR^*y}$$

Formal rule set (in the sense of Aczel) on $V = A \times A$:

$$\{\frac{\emptyset}{(x,x)}|x \in A\} \cup \{\frac{\{(x,y),(y,z)\}}{(x,z)}|x,y,z \in A\} \cup \{\frac{\emptyset}{(x,y)}|(x,y) \in R\}$$

Example: inference rules for minimal logic

$$\frac{\vdash x \Rightarrow y \qquad \vdash x}{\vdash y} \qquad \frac{}{\vdash x \Rightarrow y \Rightarrow x} \qquad \frac{}{\vdash (x \Rightarrow y \Rightarrow z) \Rightarrow (x \Rightarrow y) \Rightarrow x \Rightarrow z}$$

The corresponding rule set on $V = \mathrm{Form}$ (the set of formulas)

$$\{\frac{\{x \Rightarrow y, x\}}{y} | x, y \in \mathrm{Form}\} \cup \{\frac{\emptyset}{x \Rightarrow y \Rightarrow x} | x, y \in \mathrm{Form}\} \cup$$

$$\{\frac{\emptyset}{(x \Rightarrow y \Rightarrow z) \Rightarrow (x \Rightarrow y) \Rightarrow x \Rightarrow z} | x, y, z \in \mathrm{Form}\} \cup$$

Infinitary rules

The ω-rule is

$$\frac{(\vdash x_i)_{i\in\omega}}{\vdash \bigwedge_{i\in\omega} x_i}$$

We have the rule set

$$\{\frac{\{x_i | i \in \omega\}}{\bigwedge_{i\in\omega} x_i} | x_i \in \mathrm{Form} \text{ for all } i \in \omega\}$$

We here assume that $\bigwedge_{i\in\omega} x_i \in \mathrm{Form}$ whenever $x_i \in \mathrm{Form}$ for all $i \in \omega$.

Rules for generating natural numbers

Type-theoretic introduction rules

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

Rule set (What is V? N is given by a *fundamental* inductive definition)

$$\{\frac{\emptyset}{0}\} \cup \{\frac{\{n\}}{\mathrm{Succ}(n)} | n \in V\}$$

Monotone operator $\phi : \mathcal{P}(V) \to \mathcal{P}(V)$ which generates natural numbers:

$$\phi(X) = \{0\} \cup \{\mathrm{Succ}(n) | n \in X\} \cong 1 + X$$

Inductively defined sets generated by monotone operators

Let $\phi : \mathcal{P}(V) \to \mathcal{P}(V)$ be monotone, that is, if $X \subseteq Y \subseteq V$, then $\phi(X) \subseteq \phi(Y) \subseteq V$. Then ϕ has a least prefixed point

$$\mathcal{I}(\phi) = \bigcap \{X \subseteq V | \phi(X) \subseteq X\}$$

The induction principle is "if $\phi(X) \subseteq X$ then $\mathcal{I}(\phi) \subseteq X$". The introduction rule is "$\phi(\mathcal{I}(\phi)) \subseteq \mathcal{I}(\phi)$".

Exercise. Show that inductive generation by rule sets and monotone operators are equivalent.

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.

Towards a language for constructive mathematics

Constructivism:

- 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.

What is the data?

- Kleene's partial recursive functions: natural numbers
- Turing machines: strings of characters
- Lambda calculus (untyped): lambda expressions (mix program and data)

Code natural numbers as strings of characters or as lambda expressions.

Code functions, pairs, etc as natural numbers (Gödel coding). Even coding proofs as natural numbers (Kleene realizability).

Types of data

Natural numbers N

Higher order functions $A \to B$ (cf Gödel's T)

Propositions. Church type theory. Cf type U of small types.

More types? Cf development of programming languages.

In logic. Curry-Howard: $0, 1, A + B, A \times B, \Sigma_{x:A} B, \Pi_{x:A} B$.

This yields Martin-Löf type theory 1972. (Cf also Scott 1970: Constructive validity - check. Has also version of W-type.).

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

Formulae of minimal propositional logic

Another example of an ordinary inductive definition acceptable as a primitive notion in Martin-Löf type theory:

$$\begin{array}{rcl} \mathrm{Form} & : & \mathrm{Set} \end{array}$$

$$\begin{array}{rcl} \mathrm{Atom} & : & \mathrm{N} \to \mathrm{Form} \\ \Rightarrow & : & \mathrm{Form} \to \mathrm{Form} \to \mathrm{Form} \end{array}$$

Can it be defined in $\mathbf{MLTT_U}$?

Schema for ordinary inductive definitions of sets

Ordinary as opposed to *generalized* inductive definitions

Sets as opposed to *families* of sets.

We can introduce a new set P with finitely many constructors, where each constructor has finitely many arguments, the types of which are either P itself (an *inductive* argument/premise) or a set A (a *side condition/non-inductive* premise). The set A may depend on previous side-conditions, and may also make use of previously defined constants. It may not contain P.

The conclusion has type P.

Parameterized ordinary inductive definitions of sets

A definition can moreover depend on *parameters* which can have arbitrary types (including the type of sets). This is a third kind of argument to a constructor. The parameters always come before the side-conditions and the inductive arguments. (Inductive arguments and side-conditions can be mixed.)

All parameters appear as the initial arguments in formation, introduction, and elimination rules for P.

Remark: more general schemata exist for *inductive families*, *generalized induction*, and *induction-recursion*.

Lists as an example of a parameterized ordinary inductive definition of a set

Example, lists are given by a parameterized ordinary inductive definition of a set. The constructor

$$\mathrm{Cons} \quad : \quad (A : \mathrm{Set}) \to A \to [A] \to [A]$$

has three arguments: $A : \mathrm{Set}$ is a parameter, $a : A$ is a side-condition, $as : [A]$ is an inductive argument.

Exercise: Analyse the constructors of natural numbers, binary trees with information in the leaves, the set Form of formulas of minimal logic above. Analyse also $\times$ and Σ! What about $\to$ and Π?

The inductive family of theorems

$$\begin{aligned}
\mathrm{Thm} &: \mathrm{Form} \to \mathrm{Set} \\
\\
\mathrm{K} &: (a, b : \mathrm{Form}) \to \mathrm{Thm}\ (a \Rightarrow b \Rightarrow a) \\
\mathrm{S} &: (a, b, c : \mathrm{Form}) \to \mathrm{Thm}\ ((a \Rightarrow b \Rightarrow c) \Rightarrow (a \Rightarrow b) \Rightarrow a \Rightarrow c) \\
\mathrm{Mp} &: (a, b : \mathrm{Form}) \to \mathrm{Thm}\ (a \Rightarrow b) \to \mathrm{Thm}\ a \to \mathrm{Thm}\ b
\end{aligned}$$

Exercise. By Curry-Howard, Thm represents an inductively defined *predicate*. Define the predicate Thm in $\mathbf{MLTT}_{\mathrm{U}}$ up to logical equivalence!

Elimination rule for Thm

$$
\begin{aligned}
\mathrm{R}_{\mathrm{Thm}} \quad : \quad & (C : (a : \mathrm{Form}) \to \mathrm{Thm}\ a \to \mathrm{Set}) \to \\
& ((a, b : \mathrm{Form}) \to C\ (a \Rightarrow b \Rightarrow a)\ (\mathrm{K}\ a\ b)) \to \\
& ((a, b, c : \mathrm{Form}) \to C\ ((a \Rightarrow b \Rightarrow c) \Rightarrow (a \Rightarrow b) \Rightarrow a \Rightarrow c)\ (\mathrm{S}\ a\ b\ c)) \to \\
& ((a, b : \mathrm{Form}) \to (p : \mathrm{Thm}\ (a \Rightarrow b)) \to (q : \mathrm{Thm}\ a) \to \\
& \qquad C\ (a \Rightarrow b)\ p \to C\ a\ q \to C\ b\ (\mathrm{Mp}\ a\ b\ p\ q)) \to \\
& (a : \mathrm{Form}) \to (p : \mathrm{Thm}\ a) \to C\ a\ p
\end{aligned}
$$

Classical soundness of Thm

Exercise: use the elimination rules for Form and Thm to write the follwoing two functions:

$$\begin{aligned} \mathrm{eval} &: (\mathrm{N} \to \mathrm{Bool}) \to \mathrm{Form} \to \mathrm{Bool} \\ \mathrm{sound} &: (\rho : \mathrm{N} \to \mathrm{Bool}) \to (a : \mathrm{Form}) \to \mathrm{Thm}\ a \to (\mathrm{eval}\ \rho\ a =_{\mathrm{Bool}} \mathrm{True}) \end{aligned}$$

eval assigns classical semantics in Bool to each formula.

sound is a proof that all theorems are evaluated to True under this semantics:

Equality rules for Thm

$$\mathrm{R}_{\mathrm{Thm}}\, C\; d\; e\; f\; (a \Rightarrow b \Rightarrow a)\; (\mathrm{K}\; a\; b) \quad = \quad d\; a\; b$$

$$\mathrm{R}_{\mathrm{Thm}}\, C\; d\; e\; f\; ((a \Rightarrow b \Rightarrow c) \Rightarrow (a \Rightarrow b) \Rightarrow a))(\mathrm{S}\; a\; b\; c) \quad = \quad e\; a\; b\; c$$

$$\mathrm{R}_{\mathrm{Thm}}\, C\; d\; e\; f\; b\; (\mathrm{Mp}\; a\; b\; p\; q) \quad = \quad f\; a\; b\; (\mathrm{R}_{\mathrm{Thm}}\, C\; d\; e\; f\; (a \Rightarrow b)\; p)(\mathrm{R}_{\mathrm{Thm}}\, C\; d\; e\; f\; a\; q)$$

Schema for ordinary inductive definitions of families of sets

Like for *sets*, except that we have indices (cf Martin-Löf 1971):

We can introduce a new family of sets $P : I \to \mathrm{Set}$ with finitely many constructors, where each constructor has finitely many arguments, the types of which are either $P\ p$ (an inductive argument/premise) or a set A (a side condition/non-inductive premise). The index $p : I$ and the set A may depend on previous side-conditions, and may also make use of previously defined constants. (A must not contain P.)

The conclusion has the type $P\ q$, where again $q : I$ may depend on previous side-conditions, and may also make use of previously defined constants.

Inductive families in Agda

One can use `idata` in Agda for defining inductive families.

If the index q (in the conclusion type $P\ q$) is a variable, then one can also use `data` in Agda.

A generalized inductive definition: the Brouwer ordinals

$$
\begin{array}{rcl}
\mathcal{O} & : & \mathrm{Set} \\
 & & \\
0_{\mathcal{O}} & : & \mathcal{O} \\
\mathrm{Succ}_{\mathcal{O}} & : & \mathcal{O} \to \mathcal{O} \\
\mathrm{Sup}_{\mathcal{O}} & : & (\mathrm{N} \to \mathcal{O}) \to \mathcal{O}
\end{array}
$$

Note that the type of the argument of $\mathrm{Sup}_{\mathcal{O}}$ is a function type, representing the fact that it has an infinite number of (inductive) arguments. Note that $\mathcal{O}$ appears *strictly positively* in the argument type.

Aczel rule set for Brouwer ordinals

We get set-theoretic semantics of $\mathcal{O}$ by taking the set inductively generated by the following rule set:

$$\{\frac{\emptyset}{0}\} \cup \{\frac{\{\alpha\}}{\mathrm{Succ}(\alpha)} | \alpha \in V\} \cup \{\frac{\{\beta(n) | n \in \mathrm{N}\}}{\mathrm{Sup}(\beta)} | \beta \in \mathrm{N} \to V\}$$

Elimination rule

$$
\begin{aligned}
\mathrm{ordrec} \quad : \quad & (C : \mathcal{O} \to \mathrm{Set}) \to \\
& C\ 0_{\mathcal{O}} \to \\
& ((x : \mathcal{O}) \to C\ x \to C\ (\mathrm{Succ}_{\mathcal{O}}\ x)) \to \\
& ((f : \mathrm{N} \to \mathcal{O}) \to ((x : \mathcal{O}) \to C\ (f\ x)) \to C\ (\mathrm{Sup}_{\mathcal{O}}\ f)) \to \\
& (c : \mathcal{O}) \to \\
& C\ c
\end{aligned}
$$

Exercise: write down the equality rules.

Some Brouwer ordinals

$$\begin{aligned}
\omega &= \mathrm{Sup}_{\mathcal{O}}\ (\lambda n.\iota_{\mathrm{N}\mathcal{O}}\ n) : \mathcal{O} \\
\iota_{\mathrm{N}\mathcal{O}} &: \mathrm{N} \to \mathcal{O} \\
\iota_{\mathrm{N}\mathcal{O}}\ 0 &= 0_{\mathcal{O}} \\
\iota_{\mathrm{N}\mathcal{O}}\ (\mathrm{Succ}\ n) &= \mathrm{Succ}_{\mathcal{O}}\ (\iota_{\mathrm{N}\mathcal{O}}\ n)
\end{aligned}$$

Why is

$$2\omega = \mathrm{Sup}_{\mathcal{O}}\ (\lambda n.\mathrm{R}\ (\lambda n.\mathcal{O})\ \omega\ (\lambda y, z.\mathrm{Succ}_{\mathcal{O}}\ z))?$$

Exercise. Do some more ordinals, eg $\omega^2, \omega^\omega, \epsilon_0$. Do ordinal addition.

Well-orderings

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

Exercise: write down the elimination and equality rules.

Schema for generalized inductive definitions

Inductive arguments of constructors in a generalized inductive definition of a set P can have types

$$(x_1 : A_1) \to \ldots \to A_n \to P$$

where P does not appear in A_i. (A_i may depend on previous arguments, etc.)

It is also possible to encode all sets given by a generalized inductive definition in terms of W up to extensional equality.

Exercise: Find A and B so that $\mathrm{W}\ A\ B$ encodes N. Similar question for the Brouwer ordinals. (See Martin-Löf 1984)

The set of finitely branching trees

We can define the set of finitely branching trees with arbitrary finite branching degree (no information in the nodes)

$$\mathrm{V}_{\mathrm{fin}} = \mathrm{W}\ \mathrm{N}\ \mathrm{Fin}$$

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

Exercise: Define the empty hereditarily finite set. Define union and intersection, and power set of a hereditarily finite set! Define the finite ordinals.

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 $\mathrm{V}_{\mathrm{fin}}$.

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

Exercise: constructions in V

Check that the subset relation, the operations of union and intersection, and the finite ordinals are defined in the same way as in $\mathrm{V}_{\mathrm{fin}}$.

Construct the first infinite ordinal $\omega : \mathrm{V}$!

What happens if we try to define the powerset of an arbitrary element in V?

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. 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

The well-founded part of a relation

Given a set A and a binary relation $(>)$ on A an element x is in the well-founded part of $(>)$ if there is no infinite descending chain $x > x_1 > x_2 > \cdots$.

An alternative definition is by a generalized inductive definition: x is in the well-founded part of $(>)$ provided all elements x' which are "smaller" $(x > x')$ are in the well-founded part. In particular each "smallest" element is in the well-founded part.

$$\mathrm{Wfp} \quad : \quad A \to \mathrm{Set}$$

$$\mathrm{Sup} \quad : \quad (x : A) \to ((x' : A) \to (x > x') \to \mathrm{Wfp}\ x') \to \mathrm{Wfp}\ x$$

Exercise: correspondence between the two definitions of well-foundedness

Prove that the inductive definition implies the no-infinite descending chain definition in Martin-Löf type theory!

Wfp on the previous page was defined for a *fixed* set A and a *fixed* relation $(>)$. Rewrite the definiton so that A and $(>)$ become parameters!

Using the well-founded part to encode general recursive definitions

Encode general recursive function

```
f : A -> B
```

by

$$f' : (x : A) \to \mathrm{Wfp}\ A\ (>_{\mathtt{f}})\ x \to B$$

where $(>_{\mathtt{f}})$ is the recursive call relation: $x >_{\mathtt{f}} x'$ whenever the computation of `f` x will generate a call `f` x'.

Encoding division

For example, the partial recursive division function

```
div m n = if (n < m) then 0 else (div m (n - m))
```

has the recursive call relation $(>_{\texttt{div}\ m})$, where

$$n >_{\texttt{div}\ m} p \quad = \quad n =_{\mathrm{N}} m + p$$

What is $\mathrm{Wfp}\ (>_{\texttt{div}\ m})$? What happens if $m =_{\mathrm{N}} 0$?

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

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

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

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

Set-theoretic semantics of the universe à la Tarski

Rule set

$$\{\frac{\{(a,A)\}\cup\{(b(x),B(x))|x\in A\}}{(\hat{\Sigma}(a,b),\Sigma_{x:A}B(x))}|a,A\in V,b,B\in A\to V\}\cup\cdots$$

Exercise: give similar set-theoretic semantics to the inductive-recursive definition of sorted lists with the lower bound function!

Some references

- P. Aczel, An introduction to inductive definitions, chapter C.7 in the Handbook of Mathematical Logic, North-Holland 1977.

- Inductive and inductive-recursive definitions in Martin-Löf type theory:

  `http://www.cs.chalmers.se/~peterd/papers/inductive.html`

- The calculus of inductive constructions:

  `http://pauillac.inria.fr/cdrom/www/coq/doc/node.0.3.html`

System F

Alexandre Miquel — PPS & U. Paris 7

Alexandre.Miquel@pps.jussieu.fr

Types Summer School 2005

August 15–26 — Göteborg

Introduction

Introduction

- **System F:** independently discovered by

Introduction

- **System F:** independently discovered by

  Girard: System F (1970)

Introduction

- **System F:** independently discovered by

Girard:	System F	(1970)
Reynolds:	The polymorphic λ-calculus	(1974)

Introduction

- **System F:** independently discovered by

Girard:	System F	(1970)
Reynolds:	The polymorphic λ-calculus	(1974)

- Quite different motivations...

Girard:	Interpretation of second-order logic
Reynolds:	Functional programming

  ... connected by the Curry-Howard isomorphism

Introduction

- **System F:** independently discovered by

Girard:	System F	(1970)
Reynolds:	The polymorphic λ-calculus	(1974)

- Quite different motivations...

Girard:	Interpretation of second-order logic
Reynolds:	Functional programming

  ... connected by the Curry-Howard isomorphism

- Significant influence on the development of Type Theory
  - Interpretation of higher-order logic [Girard, Martin-Löf]
  - Type:Type [Martin-Löf 1971]
  - Martin-Löf Type Theory [1972, 1984, 1990, ...]
  - The Calculus of Constructions [Coquand 1984]

Part I

System F: Church-style presentation

System F syntax

Definition

Types	A, B	$::=$	α	$\mid A \to B$	$\mid \forall \alpha\ B$	
Terms	t, u	$::=$	x			
		$\mid$	$\lambda x : A\,.\,t$	$\mid tu$		(term abstr./app.)
		$\mid$	$\Lambda \alpha\,.\,t$	$\mid tA$		(type abstr./app.)

System F syntax

Definition

Types $A, B ::= \alpha \mid A \to B \mid \forall \alpha\ B$

Terms

$$t, u ::= x \mid \lambda x : A\,.\,t \mid tu \quad \text{(term abstr./app.)} \mid \Lambda \alpha\,.\,t \mid tA \quad \text{(type abstr./app.)}$$

Notations

- Set of free (term) variables: $FV(t)$
- Set of free type variables: $TV(t), \quad TV(A)$
- Term substitution: $u\{x := t\}$
- Type substitution: $u\{\alpha := A\}, \quad B\{\alpha := A\}$

Perform α-conversion to prevent captures of free (term/type) variables!

System F typing rules

Contexts $\Gamma \quad ::= \quad x_1 : A_1, \ \ldots, \ x_n : A_n$

Typing judgments $\Gamma \vdash t : A$

System F typing rules

Contexts $\Gamma \quad ::= \quad x_1 : A_1, \ \ldots, \ x_n : A_n$

Typing judgments $\Gamma \vdash t : A$

$$\frac{}{\Gamma \vdash x : A} \ {\scriptstyle (x:A)\in\Gamma}$$

$$\frac{\Gamma,\ x : A \vdash t : B}{\Gamma \vdash \lambda x : A \,.\, t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$$

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash \Lambda\alpha \,.\, t : \forall\alpha\ B} \ {\scriptstyle \alpha \notin TV(\Gamma)} \qquad \frac{\Gamma \vdash t : \forall\alpha\ B}{\Gamma \vdash tA : B\{\alpha := A\}}$$

System F typing rules

Contexts $\quad \Gamma \quad ::= \quad x_1 : A_1, \ \ldots, \ x_n : A_n$

Typing judgments $\quad \Gamma \vdash t : A$

$$\frac{}{\Gamma \vdash x : A} \ {\scriptstyle (x:A)\in\Gamma}$$

$$\frac{\Gamma,\ x : A \vdash t : B}{\Gamma \vdash \lambda x : A \,.\, t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$$

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash \Lambda\alpha \,.\, t : \forall\alpha\ B} \ {\scriptstyle \alpha \notin TV(\Gamma)} \qquad \frac{\Gamma \vdash t : \forall\alpha\ B}{\Gamma \vdash tA : B\{\alpha := A\}}$$

- Declaration of type variables is implicit (for each $\alpha \in TV(\Gamma)$)
- Type variables could be declared explicitly: $\alpha : *$ (cf PTS)
- One rule for each syntactic construct $\Rightarrow$ System is syntax-directed

Example: the polymorphic identity

- Set: $\mathsf{id} \equiv \Lambda\alpha\,.\,\lambda x : \alpha\,.\,x$

Example: the polymorphic identity

- Set: $\mathsf{id} \equiv \Lambda\alpha\,.\,\lambda x : \alpha\,.\,x$
- One has:

$$\mathsf{id} : \forall\alpha\ (\alpha \to \alpha)$$

Example: the polymorphic identity

- Set: $\mathsf{id} \equiv \Lambda\alpha\,.\,\lambda x : \alpha\,.\,x$
- One has:

$$\begin{array}{lcll} \mathsf{id} & : & \forall\alpha\ (\alpha \to \alpha) & \\ \mathsf{id}\,B & : & B \to B & \text{for any type } B \end{array}$$

Example: the polymorphic identity

- Set: $\mathsf{id} \equiv \Lambda\alpha\,.\,\lambda x : \alpha\,.\,x$
- One has:

$$\begin{array}{lcll} \mathsf{id} & : & \forall\alpha\ (\alpha \to \alpha) & \\ \mathsf{id}\,B & : & B \to B & \text{for any type } B \\ \mathsf{id}\,B\,u & : & B & \text{for any term } u : B \end{array}$$

Example: the polymorphic identity

- Set: $\mathsf{id} \equiv \Lambda\alpha\,.\,\lambda x : \alpha\,.\,x$
- One has:

$$\begin{array}{lcll} \mathsf{id} & : & \forall\alpha\ (\alpha \to \alpha) & \\ \mathsf{id}\,B & : & B \to B & \text{for any type } B \\ \mathsf{id}\,B\,u & : & B & \text{for any term } u : B \end{array}$$

- In particular, if we take $B \equiv \forall\alpha\ (\alpha \to \alpha)$ and $u \equiv \mathsf{id}$

Example: the polymorphic identity

- Set: $\mathsf{id} \equiv \Lambda\alpha\,.\,\lambda x : \alpha\,.\,x$
- One has:

$$\begin{array}{lcll} \mathsf{id} & : & \forall\alpha\ (\alpha \to \alpha) & \\ \mathsf{id}\,B & : & B \to B & \text{for any type } B \\ \mathsf{id}\,B\,u & : & B & \text{for any term } u : B \end{array}$$

- In particular, if we take $B \equiv \forall\alpha\ (\alpha \to \alpha)$ and $u \equiv \mathsf{id}$

$$\mathsf{id}\ (\forall\alpha\ (\alpha \to \alpha)) \quad : \quad \forall\alpha\ (\alpha \to \alpha) \to \forall\alpha\ (\alpha \to \alpha)$$

Example: the polymorphic identity

- Set: $\mathsf{id} \equiv \Lambda\alpha\,.\,\lambda x : \alpha\,.\,x$
- One has:

$$\begin{array}{lcll} \mathsf{id} & : & \forall\alpha\ (\alpha \to \alpha) & \\ \mathsf{id}\,B & : & B \to B & \text{for any type } B \\ \mathsf{id}\,B\,u & : & B & \text{for any term } u : B \end{array}$$

- In particular, if we take $B \equiv \forall\alpha\ (\alpha \to \alpha)$ and $u \equiv \mathsf{id}$

$$\begin{array}{lcl} \mathsf{id}\ \big(\forall\alpha\ (\alpha \to \alpha)\big) & : & \forall\alpha\ (\alpha \to \alpha) \to \forall\alpha\ (\alpha \to \alpha) \\ \mathsf{id}\ \big(\forall\alpha\ (\alpha \to \alpha)\big)\ \mathsf{id} & : & \forall\alpha\ (\alpha \to \alpha) \end{array}$$

Example: the polymorphic identity

- Set: $\mathsf{id} \equiv \Lambda\alpha\,.\,\lambda x : \alpha\,.\,x$
- One has:

$$\begin{array}{lcll} \mathsf{id} & : & \forall\alpha\ (\alpha \to \alpha) & \\ \mathsf{id}\,B & : & B \to B & \text{for any type } B \\ \mathsf{id}\,B\,u & : & B & \text{for any term } u : B \end{array}$$

- In particular, if we take $B \equiv \forall\alpha\ (\alpha \to \alpha)$ and $u \equiv \mathsf{id}$

$$\begin{array}{lcl} \mathsf{id}\ (\forall\alpha\ (\alpha \to \alpha)) & : & \forall\alpha\ (\alpha \to \alpha) \to \forall\alpha\ (\alpha \to \alpha) \\ \mathsf{id}\ (\forall\alpha\ (\alpha \to \alpha))\ \mathsf{id} & : & \forall\alpha\ (\alpha \to \alpha) \end{array}$$

$\Rightarrow$ Type system is impredicative (or cyclic)

Properties

Properties

Substitutivity (for types/terms):

Properties

Substitutivity (for types/terms):

- $\Gamma \vdash u : B \quad \Rightarrow \quad \Gamma\{\alpha := A\} \vdash u\{\alpha := A\} : B\{\alpha := A\}$

Properties

Substitutivity (for types/terms):

- $\Gamma \vdash u : B \quad \Rightarrow \quad \Gamma\{\alpha := A\} \vdash u\{\alpha := A\} : B\{\alpha := A\}$
- $\Gamma, x : A \vdash u : B, \quad \Gamma \vdash t : A \quad \Rightarrow \quad \Gamma \vdash u\{x := t\} : B$

Properties

Substitutivity (for types/terms):

- $\Gamma \vdash u : B \quad \Rightarrow \quad \Gamma\{\alpha := A\} \vdash u\{\alpha := A\} : B\{\alpha := A\}$
- $\Gamma, x : A \vdash u : B, \quad \Gamma \vdash t : A \quad \Rightarrow \quad \Gamma \vdash u\{x := t\} : B$

Uniqueness of type

Properties

Substitutivity (for types/terms):

- $\Gamma \vdash u : B \quad \Rightarrow \quad \Gamma\{\alpha := A\} \vdash u\{\alpha := A\} : B\{\alpha := A\}$
- $\Gamma, x : A \vdash u : B, \quad \Gamma \vdash t : A \quad \Rightarrow \quad \Gamma \vdash u\{x := t\} : B$

Uniqueness of type

$\Gamma \vdash t : A, \quad \Gamma \vdash t : A' \quad \Rightarrow \quad A = A' \quad (\alpha\text{-conv.})$

Properties

Substitutivity (for types/terms):

- $\Gamma \vdash u : B \quad \Rightarrow \quad \Gamma\{\alpha := A\} \vdash u\{\alpha := A\} : B\{\alpha := A\}$
- $\Gamma, x : A \vdash u : B, \quad \Gamma \vdash t : A \quad \Rightarrow \quad \Gamma \vdash u\{x := t\} : B$

Uniqueness of type

$\Gamma \vdash t : A, \quad \Gamma \vdash t : A' \quad \Rightarrow \quad A = A' \quad (\alpha\text{-conv.})$

Decidability of type checking / type inference

Properties

Substitutivity (for types/terms):

- $\Gamma \vdash u : B \quad \Rightarrow \quad \Gamma\{\alpha := A\} \vdash u\{\alpha := A\} : B\{\alpha := A\}$
- $\Gamma, x : A \vdash u : B, \quad \Gamma \vdash t : A \quad \Rightarrow \quad \Gamma \vdash u\{x := t\} : B$

Uniqueness of type

$\Gamma \vdash t : A, \quad \Gamma \vdash t : A' \quad \Rightarrow \quad A = A' \quad (\alpha\text{-conv.})$

Decidability of type checking / type inference

1. Given Γ, t and A, decide whether $\Gamma \vdash t : A$ is derivable

Properties

Substitutivity (for types/terms):

- $\Gamma \vdash u : B \quad \Rightarrow \quad \Gamma\{\alpha := A\} \vdash u\{\alpha := A\} : B\{\alpha := A\}$
- $\Gamma, x : A \vdash u : B, \quad \Gamma \vdash t : A \quad \Rightarrow \quad \Gamma \vdash u\{x := t\} : B$

Uniqueness of type

$\Gamma \vdash t : A, \quad \Gamma \vdash t : A' \quad \Rightarrow \quad A = A' \quad (\alpha\text{-conv.})$

Decidability of type checking / type inference

1. Given Γ, t and A, decide whether $\Gamma \vdash t : A$ is derivable
2. Given Γ and t, compute a type A such that $\Gamma \vdash t : A$ if such a type exists, or fail otherwise.

Properties

Substitutivity (for types/terms):

- $\Gamma \vdash u : B \quad \Rightarrow \quad \Gamma\{\alpha := A\} \vdash u\{\alpha := A\} : B\{\alpha := A\}$
- $\Gamma, x : A \vdash u : B, \quad \Gamma \vdash t : A \quad \Rightarrow \quad \Gamma \vdash u\{x := t\} : B$

Uniqueness of type

$\Gamma \vdash t : A, \quad \Gamma \vdash t : A' \quad \Rightarrow \quad A = A' \quad (\alpha\text{-conv.})$

Decidability of type checking / type inference

1. Given Γ, t and A, decide whether $\Gamma \vdash t : A$ is derivable
2. Given Γ and t, compute a type A such that $\Gamma \vdash t : A$ if such a type exists, or fail otherwise.

Both problems are decidable

Reduction rules

Two kinds of redexes:

Reduction rules

Two kinds of redexes:

$$(\lambda x : A \,.\, t)u \quad \succ \quad t\{x := u\} \qquad \text{1st kind redex}$$

Reduction rules

Two kinds of redexes:

$$
\begin{array}{rcll}
(\lambda x : A \,.\, t)u & \succ & t\{x := u\} & \text{1st kind redex} \\
(\Lambda \alpha \,.\, t)A & \succ & t\{\alpha := A\} & \text{2nd kind redex}
\end{array}
$$

Reduction rules

Two kinds of redexes:

$$
\begin{array}{rcll}
(\lambda x : A \,.\, t)u & \succ & t\{x := u\} & \text{1st kind redex} \\
(\Lambda \alpha \,.\, t)A & \succ & t\{\alpha := A\} & \text{2nd kind redex}
\end{array}
$$

Other combinations of abstraction and application are meaningless (and rejected by typing)

Reduction rules

Two kinds of redexes:

$$
\begin{array}{rcll}
(\lambda x : A \,.\, t)u & \succ & t\{x := u\} & \text{1st kind redex} \\
(\Lambda \alpha \,.\, t)A & \succ & t\{\alpha := A\} & \text{2nd kind redex}
\end{array}
$$

Other combinations of abstraction and application are meaningless (and rejected by typing)

Definitions

Reduction rules

Two kinds of redexes:

$$
\begin{array}{rcll}
(\lambda x : A \,.\, t)u & \succ & t\{x := u\} & \text{1st kind redex} \\
(\Lambda \alpha \,.\, t)A & \succ & t\{\alpha := A\} & \text{2nd kind redex}
\end{array}
$$

Other combinations of abstraction and application are meaningless (and rejected by typing)

Definitions

- One step β-reduction $t \succ t' \equiv$
  contextual closure of both rules above

Reduction rules

Two kinds of redexes:

$$
\begin{array}{rcll}
(\lambda x : A \,.\, t)u & \succ & t\{x := u\} & \text{1st kind redex} \\
(\Lambda \alpha \,.\, t)A & \succ & t\{\alpha := A\} & \text{2nd kind redex}
\end{array}
$$

Other combinations of abstraction and application are meaningless (and rejected by typing)

Definitions

- One step β-reduction $t \succ t' \equiv$
  contextual closure of both rules above
- β-reduction $t \succ^* t' \equiv$
  reflexive-transitive closure of $\succ$

Reduction rules

Two kinds of redexes:

$$\begin{array}{rcll} (\lambda x : A \,.\, t)u & \succ & t\{x := u\} & \text{1st kind redex} \\ (\Lambda \alpha \,.\, t)A & \succ & t\{\alpha := A\} & \text{2nd kind redex} \end{array}$$

Other combinations of abstraction and application are meaningless (and rejected by typing)

Definitions

- One step β-reduction $t \succ t' \equiv$
  contextual closure of both rules above
- β-reduction $t \succ^* t' \equiv$
  reflexive-transitive closure of $\succ$
- β-convertibility $t \simeq t' \equiv$
  reflexive-symmetric-transitive closure of $\succ$

Examples

Examples

- **The polymorphic identity, again**

Examples

- **The polymorphic identity, again**

$$\mathsf{id}\ B\ u \quad \equiv \quad (\Lambda\alpha\,.\,\lambda x : \alpha\,.\,x)\ B\ u$$

Examples

- **The polymorphic identity, again**

$$\mathsf{id}\ B\ u \quad \equiv \quad (\Lambda\alpha\,.\,\lambda x : \alpha\,.\,x)\ B\ u \quad \succ \quad (\lambda x : B\,.\,x)\ u$$

Examples

- **The polymorphic identity, again**

$$\mathsf{id}\ B\ u \quad \equiv \quad (\Lambda\alpha\,.\,\lambda x : \alpha\,.\,x)\ B\ u \quad \succ \quad (\lambda x : B\,.\,x)\ u \quad \succ \quad u$$

Examples

- **The polymorphic identity, again**

$$\mathsf{id}\ B\ u \quad \equiv \quad (\Lambda\alpha\,.\,\lambda x : \alpha\,.\,x)\ B\ u \quad \succ \quad (\lambda x : B\,.\,x)\ u \quad \succ \quad u$$

$$\mathsf{id}\ (\forall\alpha\ (\alpha{\to}\alpha))\ \mathsf{id}\ (\forall\alpha\ (\alpha{\to}\alpha))\ \cdots\ \mathsf{id}\ (\forall\alpha\ (\alpha{\to}\alpha))\ \mathsf{id}\ B\ u \quad \succ^{*} \quad u$$

Examples

- **The polymorphic identity, again**

$$\mathsf{id}\ B\ u \quad \equiv \quad (\Lambda\alpha\,.\,\lambda x:\alpha\,.\,x)\ B\ u \quad \succ \quad (\lambda x:B\,.\,x)\ u \quad \succ \quad u$$

$$\mathsf{id}\ (\forall\alpha\ (\alpha{\to}\alpha))\ \mathsf{id}\ (\forall\alpha\ (\alpha{\to}\alpha))\ \cdots\ \mathsf{id}\ (\forall\alpha\ (\alpha{\to}\alpha))\ \mathsf{id}\ B\ u \quad \succ^{*} \quad u$$

- **A little bit more complex example...**

Examples

- **The polymorphic identity, again**

$$\mathsf{id}\ B\ u \quad \equiv \quad (\Lambda\alpha\,.\,\lambda x : \alpha\,.\,x)\ B\ u \quad \succ \quad (\lambda x : B\,.\,x)\ u \quad \succ \quad u$$

$$\mathsf{id}\ (\forall\alpha\ (\alpha{\to}\alpha))\ \mathsf{id}\ (\forall\alpha\ (\alpha{\to}\alpha))\ \cdots\ \mathsf{id}\ (\forall\alpha\ (\alpha{\to}\alpha))\ \mathsf{id}\ B\ u \quad \succ^{*} \quad u$$

- **A little bit more complex example...**

$$\begin{array}{l}\big(\Lambda\alpha\,.\,\lambda x : \alpha\,.\,\lambda f : \alpha{\to}\alpha\,.\,\overbrace{f\,(\cdots\,(f}^{\text{32 times}}\,x)\cdots)\big)\\ \quad \big(\forall\alpha\ (\alpha{\to}(\alpha{\to}\alpha){\to}\alpha)\big)\ \big(\Lambda\alpha\,.\,\lambda x : \alpha\,.\,\lambda f : \alpha{\to}\alpha\,.\,f\,x\big)\\ \quad \big(\lambda n : \forall\alpha\ (\alpha{\to}(\alpha{\to}\alpha){\to}\alpha)\,.\,\Lambda\alpha\,.\,\lambda x : \alpha\,.\,\lambda f : \alpha{\to}\alpha\,.\,n\,\alpha\,(n\,\alpha\,x\,f)\,f\big)\end{array}$$

Examples

- **The polymorphic identity, again**

$$\mathsf{id}\ B\ u \quad \equiv \quad (\Lambda\alpha\,.\,\lambda x : \alpha\,.\,x)\ B\ u \quad \succ \quad (\lambda x : B\,.\,x)\ u \quad \succ \quad u$$

$$\mathsf{id}\ (\forall\alpha\ (\alpha{\to}\alpha))\ \mathsf{id}\ (\forall\alpha\ (\alpha{\to}\alpha))\ \cdots\ \mathsf{id}\ (\forall\alpha\ (\alpha{\to}\alpha))\ \mathsf{id}\ B\ u \quad \succ^{*} \quad u$$

- **A little bit more complex example...**

$$\begin{array}{l}\big(\Lambda\alpha\,.\,\lambda x : \alpha\,.\,\lambda f : \alpha{\to}\alpha\,.\,\overbrace{f\,(\cdots\,(f\,x)\cdots)}^{\text{32 times}}\big) \\ \quad \big(\forall\alpha\ (\alpha{\to}(\alpha{\to}\alpha){\to}\alpha)\big)\ \big(\Lambda\alpha\,.\,\lambda x : \alpha\,.\,\lambda f : \alpha{\to}\alpha\,.\,f\,x\big) \\ \quad \big(\lambda n : \forall\alpha\ (\alpha{\to}(\alpha{\to}\alpha){\to}\alpha)\,.\,\Lambda\alpha\,.\,\lambda x : \alpha\,.\,\lambda f : \alpha{\to}\alpha\,.\,n\,\alpha\,(n\,\alpha\,x\,f)\,f\big)\end{array}$$

$$\succ^{*} \quad \Lambda\alpha\,.\,\lambda x : \alpha\,.\,\lambda f : \alpha{\to}\alpha\,.\,\underbrace{(f\ \cdots\ (f}_{\text{4 294 967 296 times}}\,x)\cdots)$$

Properties

Properties

Confluence

$$t \succ^* t_1 \ \wedge \ t \succ^* t_2 \quad \Rightarrow \quad \exists t' \left(t_1 \succ^* t' \ \wedge \ t_2 \succ^* t' \right)$$

Properties

Confluence

$$t \succ^* t_1 \ \wedge \ t \succ^* t_2 \quad \Rightarrow \quad \exists t' \left(t_1 \succ^* t' \ \wedge \ t_2 \succ^* t' \right)$$

Proof. Roughly the same as for the untyped λ-calculus (adaptation is easy)

Properties

Confluence

$$t \succ^* t_1 \ \wedge \ t \succ^* t_2 \quad \Rightarrow \quad \exists t' \left(t_1 \succ^* t' \ \wedge \ t_2 \succ^* t' \right)$$

Proof. Roughly the same as for the untyped λ-calculus (adaptation is easy)

Church-Rosser

$$t_1 \simeq t_2 \quad \Leftrightarrow \quad \exists t' \left(t_1 \succ^* t' \ \wedge \ t_2 \succ^* t' \right)$$

Properties

Confluence

$$t \succ^* t_1 \ \wedge \ t \succ^* t_2 \quad \Rightarrow \quad \exists t' \, (t_1 \succ^* t' \ \wedge \ t_2 \succ^* t')$$

Proof. Roughly the same as for the untyped λ-calculus (adaptation is easy)

Church-Rosser

$$t_1 \simeq t_2 \quad \Leftrightarrow \quad \exists t' \, (t_1 \succ^* t' \ \wedge \ t_2 \succ^* t')$$

Subject-reduction

If $\Gamma \vdash t : A$ and $t \succ^* t'$ then $\Gamma \vdash t : A$

Properties

Confluence

$$t \succ^* t_1 \ \wedge \ t \succ^* t_2 \quad \Rightarrow \quad \exists t' \, (t_1 \succ^* t' \ \wedge \ t_2 \succ^* t')$$

Proof. Roughly the same as for the untyped λ-calculus (adaptation is easy)

Church-Rosser

$$t_1 \simeq t_2 \quad \Leftrightarrow \quad \exists t' \, (t_1 \succ^* t' \ \wedge \ t_2 \succ^* t')$$

Subject-reduction

If $\Gamma \vdash t : A$ and $t \succ^* t'$ then $\Gamma \vdash t : A$

Proof. By induction on the derivation of $\Gamma \vdash t : A$, with $t \succ t'$ (one step reduction)

Properties

Confluence

$$t \succ^* t_1 \ \wedge \ t \succ^* t_2 \quad \Rightarrow \quad \exists t' \, (t_1 \succ^* t' \ \wedge \ t_2 \succ^* t')$$

Proof. Roughly the same as for the untyped λ-calculus (adaptation is easy)

Church-Rosser

$$t_1 \simeq t_2 \quad \Leftrightarrow \quad \exists t' \, (t_1 \succ^* t' \ \wedge \ t_2 \succ^* t')$$

Subject-reduction

If $\Gamma \vdash t : A$ and $t \succ^* t'$ then $\Gamma \vdash t : A$

Proof. By induction on the derivation of $\Gamma \vdash t : A$, with $t \succ t'$ (one step reduction)

Strong normalisation

All well-typed terms of system F are strongly normalisable

Properties

Confluence

$$t \succ^* t_1 \ \wedge \ t \succ^* t_2 \quad \Rightarrow \quad \exists t' \, (t_1 \succ^* t' \ \wedge \ t_2 \succ^* t')$$

Proof. Roughly the same as for the untyped λ-calculus (adaptation is easy)

Church-Rosser

$$t_1 \simeq t_2 \quad \Leftrightarrow \quad \exists t' \, (t_1 \succ^* t' \ \wedge \ t_2 \succ^* t')$$

Subject-reduction

If $\Gamma \vdash t : A$ and $t \succ^* t'$ then $\Gamma \vdash t : A$

Proof. By induction on the derivation of $\Gamma \vdash t : A$, with $t \succ t'$ (one step reduction)

Strong normalisation

All well-typed terms of system F are strongly normalisable

Proof. Girard and Tait's method of reducibility candidates (postponed)

Part II

Encoding data types

Booleans (1/3)

Booleans (1/3)

Encoding of booleans

$$\mathsf{Bool} \equiv \forall\gamma\,(\gamma \to \gamma \to \gamma)$$

Booleans (1/3)

Encoding of booleans

$$\begin{array}{lcll} \mathsf{Bool} & \equiv & \forall\gamma\,(\gamma \to \gamma \to \gamma) & \\ \mathsf{true} & \equiv & \Lambda\gamma\,.\,\lambda x, y : \gamma\,.\,x & : \quad \mathsf{Bool} \end{array}$$

Booleans (1/3)

Encoding of booleans

$$
\begin{array}{lcll}
\mathsf{Bool} & \equiv & \forall\gamma\,(\gamma \to \gamma \to \gamma) & \\
\mathsf{true} & \equiv & \Lambda\gamma\,.\,\lambda x, y : \gamma\,.\,x & : \quad \mathsf{Bool} \\
\mathsf{false} & \equiv & \Lambda\gamma\,.\,\lambda x, y : \gamma\,.\,y & : \quad \mathsf{Bool}
\end{array}
$$

Booleans (1/3)

Encoding of booleans

$$
\begin{array}{lcll}
\mathsf{Bool} & \equiv & \forall\gamma\,(\gamma \to \gamma \to \gamma) & \\
\mathsf{true} & \equiv & \Lambda\gamma\,.\,\lambda x, y : \gamma\,.\,x & : \quad \mathsf{Bool} \\
\mathsf{false} & \equiv & \Lambda\gamma\,.\,\lambda x, y : \gamma\,.\,y & : \quad \mathsf{Bool}
\end{array}
$$

$$
\mathtt{if}_A \;\; u \;\; \mathtt{then} \;\; t_1 \;\; \mathtt{else} \;\; t_2 \quad \equiv \quad u \; A \; t_1 \; t_2
$$

Booleans (1/3)

Encoding of booleans

$$
\begin{array}{lcll}
\mathsf{Bool} & \equiv & \forall\gamma\,(\gamma \to \gamma \to \gamma) & \\
\mathsf{true} & \equiv & \Lambda\gamma\,.\,\lambda x, y : \gamma\,.\,x & : \quad \mathsf{Bool} \\
\mathsf{false} & \equiv & \Lambda\gamma\,.\,\lambda x, y : \gamma\,.\,y & : \quad \mathsf{Bool}
\end{array}
$$

$$
\mathtt{if}_A \;\; u \;\; \mathtt{then} \;\; t_1 \;\; \mathtt{else} \;\; t_2 \quad \equiv \quad u \; A \; t_1 \; t_2
$$

Correctness w.r.t. typing

Booleans (1/3)

Encoding of booleans

$$
\begin{array}{lcll}
\mathsf{Bool} & \equiv & \forall\gamma\,(\gamma \to \gamma \to \gamma) & \\
\mathsf{true} & \equiv & \Lambda\gamma\,.\,\lambda x, y : \gamma\,.\,x & : \quad \mathsf{Bool} \\
\mathsf{false} & \equiv & \Lambda\gamma\,.\,\lambda x, y : \gamma\,.\,y & : \quad \mathsf{Bool}
\end{array}
$$

$$
\mathtt{if}_A \;\; u \;\; \mathtt{then} \;\; t_1 \;\; \mathtt{else} \;\; t_2 \quad \equiv \quad u \; A \; t_1 \; t_2
$$

Correctness w.r.t. typing

$$
\frac{\Gamma \vdash u : \mathsf{Bool} \qquad \Gamma \vdash t_1 : A \qquad \Gamma \vdash t_2 : A}{\Gamma \;\vdash\; \mathtt{if}_A \;\; u \;\; \mathtt{then} \;\; t_1 \;\; \mathtt{else} \;\; t_2 \;\; : \;\; A}
$$

Booleans (1/3)

Encoding of booleans

$$
\begin{array}{lcll}
\mathsf{Bool} & \equiv & \forall\gamma\,(\gamma \to \gamma \to \gamma) & \\
\mathsf{true} & \equiv & \Lambda\gamma\,.\,\lambda x, y : \gamma\,.\,x & : \quad \mathsf{Bool} \\
\mathsf{false} & \equiv & \Lambda\gamma\,.\,\lambda x, y : \gamma\,.\,y & : \quad \mathsf{Bool}
\end{array}
$$

$$
\mathtt{if}_A \;\; u \;\; \mathtt{then} \;\; t_1 \;\; \mathtt{else} \;\; t_2 \quad \equiv \quad u \; A \; t_1 \; t_2
$$

Correctness w.r.t. typing

$$
\frac{\Gamma \vdash u : \mathsf{Bool} \qquad \Gamma \vdash t_1 : A \qquad \Gamma \vdash t_2 : A}{\Gamma \;\vdash\; \mathtt{if}_A \;\; u \;\; \mathtt{then} \;\; t_1 \;\; \mathtt{else} \;\; t_2 \;\; : \;\; A}
$$

Correctness w.r.t. reduction

Booleans (1/3)

Encoding of booleans

$$
\begin{array}{lcll}
\mathsf{Bool} & \equiv & \forall\gamma\,(\gamma \to \gamma \to \gamma) & \\
\mathsf{true} & \equiv & \Lambda\gamma\,.\,\lambda x, y : \gamma\,.\,x & : \quad \mathsf{Bool} \\
\mathsf{false} & \equiv & \Lambda\gamma\,.\,\lambda x, y : \gamma\,.\,y & : \quad \mathsf{Bool}
\end{array}
$$

$$
\mathtt{if}_A \;\; u \;\; \mathtt{then} \;\; t_1 \;\; \mathtt{else} \;\; t_2 \quad \equiv \quad u \; A \; t_1 \; t_2
$$

Correctness w.r.t. typing

$$
\frac{\Gamma \vdash u : \mathsf{Bool} \qquad \Gamma \vdash t_1 : A \qquad \Gamma \vdash t_2 : A}{\Gamma \;\vdash\; \mathtt{if}_A \;\; u \;\; \mathtt{then} \;\; t_1 \;\; \mathtt{else} \;\; t_2 \;\; : \;\; A}
$$

Correctness w.r.t. reduction

$$
\mathtt{if}_A \;\; \mathtt{true} \;\; \mathtt{then} \;\; t_1 \;\; \mathtt{else} \;\; t_2 \quad \succ^* \quad t_1
$$

Booleans (1/3)

Encoding of booleans

$$
\begin{array}{lcll}
\mathsf{Bool} & \equiv & \forall\gamma\,(\gamma \to \gamma \to \gamma) & \\
\mathsf{true} & \equiv & \Lambda\gamma\,.\,\lambda x, y : \gamma\,.\,x & : \quad \mathsf{Bool} \\
\mathsf{false} & \equiv & \Lambda\gamma\,.\,\lambda x, y : \gamma\,.\,y & : \quad \mathsf{Bool}
\end{array}
$$

$$
\mathtt{if}_A \;\; u \;\; \mathtt{then} \;\; t_1 \;\; \mathtt{else} \;\; t_2 \quad \equiv \quad u \; A \; t_1 \; t_2
$$

Correctness w.r.t. typing

$$
\frac{\Gamma \vdash u : \mathsf{Bool} \qquad \Gamma \vdash t_1 : A \qquad \Gamma \vdash t_2 : A}{\Gamma \;\vdash\; \mathtt{if}_A \;\; u \;\; \mathtt{then} \;\; t_1 \;\; \mathtt{else} \;\; t_2 \;\; : \;\; A}
$$

Correctness w.r.t. reduction

$$
\begin{array}{lcl}
\mathtt{if}_A \;\; \mathsf{true} \;\; \mathtt{then} \;\; t_1 \;\; \mathtt{else} \;\; t_2 & \succ^* & t_1 \\
\mathtt{if}_A \;\; \mathsf{false} \;\; \mathtt{then} \;\; t_1 \;\; \mathtt{else} \;\; t_2 & \succ^* & t_2
\end{array}
$$

Booleans (2/3)

Objection:

Booleans (2/3)

Objection: We can do the same in the untyped λ-calculus!

Booleans (2/3)

Objection: We can do the same in the untyped λ-calculus!

$$\begin{array}{lcl} \mathsf{true} & \equiv & \lambda x, y\,.\,x \\ \mathsf{false} & \equiv & \lambda x, y\,.\,y \end{array}$$

$$\texttt{if}\ \ u\ \ \texttt{then}\ \ t_1\ \ \texttt{else}\ \ t_2 \quad \equiv \quad u\ t_1\ t_2$$

Booleans (2/3)

Objection: We can do the same in the untyped λ-calculus!

$$\left.\begin{array}{l} \mathsf{true} \equiv \lambda x, y\,.\,x \\ \mathsf{false} \equiv \lambda x, y\,.\,y \\ \\ \texttt{if}\ u\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \equiv u\ t_1\ t_2 \end{array}\right\} \text{Same reduction rules as before}$$

Booleans (2/3)

Objection: We can do the same in the untyped λ-calculus!

$$\left.\begin{array}{l} \mathsf{true} \equiv \lambda x, y\,.\,x \\ \mathsf{false} \equiv \lambda x, y\,.\,y \\ \\ \texttt{if}\ u\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \equiv u\ t_1\ t_2 \end{array}\right\} \text{Same reduction rules as before}$$

But nothing prevents the following computation:

$$\texttt{if}\ \underbrace{\lambda x\,.\,x}_{\text{bad bool}}\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \equiv (\lambda x\,.\,x)\ t_1\ t_2 \succ \underbrace{t_1\,t_2}_{\text{meaningless result}}$$

Booleans (2/3)

Objection: We can do the same in the untyped λ-calculus!

$$\left.\begin{array}{l} \mathsf{true} \equiv \lambda x, y\,.\,x \\ \mathsf{false} \equiv \lambda x, y\,.\,y \\ \\ \texttt{if}\ u\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \equiv u\ t_1\ t_2 \end{array}\right\} \text{Same reduction rules as before}$$

But nothing prevents the following computation:

$$\texttt{if}\ \underbrace{\lambda x\,.\,x}_{\text{bad bool}}\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \equiv (\lambda x\,.\,x)\ t_1\ t_2 \succ \underbrace{t_1\,t_2}_{\text{meaningless result}}$$

Question: Does the type discipline of system F avoid this?

Booleans (3/3)

Principle (that should be satisfied by any functional programming language)

Booleans (3/3)

Principle (that should be satisfied by any functional programming language)

When a program P of type A evaluates to a value v, then v has one of the canonical forms expected by the type A.

Booleans (3/3)

Principle (that should be satisfied by any functional programming language)

When a program P of type A evaluates to a value v, then v has one of the canonical forms expected by the type A.

In ML/Haskell, a value produced by a program of type `Bool` will always be `true` or `false` (i.e. the canonical forms of type `bool`).

Booleans (3/3)

Principle (that should be satisfied by any functional programming language)

When a program P of type A evaluates to a value v, then v has one of the canonical forms expected by the type A.

In ML/Haskell, a value produced by a program of type `Bool` will always be `true` or `false` (i.e. the canonical forms of type `bool`).

In system F: Subject-reduction ensures that the normal form of a term of type Bool is a term of type Bool.

Booleans (3/3)

Principle (that should be satisfied by any functional programming language)

When a program P of type A evaluates to a value v, then v has one of the canonical forms expected by the type A.

In ML/Haskell, a value produced by a program of type `Bool` will always be `true` or `false` (i.e. the canonical forms of type `bool`).

In system F: Subject-reduction ensures that the normal form of a term of type Bool is a term of type Bool.

To conclude, it suffices to check that in system F:

Lemma (Canonical forms of type bool)

Booleans (3/3)

Principle (that should be satisfied by any functional programming language)

When a program P of type A evaluates to a value v, then v has one of the canonical forms expected by the type A.

In ML/Haskell, a value produced by a program of type `Bool` will always be `true` or `false` (i.e. the canonical forms of type `bool`).

In system F: Subject-reduction ensures that the normal form of a term of type Bool is a term of type Bool.

To conclude, it suffices to check that in system F:

Lemma (Canonical forms of type bool)

The terms $\text{true} \equiv \Lambda\gamma\,.\,\lambda x, y : \gamma\,.\,x$ and $\text{false} \equiv \Lambda\gamma\,.\,\lambda x, y : \gamma\,.\,y$ are the only closed normal terms of type $\text{Bool} \equiv \forall\gamma\ (\gamma \to \gamma \to \gamma)$

Booleans (3/3)

Principle (that should be satisfied by any functional programming language)

When a program P of type A evaluates to a value v, then v has one of the canonical forms expected by the type A.

In ML/Haskell, a value produced by a program of type `Bool` will always be `true` or `false` (i.e. the canonical forms of type `bool`).

In system F: Subject-reduction ensures that the normal form of a term of type Bool is a term of type Bool.

To conclude, it suffices to check that in system F:

Lemma (Canonical forms of type bool)

The terms $\text{true} \equiv \Lambda\gamma\,.\,\lambda x, y : \gamma\,.\,x$ and $\text{false} \equiv \Lambda\gamma\,.\,\lambda x, y : \gamma\,.\,y$ are the only closed normal terms of type $\text{Bool} \equiv \forall\gamma\ (\gamma \to \gamma \to \gamma)$

Proof. Case analysis on the derivation.

Cartesian product

Cartesian product

Encoding of the cartesian product $A \times B$

$$
\begin{array}{lcll}
A \times B & \equiv & \forall\gamma\,((A \to B \to \gamma) \to \gamma) & \\
\langle t_1, t_2 \rangle & \equiv & \Lambda\gamma\,.\,\lambda f : A \to B \to \gamma\,.\,f\ t_1\ t_2 & \\
\mathsf{fst} & \equiv & \lambda p : A \times B\,.\,p\ A\ (\lambda x : A\,.\,\lambda y : B\,.\,x) & : \quad A \times B \to A \\
\mathsf{snd} & \equiv & \lambda p : A \times B\,.\,p\ B\ (\lambda x : A\,.\,\lambda y : B\,.\,y) & : \quad A \times B \to B
\end{array}
$$

Cartesian product

Encoding of the cartesian product $A \times B$

$$
\begin{array}{lcl}
A \times B & \equiv & \forall\gamma\,((A \to B \to \gamma) \to \gamma) \\
\langle t_1, t_2 \rangle & \equiv & \Lambda\gamma\,.\,\lambda f : A \to B \to \gamma\,.\,f\ t_1\ t_2 \\
\mathsf{fst} & \equiv & \lambda p : A \times B\,.\,p\ A\ (\lambda x : A\,.\,\lambda y : B\,.\,x) \quad : \quad A \times B \to A \\
\mathsf{snd} & \equiv & \lambda p : A \times B\,.\,p\ B\ (\lambda x : A\,.\,\lambda y : B\,.\,y) \quad : \quad A \times B \to B
\end{array}
$$

Correctness w.r.t. typing and reduction

$$
\frac{\Gamma \vdash t_1 : A \qquad \Gamma \vdash t_2 : B}{\Gamma \vdash \langle t_1, t_2 \rangle : A \times B}
$$

$$
\begin{array}{lcl}
\mathsf{fst}\ \langle t_1, t_2 \rangle & \succ^* & t_1 \\
\mathsf{snd}\ \langle t_1, t_2 \rangle & \succ^* & t_2
\end{array}
$$

Cartesian product

Encoding of the cartesian product $A \times B$

$$
\begin{array}{lcl}
A \times B & \equiv & \forall\gamma\,((A{\to}B{\to}\gamma) \to \gamma) \\
\langle t_1, t_2\rangle & \equiv & \Lambda\gamma\,.\,\lambda f : A \to B \to \gamma\,.\,f\ t_1\ t_2 \\
\mathsf{fst} & \equiv & \lambda p : A \times B\,.\,p\ A\ (\lambda x : A\,.\,\lambda y : B\,.\,x) \quad : \quad A \times B \to A \\
\mathsf{snd} & \equiv & \lambda p : A \times B\,.\,p\ B\ (\lambda x : A\,.\,\lambda y : B\,.\,y) \quad : \quad A \times B \to B
\end{array}
$$

Correctness w.r.t. typing and reduction

$$
\frac{\Gamma \vdash t_1 : A \qquad \Gamma \vdash t_2 : B}{\Gamma \vdash \langle t_1, t_2\rangle : A \times B}
$$

$$
\begin{array}{lcl}
\mathsf{fst}\ \langle t_1, t_2\rangle & \succ^* & t_1 \\
\mathsf{snd}\ \langle t_1, t_2\rangle & \succ^* & t_2
\end{array}
$$

Lemma (Canonical forms of type $A \times B$)

The closed normal terms of type $A \times B$ are of the form $\langle t_1, t_2\rangle$, where t_1 and t_2 are closed normal terms of type A and B, respectively.

Disjoint union

Disjoint union

Encoding of the disjoint union $A + B$

$$A + B \quad \equiv \quad \forall\gamma\ ((A{\to}\gamma) \to (B{\to}\gamma) \to \gamma)$$

$$\begin{array}{lcllll} \mathsf{inl}(v) & \equiv & \Lambda\gamma\,.\,\lambda f : A \to \gamma\,.\,\lambda g : B \to \gamma\,.\,f\ v & : & A + B & (\text{with } v : A) \\ \mathsf{inr}(v) & \equiv & \Lambda\gamma\,.\,\lambda f : A \to \gamma\,.\,\lambda g : B \to \gamma\,.\,g\ v & : & A + B & (\text{with } v : B) \end{array}$$

$$\mathtt{case}_C\ u\ \mathtt{of}\ \ \mathsf{inl}(x) \mapsto t_1\ \ |\ \ \mathsf{inr}(y) \mapsto t_2 \quad \equiv \quad u\ C\ (\lambda x : A\,.\,t_1)\ (\lambda y : B\,.\,t_2)$$

Disjoint union

Encoding of the disjoint union $A + B$

$$A + B \quad \equiv \quad \forall\gamma\ ((A{\to}\gamma) \to (B{\to}\gamma) \to \gamma)$$

$$\begin{array}{lcllll} \mathsf{inl}(v) & \equiv & \Lambda\gamma\,.\,\lambda f : A \to \gamma\,.\,\lambda g : B \to \gamma\,.\,f\ v & : & A + B & (\text{with } v : A) \\ \mathsf{inr}(v) & \equiv & \Lambda\gamma\,.\,\lambda f : A \to \gamma\,.\,\lambda g : B \to \gamma\,.\,g\ v & : & A + B & (\text{with } v : B) \end{array}$$

$$\mathtt{case}_C\ u\ \mathtt{of}\ \ \mathsf{inl}(x) \mapsto t_1\ \ |\ \ \mathsf{inr}(y) \mapsto t_2 \quad \equiv \quad u\ C\ (\lambda x : A\,.\,t_1)\ (\lambda y : B\,.\,t_2)$$

Correctness w.r.t. typing and reduction

$$\frac{\Gamma \vdash u : A + B \qquad \Gamma,\ x : A \vdash t_1 : C \qquad \Gamma,\ y : B \vdash t_2 : C}{\Gamma \ \vdash\ \mathtt{case}_C\ u\ \mathtt{of}\ \ \mathsf{inl}(x) \mapsto t_1\ \ |\ \ \mathsf{inr}(y) \mapsto t_2\ \ :\ \ C}$$

$$\begin{array}{lllll} \mathtt{case}_C\ \mathsf{inl}(v)\ \mathtt{of}\ \ \mathsf{inl}(x) \mapsto t_1\ \ |\ \ \mathsf{inr}(y) \mapsto t_2 & \succ^* & t_1\{x := v\} \\ \mathtt{case}_C\ \mathsf{inr}(v)\ \mathtt{of}\ \ \mathsf{inl}(x) \mapsto t_1\ \ |\ \ \mathsf{inr}(y) \mapsto t_2 & \succ^* & t_2\{y := v\} \end{array}$$

Disjoint union

Encoding of the disjoint union $A + B$

$$A + B \quad \equiv \quad \forall\gamma\ ((A{\to}\gamma) \to (B{\to}\gamma) \to \gamma)$$

$$\begin{array}{lcllll} \mathsf{inl}(v) & \equiv & \Lambda\gamma\,.\,\lambda f : A \to \gamma\,.\,\lambda g : B \to \gamma\,.\,f\ v & : & A + B & (\text{with } v : A) \\ \mathsf{inr}(v) & \equiv & \Lambda\gamma\,.\,\lambda f : A \to \gamma\,.\,\lambda g : B \to \gamma\,.\,g\ v & : & A + B & (\text{with } v : B) \end{array}$$

$$\mathtt{case}_C\ u\ \mathtt{of}\ \ \mathsf{inl}(x) \mapsto t_1\ \ |\ \ \mathsf{inr}(y) \mapsto t_2 \quad \equiv \quad u\ C\ (\lambda x : A\,.\,t_1)\ (\lambda y : B\,.\,t_2)$$

Correctness w.r.t. typing and reduction

$$\frac{\Gamma \vdash u : A + B \qquad \Gamma,\ x : A \vdash t_1 : C \qquad \Gamma,\ y : B \vdash t_2 : C}{\Gamma \ \vdash\ \mathtt{case}_C\ u\ \mathtt{of}\ \ \mathsf{inl}(x) \mapsto t_1\ \ |\ \ \mathsf{inr}(y) \mapsto t_2\ \ :\ \ C}$$

$$\begin{array}{lllll} \mathtt{case}_C\ \mathsf{inl}(v)\ \mathtt{of}\ \ \mathsf{inl}(x) \mapsto t_1\ \ |\ \ \mathsf{inr}(y) \mapsto t_2 & \succ^* & t_1\{x := v\} \\ \mathtt{case}_C\ \mathsf{inr}(v)\ \mathtt{of}\ \ \mathsf{inl}(x) \mapsto t_1\ \ |\ \ \mathsf{inr}(y) \mapsto t_2 & \succ^* & t_2\{y := v\} \end{array}$$

+ Canonical forms of type $A + B$ (works as expected)

Finite types

Finite types

Encoding of Fin_n $(n \geq 0)$

$$\mathsf{Fin}_n \equiv \forall\gamma\,(\underbrace{\gamma \to \cdots \to \gamma}_{n \text{ times}} \to \gamma)$$

$$\mathsf{e}_i \equiv \Lambda\gamma\,.\,\lambda x_1 : \gamma \ldots \lambda x_n : \gamma\,.\,x_i \;:\; \mathsf{Fin}_n \qquad (1 \leq i \leq n)$$

Finite types

Encoding of Fin_n $(n \geq 0)$

$$\mathsf{Fin}_n \equiv \forall\gamma\,(\underbrace{\gamma \to \cdots \to \gamma}_{n \text{ times}} \to \gamma)$$

$$\mathbf{e}_i \equiv \Lambda\gamma\,.\,\lambda x_1 : \gamma \ldots \lambda x_n : \gamma\,.\,x_i \;:\; \mathsf{Fin}_n \qquad (1 \leq i \leq n)$$

Again, $\mathbf{e}_1, \ldots, \mathbf{e}_n$ are the only closed normal terms of type Fin_n.

Finite types

Encoding of Fin_n $(n \geq 0)$

$$\mathsf{Fin}_n \equiv \forall\gamma\,(\underbrace{\gamma \to \cdots \to \gamma}_{n \text{ times}} \to \gamma)$$

$$\mathbf{e}_i \equiv \Lambda\gamma\,.\,\lambda x_1 : \gamma \ldots \lambda x_n : \gamma\,.\,x_i \;:\; \mathsf{Fin}_n \qquad (1 \leq i \leq n)$$

Again, $\mathbf{e}_1, \ldots, \mathbf{e}_n$ are the only closed normal terms of type Fin_n.

In particular:

$$\mathsf{Fin}_2 \equiv \forall\gamma\,(\gamma \to \gamma \to \gamma) \equiv \mathsf{Bool} \qquad \text{(type of booleans)}$$

Finite types

Encoding of Fin_n $(n \geq 0)$

$$\mathsf{Fin}_n \equiv \forall\gamma\,(\underbrace{\gamma \to \cdots \to \gamma}_{n \text{ times}} \to \gamma)$$

$$\mathbf{e}_i \equiv \Lambda\gamma\,.\,\lambda x_1 : \gamma \ldots \lambda x_n : \gamma\,.\,x_i \;:\; \mathsf{Fin}_n \qquad (1 \leq i \leq n)$$

Again, $\mathbf{e}_1, \ldots, \mathbf{e}_n$ are the only closed normal terms of type Fin_n.

In particular:

$$\mathsf{Fin}_2 \equiv \forall\gamma\,(\gamma \to \gamma \to \gamma) \equiv \mathsf{Bool} \qquad \text{(type of booleans)}$$

$$\mathsf{Fin}_1 \equiv \forall\gamma\,(\gamma \to \gamma) \equiv \mathsf{Unit} \qquad \text{(unit data-type)}$$

Finite types

Encoding of Fin_n $(n \geq 0)$

$$\mathsf{Fin}_n \equiv \forall\gamma\,(\underbrace{\gamma \to \cdots \to \gamma}_{n \text{ times}} \to \gamma)$$

$$\mathbf{e}_i \equiv \Lambda\gamma\,.\,\lambda x_1 : \gamma \ldots \lambda x_n : \gamma\,.\,x_i \;:\; \mathsf{Fin}_n \qquad (1 \leq i \leq n)$$

Again, $\mathbf{e}_1, \ldots, \mathbf{e}_n$ are the only closed normal terms of type Fin_n.

In particular:

$$\mathsf{Fin}_2 \equiv \forall\gamma\,(\gamma \to \gamma \to \gamma) \equiv \mathsf{Bool} \quad \text{(type of booleans)}$$

$$\mathsf{Fin}_1 \equiv \forall\gamma\,(\gamma \to \gamma) \equiv \mathsf{Unit} \quad \text{(unit data-type)}$$

$$\mathsf{Fin}_0 \equiv \forall\gamma\,\gamma \equiv \bot \quad \text{(empty data-type)}$$

Finite types

Encoding of Fin_n $(n \geq 0)$

$$\mathsf{Fin}_n \equiv \forall\gamma\,(\underbrace{\gamma \to \cdots \to \gamma}_{n \text{ times}} \to \gamma)$$

$$\mathbf{e}_i \equiv \Lambda\gamma\,.\,\lambda x_1 : \gamma \ldots \lambda x_n : \gamma\,.\,x_i \;:\; \mathsf{Fin}_n \qquad (1 \leq i \leq n)$$

Again, $\mathbf{e}_1, \ldots, \mathbf{e}_n$ are the only closed normal terms of type Fin_n.

In particular:

$$\mathsf{Fin}_2 \equiv \forall\gamma\,(\gamma \to \gamma \to \gamma) \equiv \mathsf{Bool} \quad \text{(type of booleans)}$$

$$\mathsf{Fin}_1 \equiv \forall\gamma\,(\gamma \to \gamma) \equiv \mathsf{Unit} \quad \text{(unit data-type)}$$

$$\mathsf{Fin}_0 \equiv \forall\gamma\,\gamma \equiv \bot \quad \text{(empty data-type)}$$

(Notice that there is no closed normal term of type $\bot$.)

Natural numbers

Natural numbers

Encoding of the type of Church numerals

$$\mathsf{Nat} \equiv \forall\gamma\,(\gamma \to (\gamma \to \gamma) \to \gamma)$$

Natural numbers

Encoding of the type of Church numerals

$$
\begin{array}{lcll}
\mathsf{Nat} & \equiv & \forall\gamma\,(\gamma \to (\gamma \to \gamma) \to \gamma) & \\
 & & & \\
\overline{0} & \equiv & \Lambda\gamma\,.\,\lambda x:\gamma\,.\,\lambda f:\gamma\to\gamma\,.\,x & \\
\overline{1} & \equiv & \Lambda\gamma\,.\,\lambda x:\gamma\,.\,\lambda f:\gamma\to\gamma\,.\,f\ x & \\
\overline{2} & \equiv & \Lambda\gamma\,.\,\lambda x:\gamma\,.\,\lambda f:\gamma\to\gamma\,.\,f\ (f\ x) & \\
 & \vdots & & \\
\overline{n} & \equiv & \Lambda\gamma\,.\,\lambda x:\gamma\,.\,\lambda f:\gamma\to\gamma\,.\,\underbrace{f(\cdots(f}_{n\ \text{times}}\,x)\cdots) & : \quad \mathsf{Nat} \\
 & \vdots & &
\end{array}
$$

Natural numbers

Encoding of the type of Church numerals

$$
\begin{array}{lcll}
\mathsf{Nat} & \equiv & \forall\gamma\,(\gamma \to (\gamma \to \gamma) \to \gamma) & \\
 & & & \\
\overline{0} & \equiv & \Lambda\gamma\,.\,\lambda x:\gamma\,.\,\lambda f:\gamma{\to}\gamma\,.\,x & \\
\overline{1} & \equiv & \Lambda\gamma\,.\,\lambda x:\gamma\,.\,\lambda f:\gamma{\to}\gamma\,.\,f\ x & \\
\overline{2} & \equiv & \Lambda\gamma\,.\,\lambda x:\gamma\,.\,\lambda f:\gamma{\to}\gamma\,.\,f\ (f\ x) & \\
 & \vdots & & \\
\overline{n} & \equiv & \Lambda\gamma\,.\,\lambda x:\gamma\,.\,\lambda f:\gamma{\to}\gamma\,.\,\underbrace{f(\cdots(f}_{n\text{ times}}\,x)\cdots) & : \quad \mathsf{Nat} \\
 & \vdots & &
\end{array}
$$

Lemma (Canonical forms of type Nat)

The terms $\overline{0}, \overline{1}, \overline{2}, \ldots$ are the only closed normal terms of type Nat.

Computing with natural numbers (1/2)

Computing with natural numbers (1/2)

Intuition: Church numeral $\overline{n}$ acts as an iterator:

$$\overline{n}\ A\ f\ x \quad \succ^* \quad \underbrace{f\ (\cdots(f}_{n}\ x)\cdots) \qquad (f : A \to A, \quad x : A)$$

Computing with natural numbers (1/2)

Intuition: Church numeral $\overline{n}$ acts as an iterator:

$$\overline{n}\ A\ f\ x \quad \succ^* \quad \underbrace{f\ (\cdots (f}_{n}\ x)\cdots) \qquad (f : A \to A, \quad x : A)$$

- Successor

$$\mathsf{succ} \quad \equiv \quad \lambda n : \mathsf{Nat}\,.\,\Lambda\gamma\,.\,\lambda x : \gamma\,.\,\lambda f : \gamma \to \gamma\,.\,f\ (n\ \gamma\ x\ f)$$

Computing with natural numbers (1/2)

Intuition: Church numeral $\overline{n}$ acts as an iterator:

$$\overline{n}\ A\ f\ x \quad \succ^* \quad \underbrace{f\ (\cdots (f}_{n}\ x)\cdots) \qquad (f : A \to A, \quad x : A)$$

- Successor

$$\mathsf{succ} \quad \equiv \quad \lambda n : \mathsf{Nat}\,.\,\Lambda\gamma\,.\,\lambda x : \gamma\,.\,\lambda f : \gamma \to \gamma\,.\,f\ (n\ \gamma\ x\ f)$$

- Addition

$$\mathsf{plus} \quad \equiv \quad \lambda n, m : \mathsf{Nat}\,.\,\Lambda\gamma\,.\,\lambda x : \gamma\,.\,\lambda f : \gamma \to \gamma\,.\,m\ \gamma\ (n\ \gamma\ x\ f)\ f$$

Computing with natural numbers (1/2)

Intuition: Church numeral $\overline{n}$ acts as an iterator:

$$\overline{n}\ A\ f\ x \quad \succ^* \quad \underbrace{f\ (\cdots(f}_{n}\ x)\cdots) \qquad (f : A \to A, \quad x : A)$$

- Successor

$$\mathsf{succ} \quad \equiv \quad \lambda n : \mathsf{Nat}\,.\,\Lambda\gamma\,.\,\lambda x : \gamma\,.\,\lambda f : \gamma{\to}\gamma\,.\,f\ (n\ \gamma\ x\ f)$$

- Addition

$$\begin{array}{lcl} \mathsf{plus} & \equiv & \lambda n, m : \mathsf{Nat}\,.\,\Lambda\gamma\,.\,\lambda x : \gamma\,.\,\lambda f : \gamma{\to}\gamma\,.\,m\ \gamma\ (n\ \gamma\ x\ f)\ f \\ \mathsf{plus}' & \equiv & \lambda n, m : \mathsf{Nat}\,.\,m\ \mathsf{Nat}\ n\ \mathsf{succ} \end{array}$$

Computing with natural numbers (1/2)

Intuition: Church numeral $\overline{n}$ acts as an iterator:

$$\overline{n}\ A\ f\ x \quad \succ^* \quad \underbrace{f\ (\cdots (f}_{n}\ x) \cdots) \qquad (f : A \to A, \quad x : A)$$

- Successor

$$\mathsf{succ} \quad \equiv \quad \lambda n : \mathsf{Nat}\,.\,\Lambda\gamma\,.\,\lambda x : \gamma\,.\,\lambda f : \gamma \to \gamma\,.\,f\ (n\ \gamma\ x\ f)$$

- Addition

$$\begin{array}{lcl} \mathsf{plus} & \equiv & \lambda n, m : \mathsf{Nat}\,.\,\Lambda\gamma\,.\,\lambda x : \gamma\,.\,\lambda f : \gamma \to \gamma\,.\,m\ \gamma\ (n\ \gamma\ x\ f)\ f \\ \mathsf{plus}' & \equiv & \lambda n, m : \mathsf{Nat}\,.\,m\ \mathsf{Nat}\ n\ \mathsf{succ} \end{array}$$

- Multiplication

$$\mathsf{mult} \quad \equiv \quad \lambda n, m : \mathsf{Nat}\,.\,\Lambda\gamma\,.\,\lambda x : \gamma\,.\,\lambda f : \gamma \to \gamma\,.\,n\ \gamma\ x\ (\lambda y : \gamma\,.\,m\ \gamma\ y\ f)$$

Computing with natural numbers (1/2)

Intuition: Church numeral $\overline{n}$ acts as an iterator:

$$\overline{n}\ A\ f\ x \quad \succ^* \quad \underbrace{f\ (\cdots (f}_{n}\ x) \cdots) \qquad (f : A \to A, \quad x : A)$$

- Successor

$$\mathsf{succ} \quad \equiv \quad \lambda n : \mathsf{Nat}\,.\,\Lambda\gamma\,.\,\lambda x : \gamma\,.\,\lambda f : \gamma \to \gamma\,.\,f\ (n\ \gamma\ x\ f)$$

- Addition

$$\begin{array}{lcl} \mathsf{plus} & \equiv & \lambda n, m : \mathsf{Nat}\,.\,\Lambda\gamma\,.\,\lambda x : \gamma\,.\,\lambda f : \gamma \to \gamma\,.\,m\ \gamma\ (n\ \gamma\ x\ f)\ f \\ \mathsf{plus}' & \equiv & \lambda n, m : \mathsf{Nat}\,.\,m\ \mathsf{Nat}\ n\ \mathsf{succ} \end{array}$$

- Multiplication

$$\begin{array}{lcl} \mathsf{mult} & \equiv & \lambda n, m : \mathsf{Nat}\,.\,\Lambda\gamma\,.\,\lambda x : \gamma\,.\,\lambda f : \gamma \to \gamma\,.\,n\ \gamma\ x\ (\lambda y : \gamma\,.\,m\ \gamma\ y\ f) \\ \mathsf{mult}' & \equiv & \lambda n, m : \mathsf{Nat}\,.\,n\ \mathsf{Nat}\ \overline{0}\ (\mathsf{plus}\ m) \end{array}$$

Computing with natural numbers (2/2)

Computing with natural numbers (2/2)

- Predecessor function $\quad \mathsf{pred} \ : \ \mathsf{Nat} \longrightarrow \mathsf{Nat}$

Computing with natural numbers (2/2)

- Predecessor function $\quad \mathsf{pred} \ : \ \mathsf{Nat} \longrightarrow \mathsf{Nat}$

$$\begin{array}{lcl} \mathsf{pred}\ \overline{0} & \simeq & \overline{0} \\ \mathsf{pred}\ (\overline{n+1}) & \simeq & \overline{n} \end{array}$$

Computing with natural numbers (2/2)

- Predecessor function $\quad \mathsf{pred} \;:\; \mathsf{Nat} \longrightarrow \mathsf{Nat}$

$$\begin{array}{lcl} \mathsf{pred}\ \overline{0} & \simeq & \overline{0} \\ \mathsf{pred}\ (\overline{n+1}) & \simeq & \overline{n} \end{array}$$

$$\begin{array}{lcll} \mathsf{fst} & \equiv & \lambda p : \mathsf{Nat}\times\mathsf{Nat}\,.\, p\ \mathsf{Nat}\ (\lambda x, y : \mathsf{Nat}\,.\, x) & : \quad \mathsf{Nat}\times\mathsf{Nat} \longrightarrow \mathsf{Nat} \\ \mathsf{snd} & \equiv & \lambda p : \mathsf{Nat}\times\mathsf{Nat}\,.\, p\ \mathsf{Nat}\ (\lambda x, y : \mathsf{Nat}\,.\, y) & : \quad \mathsf{Nat}\times\mathsf{Nat} \longrightarrow \mathsf{Nat} \\ \mathsf{step} & \equiv & \lambda p : \mathsf{Nat}\times\mathsf{Nat}\,.\, \langle \mathsf{snd}\ p,\ \mathsf{succ}\ (\mathsf{snd}\ p)\rangle & : \quad \mathsf{Nat}\times\mathsf{Nat} \longrightarrow \mathsf{Nat}\times\mathsf{Nat} \\ \mathsf{pred} & \equiv & \lambda n : \mathsf{Nat}\,.\, \mathsf{fst}\ (n\ (\mathsf{Nat}\times\mathsf{Nat})\ \langle \overline{0}, \overline{0}\rangle\ \mathsf{step}) & : \quad \mathsf{Nat} \longrightarrow \mathsf{Nat} \end{array}$$

Computing with natural numbers (2/2)

- Predecessor function $\quad \mathsf{pred} \;:\; \mathsf{Nat} \longrightarrow \mathsf{Nat}$

$$\begin{array}{lcl} \mathsf{pred}\ \overline{0} & \simeq & \overline{0} \\ \mathsf{pred}\ (\overline{n+1}) & \simeq & \overline{n} \end{array}$$

$$\begin{array}{lcll} \mathsf{fst} & \equiv & \lambda p : \mathsf{Nat}\times\mathsf{Nat}\,.\, p\ \mathsf{Nat}\ (\lambda x, y : \mathsf{Nat}\,.\, x) & : \quad \mathsf{Nat}\times\mathsf{Nat} \longrightarrow \mathsf{Nat} \\ \mathsf{snd} & \equiv & \lambda p : \mathsf{Nat}\times\mathsf{Nat}\,.\, p\ \mathsf{Nat}\ (\lambda x, y : \mathsf{Nat}\,.\, y) & : \quad \mathsf{Nat}\times\mathsf{Nat} \longrightarrow \mathsf{Nat} \\ \mathsf{step} & \equiv & \lambda p : \mathsf{Nat}\times\mathsf{Nat}\,.\, \langle \mathsf{snd}\ p,\ \mathsf{succ}\ (\mathsf{snd}\ p)\rangle & : \quad \mathsf{Nat}\times\mathsf{Nat} \longrightarrow \mathsf{Nat}\times\mathsf{Nat} \\ \mathsf{pred} & \equiv & \lambda n : \mathsf{Nat}\,.\, \mathsf{fst}\ (n\ (\mathsf{Nat}\times\mathsf{Nat})\ \langle \overline{0}, \overline{0}\rangle\ \mathsf{step}) & : \quad \mathsf{Nat} \longrightarrow \mathsf{Nat} \end{array}$$

- Ackerman function $\quad \mathsf{ack} \;:\; \mathsf{Nat} \longrightarrow \mathsf{Nat} \longrightarrow \mathsf{Nat}$

Computing with natural numbers (2/2)

- Predecessor function $\quad \mathsf{pred} \;:\; \mathsf{Nat} \longrightarrow \mathsf{Nat}$

$$\begin{array}{lcl} \mathsf{pred}\ \overline{0} & \simeq & \overline{0} \\ \mathsf{pred}\ (\overline{n+1}) & \simeq & \overline{n} \end{array}$$

$$\begin{array}{lcll} \mathsf{fst} & \equiv & \lambda p : \mathsf{Nat}\times\mathsf{Nat} \,.\, p\ \mathsf{Nat}\ (\lambda x, y : \mathsf{Nat} \,.\, x) & : \quad \mathsf{Nat}\times\mathsf{Nat} \longrightarrow \mathsf{Nat} \\ \mathsf{snd} & \equiv & \lambda p : \mathsf{Nat}\times\mathsf{Nat} \,.\, p\ \mathsf{Nat}\ (\lambda x, y : \mathsf{Nat} \,.\, y) & : \quad \mathsf{Nat}\times\mathsf{Nat} \longrightarrow \mathsf{Nat} \\ \mathsf{step} & \equiv & \lambda p : \mathsf{Nat}\times\mathsf{Nat} \,.\, \langle \mathsf{snd}\ p,\ \mathsf{succ}\ (\mathsf{snd}\ p) \rangle & : \quad \mathsf{Nat}\times\mathsf{Nat} \longrightarrow \mathsf{Nat}\times\mathsf{Nat} \\ \mathsf{pred} & \equiv & \lambda n : \mathsf{Nat} \,.\, \mathsf{fst}\ (n\ (\mathsf{Nat}\times\mathsf{Nat})\ \langle \overline{0}, \overline{0} \rangle\ \mathsf{step}) & : \quad \mathsf{Nat} \longrightarrow \mathsf{Nat} \end{array}$$

- Ackerman function $\quad \mathsf{ack} \;:\; \mathsf{Nat} \longrightarrow \mathsf{Nat} \longrightarrow \mathsf{Nat}$

$$\begin{array}{llll} \mathsf{ack} & \overline{0} & \overline{m} & \simeq \quad \overline{m+1} \\ \mathsf{ack} & (\overline{n+1}) & \overline{0} & \simeq \quad \mathsf{ack}\ \overline{n}\ \overline{1} \\ \mathsf{ack} & (\overline{n+1}) & (\overline{m+1}) & \simeq \quad \mathsf{ack}\ \overline{n}\ \left(\mathsf{ack}\ (\overline{n+1})\ \overline{m}\right) \end{array}$$

Computing with natural numbers (2/2)

- Predecessor function $\quad \mathsf{pred} \;:\; \mathsf{Nat} \longrightarrow \mathsf{Nat}$

$$\begin{array}{lcl} \mathsf{pred}\ \overline{0} & \simeq & \overline{0} \\ \mathsf{pred}\ (\overline{n+1}) & \simeq & \overline{n} \end{array}$$

$$\begin{array}{lcll} \mathsf{fst} & \equiv & \lambda p : \mathsf{Nat}\times\mathsf{Nat} \,.\, p\ \mathsf{Nat}\ (\lambda x, y : \mathsf{Nat} \,.\, x) & : \quad \mathsf{Nat}\times\mathsf{Nat} \longrightarrow \mathsf{Nat} \\ \mathsf{snd} & \equiv & \lambda p : \mathsf{Nat}\times\mathsf{Nat} \,.\, p\ \mathsf{Nat}\ (\lambda x, y : \mathsf{Nat} \,.\, y) & : \quad \mathsf{Nat}\times\mathsf{Nat} \longrightarrow \mathsf{Nat} \\ \mathsf{step} & \equiv & \lambda p : \mathsf{Nat}\times\mathsf{Nat} \,.\, \langle \mathsf{snd}\ p,\ \mathsf{succ}\ (\mathsf{snd}\ p) \rangle & : \quad \mathsf{Nat}\times\mathsf{Nat} \longrightarrow \mathsf{Nat}\times\mathsf{Nat} \\ \mathsf{pred} & \equiv & \lambda n : \mathsf{Nat} \,.\, \mathsf{fst}\ (n\ (\mathsf{Nat}\times\mathsf{Nat})\ \langle \overline{0}, \overline{0} \rangle\ \mathsf{step}) & : \quad \mathsf{Nat} \longrightarrow \mathsf{Nat} \end{array}$$

- Ackerman function $\quad \mathsf{ack} \;:\; \mathsf{Nat} \longrightarrow \mathsf{Nat} \longrightarrow \mathsf{Nat}$

$$\begin{array}{llll} \mathsf{ack} & \overline{0} & \overline{m} & \simeq \quad \overline{m+1} \\ \mathsf{ack} & (\overline{n+1}) & \overline{0} & \simeq \quad \mathsf{ack}\ \overline{n}\ \overline{1} \\ \mathsf{ack} & (\overline{n+1}) & (\overline{m+1}) & \simeq \quad \mathsf{ack}\ \overline{n}\ \left(\mathsf{ack}\ (\overline{n+1})\ \overline{m}\right) \end{array}$$

$$\begin{array}{lcll} \mathsf{down} & \equiv & \lambda f : (\mathsf{Nat}\longrightarrow\mathsf{Nat}) \,.\, \lambda p : \mathsf{Nat} \,.\, p\ \mathsf{Nat}\ (f\ \overline{1})\ f & : \quad (\mathsf{Nat}\longrightarrow\mathsf{Nat}) \longrightarrow (\mathsf{Nat}\longrightarrow\mathsf{Nat}) \\ \mathsf{ack} & \equiv & \lambda n, m : \mathsf{Nat} \,.\, n\ (\mathsf{Nat}\longrightarrow\mathsf{Nat})\ \mathsf{succ}\ \mathsf{down}\ m & : \quad \mathsf{Nat} \longrightarrow \mathsf{Nat} \longrightarrow \mathsf{Nat} \end{array}$$

Computing with natural numbers (2/2)

- Predecessor function $\quad \mathsf{pred} \;:\; \mathsf{Nat} \longrightarrow \mathsf{Nat}$

$$\begin{array}{lcl} \mathsf{pred}\ \overline{0} & \simeq & \overline{0} \\ \mathsf{pred}\ (\overline{n+1}) & \simeq & \overline{n} \end{array}$$

$$\begin{array}{lcll} \mathsf{fst} & \equiv & \lambda p : \mathsf{Nat}\times\mathsf{Nat} \,.\, p\ \mathsf{Nat}\ (\lambda x, y : \mathsf{Nat} \,.\, x) & : \quad \mathsf{Nat}\times\mathsf{Nat} \longrightarrow \mathsf{Nat} \\ \mathsf{snd} & \equiv & \lambda p : \mathsf{Nat}\times\mathsf{Nat} \,.\, p\ \mathsf{Nat}\ (\lambda x, y : \mathsf{Nat} \,.\, y) & : \quad \mathsf{Nat}\times\mathsf{Nat} \longrightarrow \mathsf{Nat} \\ \mathsf{step} & \equiv & \lambda p : \mathsf{Nat}\times\mathsf{Nat} \,.\, \langle \mathsf{snd}\ p,\ \mathsf{succ}\ (\mathsf{snd}\ p)\rangle & : \quad \mathsf{Nat}\times\mathsf{Nat} \longrightarrow \mathsf{Nat}\times\mathsf{Nat} \\ \mathsf{pred} & \equiv & \lambda n : \mathsf{Nat} \,.\, \mathsf{fst}\ (n\ (\mathsf{Nat}\times\mathsf{Nat})\ \langle \overline{0}, \overline{0}\rangle\ \mathsf{step}) & : \quad \mathsf{Nat} \longrightarrow \mathsf{Nat} \end{array}$$

- Ackerman function $\quad \mathsf{ack} \;:\; \mathsf{Nat} \longrightarrow \mathsf{Nat} \longrightarrow \mathsf{Nat}$

$$\begin{array}{llll} \mathsf{ack} & \overline{0} & \overline{m} & \simeq \quad \overline{m+1} \\ \mathsf{ack} & (\overline{n+1}) & \overline{0} & \simeq \quad \mathsf{ack}\ \overline{n}\ \overline{1} \\ \mathsf{ack} & (\overline{n+1}) & (\overline{m+1}) & \simeq \quad \mathsf{ack}\ \overline{n}\ \left(\mathsf{ack}\ (\overline{n+1})\ \overline{m}\right) \end{array}$$

$$\begin{array}{lcll} \mathsf{down} & \equiv & \lambda f : (\mathsf{Nat}\longrightarrow\mathsf{Nat}) \,.\, \lambda p : \mathsf{Nat} \,.\, p\ \mathsf{Nat}\ (f\ \overline{1})\ f & : \quad (\mathsf{Nat}\longrightarrow\mathsf{Nat}) \longrightarrow (\mathsf{Nat}\longrightarrow\mathsf{Nat}) \\ \mathsf{ack} & \equiv & \lambda n, m : \mathsf{Nat} \,.\, n\ (\mathsf{Nat}\longrightarrow\mathsf{Nat})\ \mathsf{succ}\ \mathsf{down}\ m & : \quad \mathsf{Nat} \longrightarrow \mathsf{Nat} \longrightarrow \mathsf{Nat} \end{array}$$

  - ▷ SN theorem guarantees that all well-typed computations terminate

Part III

System F: Curry-style presentation

System F polymorphism

System F polymorphism

ML/Haskell polymorphism

Types	A, B	$::=$	$\alpha \mid A \to B \mid \cdots$	(user datatypes)
Schemes	S	$::=$	$\forall \vec{\alpha}\ B$	

The type scheme $\forall \alpha\ B$ is defined after its particular instances $B\{\alpha := A\}$

System F polymorphism

ML/Haskell polymorphism

Types	A, B	$::=$	$\alpha \mid A \to B \mid \cdots$	(user datatypes)
Schemes	S	$::=$	$\forall \vec{\alpha}\ B$	

The type scheme $\forall \alpha\ B$ is defined after its particular instances $B\{\alpha := A\}$
$\Rightarrow$ Type system is predicative

System F polymorphism

ML/Haskell polymorphism

Types	A, B	$::=$	α	$\mid A \to B \mid$	$\cdots$	(user datatypes)
Schemes	S	$::=$	$\forall \vec{\alpha}\ B$			

The type scheme $\forall \alpha\ B$ is defined after its particular instances $B\{\alpha{:=}\ A\}$
$\Rightarrow$ Type system is predicative

System F polymorphism

Types	A, B	$::=$	α	$\mid A \to B \mid$	$\forall \alpha\ B$

The type $\forall \alpha\ B$ and its instances $B\{\alpha{:=}A\}$ are defined simultaneously

System F polymorphism

ML/Haskell polymorphism

$$\begin{array}{lrcl} \textit{Types} & A, B & ::= & \alpha \mid A \to B \mid \cdots \quad \text{(user datatypes)} \\ \textit{Schemes} & S & ::= & \forall \vec{\alpha}\ B \end{array}$$

The type scheme $\forall \alpha\ B$ is defined after its particular instances $B\{\alpha{:=}\ A\}$
$\Rightarrow$ Type system is predicative

System F polymorphism

$$\textit{Types} \qquad A, B \quad ::= \quad \alpha \mid A \to B \mid \forall \alpha\ B$$

The type $\forall \alpha\ B$ and its instances $B\{\alpha{:=}A\}$ are defined simultaneously

$$\forall \alpha\ (\alpha \to \alpha) \qquad \text{and} \qquad \forall \alpha\ (\alpha \to \alpha) \to \forall \alpha\ (\alpha \to \alpha)$$

System F polymorphism

ML/Haskell polymorphism

$$\begin{array}{lrcl} \textit{Types} & A, B & ::= & \alpha \mid A \to B \mid \cdots \quad \text{(user datatypes)} \\ \textit{Schemes} & S & ::= & \forall \vec{\alpha}\ B \end{array}$$

The type scheme $\forall \alpha\ B$ is defined after its particular instances $B\{\alpha{:=}\ A\}$
$\Rightarrow$ Type system is predicative

System F polymorphism

$$\textit{Types} \qquad A, B \quad ::= \quad \alpha \mid A \to B \mid \forall \alpha\ B$$

The type $\forall \alpha\ B$ and its instances $B\{\alpha{:=}A\}$ are defined simultaneously

$$\forall \alpha\ (\alpha \to \alpha) \qquad \text{and} \qquad \forall \alpha\ (\alpha \to \alpha) \to \forall \alpha\ (\alpha \to \alpha)$$

$\Rightarrow$ Type system is impredicative, or cyclic

Extracting pure λ-terms

Extracting pure λ-terms

In Church-style system F, polymorphism is explicit:

$$\mathrm{id} \equiv \Lambda\alpha\,.\,\lambda x : \alpha\,.\,x \qquad \text{and} \qquad \mathrm{id}\ \mathrm{Nat}\ 2$$

- Two kind of redexes $(\lambda x : A\,.\,t)u$ and $(\Lambda\alpha\,.\,t)A$

Extracting pure λ-terms

In Church-style system F, polymorphism is explicit:

$$\mathrm{id} \equiv \Lambda\alpha\,.\,\lambda x : \alpha\,.\,x \qquad \text{and} \qquad \mathrm{id}\ \mathrm{Nat}\ 2$$

- Two kind of redexes $\quad (\lambda x : A\,.\,t)u \quad$ and $\quad (\Lambda\alpha\,.\,t)A$

Idea: Remove type abstractions/applications/annotations

Extracting pure λ-terms

In Church-style system F, polymorphism is explicit:

$$\mathrm{id} \equiv \Lambda\alpha\,.\,\lambda x : \alpha\,.\,x \qquad \text{and} \qquad \mathrm{id}\ \mathrm{Nat}\ 2$$

- Two kind of redexes $\quad (\lambda x : A\,.\,t)u \quad$ and $\quad (\Lambda\alpha\,.\,t)A$

Idea: Remove type abstractions/applications/annotations

Erasing function $t \mapsto |t|$

$$\begin{array}{rcl} |x| & = & x \\ |\lambda x : A\,.\,t| & = & \lambda x\,.\,|t| \\ |tu| & = & |t||u| \end{array} \qquad \begin{array}{rcl} |\Lambda\alpha\,.\,t| & = & |t| \\ |tA| & = & |t| \end{array}$$

Extracting pure λ-terms

In Church-style system F, polymorphism is explicit:

$$\mathrm{id} \equiv \Lambda\alpha\,.\,\lambda x : \alpha\,.\,x \qquad \text{and} \qquad \mathrm{id}\ \mathrm{Nat}\ 2$$

- Two kind of redexes $\quad (\lambda x : A\,.\,t)u \quad$ and $\quad (\Lambda\alpha\,.\,t)A$

Idea: Remove type abstractions/applications/annotations

Erasing function $t \mapsto |t|$

$$\begin{aligned} |x| &= x \\ |\lambda x : A\,.\,t| &= \lambda x\,.\,|t| \qquad & |\Lambda\alpha\,.\,t| &= |t| \\ |tu| &= |t||u| \qquad & |tA| &= |t| \end{aligned}$$

- Target language is pure λ-calculus
- Second kind redexes are erased, first kind redexes are preserved

Extending the erasing function

Erased terms have a nice computational behaviour...

- Only one kind of redex, easy to execute (Krivine's machine)
- Irrelevant part of computation has been removed
- The essence of computation has been preserved (to be justified later)

Extending the erasing function

Erased terms have a nice computational behaviour...

- Only one kind of redex, easy to execute (Krivine's machine)
- Irrelevant part of computation has been removed
- The essence of computation has been preserved (to be justified later)

... but what is their status w.r.t. typing?

Extending the erasing function

Erased terms have a nice computational behaviour...

- Only one kind of redex, easy to execute (Krivine's machine)
- Irrelevant part of computation has been removed
- The essence of computation has been preserved (to be justified later)

... but what is their status w.r.t. typing?

The erasing function, defined on terms, can be extended to:

Extending the erasing function

Erased terms have a nice computational behaviour...

- Only one kind of redex, easy to execute (Krivine's machine)
- Irrelevant part of computation has been removed
- The essence of computation has been preserved (to be justified later)

... but what is their status w.r.t. typing?

The erasing function, defined on terms, can be extended to:

- The whole syntax

Extending the erasing function

Erased terms have a nice computational behaviour...

- Only one kind of redex, easy to execute (Krivine's machine)
- Irrelevant part of computation has been removed
- The essence of computation has been preserved (to be justified later)

... but what is their status w.r.t. typing?

The erasing function, defined on terms, can be extended to:

- The whole syntax
- The judgements

Extending the erasing function

Erased terms have a nice computational behaviour...

- Only one kind of redex, easy to execute (Krivine's machine)
- Irrelevant part of computation has been removed
- The essence of computation has been preserved (to be justified later)

... but what is their status w.r.t. typing?

The erasing function, defined on terms, can be extended to:

- The whole syntax
- The judgements
- The typing rules

Extending the erasing function

Erased terms have a nice computational behaviour...

- Only one kind of redex, easy to execute (Krivine's machine)
- Irrelevant part of computation has been removed
- The essence of computation has been preserved (to be justified later)

... but what is their status w.r.t. typing?

The erasing function, defined on terms, can be extended to:

- The whole syntax
- The judgements
- The typing rules
- The derivations

Extending the erasing function

Erased terms have a nice computational behaviour...

- Only one kind of redex, easy to execute (Krivine's machine)
- Irrelevant part of computation has been removed
- The essence of computation has been preserved (to be justified later)

... but what is their status w.r.t. typing?

The erasing function, defined on terms, can be extended to:

- The whole syntax
- The judgements
- The typing rules
- The derivations

$\Rightarrow$ Induces a new formalism: Curry-style system F

Church-style system F

Types	A, B	$::=$	$\alpha \mid A \to B \mid \forall \alpha\ B$
Terms	t, u	$::=$	$x \mid \lambda x : A\,.\,t \mid tu \mid \Lambda \alpha\,.\,t \mid tA$
Judgments	Γ	$::=$	$[] \mid \Gamma,\ x{:}A$

Reduction

$$(\lambda x : A\,.\,t)u \succ t\{x := u\}$$
$$(\Lambda \alpha\,.\,t)A \succ t\{\alpha := A\}$$

Church-style system F

Types	A, B	$::=$	$\alpha \mid A \to B \mid \forall \alpha\ B$
Terms	t, u	$::=$	$x \mid \lambda x : A\,.\,t \mid tu \mid \Lambda \alpha\,.\,t \mid tA$
Judgments	Γ	$::=$	$[] \mid \Gamma,\ x{:}A$

Reduction

$$(\lambda x : A\,.\,t)u \succ t\{x := u\}$$
$$(\Lambda \alpha\,.\,t)A \succ t\{\alpha := A\}$$

Curry-style system F [Leivant 83]

Types	A, B	$::=$	$\alpha \mid A \to B \mid \forall \alpha\ B$
Terms	t, u	$::=$	$x \mid \lambda x\,.\,t \mid tu$
Judgments	Γ	$::=$	$[] \mid \Gamma,\ x{:}A$

Reduction $\quad (\lambda x\,.\,t)u \succ t\{x := u\}$

Curry-style system F [Leivant 83]

Types	A, B	$::=$	$\alpha \mid A \to B \mid \forall \alpha\ B$
Terms	t, u	$::=$	$x \mid \lambda x\,.\,t \mid tu$
Judgments	Γ	$::=$	$[] \mid \Gamma,\ x{:}A$

Reduction $(\lambda x\,.\,t)u \succ t\{x := u\}$

Remarks:

- Types (and contexts) are unchanged
- Terms are now pure λ-terms
- Only one kind of redex

Church-style system F: typing rules

$$\frac{}{\Gamma \vdash x : A} \quad (x{:}A) \in \Gamma$$

$$\frac{\Gamma,\ x : A \vdash t : B}{\Gamma \vdash \lambda x : A \,.\, t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$$

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash \Lambda\alpha \,.\, t : \forall\alpha\ B} \quad \alpha \notin TV(\Gamma) \qquad \frac{\Gamma \vdash t : \forall\alpha\ B}{\Gamma \vdash tA : B\{\alpha := A\}}$$

Curry-style system F: typing rules

$$\frac{}{\Gamma \vdash x : A}\ (x{:}A)\in\Gamma$$

$$\frac{\Gamma,\ x : A \vdash t : B}{\Gamma \vdash \lambda x\,.\,t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$$

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash t : \forall\alpha\ B}\ \alpha\notin TV(\Gamma) \qquad \frac{\Gamma \vdash t : \forall\alpha\ B}{\Gamma \vdash t : B\{\alpha := A\}}$$

Curry-style system F: typing rules

$$\frac{}{\Gamma \vdash x : A}\ (x{:}A)\in\Gamma$$

$$\frac{\Gamma,\ x : A \vdash t : B}{\Gamma \vdash \lambda x\,.\,t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$$

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash t : \forall\alpha\ B}\ \alpha\notin TV(\Gamma) \qquad \frac{\Gamma \vdash t : \forall\alpha\ B}{\Gamma \vdash t : B\{\alpha := A\}}$$

$\Rightarrow$ Rules are no more syntax directed

Curry-style system F: properties

Things that do not change

Curry-style system F: properties

Things that do not change

- Substitutivity + β-subject reduction

Curry-style system F: properties

Things that do not change

- Substitutivity + β-subject reduction
- Strong normalisation (postponed)

Curry-style system F: properties

Things that do not change

- Substitutivity + β-subject reduction
- Strong normalisation (postponed)

Things that change

Curry-style system F: properties

Things that do not change

- Substitutivity + β-subject reduction
- Strong normalisation (postponed)

Things that change

- A term may have several types

$$\Delta \equiv \lambda x\,.\,x\;x$$

Curry-style system F: properties

Things that do not change

- Substitutivity + β-subject reduction
- Strong normalisation (postponed)

Things that change

- A term may have several types

$$\Delta \equiv \lambda x\,.\,x\ x \quad : \quad \forall\alpha\ (\alpha \to \alpha) \to \forall\alpha\ (\alpha \to \alpha)$$

Curry-style system F: properties

Things that do not change

- Substitutivity + β-subject reduction
- Strong normalisation (postponed)

Things that change

- A term may have several types

$$\begin{array}{rcll} \Delta \equiv \lambda x\,.\,x\;x & : & \forall\alpha\;(\alpha \to \alpha) \to \forall\alpha\;(\alpha \to \alpha) \\ & : & \forall\alpha\;\alpha \to \forall\alpha\;\alpha \end{array}$$

Curry-style system F: properties

Things that do not change

- Substitutivity + β-subject reduction
- Strong normalisation (postponed)

Things that change

- A term may have several types

$$\begin{array}{rcll} \Delta \equiv \lambda x\,.\,x\;x & : & \forall\alpha\;(\alpha \to \alpha) \to \forall\alpha\;(\alpha \to \alpha) \\ & : & \forall\alpha\;\alpha \to \forall\alpha\;\alpha \\ & : & \forall\alpha\;\alpha \to \forall\alpha\;(\alpha \to \alpha) \end{array}$$

Curry-style system F: properties

Things that do not change

- Substitutivity + β-subject reduction
- Strong normalisation (postponed)

Things that change

- A term may have several types

$$\begin{array}{lcll} \Delta \equiv \lambda x\,.\,x\;x & : & \forall\alpha\;(\alpha \to \alpha) \to \forall\alpha\;(\alpha \to \alpha) \\ & : & \forall\alpha\;\alpha \to \forall\alpha\;\alpha \\ & : & \forall\alpha\;\alpha \to \forall\alpha\;(\alpha \to \alpha) \\ & : & \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool} \end{array}$$

Curry-style system F: properties

Things that do not change

- Substitutivity + β-subject reduction
- Strong normalisation (postponed)

Things that change

- A term may have several types

$$\begin{array}{lcll} \Delta \equiv \lambda x\,.\,x\;x & : & \forall\alpha\;(\alpha \to \alpha) \to \forall\alpha\;(\alpha \to \alpha) & \\ & : & \forall\alpha\;\alpha \to \forall\alpha\;\alpha & \\ & : & \forall\alpha\;\alpha \to \forall\alpha\;(\alpha \to \alpha) & \\ & : & \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool} & \text{('or' function!)} \end{array}$$

Curry-style system F: properties

Things that do not change

- Substitutivity + β-subject reduction
- Strong normalisation (postponed)

Things that change

- A term may have several types

$$\begin{array}{rcll} \Delta \equiv \lambda x\,.\,x\;x & : & \forall\alpha\;(\alpha \to \alpha) \to \forall\alpha\;(\alpha \to \alpha) & \\ & : & \forall\alpha\;\alpha \to \forall\alpha\;\alpha & \\ & : & \forall\alpha\;\alpha \to \forall\alpha\;(\alpha \to \alpha) & \\ & : & \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool} & \text{('or' function!)} \end{array}$$

- No principal type (cf later)

Curry-style system F: properties

Things that do not change

- Substitutivity + β-subject reduction
- Strong normalisation (postponed)

Things that change

- A term may have several types

$$\begin{array}{lcll} \Delta \equiv \lambda x\,.\,x\;x & : & \forall\alpha\;(\alpha \to \alpha) \to \forall\alpha\;(\alpha \to \alpha) & \\ & : & \forall\alpha\;\alpha \to \forall\alpha\;\alpha & \\ & : & \forall\alpha\;\alpha \to \forall\alpha\;(\alpha \to \alpha) & \\ & : & \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool} & \text{('or' function!)} \end{array}$$

- No principal type (cf later)
- Type checking/inference becomes undecidable [Wells 94]

Erasing and typing

Erasing and typing

Equivalence between Church and Curry's presentations

Erasing and typing

Equivalence between Church and Curry's presentations

1. If $\Gamma \vdash t_0 : A$ (Church), then $\Gamma \vdash |t_0| : A$ (Curry)

Erasing and typing

Equivalence between Church and Curry's presentations

1. If $\Gamma \vdash t_0 : A$ (Church), then $\Gamma \vdash |t_0| : A$ (Curry)
2. If $\Gamma \vdash t : A$ (Curry), then $\Gamma \vdash t_0 : A$ (Church) for some t_0 s.t. $|t_0| = t$

Erasing and typing

Equivalence between Church and Curry's presentations

1. If $\Gamma \vdash t_0 : A$ (Church), then $\Gamma \vdash |t_0| : A$ (Curry)
2. If $\Gamma \vdash t : A$ (Curry), then $\Gamma \vdash t_0 : A$ (Church) for some t_0 s.t. $|t_0| = t$

The erasing function maps:

Church's world **Curry's world**

Erasing and typing

Equivalence between Church and Curry's presentations

1. If $\Gamma \vdash t_0 : A$ (Church), then $\Gamma \vdash |t_0| : A$ (Curry)
2. If $\Gamma \vdash t : A$ (Curry), then $\Gamma \vdash t_0 : A$ (Church) for some t_0 s.t. $|t_0| = t$

The erasing function maps:

	Church's world		**Curry's world**	
1.	derivations	to	derivations	(isomorphism)

Erasing and typing

Equivalence between Church and Curry's presentations

1. If $\Gamma \vdash t_0 : A$ (Church), then $\Gamma \vdash |t_0| : A$ (Curry)
2. If $\Gamma \vdash t : A$ (Curry), then $\Gamma \vdash t_0 : A$ (Church) for some t_0 s.t. $|t_0| = t$

The erasing function maps:

	Church's world		**Curry's world**	
1.	derivations	to	derivations	(isomorphism)
2.	valid judgements	to	valid judgements	(surjective only)

Erasing and typing

Equivalence between Church and Curry's presentations

1. If $\Gamma \vdash t_0 : A$ (Church), then $\Gamma \vdash |t_0| : A$ (Curry)
2. If $\Gamma \vdash t : A$ (Curry), then $\Gamma \vdash t_0 : A$ (Church) for some t_0 s.t. $|t_0| = t$

The erasing function maps:

	Church's world		**Curry's world**	
1.	derivations	to	derivations	(isomorphism)
2.	valid judgements	to	valid judgements	(surjective only)

On valid judgements, erasing is not injective:

$$\begin{array}{rcl} \lambda f : (\forall\alpha\ (\alpha{\to}\alpha))\,.\,f(\forall\alpha\ (\alpha{\to}\alpha))f & : & \forall\alpha\ (\alpha{\to}\alpha) \to \forall\alpha\ (\alpha{\to}\alpha) \\ \lambda f : (\forall\alpha\ (\alpha{\to}\alpha))\,.\,\Lambda\alpha\,.\,f(\alpha \to \alpha)(f\alpha) & : & \forall\alpha\ (\alpha{\to}\alpha) \to \forall\alpha\ (\alpha{\to}\alpha) \\ \rightsquigarrow \quad \lambda f\,.\,ff & : & \forall\alpha\ (\alpha{\to}\alpha) \to \forall\alpha\ (\alpha{\to}\alpha) \end{array}$$

Erasing and reduction

Second-kind redexes are erased, first-kind redexes are preserved

Erasing and reduction

Second-kind redexes are erased, first-kind redexes are preserved

$$\text{(Church)} \qquad (\Lambda\alpha\,.\,\lambda x : \alpha\,.\,x)\,B\,y \quad \succ \quad (\lambda x : B\,.\,x)\,y \quad \succ \quad y$$

Erasing and reduction

Second-kind redexes are erased, first-kind redexes are preserved

$$\begin{array}{lccccc} \text{(Church)} & (\Lambda\alpha\,.\,\lambda x:\alpha\,.\,x)\,B\,y & \succ & (\lambda x:B\,.\,x)\,y & \succ & y \\ \quad\downarrow \text{ Erasing} & & & & & \\ \text{(Curry)} & (\lambda x\,.\,x)\,y & \equiv & (\lambda x\,.\,x)\,y & \succ & y \end{array}$$

Erasing and reduction

Second-kind redexes are erased, first-kind redexes are preserved

$$\begin{array}{lccccc} \text{(Church)} & (\Lambda\alpha\,.\,\lambda x:\alpha\,.\,x)\,B\,y & \succ & (\lambda x:B\,.\,x)\,y & \succ & y \\ \quad\downarrow \text{ Erasing} & & & & & \\ \text{(Curry)} & (\lambda x\,.\,x)\,y & \equiv & (\lambda x\,.\,x)\,y & \succ & y \end{array}$$

Fact 1 (Church to Curry):

If $t_0, t_0' \in \text{Church}$, then

$$t \succ^n t' \quad \Rightarrow \quad |t_0| \succ^p |t_0'| \qquad (\text{with } p \leq n)$$

Erasing and reduction

Second-kind redexes are erased, first-kind redexes are preserved

$$\begin{array}{lccccc} \text{(Church)} & (\Lambda\alpha\,.\,\lambda x:\alpha\,.\,x)\,B\,y & \succ & (\lambda x:B\,.\,x)\,y & \succ & y \\ \quad\downarrow \text{ Erasing} & & & & & \\ \text{(Curry)} & (\lambda x\,.\,x)\,y & \equiv & (\lambda x\,.\,x)\,y & \succ & y \end{array}$$

Fact 1 (Church to Curry):

If $t_0, t_0' \in \text{Church}$, then

$$t \succ^n t' \quad \Rightarrow \quad |t_0| \succ^p |t_0'| \qquad (\text{with } p \leq n)$$

Fact 2 (Curry to Church):

If $t_0 \in \text{Church}$, $t' \in \text{Curry}$ and t_0 well-typed, then

$$|t_0| \succ^p t' \quad \Rightarrow \quad \exists t_0' \,(|t_0'| = t' \ \wedge \ t_0 \succ^n t_0') \qquad (\text{with } n \geq p)$$

Normalisation equivalence

Normalisation equivalence

Fact 3 (Combinatorial argument):

Normalisation equivalence

Fact 3 (Combinatorial argument):

1. During the contraction of a 1st-kind redex, the number of redexes of both kinds may increase

Normalisation equivalence

Fact 3 (Combinatorial argument):

1. During the contraction of a 1st-kind redex, the number of redexes of both kinds may increase
2. During the contraction of a 2nd-kind redex

Normalisation equivalence

Fact 3 (Combinatorial argument):

1. During the contraction of a 1st-kind redex, the number of redexes of both kinds may increase
2. During the contraction of a 2nd-kind redex
   - the number of 1st-kind redexes may increase

Normalisation equivalence

Fact 3 (Combinatorial argument):

1. During the contraction of a 1st-kind redex, the number of redexes of both kinds may increase
2. During the contraction of a 2nd-kind redex
   - the number of 1st-kind redexes may increase
   - the number of 2nd-kind redexes does not increase

Normalisation equivalence

Fact 3 (Combinatorial argument):

1. During the contraction of a 1st-kind redex, the number of redexes of both kinds may increase
2. During the contraction of a 2nd-kind redex
   - the number of 1st-kind redexes may increase
   - the number of 2nd-kind redexes does not increase
   - the number of type abstractions $(\Lambda\alpha\,.\,t)$ decreases

Normalisation equivalence

Fact 3 (Combinatorial argument):

1. During the contraction of a 1st-kind redex, the number of redexes of both kinds may increase
2. During the contraction of a 2nd-kind redex
   - the number of 1st-kind redexes may increase
   - the number of 2nd-kind redexes does not increase
   - the number of type abstractions $(\Lambda\alpha\,.\,t)$ decreases

Combining facts 1, 2 and 3, we easily prove:

Theorem (Normalisation equivalence):

The following statements are combinatorially equivalent:

1. All typable terms of syst. F-Church are strongly normalisable
2. All typable terms of syst. F-Curry are strongly normalisable

Subtyping

Subtyping

In Curry-style system F, subtyping is introduced as a macro:

$$A \leq B \quad \equiv \quad x : A \vdash x \ : \ B$$

Subtyping

In Curry-style system F, subtyping is introduced as a macro:

$$A \leq B \quad \equiv \quad x : A \vdash x \ : \ B$$

Admissible rules

Subtyping

In Curry-style system F, subtyping is introduced as a macro:

$$A \leq B \quad \equiv \quad x : A \vdash x \ : \ B$$

Admissible rules

(Reflexivity, transitivity)

$$\frac{}{A \ \leq \ A} \qquad \frac{A \ \leq \ B \qquad B \ \leq \ C}{A \ \leq \ C}$$

Subtyping

In Curry-style system F, subtyping is introduced as a macro:

$$A \leq B \quad \equiv \quad x : A \vdash x \ : \ B$$

Admissible rules

(Reflexivity, transitivity) $\dfrac{}{A \ \leq \ A}$ $\qquad$ $\dfrac{A \ \leq \ B \qquad B \ \leq \ C}{A \ \leq \ C}$

(Polymorphism) $\dfrac{}{\forall \alpha \ B \ \leq \ B\{\alpha := A\}}$ $\qquad$ $\dfrac{A \ \leq \ B}{A \ \leq \ \forall \alpha \ B}$ $\alpha \notin TV(A)$

Subtyping

In Curry-style system F, subtyping is introduced as a macro:

$$A \leq B \quad \equiv \quad x : A \vdash x : B$$

Admissible rules

(Reflexivity, transitivity) $\dfrac{}{A \leq A}$ $\qquad \dfrac{A \leq B \qquad B \leq C}{A \leq C}$

(Polymorphism) $\dfrac{}{\forall\alpha\ B \leq B\{\alpha := A\}}$ $\qquad \dfrac{A \leq B}{A \leq \forall\alpha\ B}\ \alpha \notin TV(A)$

(Subsumption) $\dfrac{\Gamma \vdash t : A \qquad A \leq B}{\Gamma \vdash t : B}$

Problem with η-redexes in Curry-style system F

Problem with η-redexes in Curry-style system F

- The (desired) subtyping rule for arrow-types

$$\frac{A \leq A' \qquad B \leq B'}{A' \to B \;\leq\; A \to B'}$$

  is not admissible

Problem with η-redexes in Curry-style system F

- The (desired) subtyping rule for arrow-types

$$\frac{A \leq A' \qquad B \leq B'}{A' \to B \;\leq\; A \to B'}$$

  is not admissible

- In particular, we have: $f : \mathsf{Nat} \to \forall\beta\ \beta \not\vdash f : \forall\alpha\ \alpha \to \mathsf{Bool}$

Problem with η-redexes in Curry-style system F

- The (desired) subtyping rule for arrow-types

$$\frac{A \leq A' \qquad B \leq B'}{A' \to B \;\leq\; A \to B'}$$

  is not admissible

- In particular, we have: $f : \mathsf{Nat} \to \forall\beta\ \beta \not\vdash f : \forall\alpha\ \alpha \to \mathsf{Bool}$

  but if we η-expand: $f : \mathsf{Nat} \to \forall\beta\ \beta \vdash \lambda x\,.\,fx : \forall\alpha\ \alpha \to \mathsf{Bool}$

Problem with η-redexes in Curry-style system F

- The (desired) subtyping rule for arrow-types

$$\frac{A \leq A' \qquad B \leq B'}{A' \to B \;\leq\; A \to B'}$$

  is not admissible

- In particular, we have: $f : \mathsf{Nat} \to \forall\beta\ \beta \not\vdash f : \forall\alpha\ \alpha \to \mathsf{Bool}$

  but if we η-expand: $f : \mathsf{Nat} \to \forall\beta\ \beta \vdash \lambda x\,.\,fx : \forall\alpha\ \alpha \to \mathsf{Bool}$

- This shows that:

Problem with η-redexes in Curry-style system F

- The (desired) subtyping rule for arrow-types

$$\frac{A \leq A' \qquad B \leq B'}{A' \to B \;\leq\; A \to B'}$$

  is not admissible

- In particular, we have: $f : \mathsf{Nat} \to \forall\beta\ \beta \not\vdash f : \forall\alpha\ \alpha \to \mathsf{Bool}$

  but if we η-expand: $f : \mathsf{Nat} \to \forall\beta\ \beta \vdash \lambda x\,.\,fx : \forall\alpha\ \alpha \to \mathsf{Bool}$

- This shows that:
  1. Curry-style system F does not enjoy η-subject reduction

Problem with η-redexes in Curry-style system F

- The (desired) subtyping rule for arrow-types

$$\frac{A \leq A' \qquad B \leq B'}{A' \to B \;\leq\; A \to B'}$$

  is not admissible

- In particular, we have: $f : \mathsf{Nat} \to \forall\beta\ \beta \not\vdash f : \forall\alpha\ \alpha \to \mathsf{Bool}$

  but if we η-expand: $f : \mathsf{Nat} \to \forall\beta\ \beta \vdash \lambda x\,.\,fx : \forall\alpha\ \alpha \to \mathsf{Bool}$

- This shows that:
  1. Curry-style system F does not enjoy η-subject reduction
  2. This problem is connected with subtyping in arrow-types

Problem with η-redexes in Curry-style system F

- The (desired) subtyping rule for arrow-types

$$\frac{A \leq A' \qquad B \leq B'}{A' \to B \;\leq\; A \to B'}$$

  is not admissible

- In particular, we have: $f : \mathsf{Nat} \to \forall\beta\ \beta \not\vdash f : \forall\alpha\ \alpha \to \mathsf{Bool}$

  but if we η-expand: $f : \mathsf{Nat} \to \forall\beta\ \beta \vdash \lambda x\,.\,fx : \forall\alpha\ \alpha \to \mathsf{Bool}$

- This shows that:
  1. Curry-style system F does not enjoy η-subject reduction
  2. This problem is connected with subtyping in arrow-types

The well-typed term: $\lambda x\,.\,fx : (\forall\alpha\ \alpha) \to \mathsf{Bool}$ (Curry-style)

comes from the term $\underbrace{\lambda x : (\forall\alpha\ \alpha)\,.\,f\ (x\ \mathsf{Nat})\ \mathsf{Bool}}_{\text{not an } \eta\text{-redex}}$ (Church-style)

System F_η [Mitchell 88]

System F_η [Mitchell 88]

Extend Curry-style system F with a new rule

$$\frac{\Gamma \vdash \lambda x\,.\,tx : A}{\Gamma \vdash t : A} \quad x \notin FV(t)$$

to enforce η-subject reduction

System F_η [Mitchell 88]

Extend Curry-style system F with a new rule

$$\frac{\Gamma \vdash \lambda x\,.\,tx : A}{\Gamma \vdash t : A} \quad x \notin FV(t)$$

to enforce η-subject reduction

Properties:

System F_η [Mitchell 88]

Extend Curry-style system F with a new rule

$$\frac{\Gamma \vdash \lambda x\,.\,tx : A}{\Gamma \vdash t : A} \quad x \notin FV(t)$$

to enforce η-subject reduction

Properties:

- Substitutivity, $\beta\eta$-subject-reduction, strong normalisation

System F_η [Mitchell 88]

Extend Curry-style system F with a new rule

$$\frac{\Gamma \vdash \lambda x\,.\,tx : A}{\Gamma \vdash t : A} \quad x \notin FV(t)$$

to enforce η-subject reduction

Properties:

- Substitutivity, $\beta\eta$-subject-reduction, strong normalisation
- Subtyping rule $\dfrac{A \leq A' \qquad B \leq B'}{A' \to B \;\leq\; A \to B'}$ is now admissible

System F_η [Mitchell 88]

Extend Curry-style system F with a new rule

$$\frac{\Gamma \vdash \lambda x\,.\,tx : A}{\Gamma \vdash t : A} \quad x \notin FV(t)$$

to enforce η-subject reduction

Properties:

- Substitutivity, $\beta\eta$-subject-reduction, strong normalisation
- Subtyping rule $\dfrac{A \leq A' \qquad B \leq B'}{A' \to B \;\leq\; A \to B'}$ is now admissible

Expansion lemma

If $\Gamma \vdash t : A$ is derivable in F_η, then $\Gamma \vdash t' : A$ is derivable in system F for some η-expansion t' of the term t.

More subtyping

If we set

$$\begin{array}{rcl} \bot & := & \forall\gamma\ \gamma \\ A \times B & := & \forall\gamma\ ((A \to B \to \gamma) \to \gamma) \\ A + B & := & \forall\gamma\ ((A \to \gamma) \to (B \to \gamma) \to \gamma) \\ \mathsf{List}(A) & := & \forall\gamma\ (\gamma \to (A \to \gamma \to \gamma) \to \gamma) \end{array}$$

then, in F_η, the following subtyping rules are admissible:

$$\frac{}{\bot \ \leq\ A} \qquad \frac{A \ \leq\ A'}{\mathsf{List}(A) \ \leq\ \mathsf{List}(A')}$$

$$\frac{A \ \leq\ A' \qquad B \ \leq\ B'}{A \times B \ \leq\ A' \times B'} \qquad \frac{A \ \leq\ A' \qquad B \ \leq\ B'}{A + B \ \leq\ A' + B'}$$

More subtyping

If we set

$$\begin{array}{rcl} \bot & := & \forall\gamma\ \gamma \\ A \times B & := & \forall\gamma\ ((A \to B \to \gamma) \to \gamma) \\ A + B & := & \forall\gamma\ ((A \to \gamma) \to (B \to \gamma) \to \gamma) \\ \mathsf{List}(A) & := & \forall\gamma\ (\gamma \to (A \to \gamma \to \gamma) \to \gamma) \end{array}$$

then, in F_η, the following subtyping rules are admissible:

$$\frac{}{\bot \ \leq\ A} \qquad \frac{A \ \leq\ A'}{\mathsf{List}(A) \ \leq\ \mathsf{List}(A')}$$

$$\frac{A \ \leq\ A' \qquad B \ \leq\ B'}{A \times B \ \leq\ A' \times B'} \qquad \frac{A \ \leq\ A' \qquad B \ \leq\ B'}{A + B \ \leq\ A' + B'}$$

But most typable terms have no principal type

Adding intersection types

Extend system F_η with binary intersections

Types $\quad A, B \quad ::= \quad \alpha \quad | \quad A \to B \quad | \quad \forall \alpha\ B \quad | \quad A \cap B$

Adding intersection types

Extend system F_η with binary intersections

Types $\quad A, B \quad ::= \quad \alpha \quad | \quad A \to B \quad | \quad \forall \alpha\ B \quad | \quad A \cap B$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash t : B}{\Gamma \vdash t : A \cap B} \qquad \frac{\Gamma \vdash t : A \cap B}{\Gamma \vdash t : A} \qquad \frac{\Gamma \vdash t : A \cap B}{\Gamma \vdash t : B}$$

Adding intersection types

Extend system F_η with binary intersections

Types $\quad A, B \quad ::= \quad \alpha \quad | \quad A \to B \quad | \quad \forall \alpha\ B \quad | \quad A \cap B$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash t : B}{\Gamma \vdash t : A \cap B} \qquad \frac{\Gamma \vdash t : A \cap B}{\Gamma \vdash t : A} \qquad \frac{\Gamma \vdash t : A \cap B}{\Gamma \vdash t : B}$$

- $\beta\eta$-subject reduction, strong normalisation, etc.

Adding intersection types

Extend system F_η with binary intersections

Types $\quad A, B \quad ::= \quad \alpha \quad | \quad A \to B \quad | \quad \forall \alpha\ B \quad | \quad A \cap B$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash t : B}{\Gamma \vdash t : A \cap B} \qquad \frac{\Gamma \vdash t : A \cap B}{\Gamma \vdash t : A} \qquad \frac{\Gamma \vdash t : A \cap B}{\Gamma \vdash t : B}$$

- $\beta\eta$-subject reduction, strong normalisation, etc.
- Subtyping rules

$$\frac{}{A \cap B \ \leq \ A} \qquad \frac{}{A \cap B \ \leq \ B} \qquad \frac{C \leq A \qquad C \leq B}{C \ \leq \ A \cap B}$$

Adding intersection types

Extend system F_η with binary intersections

Types $\quad A, B \quad ::= \quad \alpha \quad | \quad A \to B \quad | \quad \forall \alpha\ B \quad | \quad A \cap B$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash t : B}{\Gamma \vdash t : A \cap B} \qquad \frac{\Gamma \vdash t : A \cap B}{\Gamma \vdash t : A} \qquad \frac{\Gamma \vdash t : A \cap B}{\Gamma \vdash t : B}$$

- $\beta\eta$-subject reduction, strong normalisation, etc.
- Subtyping rules

$$\frac{}{A \cap B \ \leq \ A} \qquad \frac{}{A \cap B \ \leq \ B} \qquad \frac{C \leq A \qquad C \leq B}{C \ \leq \ A \cap B}$$

- All the strongly normalising terms are typable...

Adding intersection types

Extend system F_η with binary intersections

Types $\quad A, B \quad ::= \quad \alpha \quad | \quad A \to B \quad | \quad \forall \alpha\ B \quad | \quad A \cap B$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash t : B}{\Gamma \vdash t : A \cap B} \qquad \frac{\Gamma \vdash t : A \cap B}{\Gamma \vdash t : A} \qquad \frac{\Gamma \vdash t : A \cap B}{\Gamma \vdash t : B}$$

- $\beta\eta$-subject reduction, strong normalisation, etc.
- Subtyping rules

$$\frac{}{A \cap B \ \leq \ A} \qquad \frac{}{A \cap B \ \leq \ B} \qquad \frac{C \leq A \qquad C \leq B}{C \ \leq \ A \cap B}$$

- All the strongly normalising terms are typable...

  ... but nothing to do with $\forall$: already true in $\lambda{\to}\cap$

Adding intersection types

Extend system F_η with binary intersections

Types $\quad A, B \quad ::= \quad \alpha \quad | \quad A \to B \quad | \quad \forall \alpha\ B \quad | \quad A \cap B$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash t : B}{\Gamma \vdash t : A \cap B} \qquad \frac{\Gamma \vdash t : A \cap B}{\Gamma \vdash t : A} \qquad \frac{\Gamma \vdash t : A \cap B}{\Gamma \vdash t : B}$$

- $\beta\eta$-subject reduction, strong normalisation, etc.
- Subtyping rules

$$\frac{}{A \cap B \ \leq \ A} \qquad \frac{}{A \cap B \ \leq \ B} \qquad \frac{C \leq A \qquad C \leq B}{C \ \leq \ A \cap B}$$

- All the strongly normalising terms are typable...

  ... but nothing to do with $\forall$: already true in $\lambda{\to}\cap$

- All typable terms have a principal type

  $\lambda x : xx\,. \quad : \quad \forall\alpha\ \forall\beta\ ((\alpha{\to}\beta) \cap \alpha \to \beta)$

Part IV

The Strong Normalisation Theorem

The meaning of second-order quantification (1/2)

Question: What is the meaning of $\forall\alpha\,(\alpha \to \alpha)$?

The meaning of second-order quantification (1/2)

Question: What is the meaning of $\forall\alpha\,(\alpha \to \alpha)$?

First scenario: an infinite Cartesian product (à la Martin-Löf)

$$\forall\alpha\,(\alpha \to \alpha) \quad \approx \quad \prod_{\alpha \text{ type}} (\alpha \to \alpha)$$

The meaning of second-order quantification (1/2)

Question: What is the meaning of $\forall\alpha\,(\alpha \to \alpha)$?

First scenario: an infinite Cartesian product (à la Martin-Löf)

$$\forall\alpha\,(\alpha \to \alpha) \approx \prod_{\alpha \text{ type}} (\alpha \to \alpha)$$

$$\approx (\bot \to \bot) \times (\mathsf{Bool} \to \mathsf{Bool}) \times (\mathsf{Nat} \to \mathsf{Nat}) \times \cdots$$

The meaning of second-order quantification (1/2)

Question: What is the meaning of $\forall\alpha\,(\alpha \to \alpha)$?

First scenario: an infinite Cartesian product (à la Martin-Löf)

$$\forall\alpha\,(\alpha \to \alpha) \approx \prod_{\alpha \text{ type}} (\alpha \to \alpha)$$
$$\approx (\bot \to \bot) \times (\mathsf{Bool} \to \mathsf{Bool}) \times (\mathsf{Nat} \to \mathsf{Nat}) \times \cdots$$

Since all the types $A \to A$ are inhabited:

The meaning of second-order quantification (1/2)

Question: What is the meaning of $\forall\alpha\,(\alpha \to \alpha)$?

First scenario: an infinite Cartesian product (à la Martin-Löf)

$$\forall\alpha\,(\alpha \to \alpha) \approx \prod_{\alpha \text{ type}} (\alpha \to \alpha)$$
$$\approx (\bot \to \bot) \times (\mathsf{Bool} \to \mathsf{Bool}) \times (\mathsf{Nat} \to \mathsf{Nat}) \times \cdots$$

Since all the types $A \to A$ are inhabited:

1. The cartesian product $\forall\alpha\,(\alpha \to \alpha)$ should be larger than all the types of the form $A \to A$

The meaning of second-order quantification (1/2)

Question: What is the meaning of $\forall\alpha\,(\alpha \to \alpha)$?

First scenario: an infinite Cartesian product (à la Martin-Löf)

$$\forall\alpha\,(\alpha \to \alpha) \approx \prod_{\alpha \text{ type}} (\alpha \to \alpha)$$
$$\approx (\bot \to \bot) \times (\mathsf{Bool} \to \mathsf{Bool}) \times (\mathsf{Nat} \to \mathsf{Nat}) \times \cdots$$

Since all the types $A \to A$ are inhabited:

1. The cartesian product $\forall\alpha\,(\alpha \to \alpha)$ should be larger than all the types of the form $A \to A$
2. In particular, $\forall\alpha\,(\alpha \to \alpha)$ should be larger than its own function space $\forall\alpha\,(\alpha \to \alpha) \to \forall\alpha\,(\alpha \to \alpha) \ldots$

The meaning of second-order quantification (1/2)

Question: What is the meaning of $\forall\alpha\,(\alpha \to \alpha)$?

First scenario: an infinite Cartesian product (à la Martin-Löf)

$$\forall\alpha\,(\alpha \to \alpha) \approx \prod_{\alpha \text{ type}} (\alpha \to \alpha)$$
$$\approx (\bot \to \bot) \times (\mathsf{Bool} \to \mathsf{Bool}) \times (\mathsf{Nat} \to \mathsf{Nat}) \times \cdots$$

Since all the types $A \to A$ are inhabited:

1. The cartesian product $\forall\alpha\,(\alpha \to \alpha)$ should be larger than all the types of the form $A \to A$
2. In particular, $\forall\alpha\,(\alpha \to \alpha)$ should be larger than its own function space $\forall\alpha\,(\alpha \to \alpha) \to \forall\alpha\,(\alpha \to \alpha) \ldots$

. . . seems to be very confusing!

The meaning of second-order quantification (2/2)

Second scenario: In F-Curry, both rules $\forall$-intro and $\forall$-elim

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash t : \forall\alpha\ B}\ \alpha \notin TV(\Gamma) \qquad \frac{\Gamma \vdash t : \forall\alpha\ B}{\Gamma \vdash t : B\{\alpha := A\}}$$

suggest that $\forall$ is not a cartesian product, but an intersection

The meaning of second-order quantification (2/2)

Second scenario: In F-Curry, both rules $\forall$-intro and $\forall$-elim

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash t : \forall\alpha\ B}\ \alpha \notin TV(\Gamma) \qquad \frac{\Gamma \vdash t : \forall\alpha\ B}{\Gamma \vdash t : B\{\alpha := A\}}$$

suggest that $\forall$ is not a cartesian product, but an intersection

Taking back our example:

The meaning of second-order quantification (2/2)

Second scenario: In F-Curry, both rules $\forall$-intro and $\forall$-elim

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash t : \forall\alpha\ B}\ \alpha \notin TV(\Gamma) \qquad \frac{\Gamma \vdash t : \forall\alpha\ B}{\Gamma \vdash t : B\{\alpha := A\}}$$

suggest that $\forall$ is not a cartesian product, but an intersection

Taking back our example:

1. The intersection $\forall\alpha\ (\alpha \to \alpha)$ is smaller than all $A \to A$

The meaning of second-order quantification (2/2)

Second scenario: In F-Curry, both rules $\forall$-intro and $\forall$-elim

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash t : \forall\alpha\ B}\ \alpha \notin TV(\Gamma) \qquad \frac{\Gamma \vdash t : \forall\alpha\ B}{\Gamma \vdash t : B\{\alpha := A\}}$$

suggest that $\forall$ is not a cartesian product, but an intersection

Taking back our example:

1. The intersection $\forall\alpha\ (\alpha \to \alpha)$ is smaller than all $A \to A$
2. In particular, $\forall\alpha\ (\alpha \to \alpha)$ is smaller than its own function space $\forall\alpha\ (\alpha \to \alpha) \to \forall\alpha\ (\alpha \to \alpha) \ldots$

The meaning of second-order quantification (2/2)

Second scenario: In F-Curry, both rules $\forall$-intro and $\forall$-elim

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash t : \forall\alpha\ B}\ \alpha \notin TV(\Gamma) \qquad \frac{\Gamma \vdash t : \forall\alpha\ B}{\Gamma \vdash t : B\{\alpha := A\}}$$

suggest that $\forall$ is not a cartesian product, but an intersection

Taking back our example:

1. The intersection $\forall\alpha\ (\alpha \to \alpha)$ is smaller than all $A \to A$
2. In particular, $\forall\alpha\ (\alpha \to \alpha)$ is smaller than its own function space $\forall\alpha\ (\alpha \to \alpha) \to \forall\alpha\ (\alpha \to \alpha) \ldots$

. . . our intuition feels much better!

The meaning of second-order quantification (2/2)

Second scenario: In F-Curry, both rules $\forall$-intro and $\forall$-elim

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash t : \forall\alpha\ B}\ \alpha \notin TV(\Gamma) \qquad \frac{\Gamma \vdash t : \forall\alpha\ B}{\Gamma \vdash t : B\{\alpha := A\}}$$

suggest that $\forall$ is not a cartesian product, but an intersection

Taking back our example:

1. The intersection $\forall\alpha\ (\alpha \to \alpha)$ is smaller than all $A \to A$
2. In particular, $\forall\alpha\ (\alpha \to \alpha)$ is smaller than its own function space $\forall\alpha\ (\alpha \to \alpha) \to \forall\alpha\ (\alpha \to \alpha) \ldots$

. . . our intuition feels much better!

$\Rightarrow$ We will prove strong normalisation for Curry-style system F

Remember that SN(F-Church) $\Leftrightarrow$ SN(F-Curry) (combinatorial equivalence)

Strong normalisation: the difficulty

Try to prove that

$$\Gamma \vdash t : A \quad \Rightarrow \quad t \text{ is SN}$$

by induction on the derivation of $\Gamma \vdash t : A$

Strong normalisation: the difficulty

Try to prove that

$$\Gamma \vdash t : A \quad \Rightarrow \quad t \text{ is SN}$$

by induction on the derivation of $\Gamma \vdash t : A$

$$\frac{}{\Gamma \vdash x : A}\ {\scriptstyle (x:A)\in\Gamma}$$

$$\frac{\Gamma,\ x : A \vdash t : B}{\Gamma \vdash \lambda x\,.\,t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B}$$

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash t : \forall \alpha\ B}\ {\scriptstyle \alpha \notin TV(\Gamma)} \qquad \frac{\Gamma \vdash t : \forall \alpha\ B}{\Gamma \vdash t : B\{\alpha := A\}}$$

Strong normalisation: the difficulty

Try to prove that

$$\Gamma \vdash t : A \quad \Rightarrow \quad t \text{ is SN}$$

by induction on the derivation of $\Gamma \vdash t : A$

$$\frac{}{\Gamma \vdash x : A}\ {\scriptstyle (x:A)\in\Gamma}$$

$$\frac{\Gamma,\ x : A \vdash t : B}{\Gamma \vdash \lambda x\,.\,t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$$

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash t : \forall \alpha\ B}\ {\scriptstyle \alpha \notin TV(\Gamma)} \qquad \frac{\Gamma \vdash t : \forall \alpha\ B}{\Gamma \vdash t : B\{\alpha := A\}}$$

All the cases successfully pass the test except application

Two terms t and u may be SN, whereas tu is not [Take $t \equiv u \equiv \lambda x\,.\,xx$]

Strong normalisation: the difficulty

Try to prove that

$$\Gamma \vdash t : A \quad \Rightarrow \quad t \text{ is SN}$$

by induction on the derivation of $\Gamma \vdash t : A$

$$\frac{}{\Gamma \vdash x : A}\ {\scriptstyle (x:A)\in\Gamma}$$

$$\frac{\Gamma,\ x : A \vdash t : B}{\Gamma \vdash \lambda x\,.\,t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$$

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash t : \forall \alpha\ B}\ {\scriptstyle \alpha \notin TV(\Gamma)} \qquad \frac{\Gamma \vdash t : \forall \alpha\ B}{\Gamma \vdash t : B\{\alpha := A\}}$$

All the cases successfully pass the test except application

Two terms t and u may be SN, whereas tu is not [Take $t \equiv u \equiv \lambda x\,.\,xx$]

$\Rightarrow$ The induction hypothesis "t is SN" is too weak (in general)

Reducibility candidates [Girard 1971]

To prove that

$$\Gamma \vdash t : A \quad \Rightarrow \quad t \text{ is SN},$$

the induction hypothesis "t is SN" is too weak.

Reducibility candidates [Girard 1971]

To prove that

$$\Gamma \vdash t : A \quad \Rightarrow \quad t \text{ is SN},$$

the induction hypothesis "t is SN" is too weak.

$\Rightarrow$ Should replace it by an invariant that depends on the type A

Reducibility candidates [Girard 1971]

To prove that

$$\Gamma \vdash t : A \quad \Rightarrow \quad t \text{ is SN}\,,$$

the induction hypothesis "t is SN" is too weak.

$\Rightarrow$ Should replace it by an invariant that depends on the type A

Intuition:

The more complex the type, the stronger its invariant,
the smaller the set of terms that fulfill this invariant

Reducibility candidates [Girard 1971]

To prove that

$$\Gamma \vdash t : A \quad \Rightarrow \quad t \text{ is SN}\,,$$

the induction hypothesis "t is SN" is too weak.

$\Rightarrow$ Should replace it by an invariant that depends on the type A

Intuition:

> *The more complex the type, the stronger its invariant,*
> *the smaller the set of terms that fulfill this invariant*

Invariants are represented by suitable sets of terms:

Reducibility candidates [Girard 1971]

To prove that

$$\Gamma \vdash t : A \quad \Rightarrow \quad t \text{ is SN},$$

the induction hypothesis "t is SN" is too weak.

$\Rightarrow$ Should replace it by an invariant that depends on the type A

Intuition:

> *The more complex the type, the stronger its invariant,*
> *the smaller the set of terms that fulfill this invariant*

Invariants are represented by suitable sets of terms:

- Reducibility candidates [Girard]

Reducibility candidates [Girard 1971]

To prove that

$$\Gamma \vdash t : A \quad \Rightarrow \quad t \text{ is SN},$$

the induction hypothesis "t is SN" is too weak.

$\Rightarrow$ Should replace it by an invariant that depends on the type A

Intuition:

> *The more complex the type, the stronger its invariant,*
> *the smaller the set of terms that fulfill this invariant*

Invariants are represented by suitable sets of terms:

- Reducibility candidates [Girard], or
- Saturated sets [Tait]

Outline of the proof

1. Define a suitable notion of reducibility candidate
   $=$ the sets of λ-terms that will interpret/represent types

   (Here, we use Tait's saturated sets)

Outline of the proof

1. Define a suitable notion of reducibility candidate
   $=$ the sets of λ-terms that will interpret/represent types

   (Here, we use Tait's saturated sets)

2. Ensure that the notion of candidate captures the property of strong normalisation (which we want to prove)

   Each candidate should only contain strongly normalisable λ-terms as elements

Outline of the proof

1. Define a suitable notion of reducibility candidate
   $=$ the sets of λ-terms that will interpret/represent types

   (Here, we use Tait's saturated sets)

2. Ensure that the notion of candidate captures the property of strong normalisation (which we want to prove)

   Each candidate should only contain strongly normalisable λ-terms as elements

3. Associate to each type A a reducibility candidate $[\![A]\!]$

   Type constructors '$\rightarrow$' and '$\forall$' have to be reflected at the level of candidates

Outline of the proof

1. Define a suitable notion of reducibility candidate
   $=$ the sets of λ-terms that will interpret/represent types

   (Here, we use Tait's saturated sets)

2. Ensure that the notion of candidate captures the property of strong normalisation (which we want to prove)

   Each candidate should only contain strongly normalisable λ-terms as elements

3. Associate to each type A a reducibility candidate $[\![A]\!]$

   Type constructors '$\rightarrow$' and '$\forall$' have to be reflected at the level of candidates

4. Check (by induction) that $\Gamma \vdash t : A$ implies $t \in [\![A]\!]$

   This is actually a little bit more complex, since we must take care of the typing context

Outline of the proof

1. Define a suitable notion of reducibility candidate
   $=$ the sets of λ-terms that will interpret/represent types

   (Here, we use Tait's saturated sets)

2. Ensure that the notion of candidate captures the property of strong normalisation (which we want to prove)

   Each candidate should only contain strongly normalisable λ-terms as elements

3. Associate to each type A a reducibility candidate $[\![A]\!]$

   Type constructors '$\rightarrow$' and '$\forall$' have to be reflected at the level of candidates

4. Check (by induction) that $\Gamma \vdash t : A$ implies $t \in [\![A]\!]$

   This is actually a little bit more complex, since we must take care of the typing context

5. Conclude that any well-typed term t is SN by step 2.

Preliminaries (1/2)

Preliminaries (1/2)

- **Notations:**

Λ	$\equiv$	set of all untyped λ-terms (open & closed)
SN	$\equiv$	set of all strongly normalisable untyped λ-terms
Var	$\equiv$	set of all (term) variables
TVar	$\equiv$	set of all type variables

Preliminaries (1/2)

- **Notations:**

Λ	$\equiv$	set of all untyped λ-terms (open & closed)
SN	$\equiv$	set of all strongly normalisable untyped λ-terms
Var	$\equiv$	set of all (term) variables
TVar	$\equiv$	set of all type variables

- A reduct of a term t is a term t' such that $t \succ t'$ (one step)

The number of reducts of a given term is finite and bounded by the number of redexes

Preliminaries (1/2)

- **Notations:**

Λ	$\equiv$	set of all untyped λ-terms (open & closed)
SN	$\equiv$	set of all strongly normalisable untyped λ-terms
Var	$\equiv$	set of all (term) variables
TVar	$\equiv$	set of all type variables

- A reduct of a term t is a term t' such that $t \succ t'$ (one step)

  The number of reducts of a given term is finite and bounded by the number of redexes

- A finite reduction sequence of a term t is a finite sequence $(t_i)_{i \in [0..n]}$ such that $t = t_0 \succ t_1 \succ \cdots \succ t_{n-1} \succ t_n$

  Infinite reduction sequences are defined similarly, by replacing $[0..n]$ by $\mathbb{N}$

Preliminaries (1/2)

- **Notations:**

Λ	$\equiv$	set of all untyped λ-terms (open & closed)
SN	$\equiv$	set of all strongly normalisable untyped λ-terms
Var	$\equiv$	set of all (term) variables
TVar	$\equiv$	set of all type variables

- A reduct of a term t is a term t' such that $t \succ t'$ (one step)

  The number of reducts of a given term is finite and bounded by the number of redexes

- A finite reduction sequence of a term t is a finite sequence $(t_i)_{i \in [0..n]}$ such that $t = t_0 \succ t_1 \succ \cdots \succ t_{n-1} \succ t_n$

  Infinite reduction sequences are defined similarly, by replacing $[0..n]$ by $\mathbb{N}$

- Finite reduction sequences of a term t form a tree, called the reduction tree of t

Preliminaries (2/2)

Preliminaries (2/2)

Definition (Strongly normalisable terms)

A term t is strongly normalisable if all the reduction sequences starting from t are finite

Preliminaries (2/2)

Definition (Strongly normalisable terms)

A term t is strongly normalisable if all the reduction sequences starting from t are finite

Proposition

The following assertions are equivalent:

1. t is strongly normalisable
2. All the reducts of t are strongly normalisable
3. The reduction tree of t is finite

Saturated sets [Tait]

Definition (Saturated set)

A set $S \subset \Lambda$ is saturated if:

(SAT1) $\quad S \subset \mathrm{SN}$

(SAT2) $\quad x \in \mathrm{Var}, \quad \vec{v} \in \mathrm{list}(\mathrm{SN}) \quad \Rightarrow \quad x\vec{v} \in S$

(SAT3) $\quad t\{x := u\}\vec{v} \in S, \quad u \in \mathrm{SN} \quad \Rightarrow \quad (\lambda x\,.\,t)u\vec{v} \in S$

Saturated sets [Tait]

Definition (Saturated set)

A set $S \subset \Lambda$ is saturated if:

(SAT1) $\quad S \subset \mathrm{SN}$

(SAT2) $\quad x \in \mathrm{Var}, \quad \vec{v} \in \mathrm{list}(\mathrm{SN}) \quad \Rightarrow \quad x\vec{v} \in S$

(SAT3) $\quad t\{x := u\}\vec{v} \in S, \quad u \in \mathrm{SN} \quad \Rightarrow \quad (\lambda x\,.\,t)u\vec{v} \in S$

- (SAT1) expresses the property we want to prove

Saturated sets [Tait]

Definition (Saturated set)

A set $S \subset \Lambda$ is saturated if:

(SAT1) $\quad S \subset \mathrm{SN}$

(SAT2) $\quad x \in \mathrm{Var}, \quad \vec{v} \in \mathrm{list}(\mathrm{SN}) \quad \Rightarrow \quad x\vec{v} \in S$

(SAT3) $\quad t\{x := u\}\vec{v} \in S, \quad u \in \mathrm{SN} \quad \Rightarrow \quad (\lambda x\,.\,t)u\vec{v} \in S$

- (SAT1) expresses the property we want to prove
- Saturated sets contain all the variables (SAT2)

  Extra-arguments $\vec{v} \in \mathrm{list}(\mathrm{SN})$ are here for technical reasons

Saturated sets [Tait]

Definition (Saturated set)

A set $S \subset \Lambda$ is saturated if:

(SAT1) $\quad S \subset \mathrm{SN}$

(SAT2) $\quad x \in \mathrm{Var}, \quad \vec{v} \in \mathrm{list}(\mathrm{SN}) \quad \Rightarrow \quad x\vec{v} \in S$

(SAT3) $\quad t\{x := u\}\vec{v} \in S, \quad u \in \mathrm{SN} \quad \Rightarrow \quad (\lambda x\,.\,t)u\vec{v} \in S$

- (SAT1) expresses the property we want to prove
- Saturated sets contain all the variables (SAT2)

  Extra-arguments $\vec{v} \in \mathrm{list}(\mathrm{SN})$ are here for technical reasons
- Saturated sets are closed under head β-expansion (SAT3)

  Notice the condition $u \in \mathrm{SN}$ to avoid a clash with (SAT1) for K-redexes

Saturated sets [Tait]

Definition (Saturated set)

A set $S \subset \Lambda$ is saturated if:

(SAT1) $\quad S \subset \mathrm{SN}$

(SAT2) $\quad x \in \mathrm{Var}, \quad \vec{v} \in \mathrm{list}(\mathrm{SN}) \quad \Rightarrow \quad x\vec{v} \in S$

(SAT3) $\quad t\{x := u\}\vec{v} \in S, \quad u \in \mathrm{SN} \quad \Rightarrow \quad (\lambda x\,.\,t)u\vec{v} \in S$

- (SAT1) expresses the property we want to prove
- Saturated sets contain all the variables (SAT2)
  Extra-arguments $\vec{v} \in \mathrm{list}(\mathrm{SN})$ are here for technical reasons
- Saturated sets are closed under head β-expansion (SAT3)
  Notice the condition $u \in \mathrm{SN}$ to avoid a clash with (SAT1) for K-redexes
- The set of all saturated sets is written **SAT** $\quad [\subset \mathfrak{P}(\mathrm{SN}) \subset \mathfrak{P}(\Lambda)]$

Properties of saturated sets

Proposition (Lattice structure)

Properties of saturated sets

Proposition (Lattice structure)

1. SN is a saturated set

Properties of saturated sets

Proposition (Lattice structure)

1. SN is a saturated set
2. **SAT** is closed under arbitrary non-empty intersections/unions:

$$I \neq \varnothing, \quad (S_i)_{i \in I} \in \mathbf{SAT}^I \quad \Rightarrow \quad \left(\bigcap_{i \in I} S_i\right), \left(\bigcup_{i \in I} S_i\right) \in \mathbf{SAT}$$

Properties of saturated sets

Proposition (Lattice structure)

1. SN is a saturated set
2. **SAT** is closed under arbitrary non-empty intersections/unions:

$$I \neq \varnothing, \quad (S_i)_{i \in I} \in \mathbf{SAT}^I \quad \Rightarrow \quad \left(\bigcap_{i \in I} S_i\right), \left(\bigcup_{i \in I} S_i\right) \in \mathbf{SAT}$$

(**SAT**, $\subset$) is a complete distributive lattice, with

$\top = \mathrm{SN}$ and $\bot = \{t \in \mathrm{SN} \mid t \succ^* xu_1 \cdots u_n\}$ (Neutral terms)

Properties of saturated sets

Proposition (Lattice structure)

1. SN is a saturated set
2. **SAT** is closed under arbitrary non-empty intersections/unions:

$$I \neq \varnothing, \quad (S_i)_{i\in I} \in \mathbf{SAT}^I \quad \Rightarrow \quad \Big(\bigcap_{i\in I} S_i\Big), \Big(\bigcup_{i\in I} S_i\Big) \in \mathbf{SAT}$$

(**SAT**, $\subset$) is a complete distributive lattice, with

$\top = \mathrm{SN}$ and $\bot = \{t \in \mathrm{SN} \mid t \succ^* xu_1 \cdots u_n\}$ (Neutral terms)

Realisability arrow: For all $S, T \subset \Lambda$ we set

$$S \to T \quad := \quad \{t \in \Lambda \mid \forall u \in S \quad tu \in T\}$$

Properties of saturated sets

Proposition (Lattice structure)

1. SN is a saturated set
2. **SAT** is closed under arbitrary non-empty intersections/unions:

$$I \neq \varnothing, \quad (S_i)_{i\in I} \in \mathbf{SAT}^I \quad \Rightarrow \quad \Big(\bigcap_{i\in I} S_i\Big), \Big(\bigcup_{i\in I} S_i\Big) \in \mathbf{SAT}$$

$(\mathbf{SAT}, \subset)$ is a complete distributive lattice, with

$\top = \mathrm{SN}$ and $\bot = \{t \in \mathrm{SN} \mid t \succ^* xu_1 \cdots u_n\}$ (Neutral terms)

Realisability arrow: For all $S, T \subset \Lambda$ we set

$$S \to T \quad := \quad \{t \in \Lambda \mid \forall u \in S \quad tu \in T\}$$

Proposition (Closure under realisability arrow)

If $S, T \in \mathbf{SAT}$, then $(S \to T) \in \mathbf{SAT}$

Interpreting types (1/2)

Principle: Interpret syntactic types by saturated sets

Interpreting types (1/2)

Principle: Interpret syntactic types by saturated sets

- Type arrow $A \to B$ is interpreted by $S \to T$ (realisability arrow)

Interpreting types (1/2)

Principle: Interpret syntactic types by saturated sets

- Type arrow $A \to B$ is interpreted by $S \to T$ (realisability arrow)
- Type quantification $\forall \alpha \ . \ .$ is interpreted by the intersection $\bigcap_{S \in \mathbf{SAT}} \cdots$

Interpreting types (1/2)

Principle: Interpret syntactic types by saturated sets

- Type arrow $A \to B$ is interpreted by $S \to T$ (realisability arrow)
- Type quantification $\forall \alpha \ . \ .$ is interpreted by the intersection $\bigcap_{S \in \mathbf{SAT}} \cdots$

Remark: this intersection is impredicative since S ranges over all saturated sets

Interpreting types (1/2)

Principle: Interpret syntactic types by saturated sets

- Type arrow $A \to B$ is interpreted by $S \to T$ (realisability arrow)
- Type quantification $\forall \alpha \ . \ .$ is interpreted by the intersection $\bigcap_{S \in \mathbf{SAT}} \cdots$

  Remark: this intersection is impredicative since S ranges over all saturated sets

Example: $\forall \alpha \ (\alpha \to \alpha)$ should be interpreted by $\bigcap_{S \in \mathbf{SAT}} (S \to S)$

Interpreting types (1/2)

Principle: Interpret syntactic types by saturated sets

- Type arrow $A \to B$ is interpreted by $S \to T$ (realisability arrow)
- Type quantification $\forall \alpha \ . \ .$ is interpreted by the intersection $\bigcap_{S \in \mathbf{SAT}} \cdots$

  Remark: this intersection is impredicative since S ranges over all saturated sets

Example: $\forall \alpha \ (\alpha \to \alpha)$ should be interpreted by $\bigcap_{S \in \mathbf{SAT}} (S \to S)$

To interpret type variables, use type valations:

Interpreting types (1/2)

Principle: Interpret syntactic types by saturated sets

- Type arrow $A \to B$ is interpreted by $S \to T$ (realisability arrow)
- Type quantification $\forall \alpha \ldots$ is interpreted by the intersection $\bigcap_{S \in \mathbf{SAT}} \cdots$

  Remark: this intersection is impredicative since S ranges over all saturated sets

Example: $\forall \alpha \, (\alpha \to \alpha)$ should be interpreted by $\bigcap_{S \in \mathbf{SAT}} (S \to S)$

To interpret type variables, use type valations:

Definition (Type valuations)

A type valuation is a function $\rho : \mathrm{TVar} \to \mathbf{SAT}$

The set of type valuations is written TVal $(= \mathrm{TVar} \to \mathbf{SAT})$

Interpreting types (2/2)

By induction on A, we define a function $[\![A]\!] : \mathsf{TVal} \to \mathbf{SAT}$

Interpreting types (2/2)

By induction on A, we define a function $[\![A]\!] : \mathrm{TVal} \to \mathbf{SAT}$

$$[\![A \to B]\!]_\rho = [\![A]\!]_\rho \to [\![B]\!]_\rho \qquad [\![\alpha]\!]_\rho = \rho(\alpha)$$

$$[\![\forall \alpha\ B]\!]_\rho = \bigcap_{S \in \mathbf{SAT}} [\![B]\!]_{\rho;\alpha \leftarrow S}$$

Note: $(\rho; \alpha \leftarrow S)$ **is defined by** $\begin{cases} (\rho; \alpha \leftarrow S)(\alpha) = S \\ (\rho; \alpha \leftarrow S)(\beta) = \rho(\beta) \quad \text{for all } \beta \neq \alpha \end{cases}$

Interpreting types (2/2)

By induction on A, we define a function $[\![A]\!] : \mathrm{TVal} \to \mathbf{SAT}$

$$[\![A \to B]\!]_\rho = [\![A]\!]_\rho \to [\![B]\!]_\rho \qquad [\![\alpha]\!]_\rho = \rho(\alpha)$$

$$[\![\forall \alpha\ B]\!]_\rho = \bigcap_{S \in \mathbf{SAT}} [\![B]\!]_{\rho;\alpha \leftarrow S}$$

Note: $(\rho; \alpha \leftarrow S)$ **is defined by** $\begin{cases} (\rho; \alpha \leftarrow S)(\alpha) = S \\ (\rho; \alpha \leftarrow S)(\beta) = \rho(\beta) \quad \text{for all } \beta \neq \alpha \end{cases}$

Problem: The implication

$$\Gamma \vdash t : A \quad \Rightarrow \quad t \in [\![A]\!]_\rho$$

cannot be proved directly. (One has to take care of the context)

Interpreting types (2/2)

By induction on A, we define a function $[\![A]\!] : \mathrm{TVal} \to \mathbf{SAT}$

$$[\![A \to B]\!]_\rho = [\![A]\!]_\rho \to [\![B]\!]_\rho \qquad [\![\alpha]\!]_\rho = \rho(\alpha)$$

$$[\![\forall \alpha\ B]\!]_\rho = \bigcap_{S \in \mathbf{SAT}} [\![B]\!]_{\rho;\alpha \leftarrow S}$$

Note: $(\rho; \alpha \leftarrow S)$ **is defined by** $\begin{cases} (\rho; \alpha \leftarrow S)(\alpha) = S \\ (\rho; \alpha \leftarrow S)(\beta) = \rho(\beta) \quad \text{for all } \beta \neq \alpha \end{cases}$

Problem: The implication

$$\Gamma \vdash t : A \quad \Rightarrow \quad t \in [\![A]\!]_\rho$$

cannot be proved directly. (One has to take care of the context)

$\Rightarrow$ Strengthen induction hypothesis using substitutions

Substitutions

Substitutions

Definition (Substitutions)

A substitution is a finite list $\sigma = [x_1 := u_1; \ldots; x_n := u_n]$ where $x_i \neq x_j$ (for $i \neq j$) and $u_i \in \Lambda$

Substitutions

Definition (Substitutions)

A substitution is a finite list $\sigma = [x_1 := u_1; \ldots; x_n := u_n]$ where $x_i \neq x_j$ (for $i \neq j$) and $u_i \in \Lambda$

Application of a substitution σ to a term t is written $t[\sigma]$

Exercise: Define it formally

Substitutions

Definition (Substitutions)

A substitution is a finite list $\sigma = [x_1 := u_1; \ldots; x_n := u_n]$
where $x_i \neq x_j$ (for $i \neq j$) and $u_i \in \Lambda$

Application of a substitution σ to a term t is written $t[\sigma]$

Exercise: Define it formally

Definition (Interpretation of contexts)

For all $\Gamma = x_1 : A_1; \ldots; x_n : A_n$ and $\rho \in \mathrm{TVal}$ set:

$$[\![\Gamma]\!]_\rho = \{\sigma = [x_1 := u_1; \ldots; x_n := u_n];\ u_i \in [\![A_i]\!]_\rho\ (i = 1..n)\}$$

Substitutions $\sigma \in [\![\Gamma]\!]_\rho$ are said to be adapted to the context Γ (in the type valuation ρ)

The strong normalisation invariant

The strong normalisation invariant

Lemma (Strong normalisation invariant)

If $\Gamma \vdash t : A$ in Curry-style system F, then

$$\forall \rho \in \mathsf{TVal} \qquad \forall \sigma \in [\![\Gamma]\!]_\rho \qquad t[\sigma] \in [\![A]\!]_\rho$$

The strong normalisation invariant

Lemma (Strong normalisation invariant)

If $\Gamma \vdash t : A$ in Curry-style system F, then

$$\forall \rho \in \mathsf{TVal} \qquad \forall \sigma \in [\![\Gamma]\!]_\rho \qquad t[\sigma] \in [\![A]\!]_\rho$$

Proof. By induction on the derivation of $\Gamma \vdash t : A$.

Exercise: Write down the 5 cases completely

The strong normalisation invariant

Lemma (Strong normalisation invariant)

If $\Gamma \vdash t : A$ in Curry-style system F, then

$$\forall \rho \in \mathsf{TVal} \qquad \forall \sigma \in [\![\Gamma]\!]_\rho \qquad t[\sigma] \in [\![A]\!]_\rho$$

Proof. By induction on the derivation of $\Gamma \vdash t : A$.

Exercise: Write down the 5 cases completely

Theorem (Strong normalisation)

The typable terms of F-Curry are strongly normalisable

The strong normalisation invariant

Lemma (Strong normalisation invariant)

If $\Gamma \vdash t : A$ in Curry-style system F, then

$$\forall \rho \in \mathsf{TVal} \qquad \forall \sigma \in [\![\Gamma]\!]_\rho \qquad t[\sigma] \in [\![A]\!]_\rho$$

Proof. By induction on the derivation of $\Gamma \vdash t : A$.

Exercise: Write down the 5 cases completely

Theorem (Strong normalisation)

The typable terms of F-Curry are strongly normalisable

Proof. Assume $x_1 : A_1; \ldots; x_n : A_n \vdash t : B$

The strong normalisation invariant

Lemma (Strong normalisation invariant)

If $\Gamma \vdash t : A$ in Curry-style system F, then

$$\forall \rho \in \mathsf{TVal} \qquad \forall \sigma \in [\![\Gamma]\!]_\rho \qquad t[\sigma] \in [\![A]\!]_\rho$$

Proof. By induction on the derivation of $\Gamma \vdash t : A$.

Exercise: Write down the 5 cases completely

Theorem (Strong normalisation)

The typable terms of F-Curry are strongly normalisable

Proof. Assume $x_1 : A_1; \ldots; x_n : A_n \vdash t : B$

Consider an arbitrary type valuation ρ (for instance: $\rho(\alpha) = \mathsf{SN}$ for all α)

The strong normalisation invariant

Lemma (Strong normalisation invariant)

If $\Gamma \vdash t : A$ in Curry-style system F, then

$$\forall \rho \in \mathrm{TVal} \qquad \forall \sigma \in [\![\Gamma]\!]_\rho \qquad t[\sigma] \in [\![A]\!]_\rho$$

Proof. By induction on the derivation of $\Gamma \vdash t : A$.

Exercise: Write down the 5 cases completely

Theorem (Strong normalisation)

The typable terms of F-Curry are strongly normalisable

Proof. Assume $x_1 : A_1; \ldots; x_n : A_n \vdash t : B$

Consider an arbitrary type valuation ρ (for instance: $\rho(\alpha) = \mathrm{SN}$ for all α)

We have: $x_1 \in [\![A_1]\!]_\rho$, $x_2 \in [\![A_2]\!]_\rho$, $\ldots$, $x_n \in [\![A_n]\!]_\rho$ (SAT2)

The strong normalisation invariant

Lemma (Strong normalisation invariant)

If $\Gamma \vdash t : A$ in Curry-style system F, then

$$\forall \rho \in \mathrm{TVal} \qquad \forall \sigma \in [\![\Gamma]\!]_\rho \qquad t[\sigma] \in [\![A]\!]_\rho$$

Proof. By induction on the derivation of $\Gamma \vdash t : A$.

Exercise: Write down the 5 cases completely

Theorem (Strong normalisation)

The typable terms of F-Curry are strongly normalisable

Proof. Assume $x_1 : A_1; \ldots; x_n : A_n \vdash t : B$

Consider an arbitrary type valuation ρ (for instance: $\rho(\alpha) = \mathrm{SN}$ for all α)

We have: $x_1 \in [\![A_1]\!]_\rho$, $x_2 \in [\![A_2]\!]_\rho$, $\ldots$, $x_n \in [\![A_n]\!]_\rho$ (SAT2), hence:

$$\sigma = [x_1 := x_1; \ldots; x_n := x_n] \in [\![x_1 : A_1; \ldots; x_n : A_n]\!]_\rho$$

From the lemma we get $t = t[\sigma] \in [\![B]\!]_\rho$, hence $t \in \mathrm{SN}$ (SAT1)

The strong normalisation invariant

Lemma (Strong normalisation invariant)

If $\Gamma \vdash t : A$ in Curry-style system F, then

$$\forall \rho \in \mathrm{TVal} \qquad \forall \sigma \in [\![\Gamma]\!]_\rho \qquad t[\sigma] \in [\![A]\!]_\rho$$

Proof. By induction on the derivation of $\Gamma \vdash t : A$.

Exercise: Write down the 5 cases completely

Theorem (Strong normalisation)

The typable terms of F-Curry are strongly normalisable

Corollary (Church-style SN)

The typable terms of F-Church are strongly normalisable

A remark on impredicativity

In the SN proof, interpretation of $\forall$ relies on the property:

If $(S_i)_{i \in I}$ $(I \neq \varnothing)$ *is a family of saturated sets,*
then $\bigcap_{i \in I} S_i$ *is a saturated set*

in the special case where $I = \mathbf{SAT}$ (impredicative intersection)

A remark on impredicativity

In the SN proof, interpretation of $\forall$ relies on the property:

If $(S_i)_{i \in I}$ $(I \neq \varnothing)$ *is a family of saturated sets,*
then $\bigcap_{i \in I} S_i$ *is a saturated set*

in the special case where $I = \mathbf{SAT}$ (impredicative intersection)

- In 'classical' mathematics, this construction is legal

A remark on impredicativity

In the SN proof, interpretation of $\forall$ relies on the property:

If $(S_i)_{i \in I}$ $(I \neq \varnothing)$ *is a family of saturated sets,*
then $\bigcap_{i \in I} S_i$ *is a saturated set*

in the special case where $I = \mathbf{SAT}$ (impredicative intersection)

- In 'classical' mathematics, this construction is legal

  $\Rightarrow$ Standard set theories (Z, ZF, ZFC) are impredicative

A remark on impredicativity

In the SN proof, interpretation of $\forall$ relies on the property:

If $(S_i)_{i \in I}$ $(I \neq \varnothing)$ *is a family of saturated sets,*
then $\bigcap_{i \in I} S_i$ *is a saturated set*

in the special case where $I = \mathbf{SAT}$ (impredicative intersection)

- In 'classical' mathematics, this construction is legal

  $\Rightarrow$ Standard set theories (Z, ZF, ZFC) are impredicative

- In (Bishop, Martin-Löf's style) constructive mathematics, this principle is rejected, mainly for philosophical reasons:
  - No convincing 'constructive' explanation
  - Suspicion about (this kind of) cyclicity

Impredicativity: An example (1/2)

Assume E is a vector space, S a set of vectors.
How to define the sub-vector space $\overline{S} \subset E$ generated by S in E ?

Impredicativity: An example (1/2)

Assume E is a vector space, S a set of vectors.
How to define the sub-vector space $\overline{S} \subset E$ generated by S in E ?

Standard 'abstract' method:

Impredicativity: An example (1/2)

Assume E is a vector space, S a set of vectors.
How to define the sub-vector space $\overline{S} \subset E$ generated by S in E ?

Standard 'abstract' method:

1. Consider the set: $\mathfrak{S} = \{F; \quad F \text{ is a sub-vector space of } E \text{ and } F \supset S\}$

Impredicativity: An example (1/2)

Assume E is a vector space, S a set of vectors.
How to define the sub-vector space $\overline{S} \subset E$ generated by S in E ?

Standard 'abstract' method:

1. Consider the set: $\mathfrak{S} = \{F; \quad F \text{ is a sub-vector space of } E \text{ and } F \supset S\}$
2. Fact: $\mathfrak{S}$ is non empty, since $E \in \mathfrak{S}$

Impredicativity: An example (1/2)

Assume E is a vector space, S a set of vectors.
How to define the sub-vector space $\overline{S} \subset E$ generated by S in E ?

Standard 'abstract' method:

1. Consider the set: $\mathfrak{S} = \{F; \quad F \text{ is a sub-vector space of } E \text{ and } F \supset S\}$
2. Fact: $\mathfrak{S}$ is non empty, since $E \in \mathfrak{S}$
3. Take: $\overline{S} = \bigcap_{F \in \mathfrak{S}} F$

Impredicativity: An example (1/2)

Assume E is a vector space, S a set of vectors.
How to define the sub-vector space $\overline{S} \subset E$ generated by S in E ?

Standard 'abstract' method:

1. Consider the set: $\mathfrak{S} = \{F; \quad F \text{ is a sub-vector space of } E \text{ and } F \supset S\}$
2. Fact: $\mathfrak{S}$ is non empty, since $E \in \mathfrak{S}$
3. Take: $\overline{S} = \bigcap_{F \in \mathfrak{S}} F$
4. By definition, S is included in all the sub-spaces of E containing S

Impredicativity: An example (1/2)

Assume E is a vector space, S a set of vectors.
How to define the sub-vector space $\overline{S} \subset E$ generated by S in E ?

Standard 'abstract' method:

1. Consider the set: $\mathfrak{S} = \{F; \quad F \text{ is a sub-vector space of } E \text{ and } F \supset S\}$
2. Fact: $\mathfrak{S}$ is non empty, since $E \in \mathfrak{S}$
3. Take: $\overline{S} = \bigcap_{F \in \mathfrak{S}} F$
4. By definition, S is included in all the sub-spaces of E containing S
5. But $\overline{S}$ is itself a sub-vector space of E containing S (so that $\overline{S} \in \mathfrak{S}$)

Impredicativity: An example (1/2)

Assume E is a vector space, S a set of vectors.
How to define the sub-vector space $\overline{S} \subset E$ generated by S in E ?

Standard 'abstract' method:

1. Consider the set: $\mathfrak{S} = \{F; \quad F \text{ is a sub-vector space of } E \text{ and } F \supset S\}$
2. Fact: $\mathfrak{S}$ is non empty, since $E \in \mathfrak{S}$
3. Take: $\overline{S} = \bigcap_{F \in \mathfrak{S}} F$
4. By definition, S is included in all the sub-spaces of E containing S
5. But $\overline{S}$ is itself a sub-vector space of E containing S (so that $\overline{S} \in \mathfrak{S}$)
6. So that $\overline{S}$ is actually the smallest of all such spaces

Impredicativity: An example (1/2)

Assume E is a vector space, S a set of vectors.
How to define the sub-vector space $\overline{S} \subset E$ generated by S in E ?

Standard 'abstract' method:

1. Consider the set: $\mathfrak{S} = \{F; \quad F \text{ is a sub-vector space of } E \text{ and } F \supset S\}$
2. Fact: $\mathfrak{S}$ is non empty, since $E \in \mathfrak{S}$
3. Take: $\overline{S} = \bigcap_{F \in \mathfrak{S}} F$
4. By definition, S is included in all the sub-spaces of E containing S
5. But $\overline{S}$ is itself a sub-vector space of E containing S (so that $\overline{S} \in \mathfrak{S}$)
6. So that $\overline{S}$ is actually the smallest of all such spaces

This definition is impredicative (step 3) (but legal in 'classical' mathematics)

Impredicativity: An example (1/2)

Assume E is a vector space, S a set of vectors.
How to define the sub-vector space $\overline{S} \subset E$ generated by S in E ?

Standard 'abstract' method:

1. Consider the set: $\mathfrak{S} = \{F; \quad F \text{ is a sub-vector space of } E \text{ and } F \supset S\}$
2. Fact: $\mathfrak{S}$ is non empty, since $E \in \mathfrak{S}$
3. Take: $\overline{S} = \bigcap_{F \in \mathfrak{S}} F$
4. By definition, S is included in all the sub-spaces of E containing S
5. But $\overline{S}$ is itself a sub-vector space of E containing S (so that $\overline{S} \in \mathfrak{S}$)
6. So that $\overline{S}$ is actually the smallest of all such spaces

This definition is impredicative (step 3) (but legal in 'classical' mathematics)

The set $\overline{S}$ is defined *from* $\mathfrak{S}$, $\underbrace{\text{that already contains } \overline{S} \textit{ as an element}}_{\text{discovered a fortiori}}$

Impredicativity: An example (2/2)

But there are other ways of defining $\overline{S}$...

Impredicativity: An example (2/2)

But there are other ways of defining $\overline{S}$...

- **Standard 'concrete' definition, by linear combinations:**

Impredicativity: An example (2/2)

But there are other ways of defining $\overline{S}$...

- **Standard 'concrete' definition, by linear combinations:**

  Let $\overline{S}$ be the set of all vectors of the form $v = \alpha_1 \cdot v_1 + \cdots + \alpha_n \cdot v_n$

  where (v_i) ranges over all the finite families of elements of S,
  and (α_i) ranges over all the finite families of scalars

Impredicativity: An example (2/2)

But there are other ways of defining $\overline{S}$...

- **Standard 'concrete' definition, by linear combinations:**

  Let $\overline{S}$ be the set of all vectors of the form $v = \alpha_1 \cdot v_1 + \cdots + \alpha_n \cdot v_n$

  where (v_i) ranges over all the finite families of elements of S,
  and (α_i) ranges over all the finite families of scalars

- **Inductive definition:**

Impredicativity: An example (2/2)

But there are other ways of defining $\overline{S}$...

- **Standard 'concrete' definition, by linear combinations:**

  Let $\overline{S}$ be the set of all vectors of the form $v = \alpha_1 \cdot v_1 + \cdots + \alpha_n \cdot v_n$

  where (v_i) ranges over all the finite families of elements of S,
  and (α_i) ranges over all the finite families of scalars

- **Inductive definition:**

  Let $\overline{S}$ be the set inductively defined by:

  1. $\vec{0} \in \overline{S}$,
  2. If $v \in S$, then $v \in \overline{S}$,
  3. If $v \in \overline{S}$ and α is a scalar, then $\alpha \cdot v \in \overline{S}$
  4. If $v_1 \in \overline{S}$ and $v_2 \in \overline{S}$, then $v_1 + v_2 \in \overline{S}$.

Impredicativity: An example (2/2)

But there are other ways of defining $\overline{S}$...

- **Standard 'concrete' definition, by linear combinations:**

  Let $\overline{S}$ be the set of all vectors of the form $v = \alpha_1 \cdot v_1 + \cdots + \alpha_n \cdot v_n$

  where (v_i) ranges over all the finite families of elements of S,
  and (α_i) ranges over all the finite families of scalars

- **Inductive definition:**

  Let $\overline{S}$ be the set inductively defined by:

  1. $\vec{0} \in \overline{S}$,
  2. If $v \in S$, then $v \in \overline{S}$,
  3. If $v \in \overline{S}$ and α is a scalar, then $\alpha \cdot v \in \overline{S}$
  4. If $v_1 \in \overline{S}$ and $v_2 \in \overline{S}$, then $v_1 + v_2 \in \overline{S}$.

$\Rightarrow$ Both definitions are predicative (and give the same object)

Normalisation of Second Order Arithmetic

Alexandre Miquel — PPS & U. Paris 7

Alexandre.Miquel@pps.jussieu.fr

Types Summer School 2005

August 15–26 — Göteborg

Syntax of HA2

Syntax of HA2

Variables	$x,\ y, z,\ \ldots$		of individuals	(i.e. natural numbers)
	$\alpha^n,\ \beta^n,\ \gamma^n,\ \ldots$		of predicates	(for each arity $n \geq 0$)
Individuals	t, u	$::=$	$x \quad \mid \quad 0 \quad \mid \quad s(t)$	
Formulæ	A, B	$::=$	$\alpha^n(t_1, \ldots, t_n)$	(for all $n \geq 0$)
		$\mid$	$A \Rightarrow B$	
		$\mid$	$\forall x\ B$	(first-order)
		$\mid$	$\forall \alpha^n\ B$	(second order, for all $n \geq 0$)
Contexts	Γ, Δ	$::=$	$A_1, \ldots, A_n$	(lists of formulæ)

Syntax of HA2

Variables	$x,\ y, z,\ \ldots$		of individuals	(i.e. natural numbers)
	$\alpha^n,\ \beta^n,\ \gamma^n,\ \ldots$		of predicates	(for each arity $n \geq 0$)
Individuals	t, u	$::=$	$x \quad \mid \quad 0 \quad \mid \quad s(t)$	
Formulæ	A, B	$::=$	$\alpha^n(t_1, \ldots, t_n)$	(for all $n \geq 0$)
		$\mid$	$A \Rightarrow B$	
		$\mid$	$\forall x\ B$	(first-order)
		$\mid$	$\forall \alpha^n\ B$	(second order, for all $n \geq 0$)
Contexts	Γ, Δ	$::=$	$A_1, \ldots, A_n$	(lists of formulæ)

- Predicate variables of arity 0 represent propositions
- Predicate variables represent sets (of numerals, of pairs, etc.)
- Real numbers can be represented as predicate variables (intuitionistic analysis)

Substitution

Substitution

- **Term substitution** $u\{x := t\} \quad \Rightarrow \quad$ defined in the usual way

Substitution

- **Term substitution** $u\{x := t\} \quad \Rightarrow \quad$ defined in the usual way
- **First-order substitution** $B\{x := t\} \quad \Rightarrow \quad$ defined in the usual way

Substitution

- **Term substitution** $u\{x := t\} \quad \Rightarrow \quad$ defined in the usual way
- **First-order substitution** $B\{x := t\} \quad \Rightarrow \quad$ defined in the usual way
- **Second-order substitution** $B\{\alpha^n := \lambda x_1, \ldots, x_n \,.\, A\}$

  In the formula B, replace each atomic subformula of the form

  $$\alpha^n(t_1, \ldots, t_n)$$

  by the (substituted) formula

  $$A\{x_1 := t_1; \ldots; x_n := t_n\}$$

Substitution

- **Term substitution** $u\{x := t\} \quad \Rightarrow \quad$ defined in the usual way
- **First-order substitution** $B\{x := t\} \quad \Rightarrow \quad$ defined in the usual way
- **Second-order substitution** $B\{\alpha^n := \lambda x_1, \ldots, x_n \,.\, A\}$

  In the formula B, replace each atomic subformula of the form

  $$\alpha^n(t_1, \ldots, t_n)$$

  by the (substituted) formula

  $$A\{x_1 := t_1; \ldots; x_n := t_n\}$$

The notation '$\lambda x_1, \ldots, x_n \,.\, A$' is not part of the syntax

Encoding missing constructions

Encoding missing constructions

- Other connectives can be encoded:

$$
\begin{array}{lcl}
\top & \equiv & \forall \gamma^0\ (\gamma^0 \Rightarrow \gamma^0) \\
\bot & \equiv & \forall \gamma^0\ \gamma^0 \\
A \wedge B & \equiv & \forall \gamma^0\ ((A \Rightarrow B \Rightarrow \gamma^0) \Rightarrow \gamma^0) \\
A \vee B & \equiv & \forall \gamma^0\ ((A \Rightarrow \gamma^0) \Rightarrow (B \Rightarrow \gamma^0) \Rightarrow \gamma^0) \\
\neg A & \equiv & A \Rightarrow \bot
\end{array}
$$

Encoding missing constructions

- Other connectives can be encoded:

$$
\begin{array}{lcl}
\top & \equiv & \forall \gamma^0\ (\gamma^0 \Rightarrow \gamma^0) \\
\bot & \equiv & \forall \gamma^0\ \gamma^0 \\
A \wedge B & \equiv & \forall \gamma^0\ ((A \Rightarrow B \Rightarrow \gamma^0) \Rightarrow \gamma^0) \\
A \vee B & \equiv & \forall \gamma^0\ ((A \Rightarrow \gamma^0) \Rightarrow (B \Rightarrow \gamma^0) \Rightarrow \gamma^0) \\
\neg A & \equiv & A \Rightarrow \bot
\end{array}
$$

- Existential quantifier (1st + 2nd order)

$$
\begin{array}{lcl}
\exists x\ B[x] & \equiv & \forall \gamma^0\ (\forall x\ (B[x] \Rightarrow \gamma^0)\ \Rightarrow\ \gamma^0) \\
\exists \alpha^n\ B[\alpha^n] & \equiv & \forall \gamma^0\ (\forall \alpha^n\ (B[\alpha^n] \Rightarrow \gamma^0)\ \Rightarrow\ \gamma^0)
\end{array}
$$

Encoding missing constructions

- Other connectives can be encoded:

$$
\begin{array}{lcl}
\top & \equiv & \forall \gamma^0\ (\gamma^0 \Rightarrow \gamma^0) \\
\bot & \equiv & \forall \gamma^0\ \gamma^0 \\
A \wedge B & \equiv & \forall \gamma^0\ ((A \Rightarrow B \Rightarrow \gamma^0) \Rightarrow \gamma^0) \\
A \vee B & \equiv & \forall \gamma^0\ ((A \Rightarrow \gamma^0) \Rightarrow (B \Rightarrow \gamma^0) \Rightarrow \gamma^0) \\
\neg A & \equiv & A \Rightarrow \bot
\end{array}
$$

- Existential quantifier (1st + 2nd order)

$$
\begin{array}{lcl}
\exists x\ B[x] & \equiv & \forall \gamma^0\ (\forall x\ (B[x] \Rightarrow \gamma^0)\ \Rightarrow\ \gamma^0) \\
\exists \alpha^n\ B[\alpha^n] & \equiv & \forall \gamma^0\ (\forall \alpha^n\ (B[\alpha^n] \Rightarrow \gamma^0)\ \Rightarrow\ \gamma^0)
\end{array}
$$

- Leibniz equality:

$$t = u \quad \equiv \quad \forall \gamma^1\ (\gamma^1(t) \Rightarrow \gamma^1(u))$$

Deduction rules of HA2

- General rules for second-order intuitionistic logic:

$$\frac{}{\Gamma \vdash A}\ {\scriptstyle A \in \Gamma}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad \frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash \forall x\ B}\ {\scriptstyle x \notin FV_1(\Gamma)} \qquad \frac{\Gamma \vdash \forall x\ B}{\Gamma \vdash B\{x := t\}}$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash \forall \alpha^n\ B}\ {\scriptstyle \alpha^n \notin FV_2(\Gamma)} \qquad \frac{\Gamma \vdash \forall \alpha^n\ B}{\Gamma \vdash B\{\alpha := \lambda x_1, \ldots, x_n\,.\,A\}}$$

Deduction rules of HA2

- General rules for second-order intuitionistic logic:

$$\frac{}{\Gamma \vdash A}\ A \in \Gamma$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad \frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash \forall x\ B}\ x \notin FV_1(\Gamma) \qquad \frac{\Gamma \vdash \forall x\ B}{\Gamma \vdash B\{x := t\}}$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash \forall \alpha^n\ B}\ \alpha^n \notin FV_2(\Gamma) \qquad \frac{\Gamma \vdash \forall \alpha^n\ B}{\Gamma \vdash B\{\alpha := \lambda x_1, \ldots, x_n\,.\,A\}}$$

- Specific rules (axioms) for arithmetic:

$$\frac{}{\Gamma \vdash \forall x\ \forall y\ (s(x) = s(y) \Rightarrow x = y)} \qquad \frac{}{\Gamma \vdash \forall x\ \neg\ s(x) = 0}$$

Remember that constructions '$t = u$' and '$\neg A$' are not primitive, but encoded!

Derivable rules (1/2)

Logical deduction rules of HA2 only talk about the primitive constructions '$\Rightarrow$' and '$\forall$' (implication + 1st/2nd-order universal quantification)

Derivable rules (1/2)

Logical deduction rules of HA2 only talk about the primitive constructions '$\Rightarrow$' and '$\forall$' (implication + 1st/2nd-order universal quantification)

But in this framework, the other constructions ($\top$, $\bot$, $\wedge$, $\vee$, $\exists$ etc.) are definable and their (standard) deduction rules can be derived:

Derivable rules (1/2)

Logical deduction rules of HA2 only talk about the primitive constructions '$\Rightarrow$' and '$\forall$' (implication + 1st/2nd-order universal quantification)

But in this framework, the other constructions ($\top$, $\bot$, $\wedge$, $\vee$, $\exists$ etc.) are definable and their (standard) deduction rules can be derived:

- Logical connectives: $\top$, $\bot$ and $\wedge$

$$\frac{}{\Gamma \vdash \top} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash C}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

Derivable rules (2/2)

- Logical connectives: $\vee$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$$

$$\frac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C \qquad \Gamma \vdash A \vee B}{\Gamma \vdash C}$$

Derivable rules (2/2)

- Logical connectives: $\vee$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$$

$$\frac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C \qquad \Gamma \vdash A \vee B}{\Gamma \vdash C}$$

- Existential quantifier: 1st and 2nd-order

$$\frac{\Gamma \vdash B\{x := t\}}{\Gamma \vdash \exists x\ B} \qquad \frac{\Gamma, B \vdash C \qquad \Gamma \vdash \exists x\ B}{\Gamma \vdash C}\ x \notin FV_1(\Gamma, C)$$

$$\frac{\Gamma \vdash B\{\alpha^n := \lambda x_1, \ldots, x_n\,.\,A\}}{\Gamma \vdash \exists \alpha^n\ B} \qquad \frac{\Gamma, B \vdash C \qquad \Gamma \vdash \exists \alpha^n\ B}{\Gamma \vdash C}\ \alpha^n \notin FV_2(\Gamma, C)$$

Equality rules

Leibniz equality is defined as: $t = u \quad \equiv \quad \forall \gamma^1 \ (\gamma^1(t) \Rightarrow \gamma^1(u))$

Equality rules

Leibniz equality is defined as: $t = u \quad \equiv \quad \forall \gamma^1 \ (\gamma^1(t) \Rightarrow \gamma^1(u))$

- The following formulæ are provable (by purely logical means):

$$\forall x \ (x = x)$$

$$\forall x \ \forall y \ (x = y \ \Rightarrow \ y = x)$$

$$\forall x \ \forall y \ \forall z \ (x = y \ \Rightarrow \ y = z \ \Rightarrow \ x = z)$$

$$\forall \alpha^1 \ \forall x \ \forall y \ (\alpha^1(x) \ \Rightarrow \ x = y \ \Rightarrow \ \alpha^1(y))$$

Equality rules

Leibniz equality is defined as: $t = u \quad \equiv \quad \forall \gamma^1 \ (\gamma^1(t) \Rightarrow \gamma^1(u))$

- The following formulæ are provable (by purely logical means):

$$\forall x \ (x = x)$$

$$\forall x \ \forall y \ (x = y \ \Rightarrow \ y = x)$$

$$\forall x \ \forall y \ \forall z \ (x = y \ \Rightarrow \ y = z \ \Rightarrow \ x = z)$$

$$\forall \alpha^1 \ \forall x \ \forall y \ (\alpha^1(x) \ \Rightarrow \ x = y \ \Rightarrow \ \alpha^1(y))$$

- Moreover, HA2 assumes the following two axioms:

(Injectivity) $\forall x \ \forall y \ (s(x) = s(y) \ \Rightarrow \ x = y)$

(Non-surjectivity) $\forall x \ \neg \ (s(x) = 0)$

Induction principle

Induction principle

- Induction can be recovered via the predicate:

$$\mathsf{Nat}(x) \quad \equiv \quad \forall \alpha^{1} \left(\alpha^{1}(0) \;\Rightarrow\; \forall y \; \left(\alpha^{1}(y) \Rightarrow \alpha^{1}(s(y)) \right) \;\Rightarrow\; \alpha^{1}(x) \right)$$

Induction principle

- Induction can be recovered via the predicate:

$$\mathsf{Nat}(x) \quad \equiv \quad \forall \alpha^1 \left(\alpha^1(0) \;\Rightarrow\; \forall y \; \left(\alpha^1(y) \Rightarrow \alpha^1(s(y)) \right) \;\Rightarrow\; \alpha^1(x) \right)$$

$\Rightarrow$ defines the smallest class containing zero and closed under successor

Induction principle

- Induction can be recovered via the predicate:

$$\mathrm{Nat}(x) \quad \equiv \quad \forall \alpha^1 \left(\alpha^1(0) \;\Rightarrow\; \forall y \; (\alpha^1(y) \Rightarrow \alpha^1(s(y))) \;\Rightarrow\; \alpha^1(x) \right)$$

  $\Rightarrow$ defines the smallest class containing zero and closed under successor

- In particular, we have: $\mathrm{Nat}(0)$ and $\forall x \; (\mathrm{Nat}(x) \Rightarrow \mathrm{Nat}(s(x)))$

Induction principle

- Induction can be recovered via the predicate:

$$\mathrm{Nat}(x) \quad \equiv \quad \forall \alpha^1 \left(\alpha^1(0) \;\Rightarrow\; \forall y \; (\alpha^1(y) \Rightarrow \alpha^1(s(y))) \;\Rightarrow\; \alpha^1(x) \right)$$

  $\Rightarrow$ defines the smallest class containing zero and closed under successor

- In particular, we have: $\mathrm{Nat}(0)$ and $\forall x \; (\mathrm{Nat}(x) \Rightarrow \mathrm{Nat}(s(x)))$

- All the first-order quantifications should be restricted to this class:

  $\Rightarrow$ Systematically use $\forall x \; (\mathrm{Nat}(x) \Rightarrow A)$ and $\exists x \; (\mathrm{Nat}(x) \wedge A)$

Induction principle

- Induction can be recovered via the predicate:

$$\mathsf{Nat}(x) \quad \equiv \quad \forall \alpha^1 \left(\alpha^1(0) \;\Rightarrow\; \forall y \; (\alpha^1(y) \Rightarrow \alpha^1(s(y))) \;\Rightarrow\; \alpha^1(x) \right)$$

  $\Rightarrow$ defines the smallest class containing zero and closed under successor

- In particular, we have: $\mathsf{Nat}(0)$ and $\forall x \; (\mathsf{Nat}(x) \Rightarrow \mathsf{Nat}(s(x)))$

- All the first-order quantifications should be restricted to this class:

  $\Rightarrow$ Systematically use $\forall x \; (\mathsf{Nat}(x) \Rightarrow A)$ and $\exists x \; (\mathsf{Nat}(x) \wedge A)$

- Thanks to this trick, induction becomes provable:

$$\forall \alpha^1 \left(\alpha^1(0) \;\Rightarrow\; \forall x \; (\mathsf{Nat}(x) \Rightarrow \alpha^1(x) \Rightarrow \alpha^1(s(x))) \;\Rightarrow\; \forall x \; (\mathsf{Nat}(x) \Rightarrow \alpha^1(x)) \right)$$

The notion of cut (1/2)

The notion of cut (1/2)

- A cut is a piece of a proof constituted by an introduction rule immediately followed by the corresponding elimination rule

The notion of cut (1/2)

- A cut is a piece of a proof constituted by an introduction rule immediately followed by the corresponding elimination rule

- Each cut can be contracted in order to make the reasoning more direct...
  ... but not necessarily shorter [And actually, usually larger!]

The notion of cut (1/2)

- A cut is a piece of a proof constituted by an introduction rule immediately followed by the corresponding elimination rule

- Each cut can be contracted in order to make the reasoning more direct...
  ... but not necessarily shorter [And actually, usually larger!]

- **Implication cut:**

$$\dfrac{\dfrac{\begin{array}{c}[\Gamma, A, \Gamma' \vdash A] \\ \vdots \\ \Gamma, A \vdash B\end{array}}{\Gamma \vdash A \Rightarrow B} \quad \begin{array}{c}\vdots \\ \Gamma \vdash A\end{array}}{\Gamma \vdash B} \quad \rightsquigarrow \quad \begin{array}{c}\vdots \\ \Gamma, \Gamma' \vdash A \\ \vdots \\ \Gamma \vdash B\end{array}$$

Here, $[\Gamma, A, \Gamma' \vdash A]$ represents all the instances of an axiom with the formula A in the proof π_1. (Such instances may occur in extended contexts of the form Γ, A, Γ'.)
These instances are then used as placeholders that are filled by the proof π_2 during the contraction of the cut (after some weakenings due to the presence of extra contexts Γ')

The notion of cut (2/2)

- **Cut of the 1st-order universal quantification**:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\ \pi\\ \Gamma \vdash B\end{array}}{\Gamma \vdash \forall x\,.\,B}}{\Gamma \vdash B\{x := t\}} \quad \rightsquigarrow \quad \begin{array}{c}\vdots\ \pi\{x{:=}t\}\\ \Gamma \vdash B\{x := t\}\end{array}$$

The first piece of proof is replaced by the proof π in which the 1st-order variable x is replaced by the term t recursively. Notice that the substitution has no effect on Γ, since $x \notin FV(\Gamma)$. (Of course, the substitution has to be performed on each context too.)

The notion of cut (2/2)

- **Cut of the 1st-order universal quantification**:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\ \pi\\ \Gamma \vdash B\end{array}}{\Gamma \vdash \forall x\,.\,B}}{\Gamma \vdash B\{x := t\}} \quad \rightsquigarrow \quad \begin{array}{c}\vdots\ \pi\{x{:=}t\}\\ \Gamma \vdash B\{x := t\}\end{array}$$

The first piece of proof is replaced by the proof π in which the 1st-order variable x is replaced by the term t recursively. Notice that the substitution has no effect on Γ, since $x \notin FV(\Gamma)$. (Of course, the substitution has to be performed on each context too.)

- **Cut of the 2nd-order universal quantification**:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\ \pi\\ \Gamma \vdash B\end{array}}{\Gamma \vdash \forall \alpha^{n}\,.\,B}}{\Gamma \vdash B\{\alpha^{n} := \lambda x_1, \ldots, x_n\,.\,A\}} \quad \rightsquigarrow \quad \begin{array}{c}\vdots\ \pi\{\alpha^{n}{:=}\cdots\}\\ \Gamma \vdash B\{\alpha^{n} := \lambda x_1, \ldots, x_n\,.\,A\}\end{array}$$

Same principle, but with a 2nd-order substitution (ie. with a predicate $\lambda x_1, \ldots, x_n\,.\,A$)

Derived cuts

Derived cuts

From the encoding of the connectives $\wedge$ and $\vee$, one can derive other cuts:

Derived cuts

From the encoding of the connectives $\wedge$ and $\vee$, one can derive other cuts:

- **Cuts of the conjunction:**

$$\dfrac{\dfrac{\overset{\pi_1}{\vdots}\ \Gamma \vdash A \qquad \overset{\pi_2}{\vdots}\ \Gamma \vdash B}{\Gamma \vdash A \wedge B}}{\Gamma \vdash A} \quad \rightsquigarrow \quad \overset{\pi_1}{\vdots}\ \Gamma \vdash A \qquad (+ \text{ symmetric cut with } \wedge\text{-elim}_2)$$

Derived cuts

From the encoding of the connectives $\wedge$ and $\vee$, one can derive other cuts:

- **Cuts of the conjunction:**

$$\dfrac{\dfrac{\vdots\,\pi_1 \quad \vdots\,\pi_2}{\Gamma \vdash A \qquad \Gamma \vdash B}}{\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}} \quad \rightsquigarrow \quad \begin{array}{c}\vdots\,\pi_1\\ \Gamma \vdash A\end{array} \qquad (+ \text{ symmetric cut with } \wedge\text{-elim}_2)$$

- **Cuts of the disjunction:**

$$\dfrac{\dfrac{[\Gamma, A, \Gamma' \vdash A] \qquad [\Gamma, B, \Gamma' \vdash B] \qquad \vdots\,\pi}{\vdots\,\pi_1 \quad \vdots\,\pi_2 \qquad \Gamma \vdash A}}{\Gamma, A \vdash C \qquad \Gamma, B \vdash C \qquad \Gamma \vdash A \vee B}}{\Gamma \vdash C} \quad \rightsquigarrow \quad \begin{array}{c}\vdots\,\pi\\ \Gamma, \Gamma' \vdash A\\ \vdots\,\pi_1\\ \Gamma \vdash C\end{array}$$

$(+ \text{ symmetric cut with } \vee\text{-intro}_2)$

Filling placeholders in π_1 with π is done in the same way as for the cut of implication

Cut-free proofs

Cut-free proofs

A cut-free proof is a proof that contains no cut

$\Rightarrow$ Cut-free proofs have a simpler structure that make them easier to analyse

Cut-free proofs

A cut-free proof is a proof that contains no cut

$\Rightarrow$ Cut-free proofs have a simpler structure that make them easier to analyse

Fact (Cut-free consistency)

Cut-free proofs

A cut-free proof is a proof that contains no cut

$\Rightarrow$ Cut-free proofs have a simpler structure that make them easier to analyse

Fact (Cut-free consistency)

1. *If π is a cut-free proof of the formula $t = u$ $[\equiv \forall \alpha^1 \, (\alpha^1(t) \Rightarrow \alpha^1(u))]$ in the empty context, then the terms t and u are syntactically identical*

Cut-free proofs

A cut-free proof is a proof that contains no cut

$\Rightarrow$ Cut-free proofs have a simpler structure that make them easier to analyse

Fact (Cut-free consistency)

1. *If π is a cut-free proof of the formula $t = u$ $[\equiv \forall\alpha^1\ (\alpha^1(t) \Rightarrow \alpha^1(u))]$ in the empty context, then the terms t and u are syntactically identical*
2. *There is no cut-free proof of $\bot$ $[\equiv \forall\alpha^0\ \alpha^0]$ in the empty context*

Cut-free proofs

A cut-free proof is a proof that contains no cut

$\Rightarrow$ Cut-free proofs have a simpler structure that make them easier to analyse

Fact (Cut-free consistency)

1. *If π is a cut-free proof of the formula $t = u$ $[\equiv \forall\alpha^1\ (\alpha^1(t) \Rightarrow \alpha^1(u))]$ in the empty context, then the terms t and u are syntactically identical*
2. *There is no cut-free proof of $\bot$ $[\equiv \forall\alpha^0\ \alpha^0]$ in the empty context*

Proof. Both properties are proved simultaneously by induction on the size of the cut-free proof. Notice that a cut-free proof of $\vdash t = t$ has one of the following two forms:

$$\dfrac{\dfrac{\dfrac{}{\alpha^1(t) \vdash \alpha^1(t)}}{\vdash \alpha^1(t) \Rightarrow \alpha^1(t)}}{\underbrace{\vdash \forall\alpha^1\ (\alpha^1(t) \Rightarrow \alpha^1(t))}_{t=t}}$$

$$\dfrac{\dfrac{\dfrac{}{\vdash \forall x\ \forall y\ (s(x) = s(y) \Rightarrow x = y)}}{\vdash \forall y\ (s(t) = s(y) \Rightarrow t = y)}}{\vdash s(t) = s(t) \Rightarrow t = t} \qquad \begin{array}{c}\text{(cut-free)}\\ \vdots \\ \vdash s(t) = s(t)\end{array}$$

$$\vdash t = t$$

Cut-free proofs

A cut-free proof is a proof that contains no cut

$\Rightarrow$ Cut-free proofs have a simpler structure that make them easier to analyse

Fact (Cut-free consistency)

1. *If π is a cut-free proof of the formula $t = u$ $[\equiv \forall\alpha^1\ (\alpha^1(t) \Rightarrow \alpha^1(u))]$ in the empty context, then the terms t and u are syntactically identical*
2. *There is no cut-free proof of $\bot$ $[\equiv \forall\alpha^0\ \alpha^0]$ in the empty context*

Proof. Both properties are proved simultaneously by induction on the size of the cut-free proof. Notice that a cut-free proof of $\vdash t = t$ has one of the following two forms:

$$\dfrac{\dfrac{\dfrac{}{\alpha^1(t) \vdash \alpha^1(t)}}{\vdash \alpha^1(t) \Rightarrow \alpha^1(t)}}{\underbrace{\vdash \forall\alpha^1\ (\alpha^1(t) \Rightarrow \alpha^1(t))}_{t=t}}$$

$$\dfrac{\dfrac{\dfrac{}{\vdash \forall x\ \forall y\ (s(x) = s(y) \Rightarrow x = y)}}{\vdash \forall y\ (s(t) = s(y) \Rightarrow t = y)}}{\vdash s(t) = s(t) \Rightarrow t = t} \qquad \begin{array}{c}\text{(cut-free)}\\ \vdots \\ \vdash s(t) = s(t)\end{array}$$

$$\vdash t = t$$

$\Rightarrow$ Reasoning on cut-free proofs is purely combinatorial

Cut-elimination

Cut-elimination

- $\bot \equiv \forall \alpha^0\ \alpha^0$ has no cut-free proof (in the empty context)
  $\Rightarrow$ Means that a proof of $\bot$ necessarily contains at least one cut

Cut-elimination

- $\bot \equiv \forall \alpha^0\ \alpha^0$ has no cut-free proof (in the empty context)
  $\Rightarrow$ Means that a proof of $\bot$ necessarily contains at least one cut
- But each cut can be individually contracted
  (Keeping in mind that contracting a cut may produce several new cuts)

Cut-elimination

- $\bot \equiv \forall \alpha^0\ \alpha^0$ has no cut-free proof (in the empty context)
  $\Rightarrow$ Means that a proof of $\bot$ necessarily contains at least one cut
- But each cut can be individually contracted
  (Keeping in mind that contracting a cut may produce several new cuts)

Question [Takeuti]

Is there a strategy for contracting cuts in a proof such that the process converges to a cut-free proof ?

Cut-elimination

- $\bot \equiv \forall\alpha^0\ \alpha^0$ has no cut-free proof (in the empty context)
  $\Rightarrow$ Means that a proof of $\bot$ necessarily contains at least one cut
- But each cut can be individually contracted
  (Keeping in mind that contracting a cut may produce several new cuts)

Question [Takeuti]

Is there a strategy for contracting cuts in a proof such that the process converges to a cut-free proof ?

Theorem (Cut-elimination [Girard])

Any strategy for contracting cuts converges to a cut-free proof (in a finite number of contraction steps)

Cut-elimination

- $\bot \equiv \forall \alpha^0\ \alpha^0$ has no cut-free proof (in the empty context)
  $\Rightarrow$ Means that a proof of $\bot$ necessarily contains at least one cut
- But each cut can be individually contracted
  (Keeping in mind that contracting a cut may produce several new cuts)

Question [Takeuti]

Is there a strategy for contracting cuts in a proof such that the process converges to a cut-free proof ?

Theorem (Cut-elimination [Girard])

Any strategy for contracting cuts converges to a cut-free proof (in a finite number of contraction steps)

Corollary (Cut-free proofs & Consistency)

1. *Any proposition that has a proof has also a cut-free proof*
2. *The proposition $\bot$ has no proof in the empty context*

Outline of the proof

Outline of the proof

Idea: Deduce cut-elimination of HA2 from strong normalisation of system F

Outline of the proof

Idea: Deduce cut-elimination of HA2 from strong normalisation of system F

1. Map each formula A of HA2 to a type A^* of system F

Outline of the proof

Idea: Deduce cut-elimination of HA2 from strong normalisation of system F

1. Map each formula A of HA2 to a type A^* of system F
2. Map each logical context Γ of HA2 to a typing context Γ^* of system F

Outline of the proof

Idea: Deduce cut-elimination of HA2 from strong normalisation of system F

1. Map each formula A of HA2 to a type A^* of system F
2. Map each logical context Γ of HA2 to a typing context Γ^* of system F
3. Map each proof π of a sequent $\Gamma \vdash A$ in HA2 to a term π^* of system F such that the judgement $\Gamma^* \vdash \pi^* : A^*$ is derivable

Outline of the proof

Idea: Deduce cut-elimination of HA2 from strong normalisation of system F

1. Map each formula A of HA2 to a type A^* of system F
2. Map each logical context Γ of HA2 to a typing context Γ^* of system F
3. Map each proof π of a sequent $\Gamma \vdash A$ in HA2 to a term π^* of system F such that the judgement $\Gamma^* \vdash \pi^* : A^*$ is derivable
4. Check that each cut of π becomes a redex in π^*

Outline of the proof

Idea: Deduce cut-elimination of HA2 from strong normalisation of system F

1. Map each formula A of HA2 to a type A^* of system F
2. Map each logical context Γ of HA2 to a typing context Γ^* of system F
3. Map each proof π of a sequent $\Gamma \vdash A$ in HA2 to a term π^* of system F such that the judgement $\Gamma^* \vdash \pi^* : A^*$ is derivable
4. Check that each cut of π becomes a redex in π^*

   [**Note:** this works only for $\Rightarrow$-cuts and 2nd-order $\forall$-cuts. The case of 1st-order $\forall$-cuts is treated separately, using a combinatorial argument similar to the one we used for 2nd-kind redexes, when we proved that SN(F-Curry) entails SN(F-Church)]

Outline of the proof

Idea: Deduce cut-elimination of HA2 from strong normalisation of system F

1. Map each formula A of HA2 to a type A^* of system F
2. Map each logical context Γ of HA2 to a typing context Γ^* of system F
3. Map each proof π of a sequent $\Gamma \vdash A$ in HA2 to a term π^* of system F such that the judgement $\Gamma^* \vdash \pi^* : A^*$ is derivable
4. Check that each cut of π becomes a redex in π^*

   [**Note:** this works only for $\Rightarrow$-cuts and 2nd-order $\forall$-cuts. The case of 1st-order $\forall$-cuts is treated separately, using a combinatorial argument similar to the one we used for 2nd-kind redexes, when we proved that SN(F-Curry) entails SN(F-Church)]

5. Conclude that cuts can be eliminated in any proof of HA2 (using any strategy)

Translating HA2 formulæ (1/2)

Translating HA2 formulæ (1/2)

- Each predicate variable of HA2 is mapped to a type variable of system F

Translating HA2 formulæ (1/2)

- Each predicate variable of HA2 is mapped to a type variable of system F

  (We keep the same names for simplicity)

Translating HA2 formulæ (1/2)

- Each predicate variable of HA2 is mapped to a type variable of system F

  (We keep the same names for simplicity)

- Formulæ of HA2 are translated into the types of system F:

$$
\begin{array}{lcl}
(\alpha^n(t_1, \ldots, t_n))^* & \equiv & \alpha \\
(A \Rightarrow B)^* & \equiv & A^* \to B^* \\
(\forall x \,.\, B)^* & \equiv & B^* \\
(\forall \alpha^n \,.\, B)^* & \equiv & \forall \alpha \; B
\end{array}
$$

Translating HA2 formulæ (1/2)

- Each predicate variable of HA2 is mapped to a type variable of system F

  (We keep the same names for simplicity)

- Formulæ of HA2 are translated into the types of system F:

$$\begin{array}{lcl} (\alpha^n(t_1, \ldots, t_n))^* & \equiv & \alpha \\ (A \Rightarrow B)^* & \equiv & A^* \to B^* \\ (\forall x\,.\,B)^* & \equiv & B^* \\ (\forall \alpha^n\,.\,B)^* & \equiv & \forall \alpha\ B \end{array}$$

- **Remarks:** – arity of predicate variables is lost
  – all the first-order constructions disappear

  $\Rightarrow$ The translation only preserves (pure) second-order constructions

Translating HA2 formulæ (1/2)

- Each predicate variable of HA2 is mapped to a type variable of system F

  (We keep the same names for simplicity)

- Formulæ of HA2 are translated into the types of system F:

$$\begin{array}{lcl} (\alpha^n(t_1,\ldots,t_n))^* & \equiv & \alpha \\ (A \Rightarrow B)^* & \equiv & A^* \to B^* \\ (\forall x\,.\,B)^* & \equiv & B^* \\ (\forall \alpha^n\,.\,B)^* & \equiv & \forall \alpha\ B \end{array}$$

- **Remarks:** – arity of predicate variables is lost
  – all the first-order constructions disappear

  $\Rightarrow$ The translation only preserves (pure) second-order constructions

- **Substitutivity:**

$$\begin{array}{lcl} (B\{x := t\}) & \equiv & A^* \\ (B\{\alpha^n := \lambda x_1,\ldots,x_n\,.\,A\})^* & \equiv & B^*\{\alpha := A^*\} \end{array}$$

Translating HA2 formulæ (2/2)

Translating HA2 formulæ (2/2)

- We can test the translation on derived formulæ:

$$(A \wedge B)^* \equiv A^* \times B^* \qquad \text{(cartesian product of system } F\text{)}$$

$$(A \vee B)^* \equiv A^* + B^* \qquad \text{(disjoint union)}$$

$$(t = u)^* \equiv (\forall \alpha^1\ \alpha^1(t) \Rightarrow \alpha^1(u))^* \equiv \forall \alpha\ \alpha \to \alpha \equiv \mathsf{Unit}$$

Translating HA2 formulæ (2/2)

- We can test the translation on derived formulæ:

$$(A \wedge B)^* \equiv A^* \times B^* \qquad \text{(cartesian product of system } F\text{)}$$

$$(A \vee B)^* \equiv A^* + B^* \qquad \text{(disjoint union)}$$

$$(t = u)^* \equiv (\forall \alpha^1 \; \alpha^1(t) \Rightarrow \alpha^1(u))^* \equiv \forall \alpha \; \alpha \to \alpha \equiv \text{Unit}$$

$\Rightarrow$ Equality proofs have no computational contents

Translating HA2 formulæ (2/2)

- We can test the translation on derived formulæ:

$$(A \wedge B)^* \equiv A^* \times B^* \qquad \text{(cartesian product of system } F\text{)}$$

$$(A \vee B)^* \equiv A^* + B^* \qquad \text{(disjoint union)}$$

$$(t = u)^* \equiv (\forall \alpha^1 \; \alpha^1(t) \Rightarrow \alpha^1(u))^* \equiv \forall \alpha \; \alpha \to \alpha \equiv \text{Unit}$$

$\Rightarrow$ Equality proofs have no computational contents

- **Translation of contexts:** Each logical context

$$\Gamma \equiv A_1, \ldots, A_n$$

is translated into a typing context of system F

$$\Gamma^* \equiv \xi_1 : A_1^*, \ldots, \xi_n : A_n^*$$

by associating a term variable ξ_i (a 'name') to each hypothesis

Translating proofs (1/4)

Translating proofs (1/4)

Principle: Translate each proof π of a sequent $\Gamma \vdash A$ into a term π^* such that $\Gamma^* \vdash \pi^* : A^*$ is derivable

Translating proofs (1/4)

Principle: Translate each proof π of a sequent $\Gamma \vdash A$ into a term π^* such that $\Gamma^* \vdash \pi^* : A^*$ is derivable

- **Axiom:**

$$\left(\frac{}{\Gamma, A \vdash A} \right)^* = \xi$$

where ξ is the variable associated to the formula A in the context Γ, A

Translating proofs (1/4)

Principle: Translate each proof π of a sequent $\Gamma \vdash A$ into a term π^* such that $\Gamma^* \vdash \pi^* : A^*$ is derivable

- **Axiom:**

$$\left(\dfrac{}{\Gamma, A \vdash A} \right)^* \quad = \quad \xi$$

where ξ is the variable associated to the formula A in the context Γ, A

- **Introduction of the implication:**

$$\left(\dfrac{\begin{array}{c}\vdots\ \pi \\ \Gamma, A \vdash B\end{array}}{\Gamma \vdash A \Rightarrow B} \right)^* \quad = \quad \lambda \xi : A^* \,.\, \pi^*$$

where ξ is the variable associated to A in the context Γ, A

Translating proofs (2/4)

- **Elimination of the implication:**

$$\left(\dfrac{\begin{array}{cc} \vdots\,\pi_1 & \vdots\,\pi_2 \\ \Gamma \vdash A \Rightarrow B & \Gamma \vdash A \end{array}}{\Gamma \vdash B} \right)^* \quad = \quad \pi_1^* \pi_2^*$$

Translating proofs (2/4)

- **Elimination of the implication:**

$$\left(\dfrac{\begin{array}{cc} \vdots\,{\scriptstyle \pi_1} & \vdots\,{\scriptstyle \pi_2} \\ \Gamma \vdash A \Rightarrow B & \Gamma \vdash A \end{array}}{\Gamma \vdash B} \right)^* \quad = \quad \pi_1^* \pi_2^*$$

- **Introduction of the 1st-order universal quantification:**

$$\left(\dfrac{\begin{array}{c} \vdots\,{\scriptstyle \pi} \\ \Gamma \vdash B \end{array}}{\Gamma \vdash \forall x\ B} \right)^* \quad = \quad \pi^*$$

Translating proofs (2/4)

- **Elimination of the implication:**

$$\left(\dfrac{\begin{array}{cc} \vdots\,\pi_1 & \vdots\,\pi_2 \\ \Gamma \vdash A \Rightarrow B & \Gamma \vdash A \end{array}}{\Gamma \vdash B} \right)^* \quad = \quad \pi_1^* \pi_2^*$$

- **Introduction of the 1st-order universal quantification:**

$$\left(\dfrac{\begin{array}{c} \vdots\,\pi \\ \Gamma \vdash B \end{array}}{\Gamma \vdash \forall x\ B} \right)^* \quad = \quad \pi^*$$

- **Elimination of the 1st-order universal quantification:**

$$\left(\dfrac{\begin{array}{c} \vdots\,\pi \\ \Gamma \vdash \forall x\ B \end{array}}{\Gamma \vdash B\{x := t\}} \right)^* \quad = \quad \pi^*$$

Remark: 1st-order $\forall$-intro/elim are invisible in the extracted system F term

Translating proofs (3/4)

- **Introduction of the 2nd-order universal quantification:**

$$\left(\dfrac{\begin{array}{c}\vdots\ \pi \\ \Gamma \vdash B\end{array}}{\Gamma \vdash \forall \alpha^{n}\ B} \right)^{*} \quad = \quad \Lambda \alpha \,.\, \pi^{*}$$

Translating proofs (3/4)

- **Introduction of the 2nd-order universal quantification:**

$$\left(\dfrac{\begin{array}{c}\vdots\ \pi \\ \Gamma \vdash B\end{array}}{\Gamma \vdash \forall \alpha^{n}\ B} \right)^{*} \quad = \quad \Lambda \alpha \,.\, \pi^{*}$$

- **Elimination of the 2nd-order universal quantification:**

$$\left(\dfrac{\begin{array}{c}\vdots\ \pi \\ \Gamma \vdash \forall \alpha^{n}\ B\end{array}}{\Gamma \vdash B\{\alpha^{n} := \lambda x_1, \ldots, x_n \,.\, A\}} \right)^{*} \quad = \quad \pi^{*} A^{*}$$

Translating proofs (3/4)

- **Introduction of the 2nd-order universal quantification:**

$$\left(\dfrac{\begin{array}{c}\vdots\ \pi \\ \Gamma \vdash B\end{array}}{\Gamma \vdash \forall \alpha^{n}\ B} \right)^{*} \quad = \quad \Lambda \alpha \,.\, \pi^{*}$$

- **Elimination of the 2nd-order universal quantification:**

$$\left(\dfrac{\begin{array}{c}\vdots\ \pi \\ \Gamma \vdash \forall \alpha^{n}\ B\end{array}}{\Gamma \vdash B\{\alpha^{n} := \lambda x_1, \ldots, x_n \,.\, A\}} \right)^{*} \quad = \quad \pi^{*} A^{*}$$

Properties:

Each stage preserves the invariant $\Gamma^{*} \vdash \pi^{*} : A^{*}$

Translating proofs (3/4)

- **Introduction of the 2nd-order universal quantification:**

$$\left(\dfrac{\begin{array}{c}\vdots\ \pi \\ \Gamma \vdash B\end{array}}{\Gamma \vdash \forall \alpha^{n}\ B} \right)^{*} \quad = \quad \Lambda \alpha \,.\, \pi^{*}$$

- **Elimination of the 2nd-order universal quantification:**

$$\left(\dfrac{\begin{array}{c}\vdots\ \pi \\ \Gamma \vdash \forall \alpha^{n}\ B\end{array}}{\Gamma \vdash B\{\alpha^{n} := \lambda x_1, \ldots, x_n \,.\, A\}} \right)^{*} \quad = \quad \pi^{*} A^{*}$$

Properties:

Each stage preserves the invariant $\Gamma^{*} \vdash \pi^{*} : A^{*}$

1. Cuts of *implication* become 1st-kind redexes

Translating proofs (3/4)

- **Introduction of the 2nd-order universal quantification:**

$$\left(\dfrac{\begin{array}{c}\vdots\ \pi \\ \Gamma \vdash B\end{array}}{\Gamma \vdash \forall \alpha^n\ B} \right)^* \quad = \quad \Lambda \alpha\,.\,\pi^*$$

- **Elimination of the 2nd-order universal quantification:**

$$\left(\dfrac{\begin{array}{c}\vdots\ \pi \\ \Gamma \vdash \forall \alpha^n\ B\end{array}}{\Gamma \vdash B\{\alpha^n := \lambda x_1, \ldots, x_n\,.\,A\}} \right)^* \quad = \quad \pi^* A^*$$

Properties:

Each stage preserves the invariant $\Gamma^* \vdash \pi^* : A^*$

1. Cuts of *implication* become 1st-kind redexes
2. Cuts of *2nd-order universal quantification* become 2nd-kind redexes ...

Translating proofs (3/4)

- **Introduction of the 2nd-order universal quantification:**

$$\left(\dfrac{\begin{array}{c}\vdots\ \pi \\ \Gamma \vdash B\end{array}}{\Gamma \vdash \forall \alpha^n\ B} \right)^* \quad = \quad \Lambda \alpha\,.\,\pi^*$$

- **Elimination of the 2nd-order universal quantification:**

$$\left(\dfrac{\begin{array}{c}\vdots\ \pi \\ \Gamma \vdash \forall \alpha^n\ B\end{array}}{\Gamma \vdash B\{\alpha^n := \lambda x_1, \ldots, x_n\,.\,A\}} \right)^* \quad = \quad \pi^* A^*$$

Properties:

Each stage preserves the invariant $\Gamma^* \vdash \pi^* : A^*$

1. Cuts of *implication* become 1st-kind redexes
2. Cuts of *2nd-order universal quantification* become 2nd-kind redexes ...
3. ... but cuts of *1st-order universal quantification* disappear

Translating proofs (4/4)

Translating proofs (4/4)

- **Injectivity:** Since

$$\big(\forall x\ \forall y\ (s(x) = s(y)\ \Rightarrow\ x = y)\big)^*\quad \equiv\quad \mathsf{Unit} \to \mathsf{Unit}$$

Translating proofs (4/4)

- **Injectivity:** Since

$$\big(\forall x\ \forall y\ (s(x) = s(y)\ \Rightarrow\ x = y)\big)^*\quad\equiv\quad \mathsf{Unit} \to \mathsf{Unit}$$

it is natural to set:

$$\left(\ \frac{}{\Gamma \vdash \forall x\ \forall y\ (s(x) = s(y)\ \Rightarrow\ x = y)}\ \right)^*\quad\equiv\quad \lambda\xi : \mathsf{Unit}\,.\,\xi$$

Translating proofs (4/4)

- **Injectivity:** Since

$$\big(\forall x\ \forall y\ (s(x) = s(y)\ \Rightarrow\ x = y)\big)^* \quad \equiv \quad \mathsf{Unit} \to \mathsf{Unit}$$

it is natural to set:

$$\left(\ \frac{}{\Gamma \vdash \forall x\ \forall y\ (s(x) = s(y)\ \Rightarrow\ x = y)}\ \right)^* \quad \equiv \quad \lambda\xi : \mathsf{Unit}\,.\,\xi$$

- **Non-surjectivity:** Quite problematic, since the type

$$(\forall x\ \neg\ s(x) = 0)^* \quad \equiv \quad \mathsf{Unit} \to \bot$$

has no closed inhabitant in system F.

Translating proofs (4/4)

- **Injectivity:** Since

$$\big(\forall x\ \forall y\ (s(x) = s(y)\ \Rightarrow\ x = y)\big)^* \quad \equiv \quad \mathsf{Unit} \to \mathsf{Unit}$$

it is natural to set:

$$\left(\ \frac{}{\Gamma \vdash \forall x\ \forall y\ (s(x) = s(y)\ \Rightarrow\ x = y)}\ \right)^* \quad \equiv \quad \lambda \xi : \mathsf{Unit}\,.\,\xi$$

- **Non-surjectivity:** Quite problematic, since the type

$$(\forall x\ \neg\ s(x) = 0)^* \quad \equiv \quad \mathsf{Unit} \to \bot$$

has no closed inhabitant in system F.

Solution (hack ?): Add a dummy constant $\Omega : \bot$ in the system and put:

$$\left(\ \frac{}{\Gamma \vdash \forall x\ \neg\ s(x) = 0}\ \right)^* \quad \equiv \quad \lambda \xi : \mathsf{Unit}\,.\,\Omega$$

Cut-elimination

Cut-elimination

1. Each proof of (intuitionistic) second-order arithmetic has been translated into a well-typed term of system F ($+$ constant Ω)

   Note: From the point of view of normalisation, system $F + \Omega$ is the same as system F: Ω merely acts as a free variable that we have declared in all contexts once and for all

Cut-elimination

1. Each proof of (intuitionistic) second-order arithmetic has been translated into a well-typed term of system F ($+$ constant Ω)

   Note: From the point of view of normalisation, system $F + \Omega$ is the same as system F: Ω merely acts as a free variable that we have declared in all contexts once and for all

2. Via the translation of proofs:
   - Cuts of implication become 1st kind redexes
   - Cuts of 2nd-order quantification become 2nd kind redexes
   - cuts of 1st-order quantification disappear

Cut-elimination

1. Each proof of (intuitionistic) second-order arithmetic has been translated into a well-typed term of system F ($+$ constant Ω)

   Note: From the point of view of normalisation, system $F + \Omega$ is the same as system F: Ω merely acts as a free variable that we have declared in all contexts once and for all

2. Via the translation of proofs:
   - Cuts of implication become 1st kind redexes
   - Cuts of 2nd-order quantification become 2nd kind redexes
   - cuts of 1st-order quantification disappear

   Treat the last kind of cuts as we did with 2nd-kind redexes when we proved $\mathrm{SN}(F\text{-Curry}) \Rightarrow \mathrm{SN}(F\text{-Church})$

Cut-elimination

1. Each proof of (intuitionistic) second-order arithmetic has been translated into a well-typed term of system F ($+$ constant Ω)

   Note: From the point of view of normalisation, system $F + \Omega$ is the same as system F: Ω merely acts as a free variable that we have declared in all contexts once and for all

2. Via the translation of proofs:
   - Cuts of implication become 1st kind redexes
   - Cuts of 2nd-order quantification become 2nd kind redexes
   - cuts of 1st-order quantification disappear

   Treat the last kind of cuts as we did with 2nd-kind redexes when we proved $\mathrm{SN}(F\text{-Curry}) \Rightarrow \mathrm{SN}(F\text{-Church})$, noticing that

Fact (Contraction of 1st-order $\forall$ cuts)

Each time we contract a cut of 1st-order quantification, the number of first-order $\forall$-intro decreases in the proof

Cut-elimination

1. Each proof of (intuitionistic) second-order arithmetic has been translated into a well-typed term of system F ($+$ constant Ω)

   Note: From the point of view of normalisation, system $F + \Omega$ is the same as system F: Ω merely acts as a free variable that we have declared in all contexts once and for all

2. Via the translation of proofs:
   - Cuts of implication become 1st kind redexes
   - Cuts of 2nd-order quantification become 2nd kind redexes
   - cuts of 1st-order quantification disappear

   Treat the last kind of cuts as we did with 2nd-kind redexes when we proved $\mathrm{SN}(F\text{-Curry}) \Rightarrow \mathrm{SN}(F\text{-Church})$, noticing that

Fact (Contraction of 1st-order $\forall$ cuts)

Each time we contract a cut of 1st-order quantification, the number of first-order $\forall$-intro decreases in the proof

3. Then we conclude that HA2 enjoys the property of cut-elimination

Natural numbers

Natural numbers

- **Problem:** The translation of formulæ and proofs erased all the terms!

Natural numbers

- **Problem:** The translation of formulæ and proofs erased all the terms!

  $\Rightarrow$ *Where did my numerals go ?*

Natural numbers

- **Problem:** The translation of formulæ and proofs erased all the terms!

  $\Rightarrow$ *Where did my numerals go ?*

- **Answer:** To benefit from induction, we restricted all the 1st-order quantifications with the predicate

$$\mathsf{Nat}(x) \quad \equiv \quad \forall \alpha^1 \left(\alpha^1(0) \ \Rightarrow \ \forall y \ \left(\alpha^1(y) \Rightarrow \alpha^1(s(y)) \right) \ \Rightarrow \ \alpha^1(x) \right)$$

Natural numbers

- **Problem:** The translation of formulæ and proofs erased all the terms!

  $\Rightarrow$ *Where did my numerals go ?*

- **Answer:** To benefit from induction, we restricted all the 1st-order quantifications with the predicate

$$\mathsf{Nat}(x) \quad \equiv \quad \forall \alpha^1 \left(\alpha^1(0) \ \Rightarrow \ \forall y \ (\alpha^1(y) \Rightarrow \alpha^1(s(y))) \ \Rightarrow \ \alpha^1(x) \right)$$

  whose translation in system F is:

$$(\mathsf{Nat}(x))^* \quad \equiv$$

Natural numbers

- **Problem:** The translation of formulæ and proofs erased all the terms!

  $\Rightarrow$ *Where did my numerals go ?*

- **Answer:** To benefit from induction, we restricted all the 1st-order quantifications with the predicate

$$\mathsf{Nat}(x) \quad \equiv \quad \forall \alpha^1 \left(\alpha^1(0) \ \Rightarrow \ \forall y \ \left(\alpha^1(y) \Rightarrow \alpha^1(s(y)) \right) \ \Rightarrow \ \alpha^1(x) \right)$$

  whose translation in system F is:

$$(\mathsf{Nat}(x))^* \quad \equiv \quad \forall \alpha \ \left(\alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \right)$$

Natural numbers

- **Problem:** The translation of formulæ and proofs erased all the terms!

  $\Rightarrow$ *Where did my numerals go ?*

- **Answer:** To benefit from induction, we restricted all the 1st-order quantifications with the predicate

$$\mathsf{Nat}(x) \quad \equiv \quad \forall \alpha^1 \left(\alpha^1(0) \ \Rightarrow \ \forall y \ \left(\alpha^1(y) \Rightarrow \alpha^1(s(y)) \right) \ \Rightarrow \ \alpha^1(x) \right)$$

  whose translation in system F is:

$$(\mathsf{Nat}(x))^* \quad \equiv \quad \forall \alpha \ \left(\alpha \to (\alpha \to \alpha) \to \alpha \right) \quad \equiv \quad \mathsf{Nat} \quad (\text{of system } F)$$

Natural numbers

- **Problem:** The translation of formulæ and proofs erased all the terms!

  $\Rightarrow$ *Where did my numerals go ?*

- **Answer:** To benefit from induction, we restricted all the 1st-order quantifications with the predicate

$$\mathsf{Nat}(x) \quad \equiv \quad \forall \alpha^1 \left(\alpha^1(0) \;\Rightarrow\; \forall y \; (\alpha^1(y) \Rightarrow \alpha^1(s(y))) \;\Rightarrow\; \alpha^1(x) \right)$$

  whose translation in system F is:

$$(\mathsf{Nat}(x))^* \quad \equiv \quad \forall \alpha \; (\alpha \to (\alpha \to \alpha) \to \alpha) \quad \equiv \quad \mathsf{Nat} \quad \text{(of system } F\text{)}$$

Fact (Translation of natural numbers)

For each term of the form $s^n(0)$ *(concrete numeral)*

Natural numbers

- **Problem:** The translation of formulæ and proofs erased all the terms!

  $\Rightarrow$ *Where did my numerals go ?*

- **Answer:** To benefit from induction, we restricted all the 1st-order quantifications with the predicate

$$\mathrm{Nat}(x) \quad \equiv \quad \forall \alpha^1 \left(\alpha^1(0) \;\Rightarrow\; \forall y \; (\alpha^1(y) \Rightarrow \alpha^1(s(y))) \;\Rightarrow\; \alpha^1(x) \right)$$

  whose translation in system F is:

$$(\mathrm{Nat}(x))^* \quad \equiv \quad \forall \alpha \; (\alpha \to (\alpha \to \alpha) \to \alpha) \quad \equiv \quad \mathrm{Nat} \quad \text{(of system } F\text{)}$$

Fact (Translation of natural numbers)

For each term of the form $s^n(0)$ *(concrete numeral)*

1. *The proposition* $\mathrm{Nat}(s^n(0))$ *has exactly one cut-free proof in HA2 . . .*

Natural numbers

- **Problem:** The translation of formulæ and proofs erased all the terms!

  $\Rightarrow$ *Where did my numerals go ?*

- **Answer:** To benefit from induction, we restricted all the 1st-order quantifications with the predicate

$$\mathrm{Nat}(x) \quad \equiv \quad \forall \alpha^1 \left(\alpha^1(0) \;\Rightarrow\; \forall y \; (\alpha^1(y) \Rightarrow \alpha^1(s(y))) \;\Rightarrow\; \alpha^1(x) \right)$$

  whose translation in system F is:

$$(\mathrm{Nat}(x))^* \quad \equiv \quad \forall \alpha \; (\alpha \to (\alpha \to \alpha) \to \alpha) \quad \equiv \quad \mathrm{Nat} \quad \text{(of system } F\text{)}$$

Fact (Translation of natural numbers)

For each term of the form $s^n(0)$ *(concrete numeral)*

1. *The proposition* $\mathrm{Nat}(s^n(0))$ *has exactly one cut-free proof in HA2 . . .*
2. *. . . whose translation in system F is precisely Church numeral* $\overline{n}$

Extracting programs from proofs

Extracting programs from proofs

Representation theorem

Any function whose totality can be proved in HA2 is representable in system F by a term of type $\mathsf{Nat} \to \mathsf{Nat}$ *[Converse is also true]*

Extracting programs from proofs

Representation theorem

Any function whose totality can be proved in HA2 is representable in system F by a term of type $\mathsf{Nat} \to \mathsf{Nat}$ *[Converse is also true]*

Proof. Consider a proof π in HA2 of a statement of the form

$$\forall x\ (\mathsf{Nat}(x) \Rightarrow \exists y\ (\mathsf{Nat}(y) \wedge P[x, y]))$$

Extracting programs from proofs

Representation theorem

Any function whose totality can be proved in HA2 is representable in system F by a term of type $\mathsf{Nat} \to \mathsf{Nat}$ *[Converse is also true]*

Proof. Consider a proof π in HA2 of a statement of the form

$$\forall x\ (\mathsf{Nat}(x) \Rightarrow \exists y\ (\mathsf{Nat}(y) \wedge P[x, y]))$$

By translating the proof π into system F, we obtain a term

$$\pi^* \quad : \quad \mathsf{Nat} \to \forall \alpha\ ((\mathsf{Nat} \times P^* \to \alpha) \to \alpha)$$

(using the 2nd-order encoding of $\exists$ given in slide 3)

Extracting programs from proofs

Representation theorem

Any function whose totality can be proved in HA2 is representable in system F by a term of type $\mathsf{Nat} \to \mathsf{Nat}$ *[Converse is also true]*

Proof. Consider a proof π in HA2 of a statement of the form

$$\forall x\ (\mathsf{Nat}(x) \Rightarrow \exists y\ (\mathsf{Nat}(y) \wedge P[x, y]))$$

By translating the proof π into system F, we obtain a term

$$\pi^* \quad : \quad \mathsf{Nat} \to \forall \alpha\ ((\mathsf{Nat} \times P^* \to \alpha) \to \alpha)$$

(using the 2nd-order encoding of $\exists$ given in slide 3), so that the term

$$\lambda \xi : \mathsf{Nat}\,.\,\pi^*\ \xi\ \mathsf{Nat}\ \mathsf{fst} \quad : \quad \mathsf{Nat} \to \mathsf{Nat}$$

(where $\mathsf{fst} : \mathsf{Nat} \times P^* \to \mathsf{Nat}$ is the first projection) actually computes the desired function

Extracting programs from proofs

Representation theorem

Any function whose totality can be proved in HA2 is representable in system F by a term of type $\mathsf{Nat} \to \mathsf{Nat}$ *[Converse is also true]*

Proof. Consider a proof π in HA2 of a statement of the form

$$\forall x\ (\mathsf{Nat}(x) \Rightarrow \exists y\ (\mathsf{Nat}(y) \wedge P[x,y]))$$

By translating the proof π into system F, we obtain a term

$$\pi^* \quad : \quad \mathsf{Nat} \to \forall \alpha\ ((\mathsf{Nat} \times P^* \to \alpha) \to \alpha)$$

(using the 2nd-order encoding of $\exists$ given in slide 3), so that the term

$$\lambda \xi : \mathsf{Nat}\,.\,\pi^*\ \xi\ \mathsf{Nat}\ \mathsf{fst} \quad : \quad \mathsf{Nat} \to \mathsf{Nat}$$

(where $\mathsf{fst} : \mathsf{Nat} \times P^* \to \mathsf{Nat}$ is the first projection) actually computes the desired function

Remark: We cheated a little bit, since π^* may contain the dummy constant Ω that could block some computations.

Extracting programs from proofs

Representation theorem

Any function whose totality can be proved in HA2 is representable in system F by a term of type $\mathsf{Nat} \to \mathsf{Nat}$ *[Converse is also true]*

Proof. Consider a proof π in HA2 of a statement of the form

$$\forall x\ (\mathsf{Nat}(x) \Rightarrow \exists y\ (\mathsf{Nat}(y) \wedge P[x,y]))$$

By translating the proof π into system F, we obtain a term

$$\pi^* \quad : \quad \mathsf{Nat} \to \forall \alpha\ ((\mathsf{Nat} \times P^* \to \alpha) \to \alpha)$$

(using the 2nd-order encoding of $\exists$ given in slide 3), so that the term

$$\lambda \xi : \mathsf{Nat}\,.\,\pi^*\ \xi\ \mathsf{Nat}\ \mathsf{fst} \quad : \quad \mathsf{Nat} \to \mathsf{Nat}$$

(where $\mathsf{fst} : \mathsf{Nat} \times P^* \to \mathsf{Nat}$ is the first projection) actually computes the desired function

Remark: We cheated a little bit, since π^* may contain the dummy constant Ω that could block some computations. There are two solutions to fix this:

Extracting programs from proofs

Representation theorem

Any function whose totality can be proved in HA2 is representable in system F by a term of type $\mathsf{Nat} \to \mathsf{Nat}$ *[Converse is also true]*

Proof. Consider a proof π in HA2 of a statement of the form

$$\forall x\ (\mathsf{Nat}(x) \Rightarrow \exists y\ (\mathsf{Nat}(y) \wedge P[x,y]))$$

By translating the proof π into system F, we obtain a term

$$\pi^* \quad : \quad \mathsf{Nat} \to \forall \alpha\ ((\mathsf{Nat} \times P^* \to \alpha) \to \alpha)$$

(using the 2nd-order encoding of $\exists$ given in slide 3), so that the term

$$\lambda \xi : \mathsf{Nat}\,.\,\pi^*\ \xi\ \mathsf{Nat}\ \mathsf{fst} \quad : \quad \mathsf{Nat} \to \mathsf{Nat}$$

(where $\mathsf{fst} : \mathsf{Nat} \times P^* \to \mathsf{Nat}$ is the first projection) actually computes the desired function

Remark: We cheated a little bit, since π^* may contain the dummy constant Ω that could block some computations. There are two solutions to fix this:

1. Use the shape of cut-free proofs of $\mathsf{Nat}(s^n(0))$ to show that this never happens

Extracting programs from proofs

Representation theorem

Any function whose totality can be proved in HA2 is representable in system F by a term of type $\mathsf{Nat} \to \mathsf{Nat}$ *[Converse is also true]*

Proof. Consider a proof π in HA2 of a statement of the form

$$\forall x\ (\mathsf{Nat}(x) \Rightarrow \exists y\ (\mathsf{Nat}(y) \wedge P[x,y]))$$

By translating the proof π into system F, we obtain a term

$$\pi^* \quad : \quad \mathsf{Nat} \to \forall \alpha\ ((\mathsf{Nat} \times P^* \to \alpha) \to \alpha)$$

(using the 2nd-order encoding of $\exists$ given in slide 3), so that the term

$$\lambda \xi : \mathsf{Nat}\,.\,\pi^*\ \xi\ \mathsf{Nat}\ \mathsf{fst} \quad : \quad \mathsf{Nat} \to \mathsf{Nat}$$

(where $\mathsf{fst} : \mathsf{Nat} \times P^* \to \mathsf{Nat}$ is the first projection) actually computes the desired function

Remark: We cheated a little bit, since π^* may contain the dummy constant Ω that could block some computations. There are two solutions to fix this:

1. Use the shape of cut-free proofs of $\mathsf{Nat}(s^n(0))$ to show that this never happens
2. Define a modified translation that avoids the use of Ω [cf Proofs and Types]

Inconsistent Type Systems

Alexandre Miquel — PPS & U. Paris 7

Alexandre.Miquel@pps.jussieu.fr

Types Summer School 2005

August 15–26 — Göteborg

Introduction

- System F [Girard 1971]
- 'A theory of types' (Type:Type) [Martin-Löf 1971]
- Inconsistency of system U [Girard 1971]

  Inconsistency of Type:Types comes as a consequence
- Inconsistency of System U^- [Coquand 1991]
- Simplification of Girard's paradox (system U^-) [Hurkens 1995]
- Russell's paradox in systems U/U^- [Miquel 2000]

System $\lambda*$ (Type:Type) [Martin-Löf 71]

System $\lambda*$ (Type:Type) [Martin-Löf 71]

Terms	M, N, T, U	$::=$	x	$\mid$	$\lambda x : T \,.\, M$	$\mid$	MN	$\mid$	Type	$\mid$	$\Pi x : T \,.\, U$
Contexts	Γ, Δ	$::=$	$[]$	$\mid$	$\Gamma,\ x : T$						

System $\lambda*$ (Type:Type) [Martin-Löf 71]

Terms $M, N, T, U ::= x \mid \lambda x : T \,.\, M \mid MN \mid \mathsf{Type} \mid \Pi x : T \,.\, U$

Contexts $\Gamma, \Delta ::= [] \mid \Gamma,\ x : T$

$$\frac{}{\vdash [] \ \mathsf{ctx}} \qquad \frac{\Gamma \vdash T : \mathsf{Type}}{\vdash \Gamma,\ x : T \ \mathsf{ctx}}\ x \notin \mathbf{Dom}(\Gamma)$$

$$\frac{\vdash \Gamma \ \mathsf{ctx}}{\Gamma \vdash x : T}\ (x:T) \in \Gamma \qquad \frac{\vdash \Gamma \ \mathsf{ctx}}{\Gamma \vdash \mathsf{Type} : \mathsf{Type}} \qquad \frac{\Gamma,\ x : T \vdash U : \mathsf{Type}}{\Gamma \vdash \Pi x : T \,.\, U : \mathsf{Type}}$$

$$\frac{\Gamma,\ x : T \vdash M : U}{\Gamma \vdash \lambda x : T \,.\, M : \Pi x : T \,.\, U} \qquad \frac{\Gamma \vdash M : \Pi x : T \,.\, U \qquad \Gamma \vdash N : T}{\Gamma \vdash MN : U\{x := N\}}$$

$$\frac{\Gamma \vdash M : T \qquad \Gamma \vdash T' : \mathsf{Type}}{\Gamma \vdash M : T'}\ T' \approx T$$

System $\lambda*$ (Type:Type) [Martin-Löf 71]

Terms $M, N, T, U ::= x \mid \lambda x : T \,.\, M \mid MN \mid \mathsf{Type} \mid \Pi x : T \,.\, U$

Contexts $\Gamma, \Delta ::= [] \mid \Gamma,\ x : T$

$$\frac{}{\vdash [] \ \mathsf{ctx}} \qquad \frac{\Gamma \vdash T : \mathsf{Type}}{\vdash \Gamma,\ x : T \ \mathsf{ctx}}\ x \notin \mathbf{Dom}(\Gamma)$$

$$\frac{\vdash \Gamma \ \mathsf{ctx}}{\Gamma \vdash x : T}\ (x{:}T) \in \Gamma \qquad \frac{\vdash \Gamma \ \mathsf{ctx}}{\Gamma \vdash \mathsf{Type} : \mathsf{Type}} \qquad \frac{\Gamma,\ x : T \vdash U : \mathsf{Type}}{\Gamma \vdash \Pi x : T \,.\, U : \mathsf{Type}}$$

$$\frac{\Gamma,\ x : T \vdash M : U}{\Gamma \vdash \lambda x : T \,.\, M : \Pi x : T \,.\, U} \qquad \frac{\Gamma \vdash M : \Pi x : T \,.\, U \qquad \Gamma \vdash N : T}{\Gamma \vdash MN : U\{x := N\}}$$

$$\frac{\Gamma \vdash M : T \qquad \Gamma \vdash T' : \mathsf{Type}}{\Gamma \vdash M : T'}\ T' \approx T$$

- Computationally correct: Church-Rosser, subject reduction

System $\lambda*$ (Type:Type) [Martin-Löf 71]

Terms $M, N, T, U ::= x \mid \lambda x : T \,.\, M \mid MN \mid \mathsf{Type} \mid \Pi x : T \,.\, U$

Contexts $\Gamma, \Delta ::= [] \mid \Gamma,\ x : T$

$$\frac{}{\vdash [] \ \mathsf{ctx}} \qquad \frac{\Gamma \vdash T : \mathsf{Type}}{\vdash \Gamma,\ x : T \ \mathsf{ctx}}\ x \notin \mathbf{Dom}(\Gamma)$$

$$\frac{\vdash \Gamma \ \mathsf{ctx}}{\Gamma \vdash x : T}\ (x:T) \in \Gamma \qquad \frac{\vdash \Gamma \ \mathsf{ctx}}{\Gamma \vdash \mathsf{Type} : \mathsf{Type}} \qquad \frac{\Gamma,\ x : T \vdash U : \mathsf{Type}}{\Gamma \vdash \Pi x : T \,.\, U : \mathsf{Type}}$$

$$\frac{\Gamma,\ x : T \vdash M : U}{\Gamma \vdash \lambda x : T \,.\, M : \Pi x : T \,.\, U} \qquad \frac{\Gamma \vdash M : \Pi x : T \,.\, U \qquad \Gamma \vdash N : T}{\Gamma \vdash MN : U\{x := N\}}$$

$$\frac{\Gamma \vdash M : T \qquad \Gamma \vdash T' : \mathsf{Type}}{\Gamma \vdash M : T'}\ T' \approx T$$

- Computationally correct: Church-Rosser, subject reduction
- Logically inconsistent: closed term of type $\bot \equiv \Pi X : \mathsf{Type} \,.\, X$

System $\lambda*$ (Type:Type) [Martin-Löf 71]

Terms $M, N, T, U ::= x \mid \lambda x : T \,.\, M \mid MN \mid \mathsf{Type} \mid \Pi x : T \,.\, U$

Contexts $\Gamma, \Delta ::= [] \mid \Gamma,\ x : T$

$$\frac{}{\vdash [] \ \mathsf{ctx}} \qquad \frac{\Gamma \vdash T : \mathsf{Type}}{\vdash \Gamma,\ x : T \ \mathsf{ctx}}\ x \notin \mathbf{Dom}(\Gamma)$$

$$\frac{\vdash \Gamma \ \mathsf{ctx}}{\Gamma \vdash x : T}\ (x:T) \in \Gamma \qquad \frac{\vdash \Gamma \ \mathsf{ctx}}{\Gamma \vdash \mathsf{Type} : \mathsf{Type}} \qquad \frac{\Gamma,\ x : T \vdash U : \mathsf{Type}}{\Gamma \vdash \Pi x : T \,.\, U : \mathsf{Type}}$$

$$\frac{\Gamma,\ x : T \vdash M : U}{\Gamma \vdash \lambda x : T \,.\, M : \Pi x : T \,.\, U} \qquad \frac{\Gamma \vdash M : \Pi x : T \,.\, U \qquad \Gamma \vdash N : T}{\Gamma \vdash MN : U\{x := N\}}$$

$$\frac{\Gamma \vdash M : T \qquad \Gamma \vdash T' : \mathsf{Type}}{\Gamma \vdash M : T'}\ T' \approx T$$

- Computationally correct: Church-Rosser, subject reduction
- Logically inconsistent: closed term of type $\bot \equiv \Pi X : \mathsf{Type} \,.\, X$
- Non (weakly) normalising, since:

Fact: Closed terms of type $\bot \equiv \Pi X : \mathsf{Type} \,.\, X$ have no head normal form

Is a type of all types like a set of all sets?

Is a type of all types like a set of all sets?

The analogy between Type:Type and the set of all sets of Cantor-Frege's (inconsistent) set theory is erroneous, since:

Is a type of all types like a set of all sets?

The analogy between Type:Type and the set of all sets of Cantor-Frege's (inconsistent) set theory is erroneous, since:

1. Typing relation and membership relation have not the same status
   - Typing belongs to the meta-language
     $\Rightarrow$ Precondition for an expression to be well-formed
   - Membership is a relation of the language
     $\Rightarrow$ Can be used to form propositions (may be negated)

Is a type of all types like a set of all sets?

The analogy between Type:Type and the set of all sets of Cantor-Frege's (inconsistent) set theory is erroneous, since:

1. Typing relation and membership relation have not the same status
   - Typing belongs to the meta-language
     $\Rightarrow$ Precondition for an expression to be well-formed
   - Membership is a relation of the language
     $\Rightarrow$ Can be used to form propositions (may be negated)
2. No comprehension scheme in Type:Type
   $\Rightarrow$ Cannot form a type of the form $\{x : T \mid P(x)\}$

Is a type of all types like a set of all sets?

The analogy between Type:Type and the set of all sets of Cantor-Frege's (inconsistent) set theory is erroneous, since:

1. Typing relation and membership relation have not the same status
   - Typing belongs to the meta-language
     $\Rightarrow$ Precondition for an expression to be well-formed
   - Membership is a relation of the language
     $\Rightarrow$ Can be used to form propositions (may be negated)
2. No comprehension scheme in Type:Type
   $\Rightarrow$ Cannot form a type of the form $\{x : T \mid P(x)\}$

Historically, the inconsistency of Type:Type has been derived [Girard] from the inconsistency of system U

Is a type of all types like a set of all sets?

The analogy between Type:Type and the set of all sets of Cantor-Frege's (inconsistent) set theory is erroneous, since:

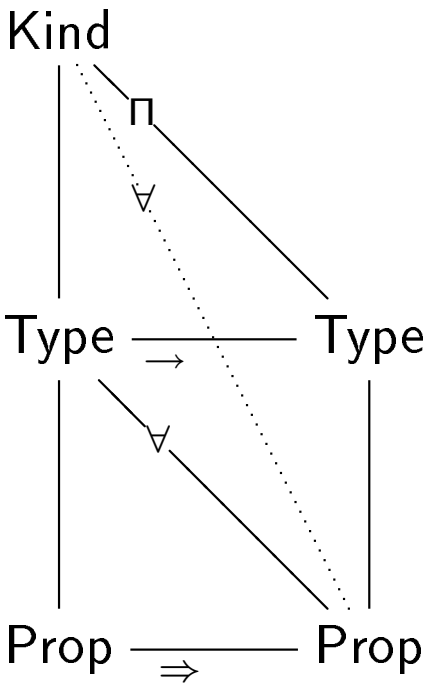
1. Typing relation and membership relation have not the same status
   - Typing belongs to the meta-language
     $\Rightarrow$ Precondition for an expression to be well-formed
   - Membership is a relation of the language
     $\Rightarrow$ Can be used to form propositions (may be negated)
2. No comprehension scheme in Type:Type
   $\Rightarrow$ Cannot form a type of the form $\{x : T \mid P(x)\}$

Historically, the inconsistency of Type:Type has been derived [Girard] from the inconsistency of system U

No cycle in the sorts (Prop : Type : Kind)...

... but two levels of impredicativity (Prop and Type)

Systems U and U^-



U^- = copy of F glued on top of $F\omega$

U = system U^- + (Kind, Prop)-quantification

- Kind = sort for kinds
- Type = sort for constructors
- Prop = sort for proof-terms

Both Type and Prop are impredicative

Higher-level is isomorphic to F:
Type inference/checking is decidable

$$
\begin{aligned}
\mathcal{S} &= \{\text{Prop}, \text{Type}, \text{Kind}\} \\
\mathcal{A} &= \{(\text{Prop} : \text{Type}), (\text{Type} : \text{Kind})\} \\
\mathcal{R} &= \{\underbrace{(\text{Prop} : \text{Prop}), (\text{Type} : \text{Prop}), (\text{Type}, \text{Type}), (\text{Kind}, \text{Type})}_{\text{system } U^-}, \underbrace{(\text{Kind}, \text{Prop})}_{U \text{ only}}\}
\end{aligned}
$$

From system $F\omega$

$$\begin{array}{lcll} \mathcal{S} & = & \mathsf{Prop}, & \mathsf{Type} \\ \mathcal{A} & = & \mathsf{Prop} : \mathsf{Type} & \\ \mathcal{R} & = & (\mathsf{Prop}, \mathsf{Prop}), & (\mathsf{Type}, \mathsf{Prop}), \quad (\mathsf{Type}, \mathsf{Type}) \end{array}$$

Kinds

$$\begin{array}{lcll} \tau, \sigma & ::= & \mathsf{Prop} & \\ & | & \tau \longrightarrow \sigma & (\mathsf{Type},\ \mathsf{Type}) \end{array}$$

Constructors

$$\begin{array}{lcll} M, N & ::= & \xi & \\ & | & \lambda x : \tau \,.\, M \quad | \quad MN & (\mathsf{Type},\ \mathsf{Type}) \\ & | & M \Rightarrow N & (\mathsf{Prop},\ \mathsf{Prop}) \\ & | & \forall x : \tau \,.\, M & (\mathsf{Type},\ \mathsf{Prop}) \end{array}$$

Proof-terms

$$\begin{array}{lcll} t, u & ::= & \xi & \\ & | & \lambda \xi : M \,.\, t \quad | \quad tu & (\mathsf{Prop},\ \mathsf{Prop}) \\ & | & \lambda x : \tau \,.\, t \quad | \quad tM & (\mathsf{Type},\ \mathsf{Prop}) \end{array}$$

From system $F\omega$...

$$
\begin{array}{lcll}
\mathcal{S} & = & \mathsf{Prop}, & \mathsf{Type}, \quad \mathsf{Kind} \\
\mathcal{A} & = & \mathsf{Prop} : \mathsf{Type}, & \mathsf{Type} : \mathsf{Kind} \\
\mathcal{R} & = & (\mathsf{Prop}, \mathsf{Prop}), & (\mathsf{Type}, \mathsf{Prop}), \quad (\mathsf{Type}, \mathsf{Type})
\end{array}
$$

Kinds

$$
\begin{array}{lcll}
\tau, \sigma & ::= & \mathsf{Prop} \mid \alpha & \\
& \mid & \tau \longrightarrow \sigma & (\mathsf{Type}, \mathsf{Type})
\end{array}
$$

Constructors

$$
\begin{array}{lcll}
M, N & ::= & \xi & \\
& \mid & \lambda x : \tau \,.\, M \mid MN & (\mathsf{Type}, \mathsf{Type}) \\
& \mid & M \Rightarrow N & (\mathsf{Prop}, \mathsf{Prop}) \\
& \mid & \forall x : \tau \,.\, M & (\mathsf{Type}, \mathsf{Prop})
\end{array}
$$

Proof-terms

$$
\begin{array}{lcll}
t, u & ::= & \xi & \\
& \mid & \lambda \xi : M \,.\, t \mid tu & (\mathsf{Prop}, \mathsf{Prop}) \\
& \mid & \lambda x : \tau \,.\, t \mid tM & (\mathsf{Type}, \mathsf{Prop})
\end{array}
$$

From system $F\omega$... to system U^-

$$
\begin{array}{lcl}
\mathcal{S} & = & \mathsf{Prop}, \quad \mathsf{Type}, \quad \mathsf{Kind} \\
\mathcal{A} & = & \mathsf{Prop} : \mathsf{Type}, \quad \mathsf{Type} : \mathsf{Kind} \\
\mathcal{R} & = & (\mathsf{Prop}, \mathsf{Prop}), \quad (\mathsf{Type}, \mathsf{Prop}), \quad (\mathsf{Type}, \mathsf{Type}), \quad (\mathsf{Kind}, \mathsf{Type})
\end{array}
$$

Kinds

$$
\begin{array}{lcll}
\tau, \sigma & ::= & \mathsf{Prop} \mid \alpha & \\
& \mid & \tau \longrightarrow \sigma & (\mathsf{Type}, \mathsf{Type}) \\
& \mid & \Pi \alpha : \mathsf{Type} \,.\, \tau & (\mathsf{Kind}, \mathsf{Type})
\end{array}
$$

Constructors

$$
\begin{array}{lcll}
M, N & ::= & \xi & \\
& \mid & \lambda x : \tau \,.\, M \mid MN & (\mathsf{Type}, \mathsf{Type}) \\
& \mid & \Lambda \alpha \,.\, M \mid M\tau & (\mathsf{Kind}, \mathsf{Type}) \\
& \mid & M \Rightarrow N & (\mathsf{Prop}, \mathsf{Prop}) \\
& \mid & \forall x : \tau \,.\, M & (\mathsf{Type}, \mathsf{Prop})
\end{array}
$$

Proof-terms

$$
\begin{array}{lcll}
t, u & ::= & \xi & \\
& \mid & \lambda \xi : M \,.\, t \mid tu & (\mathsf{Prop}, \mathsf{Prop}) \\
& \mid & \lambda x : \tau \,.\, t \mid tM & (\mathsf{Type}, \mathsf{Prop})
\end{array}
$$

From system $F\omega$... to system U

$$
\begin{array}{lcl}
\mathcal{S} & = & \mathsf{Prop}, \quad \mathsf{Type}, \quad \mathsf{Kind} \\
\mathcal{A} & = & \mathsf{Prop} : \mathsf{Type}, \quad \mathsf{Type} : \mathsf{Kind} \\
\mathcal{R} & = & (\mathsf{Prop}, \mathsf{Prop}), \quad (\mathsf{Type}, \mathsf{Prop}), \quad (\mathsf{Type}, \mathsf{Type}), \quad (\mathsf{Kind}, \mathsf{Type}), \quad (\mathsf{Kind}, \mathsf{Prop})
\end{array}
$$

Kinds	τ, σ	$::=$	$\mathsf{Prop} \mid \alpha$	
		$\mid$	$\tau \longrightarrow \sigma$	(Type, Type)
		$\mid$	$\Pi\alpha : \mathsf{Type} \,.\, \tau$	(Kind, Type)
Constructors	M, N	$::=$	ξ	
		$\mid$	$\lambda x : \tau \,.\, M \mid MN$	(Type, Type)
		$\mid$	$\Lambda\alpha \,.\, M \mid M\tau$	(Kind, Type)
		$\mid$	$M \Rightarrow N$	(Prop, Prop)
		$\mid$	$\forall x : \tau \,.\, M$	(Type, Prop)
		$\mid$	$\forall\alpha : \mathsf{Type} \,.\, M$	(Kind, Prop)
Proof-terms	t, u	$::=$	ξ	
		$\mid$	$\lambda\xi : M \,.\, t \mid tu$	(Prop, Prop)
		$\mid$	$\lambda x : \tau \,.\, t \mid tM$	(Type, Prop)
		$\mid$	$\lambda\alpha : \mathsf{Type} \,.\, t \mid t\tau$	(Kind, Prop)

Examples

(Kind, Type)	$\Pi\alpha : \text{Type} \ldots$	Polymorphism in data types
(Type : Prop)	$\forall x : \tau \ldots$	Quantification over all objects (of a given type)
(Kind, Prop)	$\forall\alpha : \text{Type} \ldots$	Quantification over all types

Examples

(Kind, Type)	$\Pi\alpha : \text{Type} \ldots$	Polymorphism in data types
(Type : Prop)	$\forall x : \tau \ldots$	Quantification over all objects (of a given type)
(Kind, Prop)	$\forall\alpha : \text{Type} \ldots$	Quantification over all types

$$\text{Nat} \quad := \quad \Pi\alpha : \text{Type} \,.\, (\alpha \to (\alpha \to \alpha) \to \alpha) \quad : \quad \text{Type}$$

Examples

(Kind, Type)	$\Pi\alpha : \text{Type} \ldots$	Polymorphism in data types
(Type : Prop)	$\forall x : \tau \ldots$	Quantification over all objects (of a given type)
(Kind, Prop)	$\forall\alpha : \text{Type} \ldots$	Quantification over all types

$$\text{Nat} \quad := \quad \Pi\alpha : \text{Type} \,.\, (\alpha \to (\alpha \to \alpha) \to \alpha) \quad : \quad \text{Type}$$

$$\text{id} \quad := \quad \lambda\alpha : \text{Type} \,.\, \lambda x : \alpha \,.\, x \quad : \quad \Pi\alpha : \text{Type} \,.\, (\alpha \to \alpha)$$

Examples

(Kind, Type)	$\Pi\alpha : \text{Type} \ldots$	Polymorphism in data types
(Type : Prop)	$\forall x : \tau \ldots$	Quantification over all objects (of a given type)
(Kind, Prop)	$\forall\alpha : \text{Type} \ldots$	Quantification over all types

$$\text{Nat} \quad := \quad \Pi\alpha : \text{Type} \,.\, (\alpha \to (\alpha \to \alpha) \to \alpha) \quad : \quad \text{Type}$$

$$\text{id} \quad := \quad \lambda\alpha : \text{Type} \,.\, \lambda x : \alpha \,.\, x \quad : \quad \Pi\alpha : \text{Type} \,.\, (\alpha \to \alpha)$$

$$x \;=_\alpha\; y \quad := \quad \forall p : (\alpha \to \text{Prop}) \,.\, (p\, x \Rightarrow p\, y) \quad : \quad \text{Prop}$$

Examples

(Kind, Type)	$\Pi\alpha : \text{Type} \ldots$	Polymorphism in data types
(Type : Prop)	$\forall x : \tau \ldots$	Quantification over all objects (of a given type)
(Kind, Prop)	$\forall\alpha : \text{Type} \ldots$	Quantification over all types

$$\text{Nat} := \Pi\alpha : \text{Type} . (\alpha \to (\alpha \to \alpha) \to \alpha) : \text{Type}$$

$$\text{id} := \lambda\alpha : \text{Type} . \lambda x : \alpha . x : \Pi\alpha : \text{Type} . (\alpha \to \alpha)$$

$$x =_\alpha y := \forall p : (\alpha \to \text{Prop}) . (p\, x \Rightarrow p\, y) : \text{Prop}$$

$$\forall\alpha : \text{Type} . \quad \forall x : \alpha . \quad \text{id}\ \alpha\ x =_\alpha x : \text{Prop}$$

Examples

(Kind, Type)	$\Pi\alpha : \mathsf{Type} \ldots$	Polymorphism in data types
(Type : Prop)	$\forall x : \tau \ldots$	Quantification over all objects (of a given type)
(Kind, Prop)	$\forall\alpha : \mathsf{Type} \ldots$	Quantification over all types

$$\mathsf{Nat} \quad := \quad \Pi\alpha : \mathsf{Type} \,.\, (\alpha \to (\alpha \to \alpha) \to \alpha) \quad : \quad \mathsf{Type}$$

$$\mathsf{id} \quad := \quad \lambda\alpha : \mathsf{Type} \,.\, \lambda x : \alpha \,.\, x \quad : \quad \Pi\alpha : \mathsf{Type} \,.\, (\alpha \to \alpha)$$

$$x \;=_\alpha\; y \quad := \quad \forall p : (\alpha \to \mathsf{Prop}) \,.\, (p\, x \Rightarrow p\, y) \quad : \quad \mathsf{Prop}$$

$$\forall\alpha : \mathsf{Type} \,. \quad \forall x : \alpha \,. \quad \mathsf{id}\ \alpha\ x \;=_\alpha\; x \quad : \quad \mathsf{Prop}$$

$$\lambda\alpha : \mathsf{Type} \,. \quad \lambda x : \alpha \,. \quad \lambda p : (\alpha \to \mathsf{Prop}) \,. \quad \lambda\xi : p\, x \,.\, \xi \quad : \quad \ldots$$

Hurkens' paradox in system U^-

Hurkens' paradox in system U^-

For any kind $\tau : \mathrm{Type}$ write: $\quad \mathfrak{P}(\tau) \;:=\; \tau \to \mathrm{Prop}$

$$
\begin{array}{lll}
\bot & : \mathrm{Prop} & := \quad \forall a : \mathrm{Prop} \,.\, a \\
\neg & : \mathrm{Prop} \to \mathrm{Prop} & := \quad \lambda a : \mathrm{Prop} \,.\, a \Rightarrow \bot \\
\mathbb{U} & : \mathrm{Type} & := \quad \Pi\alpha : \mathrm{Type} \,.\, \Big(\big(\mathfrak{P}(\mathfrak{P}(\alpha)) \to \alpha \big) \to \mathfrak{P}(\mathfrak{P}(\alpha)) \Big) \\
i & : \mathfrak{P}(\mathfrak{P}(\mathbb{U})) \to \mathbb{U} & := \quad \lambda q : \mathfrak{P}(\mathfrak{P}(\mathbb{U})) \,.\, \lambda\alpha : \mathrm{Type} \,.\, \lambda f : \big(\mathfrak{P}(\mathfrak{P}(\alpha)) \to \alpha \big) \,. \\
 & & \qquad \lambda p : \mathfrak{P}(\alpha) \,.\; q \; (\lambda x : \mathbb{U} \,.\, p \; (f \; (x \; \alpha \; f))) \\
j & : \mathbb{U} \to \mathfrak{P}(\mathfrak{P}(\mathbb{U})) & := \quad \lambda x : \mathbb{U} \,.\; x \; \mathbb{U} \; i \\
Q & : \mathfrak{P}(\mathfrak{P}(\mathbb{U})) & := \quad \lambda p : \mathfrak{P}(\mathbb{U}) \,.\; \forall x : \mathbb{U} \,.\, (j \; x \; p \Rightarrow p \; x) \\
C & : \mathfrak{P}(\mathbb{U}) & := \quad \lambda y : \mathbb{U} \,.\; \neg \forall p : \mathfrak{P}(\mathbb{U}) \,.\, \big(j \; y \; p \Rightarrow p \; (i \; (j \; y)) \big) \\
B & : \mathbb{U} & := \quad i \; Q \\
\mathrm{lem}_1 & : Q \; C & := \quad \lambda x : U \,.\, \lambda\xi^{jxC} \,.\, \lambda\zeta^{\forall p : \mathfrak{P}(\mathbb{U}) \,.\, (jxp \Rightarrow p(i(jx)))} \,. \\
 & & \qquad \zeta \; C \; \xi \; \big(\lambda p : \mathfrak{P}(\mathbb{U}) \,.\, \zeta \; (\lambda y : \mathbb{U} \,.\, p \; (i \; (j \; y))) \big) \\
A & : \mathrm{Prop} & := \quad \forall p : \mathfrak{P}(\mathbb{U}) \,.\; (Q \; p \Rightarrow p \; B) \\
\mathrm{lem}_2 & : \neg A & := \quad \lambda\xi^{A} \,.\, \xi \; C \; \mathrm{lem}_1 \; \big(\lambda p : \mathfrak{P}(\mathbb{U}) \,.\, \xi \; (\lambda y : \mathbb{U} \,.\, p \; (i \; (j \; y))) \big) \\
\mathrm{lem}_3 & : A & := \quad \lambda p : \mathfrak{P}(\mathbb{U}) \,.\, \lambda\xi^{Qp} \,.\, \xi \; B \; (\lambda x : \mathbb{U} \,.\, \xi \; (i \; (j \; x))) \\
\mathrm{paradox} & : \bot & := \quad \mathrm{lem}_2 \; \mathrm{lem}_3
\end{array}
$$

Encoding sets as pointed graphs

Encoding sets as pointed graphs

Pointed graph $=$ triple (X, A, a) where

- X : Type — the type of vertices
- $A : X \to X \to \mathrm{Prop}$ — the (local) membership relation
- $a : X$ — the root

Encoding sets as pointed graphs

Pointed graph $=$ triple (X, A, a) where

- X : Type — the type of vertices
- $A : X \to X \to \mathrm{Prop}$ — the (local) membership relation
- $a : X$ — the root

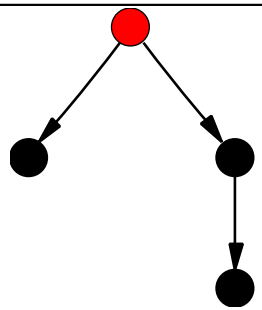
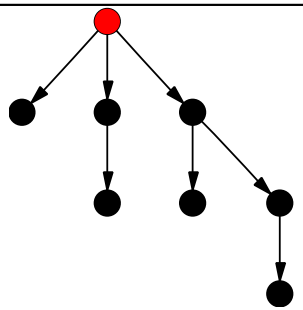
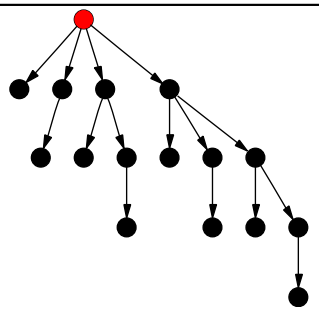
$A(x, y)$ is represented by $\bullet_x \leftarrow \bullet_y$, and the root a by $\bullet_a$

Encoding sets as pointed graphs

Pointed graph $=$ triple (X, A, a) where

- X : Type — the type of vertices
- $A : X \to X \to \mathrm{Prop}$ — the (local) membership relation
- $a : X$ — the root

$A(x, y)$ is represented by $\bullet_x \leftarrow \bullet_y$, and the root a by $\bullet_a$

$0 = \varnothing$	$1 = \{0\}$	$2 = \{0; 1\}$	$3 = \{0; 1; 2\}$	$4 = \{0; 1; 2; 3\}$
				

Identifying related pointed graphs

A set can be represented by several non-isomorphic pointed graphs

Identifying related pointed graphs

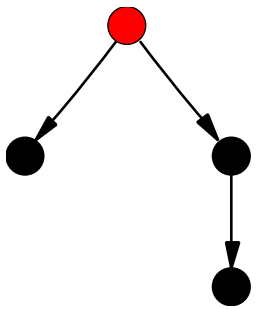
A set can be represented by several non-isomorphic pointed graphs

Example: the set $2 = \{\varnothing; \{\varnothing\}\}$

Identifying related pointed graphs

A set can be represented by several non-isomorphic pointed graphs

Example: the set $2 = \{\varnothing; \{\varnothing\}\}$

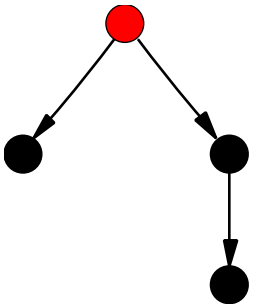


no sharing (tree)

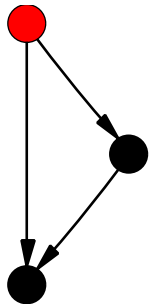
Identifying related pointed graphs

A set can be represented by several non-isomorphic pointed graphs

Example: the set $2 = \{\varnothing; \{\varnothing\}\}$



no sharing (tree)

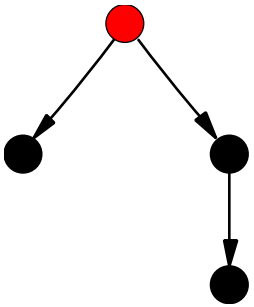


with sharing

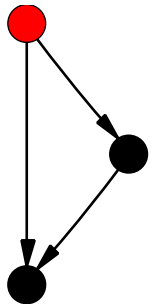
Identifying related pointed graphs

A set can be represented by several non-isomorphic pointed graphs

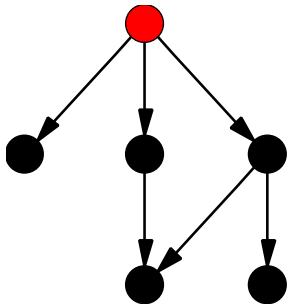
Example: the set $2 = \{\varnothing; \{\varnothing\}\}$



no sharing (tree)



with sharing

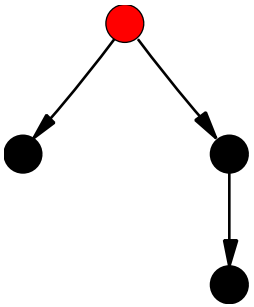


duplicate elements

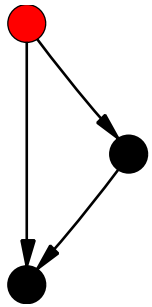
Identifying related pointed graphs

A set can be represented by several non-isomorphic pointed graphs

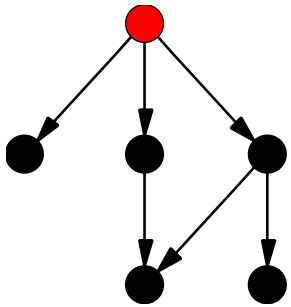
Example: the set $2 = \{\varnothing; \{\varnothing\}\}$



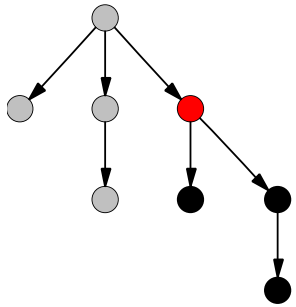
no sharing (tree)



with sharing



duplicate elements

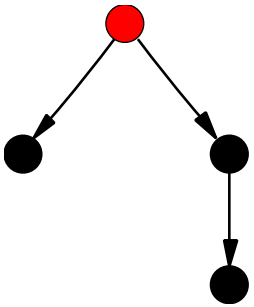


unreachable parts

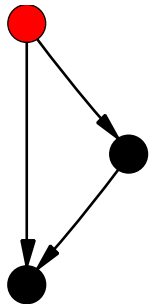
Identifying related pointed graphs

A set can be represented by several non-isomorphic pointed graphs

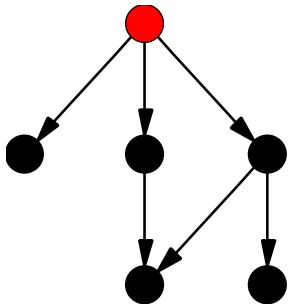
Example: the set $2 = \{\varnothing; \{\varnothing\}\}$



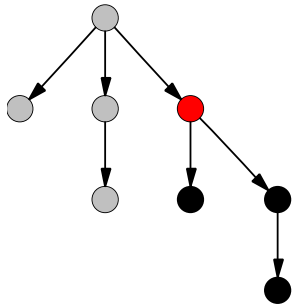
no sharing (tree)



with sharing



duplicate elements



unreachable parts

$+$ Problems related to (possible) non well-foundedness

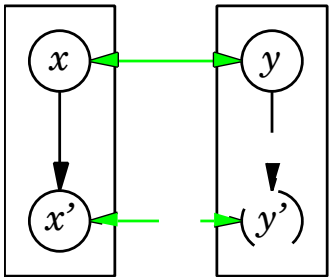
Extensional equality as bisimilarity

Extensional equality as bisimilarity

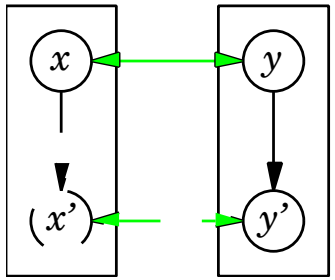
$R : X \to Y \to \mathrm{Prop}$ bisimulation between (X, A, a) and (Y, B, b) if:

1. $\forall x, x' : X \quad \forall y : Y \quad \Big(A(x', x) \wedge R(x, y) \quad \Rightarrow \quad \exists y' : Y \ \big(R(x', y') \wedge B(y', y) \big) \Big)$
2. $\forall x : X \quad \forall y, y' : Y \quad \Big(B(y', y) \wedge R(x, y) \quad \Rightarrow \quad \exists x' : X \ \big(R(x', y') \wedge A(x', x) \big) \Big)$
3. $R(a, b)$

(1)



(2)

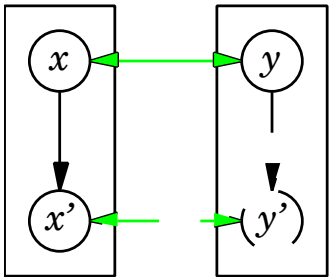


Extensional equality as bisimilarity

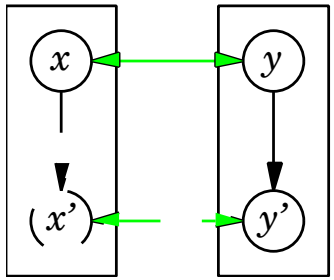
$R : X \to Y \to \mathrm{Prop}$ bisimulation between (X, A, a) and (Y, B, b) if:

1. $\forall x, x' : X \quad \forall y : Y \quad \Big(A(x', x) \wedge R(x, y) \quad \Rightarrow \quad \exists y' : Y \ \big(R(x', y') \wedge B(y', y) \big) \Big)$
2. $\forall x : X \quad \forall y, y' : Y \quad \Big(B(y', y) \wedge R(x, y) \quad \Rightarrow \quad \exists x' : X \ \big(R(x', y') \wedge A(x', x) \big) \Big)$
3. $R(a, b)$

(1)



(2)



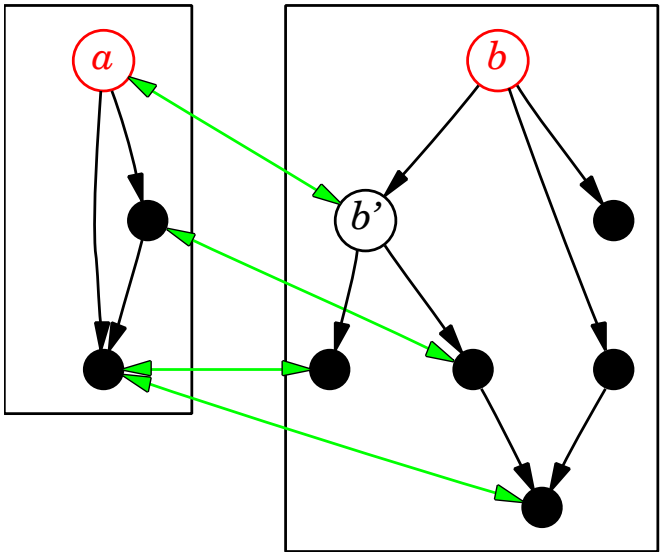
$(X, A, a) \approx (Y, B, b) \quad \equiv \quad \exists R : X \to Y \to \mathrm{Prop} \quad \text{bisimulation}$

Membership as shifted bisimilarity

$$(X, A, a) \in (Y, B, b) \quad \equiv \quad \exists b' : Y \;\; \big((X, A, a) \approx (Y, B, b') \;\wedge\; B(b', b)\big)$$

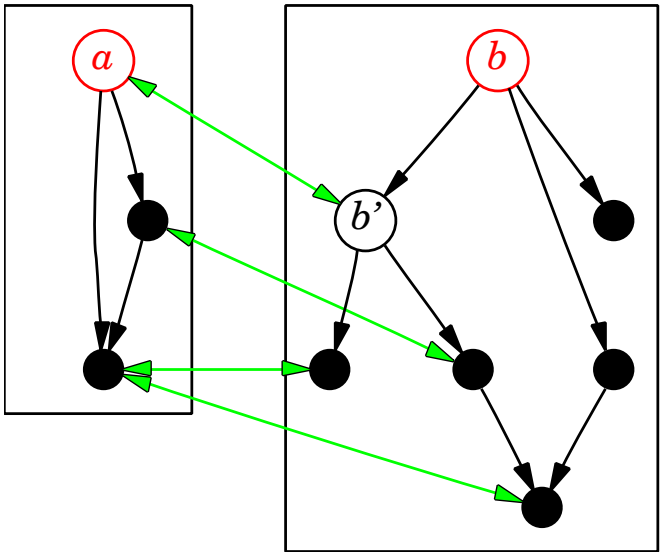
Membership as shifted bisimilarity

$$(X, A, a) \in (Y, B, b) \quad \equiv \quad \exists b' : Y \;\; \big((X, A, a) \approx (Y, B, b') \;\wedge\; B(b', b)\big)$$



Membership as shifted bisimilarity

$$(X, A, a) \in (Y, B, b) \quad \equiv \quad \exists b' : Y \;\; \big((X, A, a) \approx (Y, B, b') \;\wedge\; B(b', b)\big)$$



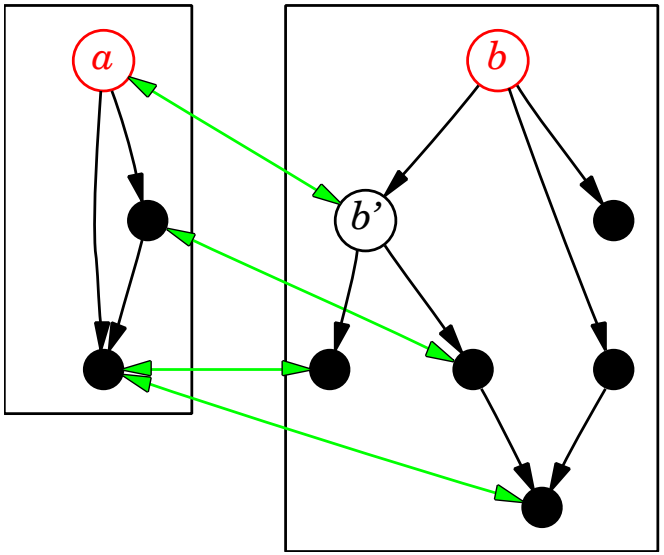
- Compatibility of $\in$ w.r.t $\approx$

$$G_1 \approx G_2 \;\wedge\; G_2 \in G_3 \;\Rightarrow\; G_1 \in G_3$$

$$G_1 \in G_2 \;\wedge\; G_2 \approx G_3 \;\Rightarrow\; G_1 \in G_3$$

Membership as shifted bisimilarity

$$(X, A, a) \in (Y, B, b) \quad \equiv \quad \exists b' : Y \ \left((X, A, a) \approx (Y, B, b') \ \wedge \ B(b', b) \right)$$



- Compatibility of $\in$ w.r.t $\approx$

  $G_1 \approx G_2 \ \wedge \ G_2 \in G_3 \ \Rightarrow \ G_1 \in G_3$

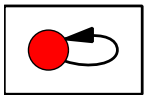
  $G_1 \in G_2 \ \wedge \ G_2 \approx G_3 \ \Rightarrow \ G_1 \in G_3$

- Extensionality of $\approx$ w.r.t. $\in$

  $\forall G \ (G \in G_1 \ \Leftrightarrow \ G \in G_2) \ \Rightarrow \ G_1 \approx G_2$

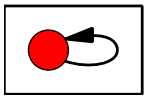
Non well-founded sets

Non well-founded sets

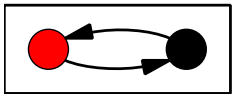


represents a set x such that $x = \{x\}$

Non well-founded sets

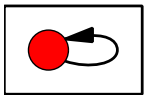


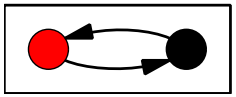
represents a set x such that $x = \{x\}$



represents a set y such that $y = \{z\}$ and $z = \{y\}$ for some z

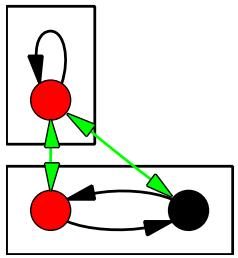
Non well-founded sets

 represents a set x such that $x = \{x\}$

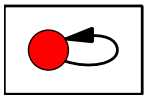
 represents a set y such that $y = \{z\}$ and $z = \{y\}$ for some z

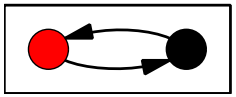
Since there is a bisimulation, we have

$x = y = z = \{x\} = \{y\} = \{z\}$



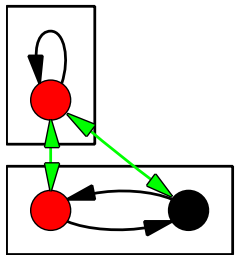
Non well-founded sets

 represents a set x such that $x = \{x\}$

 represents a set y such that $y = \{z\}$ and $z = \{y\}$ for some z

Since there is a bisimulation, we have

$x = y = z = \{x\} = \{y\} = \{z\}$



Sets as pointed graphs $+$ Equality as a bisimulation

$\Rightarrow$ Interprets the Anti-Foundation Axiom (*AFA*) [P. Aczel]

The universal type for representing pointed graphs

The universal type for representing pointed graphs

$$\text{Let} \quad U \quad := \quad \Big(\Pi T : \text{Type} \,.\, (T \to T \to \text{Prop}) \to T \to \text{Prop}\Big) \to \text{Prop}$$

$$\begin{aligned} \text{and} \quad i \quad &: \quad \Pi X : \text{Type} \,.\, (X \to X \to \text{Prop}) \to X \to U \\ &:= \quad \lambda X, A, a \,.\, \lambda f \,.\, f \;\; X \;\; A \;\; a \end{aligned}$$

The universal type for representing pointed graphs

$$\text{Let} \quad U \quad := \quad \Big(\Pi T : \text{Type}\,.\,(T \to T \to \text{Prop}) \to T \to \text{Prop}\Big) \to \text{Prop}$$

$$\begin{aligned} \text{and} \quad i \quad &: \quad \Pi X : \text{Type}\,.\,(X \to X \to \text{Prop}) \to X \to U \\ &:= \quad \lambda X, A, a\,.\,\lambda f\,.\,f\ X\ A\ a \end{aligned}$$

- Higher-level impredicativity $(\text{Kind}, \text{Type})$ ensures that $U : \text{Type}$

The universal type for representing pointed graphs

$$\text{Let} \quad U \quad := \quad \Big(\Pi T : \text{Type}\,.\,(T \to T \to \text{Prop}) \to T \to \text{Prop}\Big) \to \text{Prop}$$

$$\begin{aligned} \text{and} \quad i \quad &: \quad \Pi X : \text{Type}\,.\,(X \to X \to \text{Prop}) \to X \to U \\ &:= \quad \lambda X, A, a\,.\,\lambda f\,.\,f\ X\ A\ a \end{aligned}$$

- Higher-level impredicativity $(\text{Kind}, \text{Type})$ ensures that $U : \text{Type}$
- The map i is an embedding of pointed graphs into U

$$i(X, A, a) = i(Y, B, b) \quad \Rightarrow \quad (X, A, a) \approx (Y, B, b)$$

The universal type for representing pointed graphs

$$\text{Let} \quad U \quad := \quad \Big(\Pi T : \text{Type} \,.\, (T \to T \to \text{Prop}) \to T \to \text{Prop}\Big) \to \text{Prop}$$

$$\begin{aligned} \text{and} \quad i \quad &: \quad \Pi X : \text{Type} \,.\, (X \to X \to \text{Prop}) \to X \to U \\ &:= \quad \lambda X, A, a \,.\, \lambda f \,.\, f \;\; X \;\; A \;\; a \end{aligned}$$

- Higher-level impredicativity $(\text{Kind}, \text{Type})$ ensures that $U : \text{Type}$
- The map i is an embedding of pointed graphs into U

$$i(X, A, a) = i(Y, B, b) \quad \Rightarrow \quad (X, A, a) \;\approx\; (Y, B, b)$$

- The map i is not surjective:

$$r : U \;=\; \lambda f \,.\, \bot \quad \text{is outside the codomain of } i$$

Translating equivalence and membership on U

$$u \approx v \quad := \quad \exists X, A, a \quad \exists Y, B, b \\ \big(u = i(X, A, a) \ \wedge \ v = i(Y, B, b) \ \wedge \ (X, A, a) \approx (Y, B, b)\big)$$

$$u \in v \quad := \quad \exists X, A, a \quad \exists Y, B, b \\ \big(u = i(X, A, a) \ \wedge \ v = i(Y, B, b) \ \wedge \ (X, A, a) \in (Y, B, b)\big)$$

Translating equivalence and membership on U

$$u \approx v \quad := \quad \exists X, A, a \quad \exists Y, B, b \quad \big(u = i(X, A, a) \ \wedge \ v = i(Y, B, b) \ \wedge \ (X, A, a) \approx (Y, B, b)\big)$$

$$u \in v \quad := \quad \exists X, A, a \quad \exists Y, B, b \quad \big(u = i(X, A, a) \ \wedge \ v = i(Y, B, b) \ \wedge \ (X, A, a) \in (Y, B, b)\big)$$

$$\mathrm{set}(u) \quad := \quad \exists X, A, a \qquad u = i(X, A, a) \qquad (\equiv \text{ codomain of } i)$$

Translating equivalence and membership on U

$$u \approx v \quad := \quad \exists X, A, a \quad \exists Y, B, b \quad \big(u = i(X, A, a) \ \wedge \ v = i(Y, B, b) \ \wedge \ (X, A, a) \approx (Y, B, b)\big)$$

$$u \in v \quad := \quad \exists X, A, a \quad \exists Y, B, b \quad \big(u = i(X, A, a) \ \wedge \ v = i(Y, B, b) \ \wedge \ (X, A, a) \in (Y, B, b)\big)$$

$$\mathrm{set}(u) \quad := \quad \exists X, A, a \qquad u = i(X, A, a) \qquad (\equiv \text{ codomain of } i)$$

- $\approx$ (on U) is now a partial equivalence relation

Translating equivalence and membership on U

$$u \approx v \quad := \quad \exists X, A, a \quad \exists Y, B, b \quad \big(u = i(X, A, a) \ \wedge \ v = i(Y, B, b) \ \wedge \ (X, A, a) \approx (Y, B, b)\big)$$

$$u \in v \quad := \quad \exists X, A, a \quad \exists Y, B, b \quad \big(u = i(X, A, a) \ \wedge \ v = i(Y, B, b) \ \wedge \ (X, A, a) \in (Y, B, b)\big)$$

$$\mathrm{set}(u) \quad := \quad \exists X, A, a \qquad u = i(X, A, a) \qquad (\equiv \text{ codomain of } i)$$

- $\approx$ (on U) is now a partial equivalence relation
- Relations $\approx$ and $\in$ are defined on elements $u : U$ s.t. $\mathrm{set}(u)$

Translating equivalence and membership on U

$$u \approx v \quad := \quad \exists X, A, a \quad \exists Y, B, b \quad \big(u = i(X, A, a) \ \wedge \ v = i(Y, B, b) \ \wedge \ (X, A, a) \approx (Y, B, b)\big)$$

$$u \in v \quad := \quad \exists X, A, a \quad \exists Y, B, b \quad \big(u = i(X, A, a) \ \wedge \ v = i(Y, B, b) \ \wedge \ (X, A, a) \in (Y, B, b)\big)$$

$$\mathrm{set}(u) \quad := \quad \exists X, A, a \qquad u = i(X, A, a) \qquad (\equiv \text{ codomain of } i)$$

- $\approx$ (on U) is now a partial equivalence relation
- Relations $\approx$ and $\in$ are defined on elements $u : U$ s.t. $\mathrm{set}(u)$
- Other properties of $\approx$ and $\in$ are kept (compatibility, extensionality)

Translating equivalence and membership on U

$$u \approx v \quad := \quad \exists X, A, a \quad \exists Y, B, b \quad \big(u = i(X, A, a) \ \wedge \ v = i(Y, B, b) \ \wedge \ (X, A, a) \approx (Y, B, b)\big)$$

$$u \in v \quad := \quad \exists X, A, a \quad \exists Y, B, b \quad \big(u = i(X, A, a) \ \wedge \ v = i(Y, B, b) \ \wedge \ (X, A, a) \in (Y, B, b)\big)$$

$$\mathrm{set}(u) \quad := \quad \exists X, A, a \qquad u = i(X, A, a) \qquad (\equiv \text{ codomain of } i)$$

- $\approx$ (on U) is now a partial equivalence relation
- Relations $\approx$ and $\in$ are defined on elements $u : U$ s.t. $\mathrm{set}(u)$
- Other properties of $\approx$ and $\in$ are kept (compatibility, extensionality)
- Exists some object $r : U$ such that $\neg\mathrm{set}(r)$

The unbounded comprehension scheme

Let $P : U \to \text{Prop}$ be a predicate over objects of type U

The unbounded comprehension scheme

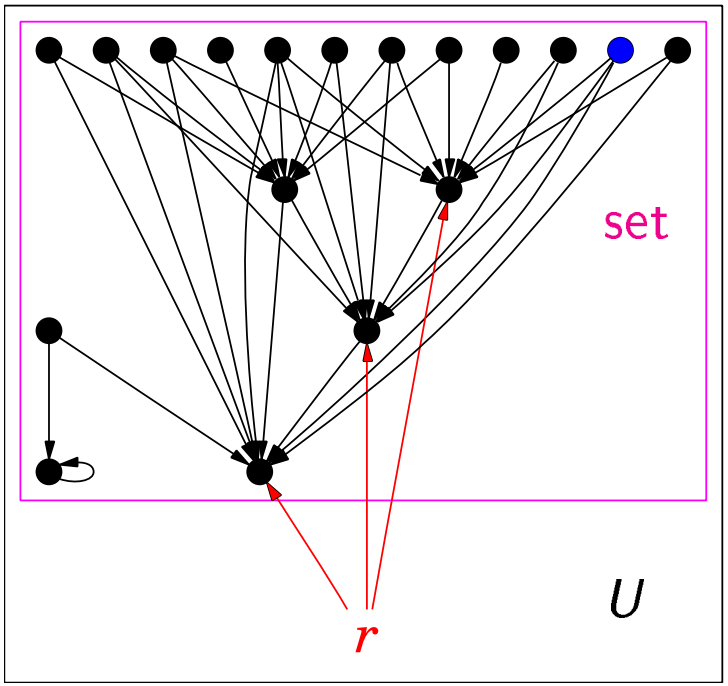
Let $P : U \to \mathrm{Prop}$ be a predicate over objects of type U

We assume P extensional: $\forall u, u' : U \,.\, (P(u) \wedge u \approx u' \Rightarrow P(u'))$

The unbounded comprehension scheme

Let $P : U \longrightarrow \mathrm{Prop}$ be a predicate over objects of type U

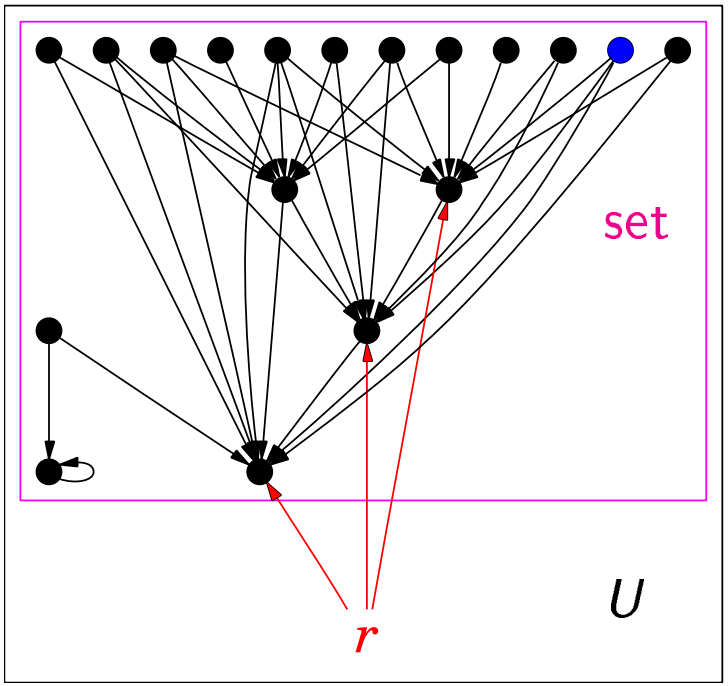
We assume P extensional: $\forall u, u' : U \,.\, (P(u) \wedge u \approx u' \Rightarrow P(u'))$



The unbounded comprehension scheme

Let $P : U \longrightarrow \mathrm{Prop}$ be a predicate over objects of type U

We assume P extensional: $\forall u, u' : U \,.\, (P(u) \wedge u \approx u' \Rightarrow P(u'))$

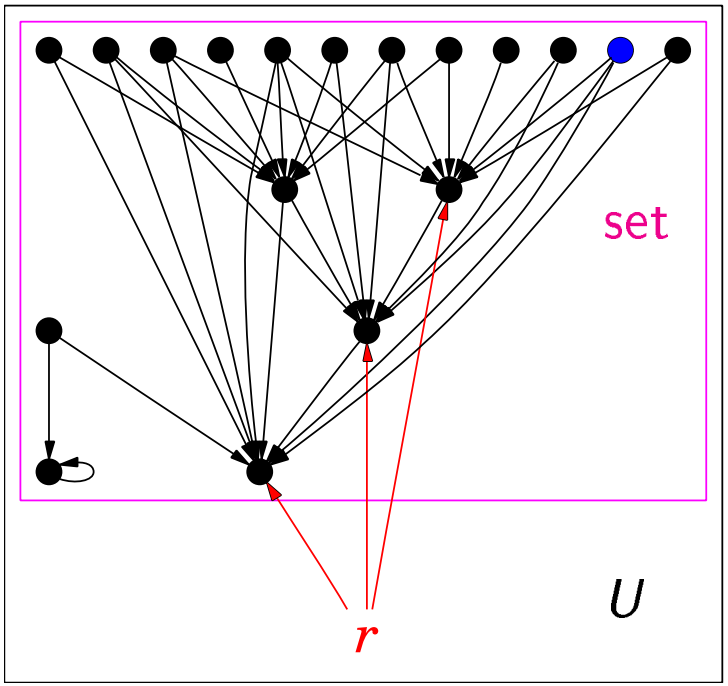


1. Connect r to all $\bullet$ s.t. $P(\bullet)$

The unbounded comprehension scheme

Let $P : U \to \mathrm{Prop}$ be a predicate over objects of type U

We assume P extensional: $\forall u, u' : U \,.\, (P(u) \wedge u \approx u' \Rightarrow P(u'))$

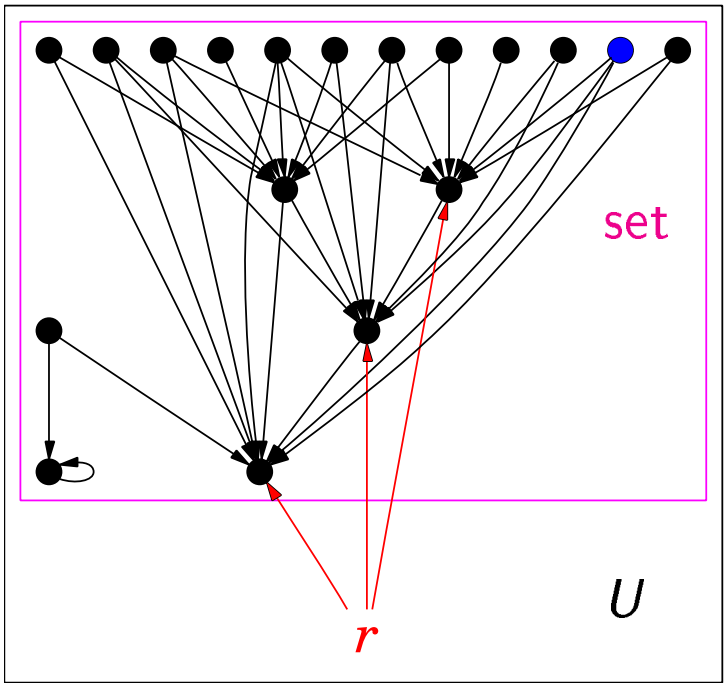


1. Connect r to all $\bullet$ s.t. $P(\bullet)$
2. Let $R_P = \{\to\} \cup \{\to\}$

The unbounded comprehension scheme

Let $P : U \to \mathrm{Prop}$ be a predicate over objects of type U

We assume P extensional: $\forall u, u' : U \,.\, (P(u) \wedge u \approx u' \Rightarrow P(u'))$



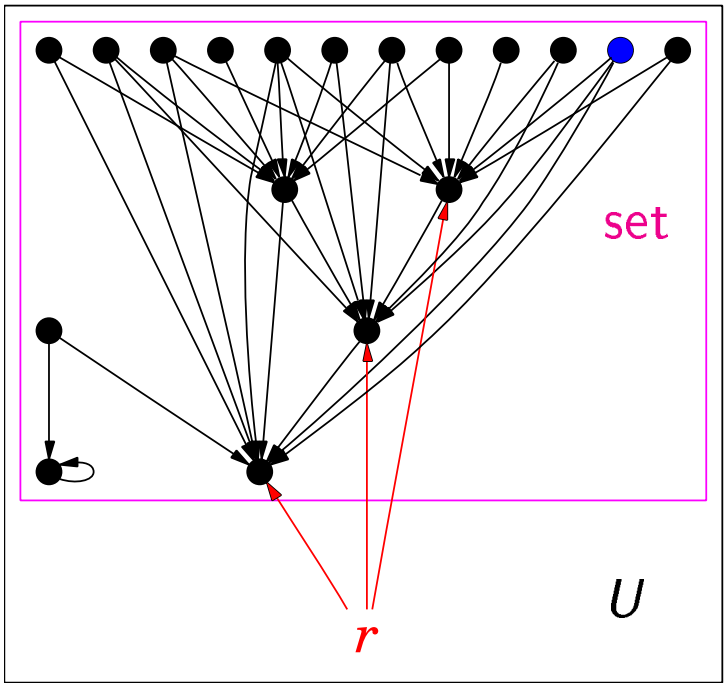
1. Connect r to all $\bullet$ s.t. $P(\bullet)$
2. Let $R_P = \{\to\} \cup \{\to\}$
3. Reflect (U, R_P, r) into U, setting

$$\mathrm{fold}(P) = i(U, R_P, r) \qquad (\equiv \bullet)$$

The unbounded comprehension scheme

Let $P : U \to \mathrm{Prop}$ be a predicate over objects of type U

We assume P extensional: $\forall u, u' : U \,.\, (P(u) \wedge u \approx u' \Rightarrow P(u'))$



1. Connect r to all $\bullet$ s.t. $P(\bullet)$
2. Let $R_P = \{\to\} \cup \{\to\}$
3. Reflect (U, R_P, r) into U, setting

$$\mathrm{fold}(P) = i(U, R_P, r) \qquad (\equiv \bullet)$$

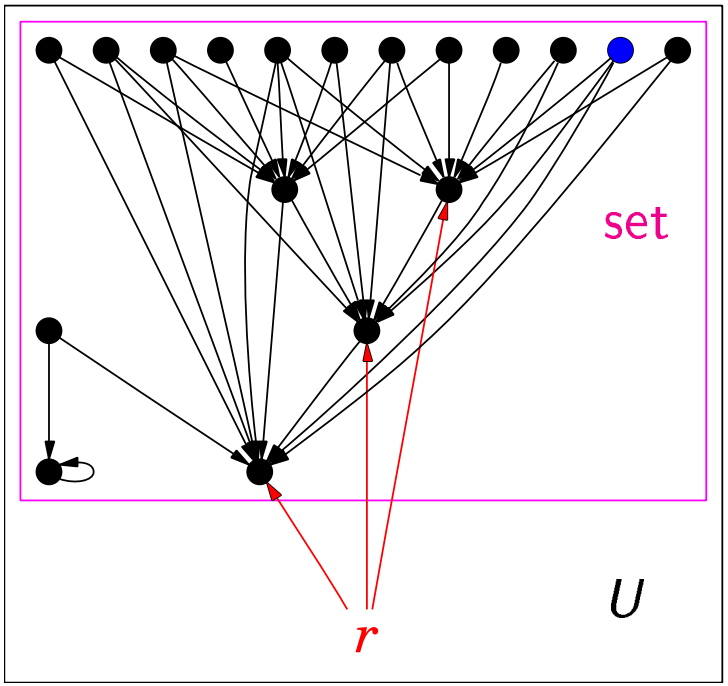
$\Rightarrow$ Relies on the embedding property

$$(X, A, a) \approx \big(U, \in, i(X, A, a)\big)$$

The unbounded comprehension scheme

Let $P : U \to \mathrm{Prop}$ be a predicate over objects of type U

We assume P extensional: $\forall u, u' : U \,.\, (P(u) \wedge u \approx u' \Rightarrow P(u'))$



1. Connect r to all $\bullet$ s.t. $P(\bullet)$
2. Let $R_P = \{\to\} \cup \{\to\}$
3. Reflect (U, R_P, r) into U, setting

   $\mathrm{fold}(P) = i(U, R_P, r) \qquad (\equiv \bullet)$

$\Rightarrow$ Relies on the embedding property

$(X, A, a) \approx \big(U, \in, i(X, A, a)\big)$

Fact (Unbounded comprehension)

$\forall u : U \,.\, (u \in i(U, R_P, r) \Leftrightarrow P(u))$ (if P is extensional)

Cantor-Frege's set theory in systems U/U^-

Type U $+$ two relations $\approx$ and $\in$

Cantor-Frege's set theory in systems U/U^-

Type U $+$ two relations $\approx$ and $\in$

- $\in$ is compatible w.r.t. $\approx$

Cantor-Frege's set theory in systems U/U^-

Type U + two relations $\approx$ and $\in$

- $\in$ is compatible w.r.t. $\approx$
- $\approx$ is extensional w.r.t. $\in$

Cantor-Frege's set theory in systems U/U^-

Type U + two relations $\approx$ and $\in$

- $\in$ is compatible w.r.t. $\approx$
- $\approx$ is extensional w.r.t. $\in$
- The unbounded comprehension scheme is derivable

Cantor-Frege's set theory in systems U/U^-

Type U + two relations $\approx$ and $\in$

- $\in$ is compatible w.r.t. $\approx$
- $\approx$ is extensional w.r.t. $\in$
- The unbounded comprehension scheme is derivable

$\Rightarrow$ An embedding of Cantor-Frege's (insonsistent) set theory into U/U^-

Cantor-Frege's set theory in systems U/U^-

Type U + two relations $\approx$ and $\in$

- $\in$ is compatible w.r.t. $\approx$
- $\approx$ is extensional w.r.t. $\in$
- The unbounded comprehension scheme is derivable

$\Rightarrow$ An embedding of Cantor-Frege's (insonsistent) set theory into U/U^-
All the usual paradoxes can be derived (Burali-Forti, Russell, ...)

Cantor-Frege's set theory in systems U/U^-

Type U + two relations $\approx$ and $\in$

- $\in$ is compatible w.r.t. $\approx$
- $\approx$ is extensional w.r.t. $\in$
- The unbounded comprehension scheme is derivable

$\Rightarrow$ An embedding of Cantor-Frege's (insonsistent) set theory into U/U^-
All the usual paradoxes can be derived (Burali-Forti, Russell, ...)

Russell's paradox: Consider the set $\mathrm{fold}(\lambda x\,.\,x \notin x)\ldots$

Cantor-Frege's set theory in systems U/U^-

Type U + two relations $\approx$ and $\in$

- $\in$ is compatible w.r.t. $\approx$
- $\approx$ is extensional w.r.t. $\in$
- The unbounded comprehension scheme is derivable

$\Rightarrow$ An embedding of Cantor-Frege's (insonsistent) set theory into U/U^-
All the usual paradoxes can be derived (Burali-Forti, Russell, ...)

Russell's paradox: Consider the set $\mathrm{fold}(\lambda x\,.\,x \notin x)\ldots$

Remark: The formalization has been presented in system U

If we only consider pointed graphs based on $X = U$, we can drop the $(\mathsf{Kind}, \mathsf{Prop})$-quantification, thus restricting to system U^-

Why is system U^- inconsistent?

Why is system U^- inconsistent?

Kinds	τ, σ	$::=$	$\mathsf{Prop} \mid \alpha$	
		$\mid$	$\tau \to \sigma$	(Type, Type)
		$\mid$	$\Pi\alpha : \mathsf{Type} \,.\, \tau$	(Kind, Type)
Constructors	M, N	$::=$	ξ	
		$\mid$	$\lambda x : \tau \,.\, M \mid MN$	(Type, Type)
		$\mid$	$\Lambda\alpha \,.\, M \mid M\tau$	(Kind, Type)
		$\mid$	$M \Rightarrow N$	(Prop, Prop)
		$\mid$	$\forall x : \tau \,.\, M$	(Type, Prop)
Proof-terms	t, u	$::=$	ξ	
		$\mid$	$\lambda x : M \,.\, t \mid tu$	(Prop, Prop)
		$\mid$	$\lambda x : \tau \,.\, t \mid tM$	(Type, Prop)

Why is system U^- inconsistent?

Kinds	τ, σ	$::=$	$\mathsf{Prop} \mid \alpha$	
		$\mid$	$\tau \to \sigma$	$(\mathsf{Type}, \mathsf{Type})$
		$\mid$	$\Pi\alpha : \mathsf{Type} \,.\, \tau$	$(\mathsf{Kind}, \mathsf{Type})$
Constructors	M, N	$::=$	ξ	
		$\mid$	$\lambda x : \tau \,.\, M \mid MN$	$(\mathsf{Type}, \mathsf{Type})$
		$\mid$	$\Lambda\alpha \,.\, M \mid M\tau$	$(\mathsf{Kind}, \mathsf{Type})$
		$\mid$	$M \Rightarrow N$	$(\mathsf{Prop}, \mathsf{Prop})$
		$\mid$	$\forall x : \tau \,.\, M$	$(\mathsf{Type}, \mathsf{Prop})$
Proof-terms	t, u	$::=$	ξ	
		$\mid$	$\lambda x : M \,.\, t \mid tu$	$(\mathsf{Prop}, \mathsf{Prop})$
		$\mid$	$\lambda x : \tau \,.\, t \mid tM$	$(\mathsf{Type}, \mathsf{Prop})$

Why is system U^- inconsistent?

Kinds	τ, σ	$::=$	$\mathsf{Prop} \mid \alpha$	
		$\mid$	$\tau \to \sigma$	(Type, Type)
		$\mid$	$\Pi\alpha : \mathsf{Type} \,.\, \tau$	(Kind, Type)
Constructors	M, N	$::=$	ξ	
		$\mid$	$\lambda x : \tau \,.\, M \mid MN$	(Type, Type)
		$\mid$	$\Lambda\alpha \,.\, M \mid M\tau$	(Kind, Type)
		$\mid$	$M \Rightarrow N$	(Prop, Prop)
		$\mid$	$\forall x : \tau \,.\, M$	(Type, Prop)
Proof-terms	t, u	$::=$	ξ	
		$\mid$	$\lambda x \,.\, t \mid tu$	(Prop, Prop)

- (Type, Prop)-abstraction/application can be erased

Why is system U^- inconsistent?

Kinds	τ, σ	$::=$	$\mathsf{Prop} \mid \alpha$	
		$\mid$	$\tau \to \sigma$	(Type, Type)
		$\mid$	$\Pi\alpha : \mathsf{Type} \,.\, \tau$	(Kind, Type)
Constructors	M, N	$::=$	ξ	
		$\mid$	$\lambda x : \tau \,.\, M \mid MN$	(Type, Type)
		$\mid$	$\Lambda\alpha \,.\, M \mid M\tau$	(Kind, Type)
		$\mid$	$M \Rightarrow N$	(Prop, Prop)
		$\mid$	$\forall x : \tau \,.\, M$	(Type, Prop)
Proof-terms	t, u	$::=$	ξ	
		$\mid$	$\lambda x \,.\, t \mid tu$	(Prop, Prop)

- (Type, Prop)-abstraction/application can be erased

Why is system U^- inconsistent?

Kinds	τ, σ	$::=$	$\mathsf{Prop} \mid \alpha$	
		$\mid$	$\tau \longrightarrow \sigma$	(Type, Type)
		$\mid$	$\Pi\alpha : \mathsf{Type} \,.\, \tau$	(Kind, Type)
Constructors	M, N	$::=$	ξ	
		$\mid$	$\lambda x \,.\, M \mid MN$	(Type, Type)
		$\mid$		(Kind, Type)
		$\mid$	$M \Rightarrow N$	(Prop, Prop)
		$\mid$	$\forall x : \tau \,.\, M$	(Type, Prop)
Proof-terms	t, u	$::=$	ξ	
		$\mid$	$\lambda x \,.\, t \mid tu$	(Prop, Prop)

- (Type, Prop)-abstraction/application can be erased
- We can erase $\Lambda\alpha \,.\, M$ + $M\tau$ + type in $\lambda x : \tau \,.\, M$...

Why is system U^- inconsistent?

Kinds	τ, σ	$::=$	$\mathsf{Prop} \mid \alpha$	
		$\mid$	$\tau \longrightarrow \sigma$	(Type, Type)
		$\mid$	$\Pi\alpha : \mathsf{Type} \,.\, \tau$	(Kind, Type)
Constructors	M, N	$::=$	ξ	
		$\mid$	$\lambda x \,.\, M \mid MN$	(Type, Type)
		$\mid$		(Kind, Type)
		$\mid$	$M \Rightarrow N$	(Prop, Prop)
		$\mid$	$\forall x : \tau \,.\, M$	(Type, Prop)
Proof-terms	t, u	$::=$	ξ	
		$\mid$	$\lambda x \,.\, t \mid tu$	(Prop, Prop)

- (Type, Prop)-abstraction/application can be erased
- We can erase $\Lambda\alpha \,.\, M$ + $M\tau$ + type in $\lambda x : \tau \,.\, M$...
  ... but makes no sense to remove τ in $\forall x : \tau \,.\, M$

Why is system U^- inconsistent?

Kinds	τ, σ	$::=$	$\mathsf{Prop} \mid \alpha$	
		$\mid$	$\tau \to \sigma$	(Type, Type)
		$\mid$	$\Pi\alpha : \mathsf{Type} \,.\, \tau$	(Kind, Type)
Constructors	M, N	$::=$	ξ	
		$\mid$	$\lambda x \,.\, M \mid MN$	(Type, Type)
		$\mid$		(Kind, Type)
		$\mid$	$M \Rightarrow N$	(Prop, Prop)
		$\mid$	$\forall x : \tau \,.\, M$	(Type, Prop)
Proof-terms	t, u	$::=$	ξ	
		$\mid$	$\lambda x \,.\, t \mid tu$	(Prop, Prop)

- (Type, Prop)-abstraction/application can be erased
- We can erase $\Lambda\alpha \,.\, M \;+\; M\tau \;+$ type in $\lambda x : \tau \,.\, M$...
  ... but makes no sense to remove τ in $\forall x : \tau \,.\, M$
  Would identify propositions $\forall x, y : \mathsf{Unit} \,.\, x = y$ with $\forall x, y : \mathsf{Bool} \,.\, x = y$

Why is system U^- inconsistent?

Kinds	τ, σ	$::=$	$\mathsf{Prop} \mid \alpha$	
		$\mid$	$\tau \to \sigma$	(Type, Type)
		$\mid$	$\Pi\alpha : \mathsf{Type} \,.\, \tau$	(Kind, Type)
Constructors	M, N	$::=$	ξ	
		$\mid$	$\lambda x \,.\, M \mid MN$	(Type, Type)
		$\mid$		(Kind, Type)
		$\mid$	$M \Rightarrow N$	(Prop, Prop)
		$\mid$	$\forall x : \tau \,.\, M$	(Type, Prop)
Proof-terms	t, u	$::=$	ξ	
		$\mid$	$\lambda x \,.\, t \mid tu$	(Prop, Prop)

- (Type, Prop)-abstraction/application can be erased
- We can erase $\Lambda\alpha \,.\, M \;+\; M\tau \;+$ type in $\lambda x : \tau \,.\, M$...
  ... but makes no sense to remove τ in $\forall x : \tau \,.\, M$
  Would identify propositions $\forall x, y : \mathsf{Unit} \,.\, x = y$ with $\forall x, y : \mathsf{Bool} \,.\, x = y$

$\Rightarrow$ (Kind, Type)-impredicativity is not parametric
i.e. cannot be reduced to an intersection

Types, Propositions and Problems

an introduction to type theoretical ideas

Bengt Nordström

Computing Science, Chalmers and University of Göteborg

Types Summer School, Hisingen, 15 August 2005

Classical logic, truth tables

Conjunction

A	B	$A \& B$
T	T	T
T	F	F
F	T	F
F	F	F

Disjunction

A	B	$A \vee B$
T	T	T
T	F	T
F	T	T
F	F	F

Implication

A	B	$A \supset B$
T	T	T
T	F	F
F	T	T
F	F	T

This assumes that a proposition is either true or false!

Brouwer

Brouwer rejected the idea that the meaning of a mathematical proposition is its truth value.
Mathematical propositions do not exist independently of us.
We cannot say that a proposition is true without having a proof of it.

Heyting

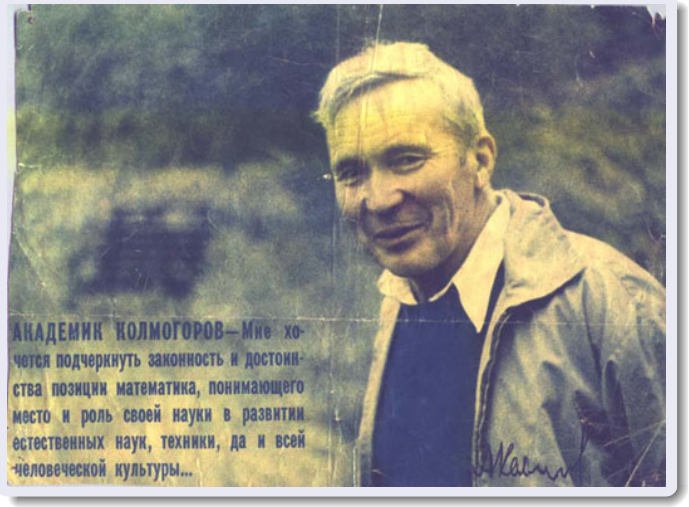
Heyting was a student of Brouwer.
He gave the following explanation of the logical constants.

Heyting's explanation of the logical constants (1930)

A proof of:	consists of:
$A \,\&\, B$	a proof of A and a proof of B
$A \vee B$	a proof of A or a proof of B
$A \supset B$	a method which takes any proof of A to a proof of B
$\neg A$	a method which takes any proof of A to a proof of absurdity
$\bot$	has no proof
$\exists x \in A.B$	an element a in A and a proof of $B[x := a]$
$\forall x \in A.B$	a method, which takes any element x in A to a proof of $B[x := a]$

Kolmogorov

Independently of Heyting, Kolmogorov interpreted propositions as problems.



Kolmogorov understood the logical constants as problems (1932)

The problem:	is solved if we can:
$A \,\&\, B$	solve A and solve B
$A \vee B$	solve A or solve B
$A \supset B$	reduce the solution of B to the solution of A
$\neg A$	show that there is no solution of A
$\bot$	has no solution

Heyting's and Kolmogorov's explanation

A proof (solution) of:	consists of:
$A \,\&\, B$	a proof (solution) of A and a proof (solution) of B
$A \vee B$	a proof (solution) of A or a proof (solution) of B
$A \supset B$	a method which takes any proof (solution) of A to a proof (solution) of B
$\neg A$	a method which takes any proof (solution) of A to a proof (solution) of absurdity
$\bot$	has no proof (solution)
$\exists x \in A.B$	an element a in A and a proof (solution) of $B[x := a]$
$\forall x \in A.B$	a method, which takes any element x in A to a proof (solution) of $B[x := a]$

Question:

Is this correct? Could not a proof (solution) of $A \,\&\, B$ be obtained by induction, for instance?

Direct and indirect proofs

When we say that we have a proof of a proposition, then we mean that we have a method which when computed yields a direct proof of it.
Compare this with mathematics and programming: When we say that $2+4$ and $\mathrm{fst}(<45^2, -9>)$ are natural numbers, then we mean that they can be *computed* to a natural number.

Terminology:

proofs:	objects:
direct vs. indirect proof	value vs. expression
canonical vs. non-canonical proof	canonical vs. non-canonical element
introduction vs. elimination proof	

Examples of indirect proofs

And-elimination

$$\frac{A \,\&\, B}{A}$$

If we have a proof of $A \,\&\, B$, then we can compute it to a direct proof. This always consists of a proof of A and a proof of B. Hence we may always obtain a proof of A from a proof of $A \,\&\, B$.

Mathematical induction

$$\frac{n \in \mathrm{N} \quad P(0) \quad (\forall n \in \mathrm{N}) P(n) \supset P(\mathrm{succ}(n))}{P(i)}$$

What is a proposition (problem)?

To summarize Heyting's and Kolmogorov's explanations:

What does it mean to understand a proposition?

I understand a proposition when I understand what a direct proof of it is.

This looks very similar to:

What does it mean to understand a set?

I understand a set when I understand what a canonical element of it is.

Propositions and sets

A proof (element) of:	consists of:
$A \,\&\, B$	a proof (solution) of A and a proof (solution) of B
$A \times B$	an element in A and an element in B
$A \vee B$	a proof (solution) of A or a proof (solution) of B
$A + B$	an element in A or an element in B
$A \supset B$	a method which takes any proof (solution) of A to a proof (solution) of B
$A \to B$	a method which takes any element in A to an element in B
$\bot$	has no proof (solution)
$\emptyset$	has no elements
$\exists x \in A.B$	an element a in A and a proof (solution) of $B[x := a]$
$\forall x \in A.B$	a method, which takes any element x in A to a proof (solution) of $B[x := a]$

This similarity leads to the

Curry-Howard isomorphism

$$A \,\&\, B = A \times B$$
$$A \vee B = A + B$$
$$A \supset B = A \to B$$
$$\bot = \emptyset$$
$$\neg A = A \to \emptyset$$

Curry's contribution

Curry noticed the formal similarity between the axioms of positive implicational logic:

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

and the type of the basic combinators:

$$K \in A \to B \to A$$
$$S \in (A \to B \to C) \to (A \to B) \to C$$

and modus ponens corresponds to the typing rule for application:

$$\frac{A \supset B \quad A}{A} \qquad \frac{f \in A \to B \quad a \in A}{f\ a \in B}$$

Proofs as Programs in a functional programming language

A direct proof of:	consists of:	As a type:
$A \vee B$	a proof of A or a proof of B	**data** Or A B = Ori1 A | Ori2 B;
$A \,\&\, B$	a proof of A and a proof of B	**data** And A B = Andi A B;
$A \supset B$	a method taking a proof of A to a proof of B	**data** Implies A B = Impi A → B;
Falsity		**data** Falsity = ;

Constructors are introduction rules

$$\frac{A}{A \vee B} \qquad \textbf{Ori1} \in A \to A \vee B$$

$$\frac{B}{A \vee B} \qquad \textbf{Ori2} \in B \to A \vee B$$

$$\frac{A \quad B}{A \,\&\, B} \qquad \textbf{Andi} \in A \to B \to A \,\&\, B$$

$$\frac{\begin{matrix}[A] \\ B\end{matrix}}{A \supset B} \qquad \textbf{Impli} \in (A \to B) \to A \supset B$$

Elimination rules can be defined

$\mathbf{orel} \in A \vee B \to (A \to C) \to (B \to C) \to C$

$\mathbf{orel}\ (\mathbf{Ori1}\ a)\ f\ g = f\ a$

$\mathbf{orel}\ (\mathbf{Ori2}\ b)\ f\ g = g\ b$

$$\frac{A \vee B \qquad \begin{matrix}[A]\\ C\end{matrix} \qquad \begin{matrix}[B]\\ C\end{matrix}}{C}\ \mathbf{orel}$$

$\mathbf{andel} \in A\,\&\,B \to (A \to B \to C) \to C$

$\mathbf{andel}\ (\mathbf{Andi}\ a\ b)\ f = f\ a\ b$

$$\frac{A\,\&\,B \qquad \begin{matrix}[A, B]\\ C\end{matrix}}{C}\ \mathbf{andel}$$

$\mathbf{implel} \in A \supset B \to A \to B$

$\mathbf{implel}\ (\mathbf{Impli}\ f)\ a = f\ a$

$$\frac{A \supset B \quad A}{B}\ \mathbf{implel}$$

Proof checking = Type checking

In this way we can prove propositional formulas in a typed functional programming language. The problem of proving for instance

$$(A\,\&\,B) \supset (B\,\&\,A)$$

is then the problem of finding a program in this type. The type checker will check if the proof is correct. In this case, we can use the following program:

$$\mathbf{Impli}\ (\lambda x.\mathbf{Andi}\ (\mathbf{andel}\ x\ \lambda y.\lambda z.z)\ (\mathbf{andel}\ x\ \lambda y.\lambda z.y))$$

What about the quantifiers?

Propositions and sets

A proof (element) of:	consists of:
$\exists x \in A.B$	an element a in A and a proof (solution) of $B[x := a]$
$\Sigma x \in A.B$	an element a in A and an element in $B[x := a]$
$\forall x \in A.B$	a method, which takes any element x in A to a proof (solution) of $B[x := a]$
$\Pi x \in A.B$	a method, which takes any element x in A to an element in $B[x := a]$

Overview of Martin Löf´s type theory

- Type theory is a small typed functional language with one basic type and two type forming operation.
- It is a **framework** for defining logics.
- A new logic is introduced by definitions.

What types are there?

- Set is a type
- $El(A)$ is a type, if $A \in \text{Set}$.
- $(x \in A) \to B$ is a type, if A is a type and B a family of types for $x \in A$.

What programs are there?

Programs are formed from variables and constants using abstraction and application:

- Application
$$\frac{c \in (x \in A) \to B \quad a \in A}{c\ a \in B[x := a]}$$
- Abstraction
$$\frac{b \in B \quad [x \in A]}{[x]b \in (x \in A) \to B}$$
- constants are either primitive or defined

Constants

There are two kinds of constants:

primitive: (not defined) have a type but no definiens (RHS):

$$\text{identifier} \in \text{Type}$$

defined: have a type and a definiens:

$$\text{identifier} = \text{expr} \in \text{Type}$$

There are two kinds of defined constants:

- explicitly defined
- implicitly defined

Primitive constants

- computes to themselves (i.e. are values).
- constructors in functional languages.
- introduction rules and formation rules in logic
- postulates

Examples:

$$\begin{array}{rcl}
\mathsf{N} & \in & \text{Set} \\
0 & \in & \mathsf{N} \\
\mathsf{s} & \in & \mathsf{N} \to \mathsf{N} \\
\& & \in & \text{Set} \to \text{Set} \to \text{Set} \\
\&\mathbf{I} & \in & (A \in \text{Set}) \to (B \in \text{Set}) \to A \to B \to A \,\&\, B \\
\Pi & \in & (A \in \text{Set}) \to (A \to \text{Set}) \to \text{Set} \\
\lambda & \in & (A \in \text{Set}) \to (B \in A \to \text{Set}) \to ((x \in A) \to B(x)) \to \\
 & & \Pi(A, B)
\end{array}$$

Explicitly defined constants

- have a type and a definiens (RHS).
- the definiens is a welltyped expression
- abbreviation
- derived rule in logic.
- names for proofs and theorems in math.

Examples:

$$\begin{aligned} 2 &\in \mathsf{N} \\ &= \mathsf{succ}(\mathsf{succ}\ 0) \\ \forall(A \in \mathsf{Set})(B \in A \to \mathsf{Set}) &\in \mathsf{Set} \\ &= \Pi\ A\ B \\ +(x \in \mathsf{N})(y \in \mathsf{N}) &\in \mathsf{N} \\ &= \mathsf{natrec}\ [x]\mathsf{N}\ x\ y\ [u, v](\mathsf{succ}\ v) \\ \supset (A \in \mathsf{Set})(B \in \mathsf{Set}) &\in \mathsf{Set} \\ &= \Pi\ A\ [x]B \end{aligned}$$

Implicitly defined constants

The definiens (RHS) may contain pattern matching and may contain occurrences of the constant itself. The correctness of the definition must in general be decided outside the system

- Recursively defined programs
- Elimination rules (the step from the definiendum to the definiens is the contraction rule).

Examples:

$$\begin{aligned} \mathbf{add}(x \in \mathsf{N})(y \in \mathsf{N}) &\in \mathsf{N} \\ \mathbf{add}\ 0\ y &= y \\ \mathbf{add}\ (\mathsf{succ}\ u)\ y &= \mathsf{succ}\ (\mathbf{add}\ u\ y) \\ \&\mathbf{E}(A \in \mathsf{Set})(B \in \mathsf{Set})(C \in A \to B \to \mathsf{Set}) & \\ (f \in (x \in A) \to (y \in B) \to C(\&\mathbf{I}\ x\ y)) & \\ (z \in A \,\&\, B) & \\ \in C(z) & \\ \&\mathbf{E}\ A\ B\ C\ f\ (\&\mathbf{I}\ a\ b) &= f\ a\ b \end{aligned}$$

Two approaches to the usage of implicit constants:

- The conservative approach: Use them only to define induction principles for sets (elimination rules). These are functions, which for an inductively defined set A produces a function in $(x \in A) \to (C\ z)$ for a family of sets $C \in A \to \mathsf{Set}$.
- The liberal approach: Use them when they are convenient.

The editing process

The idea is to build expressions from incomplete expressions with holes (placeholders). Each editing step replaces a place holder with another incomplete expression

Place holders

We use the notation

$$\Box_1, \ldots, \Box_n$$

for place holders (holes).
Each place holder has an **expected type** and a **local context** (variables which may be used to fill in the hole).

To construct an object

We start to give the name of the object to define, and the computer responds with

$$c \in \Box_1$$
$$c = \Box_2$$

We must first give the type of c by refining $\Box_1$.
We can either enter text from the keyboard, or do it stepwise, replace it by

- $(x \in \Box_3) \to \Box_4$
- Set, or
- $C\ \Box_3\ \ldots\ \Box_n$

Refinement of an object

When we have constructed the type of the constant c, we can start to define it:

$$c \in C$$
$$c = \Box_0$$

Here, *the expected type* of $\Box_0$ is C .
In general, we are in a situation like

$$c = \ldots \Box_1 \ldots \Box_2 \ldots$$

where we know the expected type of the place holders.

Refinement of an object: application

To refine a place holder

$$\Box_0 \in A$$

with a constant c (or a variable) is to replace it by

$$c\ \Box_1\ \ldots\ \Box_n \in A$$

where $\Box_1 \in B_1, \ldots, \Box_n \in B_n$.
The system computes the expected types of the new place holders and some constraints from the condition that the type of $c\ \Box_1\ \ldots\ \Box_n$ must be equal to A.

We have reduced the problem A to the subproblems $B_1, \ldots B_n$ using the rule c.

Refinement of an object: abstraction

To refine a place holder

$$\Box_0 \in A$$

with an abstraction is to replace it by

$$[x]\Box_1 \in A$$

The system checks that A is a functional type $(x \in B) \to C$ and the expected type of $\Box_1$ is C and the local context for it will contain the assumption $x \in B$.

We have reduced the problem $(x \in B) \to C$ to the problem C by using the assumption $x \in B$.

Hence: to prove is to build a proof object

- To apply a rule **c** is to construct an application of the constant **c**.
- To assume **A** is to construct an abstraction of a variable of type **A**.
- To refer to an assumption of **A** is to use a variable of type **A**.

Summary: Type Theory

- Types:
$$T \ ::= \ \text{Set} \ \mid \ El(e) \ \mid \ (x \in T) \to T'$$
- Programs:
$$e \ ::= \ e \ e' \ \mid \ [x]e \ \mid \ x \ \mid \ c$$
- Constants:
  - Primitive (without a definition):
$$c \in T$$
  - Explicitly defined:
$$c = e \in T$$
  - Implicitly defined:
$$\begin{aligned} c \ p_1 \ \ldots \ p_n \ &= \ e \\ &\vdots \\ c \ p'_1 \ \ldots \ p'_n \ &= \ e' \end{aligned}$$

Proof of normalisation using domain theory

Thierry Coquand and Arnaud Spivak

Aug. 24, 2005

Goal of the presentation

Show an example where computer science helps in simplifying an argument in proof theory

How to prove normalisation for some computation rules introduced in proof theory (variant of bar recursion)

Intuition: if the computation rules make sense, the system should be normalising

Goal of the presentation

This presentation aims to present a simplified version of

Ulrich Berger "Continuous Semantics for Strong Normalisation"
LNCS 3526, 23-34, 2005

This work itsef simplifies the argument in

W.W. Tait "Normal form theorem for bar recursive functions of finite type"
Proceedings of the Second Scandinavian Logic Symposium, North-Holland, 1971

PCF

Introduced by D. Scott in 1969

"A type-theoretical alternative to CUCH, ISWIM and OWHY"

Published in *Theoret. Comput. Sci.* 121 (1993), no. 1-2, 411–440.

This was the basis of the LCF system

PCF

G. Plotkin "LCF considered as a programming language"

Theoretical Computer Science, 5:223-255, 1977

Simply typed λ-calculus with with base types o, ι and constants

Basic operations

$tt : o,\ ff : o,\ k_n : \iota,\ (+1) : \iota \to \iota,\ (-1) : \iota \to \iota,\ Z : \iota \to o$

$\supset_\iota : o, \iota, \iota \to \iota, \quad \supset_o : o, o, o \to o$

$Y_\sigma : (\sigma \to \sigma) \to \sigma$

Operational semantics

$$\frac{}{\lambda x.t \Downarrow \lambda x.t} \qquad \frac{t \Downarrow \lambda x.t' \qquad t'(x = u) \Downarrow v}{t\ u \Downarrow v}$$

$$\frac{a \Downarrow tt \qquad b \Downarrow v}{\supset\ a\ b\ c\ \Downarrow v} \qquad \frac{a \Downarrow ff \qquad c \Downarrow v}{\supset\ a\ b\ c\ \Downarrow v}$$

$$\frac{f\ (Y\ f) \Downarrow v}{Y\ f \Downarrow v}$$

$$\frac{a \Downarrow k_n}{(+1)\ a \Downarrow k_{n+1}} \qquad \frac{a \Downarrow k_{n+1}}{(-1)\ a \Downarrow k_n} \qquad \frac{a \Downarrow k_0}{Z\ a \Downarrow tt} \qquad \frac{a \Downarrow k_{n+1}}{Z\ a \Downarrow ff}$$

Denotational semantics

A *domain* is a complete partial order D, with a least element $\bot$ and a top element $\top$

If D, E are domains, $[D \to E]$ is the complete lattice of *continuous* functions, i.e. monotone and such that $f(\vee_{i\in I} X_i) = \vee_{i\in I} f(X_i)$ for *directed* families (X_i)

We have natural choices for D_ι and D_o

$D_{\sigma\to\tau} = [D_\sigma \to D_\tau]$

We have natural choices for $[\![c]\!] \in D_\sigma$ if $c : \sigma$

$[\![Y]\!]\ f = \bigvee_{n\in\mathbb{N}} f^n\ \ \bot$ so that $[\![Y]\!] \in [[D_\sigma \to D_\sigma] \to D_\sigma]$

Denotational semantics

Given $\rho : \mathcal{V}_\sigma \to D_\sigma$ and $t : \sigma$ we define $[\![t]\!]_\rho \in D_\sigma$ by induction on t

$$[\![x]\!]_\rho = \rho(x)$$

$$[\![\lambda x.t]\!]_\rho = u \longmapsto [\![t]\!]_{(\rho, x=u)}$$

$$[\![t\ u]\!]_\rho = [\![t]\!]_\rho\ [\![u]\!]_\rho$$

$$[\![c]\!]_\rho = [\![c]\!]$$

Adequacy theorem

Theorem: *For any closed term t of base type ι and any value k_n we have $[\![t]\!] = n$ iff $t \Downarrow k_n$*

For instance $[\![t]\!] = 0$ iff $t \Downarrow k_0$

Application: transformation of programs

Assume we have a program $t = C[u]$ having u as a subprogram

If $[\![u]\!] = [\![u']\!]$ then $[\![t]\!] = [\![C[u]]\!] = [\![C[u']]\!]$

This follows from the *compositionality* property of the denotational semantics

If $t \Downarrow k_0$ then $[\![t]\!] = 0$ hence $[\![C[u']]\!] = 0$

Hence *by the adequacy theorem* $C[u'] \Downarrow k_0$

Elegant way of proving the equivalence of programs (for instance for justification of compiler optimisations)

Avoids messy syntactical details

Adequacy theorem

Plotkin's result is for a *simply typed* language

Proof by induction on the types, reminiscent of reducibility, by introduction of a *computability* predicate

The adequacy result holds for *untyped* languages!

In some sense, untyped λ-calculus has a type structure

Finite elements

$d \in D$ is finite iff $d \leq \bigvee_{i \in I} \alpha_i$ implies $d \leq \bigvee_{i \in K} \alpha_i$ for some finite $K \subseteq I$

The finite elements represent *observable pieces of information* about a program

0: the program t reduces to 0

$0 \to 0$: if we apply t to 0 the result $t\ 0$ reduces to 0

$\bot \to 0$: if we apply t to a looping program l the result $t\ l$ reduces to 0

For the last example this means intuitively that the program t does not even look at its argument during the computation

Finite elements

If d_1, d_2 are finite so is $d_1 \vee d_2$

Algebraic domains: any element is the sup of the set of finite elements below it

If D, E are algebraic then $[D \to E]$ is algebraic: the finite elements are exactly finite sups of step functions $d \to e$

$(d \to e)\ d' = e$ if $d \leq d'$

$(d \to e)\ d' = \bot$ otherwise

Finite elements

In set theory $\iota, \iota \to \iota, \ldots$ have greater and greater cardinality

For each type σ the finite elements of D_σ form a countable set

Adequacy theorems

S. Abramsky "Domain theory in logical form."
Annals of Pure and Applied Logic, 51:1-77, (199)1.

R. Amadio and P.L. Curien *Domains and Lambda-Calculi.*
Cambridge tracts in theoretical computer science, 46, (1997).

H. Barendregt, M. Coppo and M. Dezani-Ciancaglini
"A filter lambda model and the completeness of type assignment."
J. Symbolic Logic 48 (1983), no. 4, 931–940 (1984).

P. Martin-Löf "Lecture note on the domain interpretation of type theory."
Workshop on Semantics of Programming Languages, Chalmers, (1983).

An untyped programming language

$$t \ ::= \ n \mid t \ t \mid \lambda x.t \qquad\qquad n \ ::= \ x \mid c \mid f$$

Two kind of constants: *defined* $f, g, \ldots$ and *primitive* $c, c', \ldots$

f is defined by equations (computation rules) of the form

$$f \ x_1 \ \ldots \ x_n \ (c \ y_1 \ \ldots \ y_k) \rightarrow u$$

Each constant has an arity $ar(f) = n + 1, ar(c) = k$

We write $h, h', \ldots$ for a constant f or c

Operational semantics

$$\frac{}{\lambda x.t \Downarrow \lambda x.t} \qquad \frac{}{c\ \vec{t} \Downarrow c\ \vec{t}} \qquad \frac{|\vec{t}| < ar(h)}{h\ \vec{t} \Downarrow h\ \vec{t}}$$

$$\frac{t \Downarrow \lambda x.t' \qquad t'(x = u) \Downarrow v}{t\ u \Downarrow v}$$

$$\frac{t \Downarrow c\ \vec{t} \qquad u(\vec{x} = \vec{u}, \vec{y} = \vec{t}) \Downarrow v}{f\ \vec{u}\ t \Downarrow v}$$

We suppose $f\ \vec{x}\ (c\ \vec{y}) = u$

Finite elements

Given a set of constants c with arity $ar(c) \in \mathbb{N}$

$$U, V \ ::= \ \Delta \mid U \to V \mid U \cap V \mid c\,\vec{U} \mid \nabla$$

If $\vec{U}$ is a vector $U_1, \ldots, U_m$ we write $\vec{U} \to U$ for

$$U_1 \to (\cdots \to (U_m \to U) \ldots)$$

and $c\,\vec{U}$ for

$$c\,U_1 \ \ldots \ U_m$$

Finite elements as set of closed programs

Let Λ be the set of all programs

Δ is Λ, ∇ is $\emptyset$

$c\ U_1\ \ldots\ U_k = \{t \mid t \Downarrow c\ u_1\ \ldots\ u_k,\ u_i \in U_i\}$

$U \to V$ is the set of programs t such that t computes to $\lambda x.t'$ or to $h\ \vec{t}$, $|\vec{t}| < ar(h)$ and $\forall u \in U.\ t\ u \in V$

$U \cap V = \{t \mid t \in U\ \wedge\ t \in V\}$

Meet-semi lattice

$$\nabla \subseteq U \subseteq \Delta$$

$$c\ U_1\ \ldots\ U_k \cap c\ U'_1\ \ldots\ U'_k = c\ (U_1 \cap U'_1)\ \ldots\ (U_1 \cap U'_k)$$

$$c\ U_1\ \ldots\ U_k \cap (U \to V) = \nabla \qquad c\ U_1\ \ldots\ U_k \cap c'\ U'_1\ \ldots\ U'_l = \nabla$$

$$(U \to V) \cap (U \to V') = U \to (V \cap V')$$

$$U' \subseteq U,\ V \subseteq V' \Rightarrow (U \to V) \subseteq U' \to V'$$

Key property

Lemma: *We have* $\cap_{i \in I}(U_i \to V_i) \subseteq U \to V$ *iff* $(\cap_{i \in L} V_i) \subseteq V$ *where* $L = \{i \in I \mid U \subseteq U_i\}$

This holds only, a priori, for the *formal* inclusion relation

Decidability

Given U, V we can decide whether $U \subseteq V$ or not

Filters

A *filter* α is a set of types such that

(1) $\Delta \in \alpha$

(2) if $U, V \in \alpha$ then $U \cap V \in \alpha$

(3) if $U \in \alpha$ and $U \subseteq V$ then $V \in \alpha$

These elements are ordered by inclusion

$\uparrow (U \cap V) = \uparrow U \vee \uparrow V$

There is a least element $\bot = \uparrow \Delta$ and a top element $\top = \uparrow \nabla$

We identify U and $\uparrow U$

Filters

The poset of all these filters is a complete lattice D

This poset is *algebraic*: any element is the directed sup of all finite elements below it

Notice that the greatest element $\top$ is finite!

The finite elements of D are *exactly* the types

Filters

This domain D contains 0, s 0, but also s $\perp$, s (s $\perp$), ...

We have a continuous function s : D $\to$ D

D contains the sup of these elements ω such that $\omega =$ s ω

$$\omega = \{\perp, \mathsf{s}\ \perp,\ \mathsf{s}\ (\mathsf{s}\ \perp), \dots\}$$

Filters

We have an application operation on D

$$\alpha\ \beta = \{\Delta\} \cup \{V \mid \exists U.\ [U \to V] \in \alpha\ \wedge\ U \in \beta\}$$

Notice that

$\bot\ \ \beta = \bot$

$\top\ \beta = \top$

Typing rules

$$\frac{(x{:}U) \in \Gamma}{\Gamma \vdash x : U} \qquad \frac{\Gamma, x : U \vdash t : V}{\Gamma \vdash \lambda x.t : U \to V} \qquad \frac{\Gamma \vdash t : U \to V \qquad \Gamma \vdash u : U}{\Gamma \vdash t\ u : V}$$

$$\frac{\Gamma \vdash t : U \qquad \Gamma \vdash t : V}{\Gamma \vdash t : U \cap V} \qquad \frac{\Gamma \vdash t : U \qquad U \subseteq V}{\Gamma \vdash t : V} \qquad \frac{}{\Gamma \vdash t : \Delta}$$

Typing rules for constants

$$\frac{}{\vdash c : \vec{U} \to c\ \vec{U}}$$

$$\frac{\vec{x} : \vec{U}, \vec{y} : \vec{V} \vdash u : U}{\vdash f : \vec{U} \to (c\ \vec{V}) \to U}$$

We suppose $f\ \vec{x}\ (c\ \vec{y}) = u$

$$\frac{}{\vdash f : \vec{U} \to \nabla \to \nabla}$$

Typing rules for constants

If we have $0, \mathsf{s}, \mathsf{add}$ with the equations

$$\mathsf{add}\ x\ 0 = x \qquad \mathsf{add}\ x\ (\mathsf{s}\ y) = \mathsf{s}\ (\mathsf{add}\ x\ y)$$

then we have the typing rules

$$\frac{}{\mathsf{add} : U \to 0 \to U} \qquad \frac{x{:}U, y{:}W \vdash \mathsf{add}\ x\ y : V}{\mathsf{add} : U \to (\mathsf{s}\ W) \to \mathsf{s}\ V}$$

Types and finite elements

Δ corresponds to $\bot$

$U \to V$ corresponds to the step function defined by

$[U \to V]\ U' = V$ if $U \leq U'$

$[U \to V]\ U' = \bot$ otherwise

∇ corresponds to $\top$, the top element of the domain

Denotational semantics

$[\![t]\!]_\rho \in \mathsf{D}$ for $\rho : \mathcal{V} \to \mathsf{D}$

$[\![c]\!]$ (res. $[\![f]\!]$) is the filter of all types U such that $\vdash d : U$ (resp. $\vdash f : U$)

$[\![x]\!]_\rho = \rho(x)$

$[\![t\ u]\!]_\rho = [\![t]\!]_\rho\ [\![u]\!]_\rho$

$[\![\lambda x.t]\!]_\rho = \alpha \longmapsto [\![t]\!]_{(\rho, x=\alpha)}$

Typing rules and denotational semantics

Theorem: We have $\vdash t : U$ iff $U \leq [\![t]\!]$

More generally, we have $x_1{:}U_1, \ldots, x_n{:}U_n \vdash t : U$ iff

$$U \leq [\![t]\!]_{x_1=U_1,\ldots,x_n=U_n}$$

Denotational semantics

An alternative approach is to define directly $[\![t]\!]_\rho \in \mathsf{D}$ by

$$[\![t]\!]_\rho = \{U \mid x_1{:}U_1, \ldots, x_n{:}U_n \vdash t : U,\ U_i \in \rho(x_i)\}$$

Lemma: $\Gamma \vdash \lambda x.t : U \to V$ *iff* $\Gamma, x{:}U \vdash t : V$

Denotational semantics

Theorem: *We have*

$$[\![x]\!]_\rho = \rho(x)$$

$$[\![t\ u]\!]_\rho = [\![t]\!]_\rho\ [\![u]\!]_\rho$$

$$[\![\lambda x.t]\!]_\rho\ \alpha = [\![t]\!]_{(\rho, x=\alpha)}$$

if $[\![t]\!]_{\rho,x=\alpha} = [\![u]\!]_{\nu,y=\alpha}$ *for all* α *then* $[\![\lambda x.t]\!]_\rho = [\![\lambda y.u]\!]_\nu$

Denotational semantics

This alternative characterisation of the semantics of β-conversion is described in

R. Hindley and J. Seldin "*Combinators and λ-calculus*", University Press, 1986

and goes back to G. Berry

Adequacy theorem

Theorem: *If* $\vdash t : U$ *then* $t \in U$

Corollary: *If* $[\![t]\!] = c\ \vec{U}$ *then there exists* $\vec{u}$ *such that* $t \Downarrow c\ \vec{u}$

Application: Gödel system T

Weak version of the normalisation theorem in a semantical way

The constants of Gödel system T are $0, \mathsf{s}, \mathsf{natrec}$

$\mathsf{natrec}\ u\ v\ 0 = u \qquad \mathsf{natrec}\ u\ v\ (\mathsf{s}\ m) = v\ m\ (\mathsf{natrec}\ u\ v\ m)$

The base type is ι and $0 : \iota,\ \mathsf{s} : \iota \to \iota$ and $\mathsf{natrec} : \sigma \to (\iota \to \sigma \to \sigma) \to \iota \to \sigma$

Application: Gödel system T

To each type σ we associate a predicate Tot_σ on D

$a \in$ D is a *total* integer iff $a = \mathsf{s}^k\ 0$ for some $k \in \mathbb{N}$

$Tot_{\sigma\to\tau}(b)$ means that $Tot_\sigma(a)$ implies $Tot_\tau(b\ a)$

If Γ is a context define $Tot_\Gamma(\rho)$ to mean $Tot_\sigma(\rho(x))$ for all $x{:}\sigma$ in Γ

Application: Gödel system T

Lemma 1: *If* $\Gamma \vdash t : \sigma$ *and* $Tot_\Gamma(\rho)$ *then* $Tot_\sigma(\llbracket t \rrbracket_\rho)$. *In particular, if* $\vdash t : \sigma$ *then* $Tot_\sigma(\llbracket t \rrbracket)$.

Lemma 2: *If* $Tot_\sigma(a)$ *then* $a \neq \bot$

Corollary: *If* $\vdash t : \iota$ *then* $t \Downarrow 0$ *or there exists* t' *such that* $t \Downarrow \mathsf{s}\ t'$

Strong Normalisation

As explained in the talk of Benjamin Grégoire for the (total) correctness of the type-checking algorithm we need a (strong) normalisation theorem

B. Grégoire and X. Leroy
A compiled implementation of strong reduction, *ICFP* 2002, 235-246.

Strong Normalisation

$\mathcal{N}$ subset of *strongly normalisable* terms

We write w, w' for strongly normalisable terms

Simple terms

$$s \ ::= \ x \mid s \ w \mid f \ \vec{w} \ s$$

Head-reduction

$$(\lambda x.u)\ v \succ u(x = v)$$

$$f\ \vec{u}\ (c\ \vec{v}) \succ u(\vec{x} = \vec{u}, \vec{y} = \vec{v})$$

$$\frac{u \succ u'}{u\ v \succ u'\ v} \qquad \frac{u \succ u'}{f\ \vec{u}\ u \succ f\ \vec{u}\ u'}$$

We say that u is of *head-redex form* iff there exists u' such that $u \succ u'$

Head-reduction and reduction

We let $\mathcal{S} \subseteq \mathcal{N}$ be the set of strongly normalisable terms that reduce to a simple term

$\mathcal{S} \subseteq \mathcal{N} \subseteq \Lambda$

We write $u \to u'$ ordinary reduction and

$$\to (u) = \{u' \mid u \to u'\}$$

Saturated set

$X \subseteq \Lambda$ is *saturated* iff

(CR1) $\mathcal{S} \subseteq X \subseteq \mathcal{N}$

(CR2) if $t \in X$ then $\rightarrow (t) \subseteq X$

(CR3) if t is of head-redex form and $\rightarrow (t) \subseteq X$ then $t \in X$

Saturated subsets

lemma: *If $I \neq \emptyset$ and X_i saturated then $\cap_{i \in I} X_i$ are saturated*

If $X, Y \subseteq \Lambda$ then we define

$$X \to Y = \{t \in \Lambda \mid \forall u \in X.\ t\ u \in Y\}$$

lemma: *If X and Y are saturated then so is $X \to Y$*

Saturated subsets

If $X_1, \ldots, X_k \subseteq \Lambda$ then $c\ X_1\ \ldots\ X_k$ is the set of terms defined inductively as follows

if $t_1 \in X_1, \ldots, t_k \in X_k$ then $c\ \vec{t} \in c\ \vec{X}$

if $t \in \mathcal{S}$ then $t \in c\ \vec{X}$

if t is of head-redex form and $\rightarrow (t) \subseteq c\ \vec{X}$ then $t \in c\ \vec{X}$

Finite elements as saturated sets

We consider the new set of finite elements (types)

$$U \ ::= \Delta \mid W \qquad W, V \ ::= \ c\,\vec{W} \mid W \cap W \mid W \to W \mid \nabla$$

Each finite element W can be interpreted as a saturated set

Notice that if $c\ \vec{u} \in W$ then $|\vec{u}| = ar(c)$

Meet-semi lattice

$$\nabla \subseteq U \subseteq \Delta$$

$$c\ W_1\ \ldots\ W_k \cap c\ W_1'\ \ldots\ W_k' = c\ (W_1 \cap W_1')\ \ldots\ (W_1 \cap W_k')$$

$$c\ W_1\ \ldots\ W_k \cap (W \to V) = \nabla \qquad c\ W_1\ \ldots\ W_k \cap c'\ W_1'\ \ldots\ W_l' = \nabla$$

$$(W \to V) \cap (W \to V') = W \to (V \cap V')$$

$$W' \subseteq W,\ V \subseteq V' \Rightarrow (W \to V) \subseteq W' \to V'$$

Meet-semi lattice

The filters over this lattice define a new domain E

As before we have an application

$$\alpha\ \beta = \{\Delta\} \cup \{W \mid \exists V.\ V \in \beta \wedge (V \to W) \in \alpha\}$$

Notice that $\alpha\ \ \bot = \bot$ for all α

Strict semantics

We consider the new typing system with only judgements of the form $\Gamma \vdash t : W$

Lemma: If $\vdash t : W$ then t belongs to the saturated set W

Typing rules

$$\frac{(x{:}W) \in \Gamma}{\Gamma \vdash x : W} \qquad \frac{\Gamma, x : W \vdash t : V}{\Gamma \vdash \lambda x.t : W \to V} \qquad \frac{\Gamma \vdash t : W \to V \qquad \Gamma \vdash u : W}{\Gamma \vdash t\ u : V}$$

$$\frac{\Gamma \vdash t : W \qquad \Gamma \vdash t : V}{\Gamma \vdash t : W \cap V}$$

$$\frac{\Gamma \vdash t : W \qquad W \subseteq V}{\Gamma \vdash t : V}$$

Typing rules for constants

$$\frac{}{\vdash c : \vec{W} \to c\ \vec{W}}$$

$$\frac{\vec{x} : \vec{W}, \vec{y} : \vec{V} \vdash u : W}{\vdash f : \vec{W} \to (c\ \vec{V}) \to W}$$

We suppose $f\ \vec{x}\ (c\ \vec{y}) = u$

$$\frac{}{\vdash f : \vec{W} \to \nabla \to \nabla}$$

Strict semantics

We define $[t]_\rho \in \mathsf{E}$ to be the following filter: $U \in [t]_\rho$ iff

(1) $U = \Delta$, or

(2) $x_1{:}W_1, \ldots, x_n{:}W_n \vdash t : U$ in the new system, with $W_i \in \rho(x_i)$

Strict semantics

Theorem: *We have*

$[x]_\rho = \rho(x)$

$[t\ u]_\rho = [t]_\rho\ [u]_\rho$

$[\lambda x.t]_\rho\ \alpha = [t]_{(\rho, x=\alpha)}$ if $\alpha \neq \perp$

if $[t]_{\rho, x=\alpha} = [u]_{\nu, y=\alpha}$ *for all* $\alpha \neq \perp$ *then* $[\lambda x.t]_\rho = [\lambda y.u]_\nu$

Strict semantics

Theorem: If $[t] \neq \bot$ then t is strongly normalisable

If $[\![u]\!]_\rho \neq \bot$ then

$$[\![(\lambda x.t)\ u]\!]_\rho = [\![t(x = u)]\!]_\rho$$

Application: Gödel's system T

Theorem: *If* $\Gamma \vdash t : \sigma$ *and* $Tot_\Gamma(\rho)$ *then* $Tot_\sigma([t]_\rho)$

The crucial case is the application: if $\vdash t : \sigma \to \tau$ and $u : \sigma$ then by induction $Tot_{\sigma\to\tau}([t])$ and $Tot_\tau([u])$. Hence $[u] \neq \bot$ and

$$[t\ u] = [t]\ [u]$$

Corollary: *If* $\vdash t : \sigma$ *then* t *is strongly normalisable*

Interpretation of $\top$

The special element $\top \in \mathsf{D}$ satisfies

$$\top\ \beta = \top$$

if $\beta \neq \bot$, but also

$$f\ \alpha_1\ \ldots\ \alpha_n\ \top = \top$$

if $\alpha_1 \neq \bot,\ \ldots,\ \alpha_n \neq \bot$

An idea coming from ...

- higher-order substitution (Russell, Withehead, Church, Curry, Henkin, ...)
- λ-calculus (Church, Curry, ...)
- type theory (de Bruijn, Martin-Löf, Coquand, Huet, ...)
- automated deduction (Plotkin, Peterson, Stickel, ...)
- proof-checking (Boyer, Moore, ...)
- the practice of mathematic (Appel, Haken, Hales, ...)

Proofs are built with

Deduction rules, axioms

Proofs are built with

Deduction rules, axioms and computation rules

A simple expression of this idea

In predicate logic: deduction modulo

I. Deduction modulo

II. A uniform proof language

Assumed

1. Syntax of terms and formulae in predicate logic
2. Natural deduction rules (for constructive logic)
3. Many sorted predicate logic

Deduction modulo

Proof: sequence of deduction steps

Theory: set of axioms

Deduction modulo

Proof: sequence of deduction steps and computation steps

Theory: set of axioms and computation rules

A terminating and confluent system of computation rules

Computation rules apply to terms, *e.g.*

$$0 + y \longrightarrow y$$

and to atomic formulae, *e.g.*

$$x \times y = 0 \longrightarrow x = 0 \vee y = 0$$

An example: Arithmetic

$$\forall x\ (x = x)$$

$$\forall x\ \forall y\ (x = y \Rightarrow (x/z)A \Rightarrow (y/z)A)$$

$$\forall x \forall y\ (S(x) = S(y) \Rightarrow x = y)$$

$$\forall x\ \neg 0 = S(x)$$

$$(0/z)A \Rightarrow \forall x\ ((x/z)A \Rightarrow (S(x)/z)A) \Rightarrow \forall n\ (n/z)A$$

$$\forall y\ 0 + y = y$$

$$\forall x \forall y\ S(x) + y = S(x + y)$$

$$\forall y\ 0 \times y = 0$$

$$\forall x \forall y\ S(x) \times y = x \times y + y$$

An example: Arithmetic

$$\forall x\ (x = x)$$

$$\forall x\ \forall y\ (x = y \Rightarrow (x/z)A \Rightarrow (y/z)A)$$

$$\forall x \forall y\ (S(x) = S(y) \Rightarrow x = y)$$

$$\forall x\ \neg 0 = S(x)$$

$$(0/z)A \Rightarrow \forall x\ ((x/z)A \Rightarrow (S(x)/z)A) \Rightarrow \forall n\ (n/z)A$$

$$0 + y \longrightarrow y$$

$$S(x) + y \longrightarrow S(x + y)$$

$$0 \times y \longrightarrow 0$$

$$S(x) \times y \longrightarrow x \times y + y$$

Congruence

The computation rules define a congruence on formulae *e.g.*

$$(2 \times 2 = 4) \equiv (4 = 4)$$

Smallest relation that

- is an equivalence relation
- is a congruence (compatible with all the symbols)
- contains $l \equiv r$ for each computation rule $l \longrightarrow r$

The congruence $\equiv$ is decidable (thanks to termination and confluence)

Deduction rules

Deduction rules parametrized by the congruence $\equiv$, *e.g.*

$$\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \Rightarrow\text{-elim}$$

$$\frac{\Gamma \vdash C \quad \Gamma \vdash A}{\Gamma \vdash B} \Rightarrow\text{-elim if } C \equiv (A \Rightarrow B)$$

How to squeeze a proof on a single slide ?

$$\dfrac{\dfrac{\dfrac{}{\forall x\ x = x \vdash \forall x\ x = x}\ \text{Axiom}}{\forall x\ x = x \vdash 2 \times 2 = 4}\ \forall\text{-elim}}{\forall x\ x = x \vdash \exists y\ 2 \times y = 4}\ \exists\text{-intro}$$

Business as usual (The equivalence lemma)

For each congruence $\equiv$, there is a theory $\mathcal{T}$ such that

$$\Gamma \vdash_{\equiv} A$$

iff

$$\mathcal{T}, \Gamma \vdash A$$

e.g. $\mathcal{T} = \{\overline{\forall}\ (P \Leftrightarrow Q) \mid P \equiv Q\}$

Nothing new from the provability point of view

Something new from the proof structure point of view

Proofs and certificates

A test: is 221 prime or composite ?

Proofs and certificates

A test: is 221 prime or composite ?

Four answers: 221 composite

- as you can check yourself
- because 13 is a divisor
- because $221 = 13 \times 17$
- because

$$
\begin{array}{r}
\not{2} \\
1\ 7 \\
1\ 3 \\
\hline
5\ 1 \\
{\scriptstyle 1}\ 1\ 7 \\
\hline
2\ 2\ 1
\end{array}
$$

- C defined by computation rules:

$$\frac{}{C(221)}\ \top\text{-intro}$$

(as you can check yourself)

- $\mid$ defined by computation rules $C(x) = \exists y\ y \mid x$

$$\frac{\dfrac{}{13 \mid 221}\ \top\text{-intro}}{C(221)}\ \exists\text{-intro}$$

(because 13 is a divisor)

- $\times$ defined by computation rules $C(x) = \exists y \exists z\ x = y \times z$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x\ (x = x)}\ \text{axiom}}{221 = 13 \times 17}\ \forall\text{-elim}}{\exists z\ (221 = 13 \times z)}\ \exists\text{-intro}}{C(221)}\ \exists\text{-intro}$$

(because $221 = 13 \times 17$)

- only axioms (each step of the computation)

$$\dfrac{\dfrac{\dfrac{\dots}{221 = 13 \times 17}}{\exists z\ (221 = 13 \times z)}\ \exists\text{-intro}}{C(221)}\ \exists\text{-intro}$$

Purely computational theories

No axioms

Only computation rules and deduction rules

Examples: arithmetic, simple type theory, set theory

Peano fourth and fifth axioms

$$\forall x \forall y \ (S(x) = S(y) \Rightarrow x = y)$$

$$\forall x \ \neg 0 = S(x)$$

Peano fourth and fifth axioms

$$\forall x \forall y \ (S(x) = S(y) \Rightarrow x = y)$$

$$\forall x \ \neg 0 = S(x)$$

$Pred(S(x)) \longrightarrow x$

No term rule for the fourth axiom: no one point model

$Null(0) \longrightarrow \top$

$Null(S(x)) \longrightarrow \bot$

Exercise: prove the two axioms

Where are we ?

$$\forall x\ (x = x)$$

$$\forall x\ \forall y\ (x = y \Rightarrow (x/z)A \Rightarrow (y/z)A)$$

$$\forall x \forall y\ (S(x) = S(y) \Rightarrow x = y)$$

$$\forall x\ \neg 0 = S(x)$$

$$(0/z)A \Rightarrow \forall x\ ((x/z)A \Rightarrow (S(x)/z)A) \Rightarrow \forall n\ (n/z)A$$

$$\forall y\ 0 + y = y$$

$$\forall x \forall y\ S(x) + y = S(x + y)$$

$$\forall y\ 0 \times y = 0$$

$$\forall x \forall y\ S(x) \times y = x \times y + y$$

Equality

$$\forall x\ \forall y\ (x = y \Rightarrow (x/z)A \Rightarrow (y/z)A)$$

Example:

$$\forall x\ \forall y\ (x = y \Rightarrow x \leq 4 \Rightarrow y \leq 4)$$

A second sort for sets and the set $\{z \mid z \leq 4\}$: $f_{z,z\leq 4}$

$$z \in \{z \mid z \leq 4\} \Leftrightarrow z \leq 4$$

$$\forall x\ \forall y\ (x = y \Rightarrow \forall E\ (x \in E \Rightarrow y \in E))$$

Equality

$$\forall x\ \forall y\ (x = y \Rightarrow (x/z)A \Rightarrow (y/z)A)$$

Example:

$$\forall x\ \forall y\ (x = y \Rightarrow x \leq 4 \Rightarrow y \leq 4)$$

A second sort for sets and the set $\{z \mid z \leq 4\}$: $f_{z,z\leq 4}$

$$z \in \{z \mid z \leq 4\} \Leftrightarrow z \leq 4$$

$$\forall x\ \forall y\ (x = y \Leftrightarrow \forall E\ (x \in E \Rightarrow y \in E))$$

Equality

$$\forall x\ \forall y\ (x = y \Rightarrow (x/z)A \Rightarrow (y/z)A)$$

Example:

$$\forall x\ \forall y\ (x = y \Rightarrow x \leq 4 \Rightarrow y \leq 4)$$

A second sort for sets and the set $\{z \mid z \leq 4\}$: $f_{z,z\leq 4}$

$$z \in \{z \mid z \leq 4\} \longrightarrow z \leq 4$$

$$x = y \longrightarrow \forall E\ (x \in E \Rightarrow y \in E)$$

Induction

$$(0/z)A \Rightarrow \forall x\ ((x/z)A \Rightarrow (S(x)/z)A) \Rightarrow \forall n\ (n/z)A$$

Induction

$$\forall E\ (0 \in E \Rightarrow \forall x\ (x \in E \Rightarrow S(x) \in E) \Rightarrow \forall n\ n \in E)$$

Induction

$$\forall E\ (0 \in E \Rightarrow \forall x\ (x \in E \Rightarrow S(x) \in E) \Rightarrow \forall n\ (N(n) \Rightarrow n \in E))$$

Induction

$$\forall n\ (N(n) \Rightarrow \forall E\ (0 \in E \Rightarrow \forall x\ (x \in E \Rightarrow S(x) \in E) \Rightarrow n \in E))$$

Induction

$$\forall n\ (N(n) \Leftrightarrow \forall E\ (0 \in E \Rightarrow \forall x\ (x \in E \Rightarrow S(x) \in E) \Rightarrow n \in E))$$

Induction

$$N(n) \longrightarrow \forall E\ (0 \in E \Rightarrow \forall x\ (x \in E \Rightarrow S(x) \in E) \Rightarrow n \in E)$$

$$x \in f_{x,y_1,\ldots,y_n,P}(y_1,\ldots,y_n) \longrightarrow P$$

$$y = z \longrightarrow \forall E\ (y \in E \Rightarrow z \in E)$$

$$Pred(0) \longrightarrow 0 \qquad Pred(S(x)) \longrightarrow x$$

$$Null(0) \longrightarrow \top \qquad Null(S(x)) \longrightarrow \bot$$

$$N(n) \longrightarrow \forall E\ (0 \in E \Rightarrow \forall y\ (y \in E \Rightarrow S(y) \in E) \Rightarrow n \in E)$$

$$0 + y \longrightarrow y$$

$$S(x) + y \longrightarrow S(x + y)$$

$$0 \times y \longrightarrow 0$$

$$S(x) \times y \longrightarrow x \times y + y$$

I. Deduction modulo

II. A uniform proof language

a. $\lambda\Pi$

b. Axioms, non logical deduction rules, computation rules

c. An example: polymorphism

Type theories

A language to express (among other things) proofs

Proofs in which theory ?

It depends: propositional logic (simply typed λ-calculus), predicate logic ($\lambda\Pi$), arithmetic (T), second-order propositional logic (F), predicative higher-order arithmetic (ITT), second-order arithmetic (AF2), higher-order logic (CoC, Fω), full higher-order arithmetic (CIC), ...

A uniform approach ?

Representing proofs

Proof trees (2-dimensional) are tedious to draw

Data bases, communication

A useful operation on proofs: from a proof of $\Gamma, A \vdash B$ and a proof of $\Gamma \vdash A$ build a proof of $\Gamma \vdash B$

- suppress hypothesis A in all sequents
- replace axiom rules using A by the proof of $\Gamma \vdash A$

A better notation for proofs

In $A_1, ..., A_n \vdash B$, associate a variable ξ_i to each hypothesis A_i

A proof of $A_1, ..., A_n \vdash B$ = a term containing the variables $\xi_1, ..., \xi_n$

$$\frac{}{A_1, ..., A_n \vdash A_i}\ \text{Axiom}$$

ξ_i

To each rule: a function symbol (some are binders)

$$\frac{\dfrac{\pi_1}{\Gamma \vdash A} \qquad \dfrac{\pi_2}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\ \wedge\text{-intro}$$

$f(\pi_1, \pi_2)$

$$\frac{\dfrac{\pi_1}{\Gamma, A \vdash B}}{\Gamma \vdash A \Rightarrow B}\ \Rightarrow\text{-intro}$$

$g(\xi\pi_1)$

The operation

$$\begin{array}{ccc} \pi_2 & \pi_2 & \pi_2 \\ & \pi_1 & \end{array}$$

is substitution

$$(\pi_2/\xi)\pi_1$$

A notation for proofs

$$
\begin{array}{lll}
\pi ::= & \xi & \\
| & \xi \mapsto \pi \mid (\pi_1\ \pi_2) & app(\pi_1, \pi_2) \\
| & \langle \pi_1, \pi_2 \rangle \mid fst(\pi) \mid snd(\pi) & \\
| & i(\pi) \mid j(\pi) \mid (\delta\ \pi_1\ \xi_1\pi_2\ \xi_2\pi_3) & \\
| & I & \\
| & (\delta_\perp\ \pi) & \\
| & x \mapsto \pi \mid (\pi\ t) & \\
| & \langle t, \pi \rangle \mid (\delta_\exists\ \pi_1\ x\xi\pi_2) &
\end{array}
$$

Brouwer-Heyting-Kolmogorov interpretation of proofs

A proof of $A \wedge B$ is an ordered pair formed with a proof of A and a proof of B

A proof of $A \Rightarrow B$ is an algorithmic function mapping a proof of A to a proof of B

A proof of $\forall x\ A(x)$ is an algorithmic function mapping n to a proof of $A(n)$

Explains the notation $\xi \mapsto \pi$ and $(\pi_1\ \pi_2)$

Curry-de Bruijn-Howard isomorphism

A proof of $A \Rightarrow B$ is an algorithmic function mapping a proof of A to a proof of B

If ΦA is the type of the proofs of A then

$$\Phi(A \Rightarrow B) = \Phi A \rightarrow \Phi B$$

Φ isomorphism between formulae and types

Propositions play the role of types of their proofs

Dependent types

A proof of $\forall x\ (even(x) \vee odd(x))$ is an algorithmic function mapping n to a proof of $even(n) \vee odd(n)$

$$f(n) : even(n) \vee odd(n)$$

$$f : (x : nat) \rightarrow (even(x) \vee odd(x))$$

$$f : \Pi x : nat\ (even(x) \vee odd(x))$$

A choice

Three languages:

$S(S(0))$ terms

$S(S(0)) = 0$ formulae

$x \mapsto x$ proofs

One language:

$S(S(0))$, $S(S(0)) = 0$ and $x \mapsto x$ are all terms of the language

One language: $\lambda\Pi$-calculus

A type T (*e.g.* nat) for the objects of the theory

$T : Type$

One language: $\lambda\Pi$-calculus

A type T (*e.g.* nat) for the objects of the theory

$T : Type$

Translate each term as a term of type T

To each function symbol f $f : T \to ... \to T \to T$

Translate each atomic formula $P(t, u)$ to a term of type $Type$

To each predicate symbol P $P : T \to ... \to T \to Type$

One language: $\lambda\Pi$-calculus

A type T (*e.g.* nat) for the objects of the theory

$T : Type$

Translate each term as a term of type T

To each function symbol f $f : T \to ... \to T \to T$

Translate each atomic formula $P(t, u)$ to a term of type $Type$

To each predicate symbol P $P : T \to ... \to T \to Type$

Translate $A \Rightarrow B$ to $A \to B$

Translate $\forall x\ A(x)$ to $\Pi x : T\ A(x)$

...

An example

Assume π is a proof of $\forall x\ \forall y\ (S(x) = S(y) \Rightarrow x = y)$

And π' of $1 = 2$

Find a proof of $0 = 1$

Theories

So far: predicate logic

Need to be extended to theories (*e.g.* arithmetic, simple type theory, set theory, ...) ?

A first way to arithmetic

For each axiom: a constant

$p_3 : \forall x\ \forall y\ (S(x) = S(y) \Rightarrow x = y)$

$p_4 : \forall x\ (0 = S(x) \Rightarrow \bot)$

$Rec_A : (0/z)A \Rightarrow \forall x\ ((x/z)A \Rightarrow (S(x)/z)A) \Rightarrow \forall n\ (n/z)A$

...

$\xi : 1 = 2 \vdash (p_3\ 0\ 1\ \xi) : 0 = 1$

$\xi : 1 = 2 \vdash (p_4\ 0\ (p_3\ 0\ 1\ \xi)) : \bot$

$\vdash \xi : 1 = 2 \mapsto (p_4\ 0\ (p_3\ 0\ 1\ \xi)) : (1 = 2) \Rightarrow \bot$

A second way

Replace axioms by non logical deduction rules

$$\frac{\Gamma \vdash (0/z)A \quad \Gamma \vdash \forall x\ ((x/z)A \Rightarrow (S(x)/z)A)}{\Gamma \vdash \forall n\ (n/z)A}$$

Introduce a new construction in the proof language: $Rec(\pi_1, \pi_2)$

Almost the same but Rec a construction (like app), not a constant

A particular case, the folding and unfolding rules

$$\frac{x \in (A \cap B)}{x \in A \vee x \in B}$$

Axioms poison proof reduction

In predicate logic: a normal closed terms is an introduction

Consistency, disjunction, witness, finite failure of search of $\bot$, ...

With constants normal closed terms need not be introductions

e.g. with an axiom $\exists x\ P(x)$

No witness property, no finite failure of search of $\bot$, ...

Extra reduction rules

e.g.

$$Rec\ a\ b\ 0 \longrightarrow 0$$

$$Rec\ a\ b\ S(x) \longrightarrow (b\ x\ (Rec\ a\ b\ x))$$

ι-reduction (inductive types)

Each new axiom: a new constant and a new proof-reduction rule

Replacing axioms by computation rules: $\lambda\Pi$ modulo

$Null(0) \longrightarrow \top$

$Null(S(x)) \longrightarrow \bot$

$x \in f_{Null} \longrightarrow Null(x)$

$\forall x\ \neg(0 = S(x))$ now has the proof

$$x : nat \mapsto \alpha : (0 = S(x)) \mapsto (\alpha\ f_{Null}\ I)$$

Not a constant

No need for extra reduction rules, the terms reduces for itself

Polymorphism

Simple type theory (higher-order logic)

$$\forall P\ (P \Rightarrow P)$$

$$(Q \Rightarrow Q) \Rightarrow (Q \Rightarrow Q)$$

Polymorphism

$$\Pi P : \mathit{Type}\ (P \Rightarrow P)$$

built on the same pattens as

$$\Pi x : \mathit{nat}\ (x = x)$$

But the red part nat must be of type Type

$\mathit{Type} : \mathit{Type}$? No way

Extra typing rules to form more products (polymorphism)

Polymorphism through rewriting

Express simple type theory as a first-order theory

$$\forall P\ (P \Rightarrow P)$$

$$\forall p\ (\varepsilon(p) \Rightarrow \varepsilon(p))$$

substitute by $ar(q, q)$

$$\varepsilon(ar(q, q)) \Rightarrow \varepsilon(ar(q, q))$$

Polymorphism through rewriting

Express simple type theory as a first-order theory

$$\forall P\ (P \Rightarrow P)$$

$$\forall p\ (\varepsilon(p) \Rightarrow \varepsilon(p))$$

substitute by $ar(q, q)$

$$\varepsilon(ar(q, q)) \Rightarrow \varepsilon(ar(q, q))$$

Need a rule

$$\varepsilon(ar(x, y)) \longrightarrow \varepsilon(x) \Rightarrow \varepsilon(y)$$

Proofs of simple type theory in $\lambda\Pi$ modulo

A uniform proof language

$\lambda\Pi$ modulo

A simple extension of $\lambda\Pi$ with a (parametric) congruence on types

Proofs of all axiom free theories in deduction modulo

Unifies many (more or less esoteric) type theories

Allows to design new type theories (for set theory, ...)

Uniformity allows uniform meta-theory (*e.g.* termination criteria)

Introduction to Coq

Yves Bertot

August 2005

Running Coq

- the plain command : `coqtop`
  - use your favorite line-editor,
- the compilation command : `coqc`
- the interactive environment : `coqide`
- with the Emacs environment : open a file with suffix ".v"
- Also Pcoq developed at Sophia
- All commands terminate with a period at the end of a line.

The Check command

- Useful first step: load collections of known facts and functions.

```
Require Import Arith.  Require Import ArithRing.
Require Import Omega.
```

- First know how to construct well-formed terms.

```
Check 3.
```

3 : nat

```
Check plus.
```

plus : nat->nat->nat

```
Check (nat->(nat->nat)).
```

nat->nat->nat : Set

```
Check (plus 3).
```

plus 3 : nat->nat

Basic constructs

- abstractions, applications.
  `Check (fun x => plus x x).`
  fun x:nat ⇒ x + x : nat->nat
- product types.
  `Check (fun (A:Set)(x:A)=>x).`
  fun (A:Set)(x:A) ⇒ x : forall A:Set, A->A
- Common notations.
  `Check (3=4).`
  3=4 : Prop
  `Check (fun (A:Set)(x:A)=>(3,x)).`
  *fun (A:Set)(x:A)⇒(3,x) : forall A:Set, A ->nat*A*

Basic constructs (continued)

- Logical statements.

```
Check (forall x y, x <= y -> y <= x -> x = y).
```

forall x y:nat, x <= y -> y <= x -> x = y : Prop

- proofs.

```
Check le_S.
```

le_S : forall n m:nat, n <= m -> n <= S m

```
Check (le_S 3 3).
```

le_S 3 3 : 3 <= 3 -> 3 <= 4

```
Check le_n.
```

le_n : forall n:nat, n <= n

```
Check (le_S 3 3 (le_n 3))
```

le_S 3 3 (le_n 3) : 3 <= 4

Logical notations

- conjunction, disjunction, negation.
  `Check (forall A B, A /\ (B \/ ~ A)).`
  forall A B:Prop, A /\ (B \/ ~ A) : Prop
- Well-formed statements are not always true or provable.
- Existential quantification.
  `Check (exists x:nat, x = 3).`
  exists x:nat, x = 3 : Prop

Notations

- Know what function is hidden behind a notation:
  `Locate "_ + _".`
  Notation Scope
  "x + y" := sum x y : type_scope
  "x + y" := plus x y : nat_scope
  (default interpretation)

Computing

- Unlike Haskell, ML, or OCaml, values are not computed by default

  `Check (plus 3 4).`

  3+4:nat
- A command to require computation.

  `Eval compute in ((3+4)*5).`

  $= 35 : nat$
- A proposition is not a boolean value.

  `Eval compute in ((3+4)*5=61).`

  $= 35=61:Prop$
- Fast computation is not the main concern.

Definitions

- Define an object by providing a name and a value.
  ```
  Definition ex1 := fun x => x + 3.
  ```
  ex1 is defined
- Special notation for functions.
  ```
  Definition ex2 (x:nat) := x + 3.
  ```
  ex2 is defined
- See the value associated to definitions.
  ```
  Print ex1.
  ```
  ex1 = fun x : nat ⇒ x + 3 : nat −> nat
  Argument scope is [nat_scope]
  ```
  Print ex2.
  ```
  ex2 = fun x : nat ⇒ x + 3 : nat −> nat
  Argument scope is [nat_scope]

Sections

- Sections make it possible to have a local context.

```
Section sectA.
  Variable A:Set.
```

A is assumed

```
  Variables (x:A)(P:A->Prop)(R:A->A->Prop).
```

x is assumed

```
...
  Hypothesis Hyp1 :  forall x y, R x y -> P y.
...
  Check (Hyp1 x x).
```

Hyp x x : R x x −> P x

Sections (continued)

- Definitions can use local variables.

```
Definition ex3 (z:A) := Hyp1 z z.
Print ex3.
```

ex3 = fun z:A ⇒ Hyp1 z z : forall z:A, R z z -> P z

- Defined values change at closing time.

```
End sectA.
```

ex3 is discharged.

```
Print ex3.
```

ex3 =

fun (A:Set)(P:A->Prop)(R:A->A->Prop)
(Hyp1:forall x y:A,Rxy->P y)(z :A)⇒Hyp1 z z
: forall(A:Set)(P:A->Prop)(R:A->A->Prop),
(forall x y:A, R x y->P y)->forall z:A, R z z->P z

Parameters and Axioms

- Declaring variables and Hypotheses outside sections.
- Proofs will never be required for axioms.
- Make it possible to extend the logic.
- Make partial experiments easier.
- Beware of inconsistency!

Goal directed proof

- Finding inhabitants in types.
- Recursive technique:
  - observe a type in a given context.
  - find the shape of a term with holes with this type.
  - restart recursively with the new holes in new contexts.
- The commands to fill holes are called *tactics*.
  - arrow or forall types are function types and can be filled by an abstraction: the context increases (tactic `intro`).
  - For other types one may use existing functions or theorems (tactics `exact`, `apply`).
  - special tactics take care of classes of constructs (tactics `elim`, `split`, `exist`, `rewrite`, `omega`, `ring`).
- When no hole remains, the proof needs to be saved.

Example proof

```
Theorem example2 :  forall a b:Prop, a/\ b -> b /\ a.
```

1 subgoal

```
  ============================
   forall a b : Prop, a /\ b -> b /\ a
```

```
Proof.
intros a b H.
```

1 subgoal

```
  a : Prop
  b : Prop
  H : a /\ b
  ============================
   b /\ a
```

Example proof (continued)

```
split.
```

2 subgoals

...

H : a /\ b

==========

b

subgoal 2 is:

a

Example proof (continued)

```
elim H.
...
  H : a /\ b
  ============================
    a -> b -> b
...
intros H1 H2.
...
  H1 : a
  H2 : b
  ============================
   b
```

Example proof (continued)

```
 exact b
1 subgoal ...
  ============================
   a
intuition.
Proof completed.
Qed.
intros a b H.
...
 intuition.
example2 is defined
```

Second example

```
Theorem square_lt :  forall n m, n < m -> n*n < m*m.
Proof.
intros n m H.
SearchPattern (_*_ < _*_).
```

mult_S_lt_compat_l:
    *forall n m p : nat, m < p -> S n * m < S n * p*

mult_lt_compat_r:
    *forall n m p : nat, n < m -> 0 < p -> n * p < m * p*

```
Check le_lt_trans.
```

le_lt_trans : forall n m p : nat, n <= m -> m < p -> n < p

Second example (continued)

```
apply le_lt_trans with (n * m).
...
  H : n < m
  ============================
   n * n <= n * m
...
SearchPattern (_ * _ <= _ * _).
mult_le_compat_l: forall n m p : nat, n <= m -> p * n <= p * m
mult_le_compat_r: forall n m p : nat, n <= m -> n * p <= m * p
...
Check lt_le_weak.
lt_le_weak : forall n m : nat, n < m -> n <= m
```

Second example (continued)

```
apply mult_le_compat_l; apply lt_le_weak; exact H.
```

```
...
  H : n < m
  ============================
   n * m < m * m
```

```
apply mult_lt_compat_r.
```

```
2 subgoals
...
  H : n < m
  ============================
   n < m
subgoal 2 is:
 0 < m
```

```
assumption.
```

Second example (continued)

```
Show.
```

$H : n < m$

==========

$0 < m$

```
omega.
```

Proof completed.

```
Qed.
```

Proofs : a synopsis

	$\Rightarrow$	$\forall$	$\wedge$	$\vee$	$\exists$
Hypothesis	`apply`	`apply`	`elim`	`elim`	`elim`
goal	`intros`	`intros`	`split`	`left` or `right`	`exists` *v*
	$\neg$	$=$			
Hypothesis	`elim`	`rewrite`			
goal	`intro`	`reflexivity`			

- Automatic tactics: `auto`, `auto with` *database*, `intuition`, `omega`, `ring`, `fourier`, `field`.
- Possibility to define your own tactics: `Ltac`.

Automatic tactics

- `intuition` Automatic proofs for 1st order intuitionnistic logic,
- `omega` Presburger arithmetic on types `nat` and `Z`,
- `ring` Polynomial equalities on types `Z` and `nat` (no subtraction for the latter)
- `fourier` Inequations between linear formulas in `R`,
- `field` Equations between fractional expressions in `R`.

Forward reasoning

- apply only supports backward reasoning (it does not implement ∀-elimination or -¿-elimination),
- Problem "I have `H: forall x, P x`" how can I add `P a` to the context"
  - `assert (H2 : P a)`, prove this by `apply H` and proceed,
  - alternatively `generalize (H a); intros H2`.
- Problem "I have `H: A -> B`" how can add `B` to the context and have an extra goal to prove `A`.
  - Use assert again,
  - alternatively use "lapply H;[intros H2 — idtac]".

Inductive types

- Inductive types extend the recursive (algebraic) data-types of Haskell, ML, ...
- An inductive type definition provides three kinds of elements:
  - A type (or a family of types),
  - Constructors,
  - A computation process (case-analysis and recursion),
  - A proof by induction principle.

```
Inductive bin :  Set :=
  L : bin
| N : bin -> bin -> bin.
```

Computation process

- Pattern-matching and structural recursion.

```
Fixpoint size (t1:bin):  nat :=
  match t1 with
    L => 1
  | N t1 t2 => 1 + size t1 + size t2
  end.
Fixpoint flatten_aux (t1 t2:bin) {struct t1} :  bin
:=
  match t1 with
    L => N L t2
  | N t'1 t'2 =>
    flatten_aux t'1 (flatten_aux t'2 t2)
  end.
```

Recursive definition (continued)

```
Fixpoint flatten (t:bin) :  bin :=
  match t with
    L => L
  | N t1 t2 => flatten_aux t1 (flatten t2)
  end.
```

Proof by induction principle

- Quantification over a predicate on the inductive type,
- Premises for all the cases represented by the constructors,
- Induction hypotheses for the subterms in the type.

Proof by induction principle

- Quantification over a predicate on the inductive type,
- Premises for all the cases represented by the constructors,
- Induction hypotheses for the subterms in the type.

```
Check bin_ind.
bin_ind : forall P:bin->Prop,
 P L ->
 (forall b:bin, P b -> forall b0:bin, P b0 -> P (N b b0)) ->
 forall b : bin, P b
```

Proof by induction principle

- Quantification over a predicate on the inductive type,
- Premises for all the cases represented by the constructors,
- Induction hypotheses for the subterms in the type.

```
Check bin_ind.
bin_ind : forall P:bin->Prop,
 P L ->
 (forall b:bin, P b -> forall b0:bin, P b0 -> P (N b b0)) ->
 forall b : bin, P b
```

- The tactic `elim` uses this theorem automatically.

Example proof by induction

```
Theorem forall_aux_size :
 forall t1 t,
    size(flatten_aux t1 t) = size t1+size t+1.
Proof.
 intros t1; elim t1.
...
```

==========

forall t : bin, size (flatten_aux L t) = size L + size t + 1

subgoal 2 is:

...

size (flatten_aux (N b b0) t) = size (N b b0)+size t+1

Proof by induction (continued)

```
simpl.
...
============================
forall t : bin, S (S (size t)) = S (size t + 1)
...
intros; ring_nat.
```

Proof by induction (continued)

```
...
  ============================
   forall t : bin, size (flatten_aux L t) = size L + size t + 1
```

```
simpl.
...
============================
forall t : bin, S (S (size t)) = S (size t + 1)
...
intros; ring_nat.
```

- This goal is solved.

Proof by induction (continued)

```
  ============================
   forall b : bin,
   (forall t : bin, size (flatten_aux b t) = size b+size t+1) ->
   forall b0 : bin,
   (forall t : bin, size (flatten_aux b0 t) = size b0+size t+1) ->
   forall t : bin, size (flatten_aux (N b b0) t) =
                   size (N b b0)+size t+1
intros b Hrecb c Hrec t; simpl.
...
  ============================
   size(flatten_aux b (flatten_aux c t))=S(size b+size c+size t+1)
```

Proof by induction (continued)

```
...
  Hrec : forall t : bin, size(flatten_aux c t) = size c+size t+1
  t : bin
  ============================
   size(flatten_aux b (flatten_aux c t))=S(size b+size c+size t+1)
rewrite Hrecb.
...
  ============================
   size b+size(flatten_aux c t)+1 = S(size b+size c+size t+1)
rewrite Hrec; ring_nat.
Qed.
```

Inductive type and equality

- For inductive types of type `Set`, `Type`,
  - Constructors are distinguishable (strong elimination),
  - Constructors are injective.
  - Tactics: `discriminate` and `injection`.
- Not for inductive type of type `Prop`, bad interaction with impredicativity.

Discriminate example

```
Theorem discriminate_example :  forall t1 t2, L = N
t1 t2 -> 2 = 3.
...
intros t1 t2 H.
...
```

H : L = N t1 t2

==========

2 = 3

```
discriminate H.
```

Proof completed.

- With no argument, discriminate finds an hypothesis that fits.

Injection example

```
Theorem injection_example :
     forall t1 t2 t3, N t1 t2 = N t3 t3 -> t1 = t2.
...
intros t1 t2 t3 H.
  H : N t1 t2 = N t3 t3
  ============================
   t1 = t2
...
injection H.
...
  ============================
   t2 = t3 -> t1 = t3 -> t1 = t2
intros H1 H2; rewrite H1; auto.
Proof completed.
```

Usual inductive data-types in Coq

- Most number types are inductive types,
  - Natural numbers *à la Peano*, the induction principle coincides with mathematical induction, `nat`,
  - Strictly positive integers as sequences of bits, `positive`,
  - Integers, as a three-branch disjoint sum, `Z`,
  - Strictly positive rational numbers can also be represented as an inductive type.
- Data structures: lists, binary search trees, finite sets.

Inductive propositions

- Dependent inductive types of sort `Prop`,
- The types of the constructors are logical statements,
- The induction principle is a simplified,
- Easy to understand as a minimal property for which the constructor hold.

Inductive proposition example

```
Inductive even :  nat -> Prop :=
  even0 :  even 0
| evenS : forall x:nat, even x -> even (S (S x)).
```

- `even` is a function that returns a type,
- When `x` varies, `even x` intuitively has one or zero element.

Simplified induction principle

```
Check even_ind.
even_ind : forall P : nat -> Prop,
      P 0 ->
      (forall x : nat, even x -> P x -> P (S (S x))) ->
      forall n : nat, even n -> P n
```

- quantification over a predicate on the potential arguments of the inductive type,
- No universal quantification over elements of the type, only implication (*proof irrelevance*).

Example proof by induction on a proposition

```
Theorem even_mult :  forall x, even x -> exists y, x
= 2*y.
intros x H; elim H.
...
```

*exists y : nat, 0 = 2 * y*

subgoal 2 is:

forall x0 : nat,
*even x0 -> (exists y : nat, x0 = 2 * y) ->*
*exists y : nat, S (S x0) = 2 * y*

Proof by induction on a proposition (continued)

```
exists 0; ring_nat.
intros x0 Hevenx0 IHx.
...
  IHx : exists y : nat, x0 = 2 * y
  ============================
    exists y : nat, S (S x0) = 2 * y
destruct IHx as [y Heq]; rewrite Heq.
(*alternative to elim IHx; intros y Heq; rewrite Heq
*)
exists (S y); ring_nat.
Qed.
```

Inversion

- sometimes assumptions are false because no constructor proves them,
- sometimes the hypothesis of a constructor have to be tree because only this constructor could have been used.

Example inversion

```
 not_even_1 :  ~even 1.
intros even1. ...
even1 : even 1
============================
 False
inversion even1.
Qed.
```

Usual inductive propositions in Coq

- The order <= on natural numbers (type `le`).
- The logical connectives.
- The accessibility predicate with respect to a binary relation,

Logical connectives as inductive propositions

- Parallel with usual present of logic in sequent style,
- Right introduction rules are replaced by constructors,
- Left introduction is automatically given by the induction principle.

Inductive view of `False`

- No right introduction rule: no constructor.

```
Inductive False :  Prop := .
Check False_ind.
False_ind
     : forall P : Prop, False -> P
```

Inductive view of and

- one constructor,
- two left introduction rules, but can be modeled as just one.

```
Print and.
Inductive and (A : Prop) (B : Prop) : Prop :=
    conj : A -> B -> A /\ B
Check and_ind.
and_ind
    : forall A B P : Prop, (A -> B -> P) -> A /\ B -> P
```

Inductive view of or

```
Print or.
```

Inductive or (A : Prop) (B : Prop) : Prop :=
or_introl : A -> A \/ B | or_intror : B -> A \/ B

```
Check or_ind.
```

or_ind : forall A B P : Prop, (A->P)->(B->P)->A \/ B->P

Inductive view of `exists`

```
Print ex.
```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
* ex_intro : forall x : A, P x -> ex P*
```
Check ex_ind.
```
ex_ind : forall (A : Type) (P : A -> Prop) (P0 : Prop),
* (forall x : A, P x -> P0) -> ex P -> P0*

Inductive view of eq

```
Print eq.
```
Inductive eq (A : Type) (x : A) : A -> Prop :=
* refl_equal : x = x*
```
Check eq_ind.
```
eq_ind : forall (A : Type) (x : A) (P : A -> Prop),
P x -> forall y : A, x = y -> P y

Dependently typed pattern-matching

- Well-formed pattern-matching constructs where each branch has a different type,
- Still a constraint of being well-typed,
- Determine the type of the whole expression,
- Verify that each branch is well-typed,
- Dependence on the matched expression.

Syntax of dependently typed pattern-matching

- `match` e `as` x `return` T `with`
  p_1 => v_1
  | p_2 => v_2
  ...
  `end`
- The whole expression has type $T[e/x]$,
- Each value v_1 must have type $T[p_1/x]$.

Example of dependently typed programming

```
Print nat.
Inductive nat : Set := O : nat | S : nat -> nat
Fixpoint nat_ind (P:nat->Prop)(v0:P 0)
      (f:forall n, P n -> P (S n))
      (n:nat) {struct n} :  P n :=
   match n return P n with
     0 => v0
   | S p => f p (nat_ind P v0 f p)
   end.
```

- Dependently-typed programming for logical purposes

Dependent pattern-matching with dependent inductive types

```
Fixpoint even_ind2 (P:nat->Prop)(v0:P 0)
      (f:forall n, P n -> P (S (S n)))
      (n:nat) (h:even n) {struct h} :  P n :=
      match h in even x return P x with
        even0 => v0
      | evenS a h' => f a (even_ind2 P v0 f a h')
      end.
```

Exercices in Coq

Yves Bertot

August 23, 2005

1. Define a function of type `forall x:nat, {y:nat|2*y<=x<2*y+1}`,

2. Define a function `twopower` of type `nat -> nat` that computes 2^n for every natural number n, and then define a function of type

   ```
   forall x:nat,
         {y : nat | twopower y <= x < twopower(y+1)}+{x=0}
   ```

3. Define a sorting function for an abitrary binary relation on an arbitrary type. You should define suitable predicates `sorted` and `permutation`, and then provide a function with the following type:

   ```
   forall A:Set, forall R:Set,
   forall testR : forall x y,{R x y}+{R y x},
   forall l : list A,
     {l' : list A | sorted A R l' /\ permutation A l l'}
   ```

   I suggest using insertion sort, which is reasonably simple. As a first exercise, you may leave aside the notion of permutation and construct a function that only has the following type:

   ```
   forall A:Set, forall R:Set,
   forall testR : forall x y,{R x y}+{R y x},
   forall l : list A, {l' : list A | sorted A R l'}
   ```

   But because this specification is weak, you should refrain from cheating (for instance by providing the constant function that returns the empty list).

Coq tutorial
Program verification using Coq

Jean-Christophe Filliâtre

CNRS – Université Paris Sud

TYPES summer school – August 23th, 2005

Introduction

This lecture: how to use Coq to verify purely functional programs

Thursday's lecture: verification of imperative programs (using Coq and other provers)

Introduction

This lecture: how to use Coq to verify purely functional programs

Thursday's lecture: verification of imperative programs (using Coq and other provers)

The goal

To get a purely functional (ML) program which is proved correct

There are mostly two ways:

1. define your ML function in Coq and prove it correct
2. give the Coq function a richer type (= the specification) and get the ML function via program extraction

The goal

To get a purely functional (ML) program which is proved correct

There are mostly two ways:

1. define your ML function in Coq and prove it correct
2. give the Coq function a richer type (= the specification) and get the ML function via program extraction

The goal

To get a purely functional (ML) program which is proved correct

There are mostly two ways:

1. define your ML function in Coq and prove it correct
2. give the Coq function a richer type (= the specification) and get the ML function via program extraction

The goal

To get a purely functional (ML) program which is proved correct

There are mostly two ways:

1. define your ML function in Coq and prove it correct
2. give the Coq function a richer type (= the specification) and get the ML function via program extraction

Program extraction

Two sorts:

Prop : the sort of logic terms

Set : the sort of informative terms

Program extraction turns the informative contents of a Coq term into an ML program while removing the logical contents

Program extraction

Two sorts:

Prop : the sort of logic terms

Set : the sort of informative terms

Program extraction turns the informative contents of a Coq term into an ML program while removing the logical contents

Outline

1. Direct method (ML function defined in Coq)
2. Use of Coq dependent types
3. Modules and functors

Running example

Finite sets library implemented with balanced binary search trees

1. useful
2. complex
3. purely functional

The Ocaml library Set was verified using Coq
One (balancing) bug was found (fixed in current release)

Running example

Finite sets library implemented with balanced binary search trees

1. useful
2. complex
3. purely functional

The Ocaml library Set was verified using Coq
One (balancing) bug was found (fixed in current release)

Running example

Finite sets library implemented with balanced binary search trees

1. useful
2. complex
3. purely functional

The Ocaml library Set was verified using Coq
One (balancing) bug was found (fixed in current release)

Direct method

Most ML functions can be defined in Coq

$$f \quad : \quad \tau_1 \longrightarrow \tau_2$$

A specification is a relation $S : \tau_1 \longrightarrow \tau_2 \longrightarrow \texttt{Prop}$
f verifies S if

$$\forall x : \tau_1.\ (S\ x\ (f\ x))$$

The proof is conducted following the definition of f

Direct method

Most ML functions can be defined in Coq

$$f \quad : \quad \tau_1 \longrightarrow \tau_2$$

A specification is a relation $S : \tau_1 \longrightarrow \tau_2 \longrightarrow \texttt{Prop}$
f verifies S if

$$\forall x : \tau_1.\ (S\ x\ (f\ x))$$

The proof is conducted following the definition of f

Direct method

Most ML functions can be defined in Coq

$$f \quad : \quad \tau_1 \longrightarrow \tau_2$$

A specification is a relation $S : \tau_1 \longrightarrow \tau_2 \longrightarrow \texttt{Prop}$
f verifies S if

$$\forall x : \tau_1.\ (S\ x\ (f\ x))$$

The proof is conducted following the definition of f

Binary search trees

The type of trees

```
Inductive tree : Set :=
  | Empty
  | Node : tree → Z → tree → tree.
```

The membership relation

```
Inductive In (x:Z) : tree → Prop :=
  | In_left : ∀l r y, In x l → In x (Node l y r)
  | In_right : ∀l r y, In x r → In x (Node l y r)
  | Is_root : ∀l r, In x (Node l x r).
```

Binary search trees

The type of trees

```
Inductive tree : Set :=
  | Empty
  | Node : tree → Z → tree → tree.
```

The membership relation

```
Inductive In (x:Z) : tree → Prop :=
  | In_left : ∀l r y, In x l → In x (Node l y r)
  | In_right : ∀l r y, In x r → In x (Node l y r)
  | Is_root : ∀l r, In x (Node l x r).
```

The function is_empty

ML

```
let is_empty = function Empty → true | _ → false
```

Coq

```
Definition is_empty (s:tree) : bool := match s with
  | Empty ⇒ true
  | _ ⇒ false end.
```

Correctness

```
Theorem is_empty_correct :
  ∀s, (is_empty s)=true ↔ (∀x, ¬(In x s)).
Proof.
  destruct s; simpl; intuition.
  ...
```

The function is_empty

ML

```
let is_empty = function Empty → true | _ → false
```

Coq

```
Definition is_empty (s:tree) : bool := match s with
  | Empty ⇒ true
  | _ ⇒ false end.
```

Correctness

```
Theorem is_empty_correct :
  ∀s, (is_empty s)=true ↔ (∀x, ¬(In x s)).
Proof.
  destruct s; simpl; intuition.
  ...
```

The function is_empty

ML

```
let is_empty = function Empty → true | _ → false
```

Coq

```
Definition is_empty (s:tree) : bool := match s with
  | Empty ⇒ true
  | _ ⇒ false end.
```

Correctness

```
Theorem is_empty_correct :
  ∀s, (is_empty s)=true ↔ (∀x, ¬(In x s)).
Proof.
  destruct s; simpl; intuition.
  ...
```

The function is_empty

ML

```
let is_empty = function Empty → true | _ → false
```

Coq

```
Definition is_empty (s:tree) : bool := match s with
  | Empty ⇒ true
  | _ ⇒ false end.
```

Correctness

```
Theorem is_empty_correct :
  ∀s, (is_empty s)=true ↔ (∀x, ¬(In x s)).
Proof.
  destruct s; simpl; intuition.
  ...
```

The function mem

ML

```
let rec mem x = function
  | Empty →
      false
  | Node (l, y, r) →
      let c = compare x y in
      if c < 0 then mem x l
      else if c = 0 then true
      else mem x r
```

The function mem

Coq

```
Fixpoint mem (x:Z) (s:tree) {struct s} : bool :=
  match s with
  | Empty ⇒
     false
  | Node l y r ⇒ match compare x y with
     | Lt ⇒ mem x l
     | Eq ⇒ true
     | Gt ⇒ mem x r
    end
  end.
```

assuming

```
Inductive order : Set := Lt | Eq | Gt.
Hypothesis compare : Z → Z → order.
```

The function mem

Coq

```
Fixpoint mem (x:Z) (s:tree) {struct s} : bool :=
  match s with
  | Empty ⇒
     false
  | Node l y r ⇒ match compare x y with
     | Lt ⇒ mem x l
     | Eq ⇒ true
     | Gt ⇒ mem x r
    end
  end.
```

assuming

```
Inductive order : Set := Lt | Eq | Gt.
Hypothesis compare : Z → Z → order.
```

Correctness of the function mem

to be a binary search tree

```
Inductive bst : tree → Prop :=
  | bst_empty :
     bst Empty
  | bst_node :
     ∀x l r,
     bst l → bst r →
     (∀y, In y l → y < x) →
     (∀y, In y r → x < y) → bst (Node l x r).

Theorem mem_correct :
  ∀x s, bst s → ((mem x s)=true ↔ In x s).
```

specification S has the shape $P\ x \to Q\ x\ (f\ x)$

Correctness of the function mem

to be a binary search tree

```
Inductive bst : tree → Prop :=
  | bst_empty :
     bst Empty
  | bst_node :
     ∀x l r,
     bst l → bst r →
     (∀y, In y l → y < x) →
     (∀y, In y r → x < y) → bst (Node l x r).

Theorem mem_correct :
  ∀x s, bst s → ((mem x s)=true ↔ In x s).
```

specification S has the shape $P\ x \rightarrow Q\ x\ (f\ x)$

Correctness of the function mem

to be a binary search tree

```
Inductive bst : tree → Prop :=
  | bst_empty :
     bst Empty
  | bst_node :
     ∀x l r,
     bst l → bst r →
     (∀y, In y l → y < x) →
     (∀y, In y r → x < y) → bst (Node l x r).

Theorem mem_correct :
  ∀x s, bst s → ((mem x s)=true ↔ In x s).
```

specification S has the shape $P\ x \to Q\ x\ (f\ x)$

Modularity

To prove mem correct requires a property for compare

```
Hypothesis compare_spec :
  ∀x y, match compare x y with
    | Lt ⇒ x < y
    | Eq ⇒ x = y
    | Gt ⇒ x > y
  end.

Theorem mem_correct :
  ∀x s, bst s → ((mem x s)=true ↔ In x s).
Proof.
  induction s; simpl.
  ...
  generalize (compare_spec x y); destruct (compare x y).
  ...
```

Modularity

To prove mem correct requires a property for compare

```
Hypothesis compare_spec :
  ∀x y, match compare x y with
    | Lt ⇒ x < y
    | Eq ⇒ x = y
    | Gt ⇒ x > y
  end.

Theorem mem_correct :
  ∀x s, bst s → ((mem x s)=true ↔ In x s).
Proof.
  induction s; simpl.
  ...
  generalize (compare_spec x y); destruct (compare x y).
  ...
```

Modularity

To prove mem correct requires a property for compare

```
Hypothesis compare_spec :
  ∀x y, match compare x y with
    | Lt ⇒ x < y
    | Eq ⇒ x = y
    | Gt ⇒ x > y
  end.

Theorem mem_correct :
  ∀x s, bst s → ((mem x s)=true ↔ In x s).
Proof.
  induction s; simpl.
  ...
  generalize (compare_spec x y); destruct (compare x y).
  ...
```

Partial functions

If the function f is partial, it has the Coq type

$$f : \forall x : \tau_1.\ (P\ x) \to \tau_2$$

Example: `min_elt` returning the smallest element of a tree

$$\texttt{min\_elt} : \forall s : \texttt{tree}.\ \neg s = \texttt{Empty} \to \texttt{Z}$$

specification

$$\forall s.\ \forall h : \neg s = \texttt{Empty}.\ \texttt{bst}\ s \to \\ \texttt{In}\ (\texttt{min\_elt}\ s\ h)\ s\ \wedge\ \forall x.\ \texttt{In}\ x\ s \to \texttt{min\_elt}\ s\ h \leq x$$

Partial functions

If the function f is partial, it has the Coq type

$$f : \forall x : \tau_1.\ (P\ x) \to \tau_2$$

Example: `min_elt` returning the smallest element of a tree

$$\texttt{min\_elt} : \forall s : \texttt{tree}.\ \neg s = \texttt{Empty} \to \texttt{Z}$$

specification

$$\forall s.\ \forall h : \neg s = \texttt{Empty}.\ \texttt{bst}\ s \to$$
$$\texttt{In}\ (\texttt{min\_elt}\ s\ h)\ s\ \wedge\ \forall x.\ \texttt{In}\ x\ s \to \texttt{min\_elt}\ s\ h \leq x$$

Partial functions

If the function f is partial, it has the Coq type

$$f : \forall x : \tau_1.\ (P\ x) \to \tau_2$$

Example: `min_elt` returning the smallest element of a tree

$$\texttt{min\_elt} : \forall s : \texttt{tree}.\ \neg s = \texttt{Empty} \to \texttt{Z}$$

specification

$$\forall s.\ \forall h : \neg s = \texttt{Empty}.\ \texttt{bst}\ s \to$$
$$\texttt{In}\ (\texttt{min\_elt}\ s\ h)\ s\ \wedge\ \forall x.\ \texttt{In}\ x\ s \to \texttt{min\_elt}\ s\ h \leq x$$

Partial functions

If the function f is partial, it has the Coq type

$$f : \forall x : \tau_1.\ (P\ x) \to \tau_2$$

Example: `min_elt` returning the smallest element of a tree

$$\texttt{min\_elt} : \forall s : \texttt{tree}.\ \neg s = \texttt{Empty} \to \texttt{Z}$$

specification

$$\forall s.\ \forall h : \neg s = \texttt{Empty}.\ \texttt{bst}\ s \to$$
$$\texttt{In}\ (\texttt{min\_elt}\ s\ h)\ s\ \wedge\ \forall x.\ \texttt{In}\ x\ s \to \texttt{min\_elt}\ s\ h \leq x$$

Even the definition of a partial function is not easy

ML

```
let rec min_elt = function
  | Empty → assert false
  | Node (Empty, x, _) → x
  | Node (l, _, _) → min_elt l
```

Coq

1. `assert false` ⇒ elimination on a proof of False
2. recursive call requires a proof that `l` is not empty

Even the definition of a partial function is not easy

ML

```
let rec min_elt = function
  | Empty → assert false
  | Node (Empty, x, _) → x
  | Node (l, _, _) → min_elt l
```

Coq

1. `assert false` ⇒ elimination on a proof of False
2. recursive call requires a proof that `l` is not empty

Even the definition of a partial function is not easy

ML

```
let rec min_elt = function
  | Empty → assert false
  | Node (Empty, x, _) → x
  | Node (l, _, _) → min_elt l
```

Coq

1. `assert false` ⇒ elimination on a proof of False
2. recursive call requires a proof that `l` is not empty

min_elt: a solution

```
Fixpoint min_elt (s:tree) (h:¬s=Empty) { struct s } : Z :=
  match s                                  with
  | Empty ⇒

  | Node l x _ ⇒
               match l                                 with
               | Empty ⇒                  x
               | _ ⇒                  min_elt l

               end
  end   .
```

min_elt: a solution

```
Fixpoint min_elt (s:tree) (h:¬s=Empty) { struct s } : Z :=
  match s return ¬s=Empty → Z with
  | Empty ⇒
    (fun (h:¬Empty=Empty) ⇒
       False_rec _ (h (refl_equal Empty)))
  | Node l x _ ⇒
    (fun h ⇒ match l as a return a=l → Z with
                | Empty ⇒ (fun _ ⇒ x)
                | _ ⇒ (fun h ⇒ min_elt l
                                 (Node_not_empty _ _ _ _ h))
              end (refl_equal l))
  end h.
```

Definition by proof

Idea: use the proof editor to build the whole definition

```
Definition min_elt : ∀s, ¬s=Empty → Z.
Proof.
  induction s; intro h.
  elim h; auto.
  destruct s1.
  exact z.
  apply IHs1; discriminate.
Defined.
```

But did we define the right function?

Definition by proof

Idea: use the proof editor to build the whole definition

```
Definition min_elt : ∀ s, ¬s=Empty → Z.
Proof.
  induction s; intro h.
  elim h; auto.
  destruct s1.
  exact z.
  apply IHs1; discriminate.
Defined.
```

But did we define the right function?

Definition by proof

Idea: use the proof editor to build the whole definition

```
Definition min_elt : ∀ s, ¬s=Empty → Z.
Proof.
  induction s; intro h.
  elim h; auto.
  destruct s1.
  exact z.
  apply IHs1; discriminate.
Defined.
```

But did we define the right function?

Definition by proof

Idea: use the proof editor to build the whole definition

```
Definition min_elt : ∀ s, ¬s=Empty → Z.
Proof.
  induction s; intro h.
  elim h; auto.
  destruct s1.
  exact z.
  apply IHs1; discriminate.
Defined.
```

But did we define the right function?

Definition by proof (cont'd)

We can check the extracted code:

```
Extraction min_elt.

(** val min_elt : tree → z **)

let rec min_elt = function
  | Empty → assert false (* absurd case *)
  | Node (t0, z0, t1) →
      (match t0 with
         | Empty → z0
         | Node (s1_1, z1, s1_2) → min_elt t0)
```

The refine tactic

```
Definition min_elt : ∀s, ¬s=Empty → Z.
Proof.
  refine
    (fix min (s:tree) (h:¬s=Empty) { struct s } : Z :=
    match s return ¬s=Empty → Z with
    | Empty ⇒
        (fun h ⇒ _)
    | Node l x _ ⇒
        (fun h ⇒ match l as a return a=l → Z with
                  | Empty ⇒ (fun _ ⇒ x)
                  | _ ⇒ (fun h ⇒ min l _)
                  end _)
    end h).
...
```

The refine tactic

```
Definition min_elt : ∀s, ¬s=Empty → Z.
Proof.
  refine
    (fix min (s:tree) (h:¬s=Empty) { struct s } : Z :=
    match s return ¬s=Empty → Z with
    | Empty ⇒
        (fun h ⇒ _)
    | Node l x _ ⇒
        (fun h ⇒ match l as a return a=l → Z with
                  | Empty ⇒ (fun _ ⇒ x)
                  | _ ⇒ (fun h ⇒ min l _)
                  end _)
    end h).
...
```

A last solution

To make the function total

```
Fixpoint min_elt (s:tree) : Z := match s with
  | Empty ⇒ 0
  | Node Empty z _ ⇒ z
  | Node l _ _ ⇒ min_elt l
end.
```

Correctness theorem almost unchanged:

```
Theorem min_elt_correct :
  ∀s, ¬s=Empty → bst s →
    In (min_elt s) s ∧
    ∀x, In x s → min_elt s <= x.
```

A last solution

To make the function total

```
Fixpoint min_elt (s:tree) : Z := match s with
  | Empty ⇒ 0
  | Node Empty z _ ⇒ z
  | Node l _ _ ⇒ min_elt l
end.
```

Correctness theorem almost unchanged:

```
Theorem min_elt_correct :
  ∀s, ¬s=Empty → bst s →
    In (min_elt s) s ∧
    ∀x, In x s → min_elt s <= x.
```

A last solution

To make the function total

```
Fixpoint min_elt (s:tree) : Z := match s with
  | Empty ⇒ 0
  | Node Empty z _ ⇒ z
  | Node l _ _ ⇒ min_elt l
end.
```

Correctness theorem almost unchanged:

```
Theorem min_elt_correct :
  ∀s, ¬s=Empty → bst s →
    In (min_elt s) s ∧
    ∀x, In x s → min_elt s <= x.
```

Functions that are not structurally recursive

One solution is to use a well-founded induction principle such as

```
well_founded_induction
    : ∀(A : Set) (R : A → A → Prop),
      well_founded R →
      ∀P : A → Set,
      (∀x : A, (∀y : A, R y x → P y) → P x) →
      ∀a : A, P a
```

Defining the function requires to build proof terms (of `R y x`)
similar to partial functions ⇒ similar solutions

Functions that are not structurally recursive

One solution is to use a well-founded induction principle such as

```
well_founded_induction
    : ∀(A : Set) (R : A → A → Prop),
      well_founded R →
      ∀P : A → Set,
      (∀x : A, (∀y : A, R y x → P y) → P x) →
      ∀a : A, P a
```

Defining the function requires to build proof terms (of `R y x`) similar to partial functions ⇒ similar solutions

Example: the subset function

```
let rec subset s1 s2 = match (s1, s2) with
  | Empty, _ →
    true
  | _, Empty →
    false
  | Node (l1, v1, r1), Node (l2, v2, r2) →
    let c = compare v1 v2 in
    if c = 0 then
      subset l1 l2 && subset r1 r2
    else if c < 0 then
      subset (Node (l1, v1, Empty)) l2 && subset r1 s2
    else
      subset (Node (Empty, v1, r1)) r2 && subset l1 s2
```

Example: the subset function

```
let rec subset s1 s2 = match (s1, s2) with
  | Empty, _ →
      true
  | _, Empty →
      false
  | Node (l1, v1, r1), Node (l2, v2, r2) →
      let c = compare v1 v2 in
      if c = 0 then
        subset l1 l2 && subset r1 r2
      else if c < 0 then
        subset (Node (l1, v1, Empty)) l2 && subset r1 s2
      else
        subset (Node (Empty, v1, r1)) r2 && subset l1 s2
```

Induction over two trees

```
Fixpoint cardinal_tree (s:tree) : nat := match s with
  | Empty ⇒
      0
  | Node l _ r ⇒
      (S (plus (cardinal_tree l) (cardinal_tree r)))
end.

Lemma cardinal_rec2 :
  ∀(P:tree→tree→Set),
  (∀(x x':tree),
     (∀(y y':tree),
       (lt (plus (cardinal_tree y) (cardinal_tree y'))
           (plus (cardinal_tree x) (cardinal_tree x'))) →
     → (P x x')) →
  ∀(x x':tree), (P x x').
```

Induction over two trees

```
Fixpoint cardinal_tree (s:tree) : nat := match s with
  | Empty ⇒
      0
  | Node l _ r ⇒
      (S (plus (cardinal_tree l) (cardinal_tree r)))
end.

Lemma cardinal_rec2 :
  ∀(P:tree→tree→Set),
  (∀(x x':tree),
     (∀(y y':tree),
       (lt (plus (cardinal_tree y) (cardinal_tree y'))
           (plus (cardinal_tree x) (cardinal_tree x'))) →
     → (P x x')) →
  ∀(x x':tree), (P x x').
```

Defining the subset function

```
Definition subset : tree → tree → bool.
Proof.
  intros s1 s2; pattern s1, s2; apply cardinal_rec2.
  destruct x. ... destruct x'. ...
  intros; case (compare z z0).
  (* z < z0 *)
  refine (andb (H (Node x1 z Empty) x'2 _)
               (H x2 (Node x'1 z0 x'2) _)); simpl; omega.
  (* z = z0 *)
  refine (andb (H x1 x'1 _) (H x2 x'2 _)); simpl ; omega.
  (* z > z0 *)
  refine (andb (H (Node Empty z x2) x'2 _)
               (H x1 (Node x'1 z0 x'2) _)); simpl ; omega.
Defined.
```

Defining the subset function

```
Definition subset : tree → tree → bool.
Proof.
  intros s1 s2; pattern s1, s2; apply cardinal_rec2.
  destruct x. ... destruct x'. ...
  intros; case (compare z z0).
  (* z < z0 *)
  refine (andb (H (Node x1 z Empty) x'2 _)
               (H x2 (Node x'1 z0 x'2) _)); simpl; omega.
  (* z = z0 *)
  refine (andb (H x1 x'1 _) (H x2 x'2 _)); simpl ; omega.
  (* z > z0 *)
  refine (andb (H (Node Empty z x2) x'2 _)
               (H x1 (Node x'1 z0 x'2) _)); simpl ; omega.
Defined.
```

Defining the subset function

```
Definition subset : tree → tree → bool.
Proof.
  intros s1 s2; pattern s1, s2; apply cardinal_rec2.
  destruct x. ... destruct x'. ...
  intros; case (compare z z0).
  (* z < z0 *)
  refine (andb (H (Node x1 z Empty) x'2 _)
               (H x2 (Node x'1 z0 x'2) _)); simpl; omega.
  (* z = z0 *)
  refine (andb (H x1 x'1 _) (H x2 x'2 _)); simpl ; omega.
  (* z > z0 *)
  refine (andb (H (Node Empty z x2) x'2 _)
               (H x1 (Node x'1 z0 x'2) _)); simpl ; omega.
Defined.
```

Defining the subset function

```
Definition subset : tree → tree → bool.
Proof.
  intros s1 s2; pattern s1, s2; apply cardinal_rec2.
  destruct x. ... destruct x'. ...
  intros; case (compare z z0).
  (* z < z0 *)
  refine (andb (H (Node x1 z Empty) x'2 _)
               (H x2 (Node x'1 z0 x'2) _)); simpl; omega.
  (* z = z0 *)
  refine (andb (H x1 x'1 _) (H x2 x'2 _)); simpl ; omega.
  (* z > z0 *)
  refine (andb (H (Node Empty z x2) x'2 _)
               (H x1 (Node x'1 z0 x'2) _)); simpl ; omega.
Defined.
```

Defining the subset function

```
Definition subset : tree → tree → bool.
Proof.
  intros s1 s2; pattern s1, s2; apply cardinal_rec2.
  destruct x. ... destruct x'. ...
  intros; case (compare z z0).
  (* z < z0 *)
  refine (andb (H (Node x1 z Empty) x'2 _)
               (H x2 (Node x'1 z0 x'2) _)); simpl; omega.
  (* z = z0 *)
  refine (andb (H x1 x'1 _) (H x2 x'2 _)); simpl ; omega.
  (* z > z0 *)
  refine (andb (H (Node Empty z x2) x'2 _)
               (H x1 (Node x'1 z0 x'2) _)); simpl ; omega.
Defined.
```

Defining the subset function

```
Definition subset : tree → tree → bool.
Proof.
  intros s1 s2; pattern s1, s2; apply cardinal_rec2.
  destruct x. ... destruct x'. ...
  intros; case (compare z z0).
  (* z < z0 *)
  refine (andb (H (Node x1 z Empty) x'2 _)
               (H x2 (Node x'1 z0 x'2) _)); simpl; omega.
  (* z = z0 *)
  refine (andb (H x1 x'1 _) (H x2 x'2 _)); simpl ; omega.
  (* z > z0 *)
  refine (andb (H (Node Empty z x2) x'2 _)
               (H x1 (Node x'1 z0 x'2) _)); simpl ; omega.
Defined.
```

Extraction

```
Extraction well_founded_induction.
  let rec well_founded_induction x a =
    x a (fun y _ → well_founded_induction x y)
```

```
Extraction Inline cardinal_rec2 ...
Extraction subset.
```

gives the expected ML code

Extraction

```
Extraction well_founded_induction.
  let rec well_founded_induction x a =
    x a (fun y _ → well_founded_induction x y)
```

```
Extraction Inline cardinal_rec2 ...
Extraction subset.
```

gives the expected ML code

To sum up, defining an ML function in Coq and prove it correct seems the obvious way, but it can be rather complex when the function

- is partial, and/or
- is not structurally recursive

Use of dependent types

Instead of

1. defining a pure function, and
2. proving its correctness

let us do both at the same time

We can give Coq functions richer types that are specifications

Example

$$f : \{n : Z \mid n \geq 0\} \rightarrow \{p : Z \mid \texttt{prime}\ p\}$$

Use of dependent types

Instead of

1. defining a pure function, and
2. proving its correctness

let us do both at the same time

We can give Coq functions richer types that are specifications

Example

$$f : \{n : Z \mid n \geq 0\} \rightarrow \{p : Z \mid \texttt{prime}\ p\}$$

Use of dependent types

Instead of

1. defining a pure function, and
2. proving its correctness

let us do both at the same time

We can give Coq functions richer types that are specifications
Example

$$f : \{n : Z \mid n \geq 0\} \rightarrow \{p : Z \mid \texttt{prime}\ p\}$$

The type $\{x : A \mid P\}$

Notation for `sig` A (`fun` $x \Rightarrow P$) where

```
Inductive sig (A : Set) (P : A → Prop) : Set :=
  exist : ∀x:A, P x → sig P
```

In practice, we adopt the more general specification

$$f : \forall\, x : \tau_1,\ P\ x \to \{y : \tau_2 \mid Q\ x\ y\}$$

The type $\{x : A \mid P\}$

Notation for `sig` A (`fun` $x \Rightarrow P$) where

```
Inductive sig (A : Set) (P : A → Prop) : Set :=
  exist : ∀x:A, P x → sig P
```

In practice, we adopt the more general specification

$$f : \forall\ x : \tau_1,\ P\ x \rightarrow \{y : \tau_2 \mid Q\ x\ y\}$$

The type $\{x : A \mid P\}$

Notation for `sig` A (`fun` $x \Rightarrow P$) where

```
Inductive sig (A : Set) (P : A → Prop) : Set :=
  exist : ∀x:A, P x → sig P
```

In practice, we adopt the more general specification

$$f : \forall\ x : \tau_1,\ P\ x \longrightarrow \{y : \tau_2 \mid Q\ x\ y\}$$

Example: the min_elt function

```
Definition min_elt :
  ∀s, ¬s=Empty → bst s →
    { m:Z | In m s ∧ ∀x, In x s → m <= x }.
```

We usually adopt a definition-by-proof
(which is now a definition-and-proof)

Still the same ML program

```
Coq < Extraction sig.
type 'a sig = 'a
  (* singleton inductive, whose constructor was exist *)
```

Example: the min_elt function

```
Definition min_elt :
  ∀s, ¬s=Empty → bst s →
    { m:Z | In m s ∧ ∀x, In x s → m <= x }.
```

We usually adopt a definition-by-proof
(which is now a definition-and-proof)

Still the same ML program

```
Coq < Extraction sig.
type 'a sig = 'a
  (* singleton inductive, whose constructor was exist *)
```

Example: the min_elt function

```
Definition min_elt :
  ∀s, ¬s=Empty → bst s →
    { m:Z | In m s ∧ ∀x, In x s → m <= x }.
```

We usually adopt a definition-by-proof
(which is now a definition-and-proof)

Still the same ML program

```
Coq < Extraction sig.
type 'a sig = 'a
  (* singleton inductive, whose constructor was exist *)
```

Specification of a boolean function: {A}+{B}

Notation for `sumbool A B` where

```
Inductive sumbool (A : Prop) (B : Prop) : Set :=
  | left : A → sumbool A B
  | right : B → sumbool A B
```

this is an informative disjunction

Example:

```
Definition is_empty : ∀s, { s=Empty } + { ¬ s=Empty }.
```

Extraction is a boolean

```
Coq < Extraction sumbool.
type sumbool = Left | Right
```

Specification of a boolean function: {A}+{B}

Notation for `sumbool A B` where

```
Inductive sumbool (A : Prop) (B : Prop) : Set :=
  | left : A → sumbool A B
  | right : B → sumbool A B
```

this is an informative disjunction

Example:

```
Definition is_empty : ∀s, { s=Empty } + { ¬ s=Empty }.
```

Extraction is a boolean

```
Coq < Extraction sumbool.
type sumbool = Left | Right
```

Specification of a boolean function: {A}+{B}

Notation for `sumbool A B` where

```
Inductive sumbool (A : Prop) (B : Prop) : Set :=
  | left : A → sumbool A B
  | right : B → sumbool A B
```

this is an informative disjunction

Example:

```
Definition is_empty : ∀s, { s=Empty } + { ¬ s=Empty }.
```

Extraction is a boolean

```
Coq < Extraction sumbool.
type sumbool = Left | Right
```

Specification of a boolean function: {A}+{B}

Notation for `sumbool A B` where

```
Inductive sumbool (A : Prop) (B : Prop) : Set :=
  | left : A → sumbool A B
  | right : B → sumbool A B
```

this is an informative disjunction

Example:

```
Definition is_empty : ∀s, { s=Empty } + { ¬ s=Empty }.
```

Extraction is a boolean

```
Coq < Extraction sumbool.
type sumbool = Left | Right
```

Variant sumor A+{B}

```
Inductive sumor (A : Set) (B : Prop) : Set :=
  | inleft : A → A + {B}
  | inright : B → A + {B}
```

Extracts to an option type

Example:

```
Definition min_elt :
  ∀s, bst s →
  { m:Z | In m s ∧ ∀x, In x s → m <= x } + { s=Empty }.
```

Variant sumor A+{B}

```
Inductive sumor (A : Set) (B : Prop) : Set :=
  | inleft : A → A + {B}
  | inright : B → A + {B}
```

Extracts to an option type

Example:

```
Definition min_elt :
  ∀s, bst s →
  { m:Z | In m s ∧ ∀x, In x s → m <= x } + { s=Empty }.
```

Variant sumor A+{B}

```
Inductive sumor (A : Set) (B : Prop) : Set :=
  | inleft : A → A + {B}
  | inright : B → A + {B}
```

Extracts to an option type

Example:

```
Definition min_elt :
  ∀s, bst s →
  { m:Z | In m s ∧ ∀x, In x s → m <= x } + { s=Empty }.
```

The mem function

```
Hypothesis compare : ∀x y, {x<y} + {x=y} + {x>y}.

Definition mem : ∀x s, bst s → { In x s }+{ ¬(In x s) }.
Proof.
  induction s; intros.
  (* s = Empty *)
  right; intro h; inversion_clear h.
  (* s = Node s1 z s2 *)
  destruct (compare x z) as [[h1 | h2] | h3].
  ...
Defined.
```

The mem function

```
Hypothesis compare : ∀x y, {x<y} + {x=y} + {x>y}.

Definition mem : ∀x s, bst s → { In x s }+{ ¬(In x s) }.
Proof.
  induction s; intros.
  (* s = Empty *)
  right; intro h; inversion_clear h.
  (* s = Node s1 z s2 *)
  destruct (compare x z) as [[h1 | h2] | h3].
  ...
Defined.
```

The mem function

```
Hypothesis compare : ∀x y, {x<y} + {x=y} + {x>y}.

Definition mem : ∀x s, bst s → { In x s }+{ ¬(In x s) }.
Proof.
  induction s; intros.
  (* s = Empty *)
  right; intro h; inversion_clear h.
  (* s = Node s1 z s2 *)
  destruct (compare x z) as [[h1 | h2] | h3].
  ...
Defined.
```

The mem function

```
Hypothesis compare : ∀x y, {x<y} + {x=y} + {x>y}.

Definition mem : ∀x s, bst s → { In x s }+{ ¬(In x s) }.
Proof.
  induction s; intros.
  (* s = Empty *)
  right; intro h; inversion_clear h.
  (* s = Node s1 z s2 *)
  destruct (compare x z) as [[h1 | h2] | h3].
  ...
Defined.
```

The mem function

```
Hypothesis compare : ∀x y, {x<y} + {x=y} + {x>y}.

Definition mem : ∀x s, bst s → { In x s }+{ ¬(In x s) }.
Proof.
  induction s; intros.
  (* s = Empty *)
  right; intro h; inversion_clear h.
  (* s = Node s1 z s2 *)
  destruct (compare x z) as [[h1 | h2] | h3].
  ...
Defined.
```

To sum up, using dependent types

- we replace a definition and a proof by a single proof
- the ML function is still available using extraction

On the contrary, it is more difficult to prove several properties of the same function

To sum up, using dependent types

- we replace a definition and a proof by a single proof
- the ML function is still available using extraction

On the contrary, it is more difficult to prove several properties of the same function

Modules and functors

Coq has a module system similar to Objective Caml's one

Coq modules can contain definitions but also proofs, notations, hints for the `auto` tactic, etc.

As Ocaml, Coq has functors i.e. functions from modules to modules

Modules and functors

Coq has a module system similar to Objective Caml's one

Coq modules can contain definitions but also proofs, notations, hints for the `auto` tactic, etc.

As Ocaml, Coq has functors i.e. functions from modules to modules

Modules and functors

Coq has a module system similar to Objective Caml's one

Coq modules can contain definitions but also proofs, notations, hints for the `auto` tactic, etc.

As Ocaml, Coq has functors i.e. functions from modules to modules

ML modules

```
module type OrderedType = sig
  type t
  val compare: t → t → int
end

module Make(Ord: OrderedType) : sig
  type t
  val empty : t
  val mem : Ord.t → t → bool
  ...
end
```

Coq modules

```
Module Type OrderedType.
  Parameter t : Set.
  Parameter eq : t → t → Prop.
  Parameter lt : t → t → Prop.
  Parameter compare : ∀x y, {lt x y}+{eq x y}+{lt y x}.
  Axiom eq_refl : ∀x, eq x x.
  Axiom eq_sym : ∀x y, eq x y → eq y x.
  Axiom eq_trans : ∀x y z, eq x y → eq y z → eq x z.
  Axiom lt_trans : ∀x y z, lt x y → lt y z → lt x z.
  Axiom lt_not_eq : ∀x y, lt x y → ¬(eq x y).
  Hint Immediate eq_sym.
  Hint Resolve eq_refl eq_trans lt_not_eq lt_trans.
End OrderedType.
```

Coq modules

```
Module Type OrderedType.
  Parameter t : Set.
  Parameter eq : t → t → Prop.
  Parameter lt : t → t → Prop.
  Parameter compare : ∀x y, {lt x y}+{eq x y}+{lt y x}.
  Axiom eq_refl : ∀x, eq x x.
  Axiom eq_sym : ∀x y, eq x y → eq y x.
  Axiom eq_trans : ∀x y z, eq x y → eq y z → eq x z.
  Axiom lt_trans : ∀x y z, lt x y → lt y z → lt x z.
  Axiom lt_not_eq : ∀x y, lt x y → ¬(eq x y).
  Hint Immediate eq_sym.
  Hint Resolve eq_refl eq_trans lt_not_eq lt_trans.
End OrderedType.
```

Coq modules

```
Module Type OrderedType.
  Parameter t : Set.
  Parameter eq : t → t → Prop.
  Parameter lt : t → t → Prop.
  Parameter compare : ∀x y, {lt x y}+{eq x y}+{lt y x}.
  Axiom eq_refl : ∀x, eq x x.
  Axiom eq_sym : ∀x y, eq x y → eq y x.
  Axiom eq_trans : ∀x y z, eq x y → eq y z → eq x z.
  Axiom lt_trans : ∀x y z, lt x y → lt y z → lt x z.
  Axiom lt_not_eq : ∀x y, lt x y → ¬(eq x y).
  Hint Immediate eq_sym.
  Hint Resolve eq_refl eq_trans lt_not_eq lt_trans.
End OrderedType.
```

Coq modules

```
Module Type OrderedType.
  Parameter t : Set.
  Parameter eq : t → t → Prop.
  Parameter lt : t → t → Prop.
  Parameter compare : ∀x y, {lt x y}+{eq x y}+{lt y x}.
  Axiom eq_refl : ∀x, eq x x.
  Axiom eq_sym : ∀x y, eq x y → eq y x.
  Axiom eq_trans : ∀x y z, eq x y → eq y z → eq x z.
  Axiom lt_trans : ∀x y z, lt x y → lt y z → lt x z.
  Axiom lt_not_eq : ∀x y, lt x y → ¬(eq x y).
  Hint Immediate eq_sym.
  Hint Resolve eq_refl eq_trans lt_not_eq lt_trans.
End OrderedType.
```

The Coq functor for binary search trees

```
Module BST (X: OrderedType).
  Inductive tree : Set :=
  | Empty
  | Node : tree → X.t → tree → tree.

  Fixpoint mem (x:X.t) (s:tree) {struct s} : bool := ...

  Inductive In (x:X.t) : tree → Prop := ...
  Hint Constructors In.

  Inductive bst : tree → Prop :=
  | bst_empty : bst Empty
  | bst_node : ∀x l r, bst l → bst r →
     (∀y, In y l → X.lt y x) → ...
```

Conclusion

Coq is a tool of choice for the verification of purely functional programs, up to modules

ML or Haskell code can be obtained via program extraction

Conclusion

Coq is a tool of choice for the verification of purely functional programs, up to modules

ML or Haskell code can be obtained via program extraction

Agda

Catarina Coquand

August 16, 2005

Background

- Agda is an interactive system for developing proofs in a variant of Martin-Löf's type theory
- It is based on the idea of direct manipulation of proof-term and not on tactics. The proof is a term, not a script.
- The language has ordinary programming constructs such as data-types and case-expressions, signatures and records, let-expressions and modules.
- Has an emacs-interface and a graphical interface, Alfa

System

Agda is an interactive system.

- It consists of a type checker and a termination checker
- Implemented in Haskell
- You will use a simpler version of Agda (with a small library)

A proof of $A \to A$

- The proof of $A \to A$ is the term $\lambda x : A.x$
- In Agda

```
\x -> x
-- alternative: \(x::A) -> x
```

- The syntax of Agda is rather close to Haskell

The identity function

- Function definition

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

- Application:

```
id 0
id 'c'
```

Syntactic Sugar for Function Definitions

```
id (A::Set) :: A -> A
id a =  a
```

Inbuilt type: Pairs

- Pairs are written `A` $\times$ `B`
- A pair is written `(a,b)`
- Projection functions
  - `fst :: A` $\times$ `B -> A`
  - `snd :: A` $\times$ `B -> B`
- Corresponds to logical and

Rule for And

$$\frac{\begin{array}{c}[A\&B]\\ C\end{array} \qquad \begin{array}{cc}A & B\end{array}}{C}$$

```
curry (A,B,C::Set) :: (A × B -> C) -> A -> B -> C
curry f a b = f (a,b)
```

And-elimination

Stating the $\&$-elimination rule:

$$\frac{\begin{array}{cc}[A & B] \\ C & \end{array} \qquad A\&B}{C}$$

```
uncurry(A,B,C::Set) :: (A -> B -> C)  -> A × B -> C
```

Swap

Bengt's proof of $A\&B \Longrightarrow B\&A$. We use the $\&$-elimination i.e. `uncurry`

```
swap (A,B::Set) :: (A × B) -> B × A
swap p = uncurry (\x y -> (y,x)) p
```

Inbuilt Type: Booleans

- Type is `Bool`
- Constructed by `True` and `False`
- We have the ordinary `if_then_else` construction

Inbuilt Types: Lists

- Type is `List A`
- Constructed by `Nil` and `:`
- The list `[]` is syntactic sugar for `Nil`
- The list `[1,2,5]` is syntactic sugar for `1:[2,5]`
- The list `[1,2,5]` is syntactic sugar for `1:2:5:Nil`

More Inbuilt Types

- `Integer`: Infinite integers with usual operations except division
- `Char`: Characters with some standard operations
- `String`: Strings are lists of characters

Let-expressions

We can also use let-notation

```
ex :: Integer
ex = let {
         big :: Integer;
         big = 12324567891234566789;
         neg :: Integer;
         neg = negate 1000;
         }
     in big*neg
```

Layout rule

```
ex :: Integer
ex = let big :: Integer
         big = 123245678912345667 89
         neg :: Integer
         neg = negate 1000
     in big*neg
```

Equality Type

- We write equality as `a == b`
- It is reflexive, symmetric, transitive, and substitutive
- Equivalent to Leibniz-equality

Typechecking a proof of Reflexivity

We have `refId x` is of type `x == x`,

```
refId 6 ::       6 == 6 -- also the inferred type
refId 6 ::       2 * 3 == 4 + 2
```

This is so since $6 == 6$ and $2 * 3 == 4 + 2$ are convertible. (See Herman Geuver's note on type checking)

Stating a Quantified Theorem

State that $==$ is symmetrical: $\forall\ x\ y.x == y \Longrightarrow y == x$

```
symmEq (A::Set):: (x,y::A) -> x == y -> y == x
symmEq x y  =  ....
```

Equivalent to

```
symmEq (A::Set) :: (x::A) -> (y::A) -> x == y -> y == x
symmEq x y  =  ....
```

Defining Type Synonyms

```
Pred :: Set -> Type
Pred X = X -> Set

Rel ::  Set -> Type
Rel X = X -> X -> Set

Symmetrical (X::Set) ::  (R::Rel X) -> Set
Symmetrical R = (x1,x2::X) |-> (x1 `R` x2 -> x2 `R` x1)

symmEq (A::Set) :: Symmetrical (==)
symmEq x1 x2 = ...
```

Language Constructions : Data Types

We introduce a new type by data-type construction

```
data Bool = True | False
data List (A::Set) = Nil | (:) (a::A) (l::List A)
```

Language Constructions :Case Expressions

We can introduce implicitly defined constants by case-expressions. (Should be thought of as defining functions with pattern-equations.)

```
(++)(A::Set) :: List A -> List A -> List A
(++) xs ys = case xs of
                (Nil) -> ys
                (x : xs') -> x:xs'++ys
```

Has to cover all possible cases. The term `xs ++ ys` is on normal form.

Elimination Rule for Lists

```
elimList (A ::Set) ::
         (C::List A -> Set) ->
         C [] ->
         ((x::A) -> (xs::List A) -> C xs -> C (x:xs)) ->
         (xs::List A) ->
         C xs
elimList C c_nil c_con xs =
          case xs of
                 (Nil) -> c_nil
                 (x : xs') -> c_con x xs' (elimList  C c_nil c_con xs')
```

Logic

- Or: `data Plus (X,Y::Set)  = Inl (x::X) | Inr (y::Y)`
- Exists: `data Sigma (X::Set) (Y::X -> Set) = dep_pair (x::X)(y::Y x)`
- Truth: `data Unit :: Set  = unit`
- Absurdity: `data Empty :: Set =`

Or- elimination

```
elimPlus (X,Y::Set) ::
    (C::Plus X Y -> Set) ->
    (c_lft::(x::X) -> C (Inl x)) ->
    (c_rgt::(y::Y) -> C (Inr y)) ->
    (xy::Plus X Y) ->
    C xy
elimPlus C c_lft c_rgt xy = case xy of
                                (Inl x) -> c_lft x
                                (Inr y) -> c_rgt y

whenPlus (X,Y,Z::Set) ::  (f::X -> Z) -> (g::Y -> Z) -> (Plus X Y -> Z)
whenPlus = elimPlus (\h -> Z)
```

Absurdity

```
data Empty :: Set =

elimEmpty ::  (C::Empty -> Set) -> (z::Empty) -> C z
elimEmpty C z = case z of { }

whenEmpty ::  (X::Set) -> Empty -> X
whenEmpty X z = case z of { }

Not :: Set -> Set
Not X = X -> Empty
absurdElim (A::Set) :: A -> Not A -> (X::Set) -> X
absurdElim h h' X = whenEmpty X (h' h)
```

Inductive families

```
idata (==) (X::Set)  :: X  -> X -> Set where
       refId (x::X) :: (==) x  x
```

Use elimination rules and not case for inductive families.

Language Constructions : Structures/Signature

```
PlusSig :: (A::Set) -> Set
PlusSig A = sig
   zer :: A
   plus :: A -> A -> A

IntPluSig :: PlusSig Integer
IntPluSig = struct
              zer ::  Integer
              zer  = 0
              plus ::  Integer -> Integer -> Integer
              plus  = (+)
```

Another Instance

```
ListPluSig :: (A::Set) -> PlusSig (List A)
ListPluSig A = struct
                 zer ::  List A
                 zer  = []
                 plus ::  List A -> List A -> List A
                 plus  = (++)
```

Using Struct/Sig

```
f :: Integer
f = IntPlusSig.plus IntPlusSig.zer (IntPlusSig.zer +1)

f :: Integer
f  = let open IntPlusSig use plus, zer
     in plus zer (zer + 1)
```

Packages

Packages

```
package Natural where
  open Prelude  use  Pred
  open Boolean  use  Bool, False, True

  data Nat   = Zero | Succ (n::Nat)

  natrec (C::Pred Nat)(bc::C Zero)
         (ic::(n::Nat) -> C n -> C (Succ n))
         (m::Nat)
    :: C m =  ....
  isZero (a::Nat) :: Bool
     = ....
```

Examples : typechecking

```
F :: Set
F  = Bool
f :: Bool -> F
f = \a -> a
```

Gives an equality constraint:

```
Bool = F
```

We must compute F to see that they are equal.

Example : Typechecking

```
F :: (A::Set) -> Set
F = \A -> A

f :: (B::Set) -> B -> F B
f = \B -> \a -> a
```

Gives the equality constraint:

```
B = F B
```

Meta-variables

- A meta-variable can only occur in **one** typing constraint.
- The result of typechecking is a set of typing constraints and equality constraints instead of a yes and no answer when type-checking terms with meta-variables.
- Using higher-order unification will sometimes (often) solve the constraints.

Meta-variables

```
f :: (A::Set) -> (a::A) -> A
f  = \(B::Set) -> \(b::B) -> ?
```

Is type correct if

$$B : Set, b : B \vdash \ ? : B$$

Examples Meta Variables ctd

```
f :: (A::Set) -> (a::A) ->  A
f  = \(B::Set) -> \(b::?) -> b
```

Is type correct if

$$B : Set \vdash \ ? \text{ type}$$

and

$$A \equiv \ ?(B = A)$$

Hidden Arguments

We do not have polymorhism, but hidden arguments

```
id (A::Set) :: A -> A
id   a =  a
id 'c'
```

is translated into `id |? 'c' .`

Hidden Arguments ctd

We can write more explicitly

```
id :: (A::Set) |->  A -> A
id   = \(A::Set) |-> \a -> a

id |Char 'c'
```

Emacs-symbols

```
(global-set-key (kbd "C-*") (lambda () (interactive) (insert "\327")))
;;;  Cartesian product
(global-set-key (kbd "C-.") (lambda () (interactive) (insert "\260")))
;;;; Ring
(global-set-key (kbd "C-!") (lambda () (interactive) (insert "\254")))
;;;; not
(global-set-key [f9] (lambda () (interactive) (insert "\330")))
;;;; Empty set
(global-set-key [f10] (lambda () (interactive) (insert "\267"))) ;;;; M
(global-set-key [f11] (lambda () (interactive) (insert "\367")))
```

Agda Commands

[Agda-documentation team at AIST CVS]

August 15, 2005

Abstract

Contents

A List of commands

All `Agda` commands can be invoked by key operations, or by selecting items in menus. The commands which are effective in the whole of the code are found in Agda menu in the menu bar. On the other hand, the commands for goals are found in the popup menu by right-clicking on a goal. Most of items in goal menu depend on the context.

Commands are classified to four categories roughly.

Necessary commands you must know.

Important commands used very often.

Often commands which help you use `Agda` effectively. You can do without them.

A.1 Agda menu

Restart
key: `C-c C-x C-c`
category: *often*
(Re-)initializes the type-checker.

Quit
key: `C-c C-q`
category: *necessary*
Quits and cleans up after agda. If you do not want Emacs to warn in quiting Emacs, then you should invoke this command every time.

Goto error
key: `C-c ‘`
category: *important*
Jumps to the line the first error occurs.

Load
key: `C-c C-x C-b`
category: *often*
Reads and type-checks the current buffer.

Chase-load
key: `C-c C-x RET`
category: *necessary*
Reads and type-checks the current buffer and included files.

Show constraints
key: `C-c C-e`
category: *often*
Shows all constraints in the code. A constraint is an equation of two goals or of a goal and an expression.

Compute
key: `C-c C-x >`
category: *often*
Compute a closed top-level expression. Does not reduce under lambda.

Suggest

key: `C-c C-x C-s`
category: *often*

Suggests suitable expressions.

Show goals

key: `C-c C-x C-a`
category: *often*

Shows all goals in the current buffer.

Next goal

key: `C-c C-f`
category: *often*

Moves the cursor to the next goal, if any.

Previous goal

key: `C-c C-b`
category: *often*

Moves the cursor to the previous goal, if any.

Undo

key: `C-c C-u`
category: *important*

Cancels the last `Agda` command or typing.

Text state

key: `C-c '`
category: *necessary*

Resets agda to the state that the current buffer is loaded.

Check termination

key: `C-c C-x C-t`
category: *often*

Runs termination check on the current buffer. You will need to retype-check the buffer.

Submitting bug report

key:
category:

(not implemented)

A.2 Goal commands.

When a goal is replaced with a new expression by the commands below, we know that it is type-correct.

Give

key: `C-c C-g`
category: *often*

Substitute a given expression in the goal.

Intro

key: `C-c TAB`
category: *often*

Introduces the canonical expression of the type the goal i.e. an abstraction, a record, or a constructor if only one possible exists

Refine

key: `C-c C-r`
category: *important*

Given an expression, e, this command will apply the minimum number of meta-variables needed for the expression e ? ? ..? to be type-checkable

Refine(exact)

key: `C-c C-s`
category: *often*

Given a expression with arity n it applies n meta-variables to the given expression.

Refine(projection)

key: `C-c C-p`
category: *often*

Refines the goal with a expression in a given package. For example, a function `cat` is defined in the package `OpList`. When a goal is filled with "`OpList cat`", the Refine (projection) command accepts it and refine the goal but refine command fails. (In case a goal is filled with "`OpList.cat`", refine command works.)

Case

key: `C-c C-c`
category: *often*

Makes a template of a case expression with a given formal parameter.

Let

key: `C-c C-l`
category: *often*

Makes a template of a let expression with given formal parameters.

Abstraction

key: `C-c C-a`
category: *often*

Makes a template of a function expression with given formal parameters.

Goal type

key: `C-c C-t`
category: *often*

Shows the type of the goal.

Goal type(unfolded)

key: `C-c C-x C-r`
category: *often*

Shows the reduced type of the goal.

Context

key: `C-c |`
category: *important*

Shows context (names already defined) of the goal.

Infer type

key: `C-c :`

category: *important*

Prompts an expression and infers the type of it, under the current context.

Infer type(unfolded)

key: `C-c C-x :`

category: *often*

Prompts an expression and infers the reduced type of it, under the current context.

References

[1] Programming Logic Team at Chalmers and AIST. Agda, 2000. http://www.coverproject.org/AgdaPage/.

[2] Lena Magnusson and Bengt Nordstrm. The alf proof editor and its proof engine. In *TYPES '93: Proceedings of the international workshop on Types for proofs and programs*, pages 213–237. Springer-Verlag New York, Inc., 1994.

[3] Nordström, B., Petersson, K. and Smith, J.M., *Programming in Martin-Löf's Type Theory*, available at
http://www.cs.chalmers.se/Cs/Research/Logic/book/.

[4] Nordström, B., Petersson, K. and Smith, J.M., *Martin-Löf's Type Theory*, pp. 1 - 37 in Handbook of Logic in Computer Science, vol. 5 (2000), Oxford Science Publication.

Exercises in Agda

Catarina Coquand

August 11, 2005

In the library `/usr/local/share/summerschool/agda/SummerSchool05` (on the Summer School's computers) you will find a small library with some standard files. It is good to look around in the different files.

1. In the file with `Exercises/Logic.agda`, there are many basic exercises on the Curry-Howard correspondence between propositions and sets and in the file `Exercises/ListProps.agda` there is some exercises on lists

2. Formulate and prove in type theory

$$((\forall x : A)(\exists y : B)R\ x\ y) \rightarrow (\exists f : A \rightarrow B)(\forall x : A)R\ x\ (f\ x)$$

   This is a possible formulation of the axiom of choice. It was stressed by Bishop that this form of axiom of choice is constructively valid.

3. In the file `Nat.agda` you can find the definition of natural numbers, Nat. Define the equality, $eqNat$, on Nat by double recursion and show that this equality is substitutive (if $eqNat\ x\ y$ and $P\ x.$, then $P\ y$

4. Define the set of well-founded trees (well-orderings), W: given a family of sets $B(x)$ over a set A, the set $W\ A\ B$ has one constructor sup, where $sup\ x\ f : W\ A\ B$ if $x : A$ and $f : B(x) \rightarrow W\ A\ B$. Write the corresponding elimination rule in Agda. Prove that

$$W\ A\ B \rightarrow \neg(\forall x : A.B\ x)$$

   that is, the two propositions $W\ A\ B$ and $\Pi\ A\ B$ are incompatible. Explain intuitively why

$$W\ A\ B \rightarrow \exists x : A.\neg(B\ x)$$

   should not be provable in type theory.

5. Define the type

$$F\ A\ n\ =\ \underbrace{A \rightarrow \ldots \rightarrow A}_{n} \rightarrow A$$

   i.e. a function F that takes a set A, a natural number, n, and returns a set.

Use this type to define a tautology function, i.e. a function that takes as arguments a number n, a boolean function with n arguments, and returns $True$ if and only if this boolean function is a tautology.

6. Let A be a set with a well-founded relation, $<$, on it. Show that if $f : Nat \to A$ then $\neg((\forall n : Nat)\ f\ (n+1) < f\ n)$, i.e. there is no infinite decreasing sequence. Show that if $<$ is decidable then we have $(\exists n : Nat)\neg(f\ (n+1) < f\ n)$

7. Let A be a set with a well-founded relation, $<$, on it (see `WellOrder.agda`). Prove that if A is inhabited we have

$$((\forall x : A)(P\ x \vee (f\ x < x))) \to (\exists x : A)P\ x$$

Explain how this proof corresponds to a "for-loop" program.

Introduction to Co-Induction in Coq

Yves Bertot

August 2005

Motivation

- Reason about infinite data-structures,
- Reason about lazy computation strategies,
- Reason about infinite processes, abstracting away from dates.
  - Finite state automata,
  - Temporal logic,
  - Computation on streams of data.

Inductive types as least fixpoint types

- Inductive types are fixpoints of "abstract functions",
  - If $\{c_i\}_{i\in\{1,\ldots,j\}}$ are the constructors of I and $c_i\ a_1\ \cdots\ a_k$ is well-typed then $c_i\ a_1\ \cdots\ a_k \in I$
  - Fixpoint property also gives pattern-matching: if $c_i : T_{i,1}\ \cdots\ T_{i,k} \to I$ and $f_i : T_{i,1}\ \cdots\ T_{i,k} \to B$, then there exists a single function $\phi : I \to B$ such that $\phi(c_i\ a_1 \ldots\ a_k) = f_i\ a_1\ \cdots\ a_k$.
- Initiality:
  - if f_i are functions with type $f_i : T_{i,1}[A/I]\ \cdots\ T_{i,k}[A/I] \to A$, then there exists a single function $\phi : I \to A$ such that $\phi(c_1\ a_1\ \cdots\ a_k) = f_i\ a'_1\ \cdots\ a'_k$, where $a'_m = \phi(a_m)$ if $T_m = I$ and $a'_m = a_m$ otherwise.
  - Initiality gives structural recursion.

CoInductive types

- Consider a type C with the first two fixpoint properties,
  - Images of constructors are in C (the co-inductive type),
  - Functions on C can be defined by pattern-matching,
- Take a closer look at pattern-matching:
  - With pattern matching you can define a function $\sigma : C \to (T_{11} * \cdots * T_{1k_1}) + (T_{21} * \cdots * T_{2k_2}) + \cdots$ so that $\sigma(t) = (a_1, \ldots a_{k_i}) \in (T_{i1} * \cdots T_{ik_i})$ when $t = c_i\ a_1\ \cdots\ a_k$
- Replace *initiality* with *co-initiality*, i.e.,
  - If $f : A \to (T_{11} * \cdots * T_{1k_1})[A/C] + (T_{21} * \cdots * T_{2k_2})[A/C] + \cdots$, then there exists a single $\phi : A \to C$ such that $\phi(a) = c_i\ a'_1\ \cdots\ a'_{k_i}$ when $f(a) = (T_{i1} * \cdots * T_{ik_i})[A/C]$ and $a'_j = \phi(a_j)$ if $T_{ij} = C$ and $a'_j = a_j$ otherwise.

Practical reading of theory

- For both kinds of types,
  - constructors and pattern-matching can be used in a similar way,
- For inductive types,
  - Recursion is only used to consume elements of the type,
  - Arguments of recursive calls can only be sub-components of constructors,
- For co-inductive types,
  - Co-recursion is only used to produce elements of the type,
  - Co-recursive calls can only produce sub-components of constructors.

Theory on an example

- Consider the two definitions:

```
Inductive list (A:Set) :  Set :=
  nil :  list A | cons :  A -> list A -> list A.
CoInductive Llist (A:Set) :  Set :=
   Lnil :  Llist A
 | Lcons :  A -> Llist A -> Llist A.
Implicit Arguments Lcons.
```

- given values and functions `v:B` and `f:A->B->B`, we can define a function `phi :  list A -> B` by the following

```
Fixpoint phi (l:list A) : B :=
  match l with
    nil => v | const a t => f a (phi t)
  end.
```

Theory on an example (continued)

- The "natural result type" of pattern-matching on inductive lists is: `unit+(A*list A)`

```
Definition sigma1(A:Set)(l:list A):unit+(A*list A):=
  match l with
    nil => inl (B:=A*list A) tt
  | cons a tl => inr (A:=unit) (a,tl)
  end.
```

- The natural result type of pattern matching on co-inductive lists (type `Llist`) is similar: `unit+(A*Llist A)`
- We can define a co-recursive function `phi : B -> Llist A` if we are able to inhabit the type `B -> unit+(A*B)`.

Categorical terminology

- In the category **Set**, collections of constructors define a functor F,
- for a given object A, $F(A)$ corresponds to the natural result type for pattern-matching as described in the previous slide,
- An F-algebra is an object with a morphism $F(A) \longrightarrow A$,
- F-algebras form a category, and the inductive type is an initial object in this category,
- An F-coalgebra is an object with a morphism $A \longrightarrow F(A)$,
- F-coalgebras form a category, and the coinductive type is a final object in this category.

Co-Inductive types in Coq

- Syntactic form of definitions is similar to inductive types (given a few frames before),
- pattern-matching with the same syntax as for inductive types.
- Elements of the co-inductive type can be obtained by:
  - Using the constructors,
  - Using the pattern-matching construct,
  - Using co-recursion.

Constructing co-inductive elements

```
Definition ll123 :=
    Lcons 1 (Lcons 2 (Lcons 3 (Lnil nat))).
Fixpoint list_to_llist (A:Set) (l:list A)
    {struct l} :  Llist A :=
 match l with
   nil => Lnil A
 | a::tl => Lcons a (list_to_llist A tl)
 end.
Definition ll123' := list_to_llist nat (1::2::3::nil).
```

- `list_to_llist` uses plain structural recursion on lists and plain calls to constructors.

Infinite elements

- `list_to_llist` shows that `list A` is isomorphic to a subset of `Llist A`
- Lists in `list A` are finite, recursive traversal on them terminates,
- There are infinite elements:
  ```
  CoFixpoint lones :  Llist nat := Lcons 1 lones.
  ```
- `lones` is the value of the co-recursive function defined by the *finality* statement for the following `f`:
  ```
  Definition f :  unit -> unit+(nat*unit) :=
      fun _ => inr unit (1,tt).
  ```

Infinite elements (continued)

- Here is a definition of what is called the *finality* statement in this lecture:

```
CoFixpoint Llist_finality
    (A:Set)(B:Set)(f:B->unit+(A*B)):B->Llist A:=
fun b:B => match f b with
  inl tt => Lnil A
| inr (a,b2) => Lcons a (Llist_finality A B f b2)
end.
```

- The *finality* statement is never used in Coq.
- Instead syntactic check on recursive definitions (guarded-by-constructors criterion).

Streams

```
CoInductive stream (A:Set) :  Set :=
  Cons :  A -> stream A -> stream A.
Implicit Arguments Cons.
```

- an example of type where no element could be built without co-recursion.

  ```
  CoFixpoint nums (n:nat) :  stream nat :=
    Cons n (nums (n+1)).
  ```

Computing with co-recursive values

- Unleashed unfolding of co-recursive definitions would lead to infinite reduction,
- A redex appears only when patern-matching is applied on a co-recursive value.
- Unfolding is performed (only) as needed.

Proving properties of co-recursive values

```
Definition Llist_decompose (A:Set)(l:Llist A) : Llist
A :=
  match l with Lnil => Lnil A | Lcons a tl => Lcons a
tl end.
Implicit Arguments Llist_decompose.
```

- Proofs by pattern-matching as in inductive types.

```
Theorem Llist_dec_thm :
   forall (A:Set)(l:Llist A), l = Llist_decompose l.
Proof.
 intros A l; case l; simpl; trivial.
Qed.
```

Unfolding techniques

- The theorem `Llist_dec_thm` is not just an example,
- A tool to force co-recursive functions to unfold.
- Create a redex that maybe reduced by unfolding recursion.

```
Theorem lones_dec :  Lcons 1 lones = lones.
 simpl.
  ============================
   Lcons 1 lones = lones
pattern lones at 2; rewrite (Llist_dec_thm nat lones);
simpl.
  ============================
   Lcons 1 lones = Lcons 1 lones
```

Proving equality

- Usual equality is an "inductive concept" with no recursion,
- Co-recursion can only provide new values in co-recursive types,
- Need a co-recursive notion of equality.
- Express that two terms are "equal" when then cannot be distinguished by any amount of pattern-matching,
- specific notion of equality for each co-inductive type.

Co-inductive equality

```
CoInductive bisimilar (A:Set) :  Llist A -> Llist A
-> Prop :=
  bisim0 :  bisimilar A (Lnil A)(Lnil A)
| bisim1 :  forall x t1 t2, bisimilar A t1 t2 ->
              bisimilar A (Lcons x t1) (Lcons x t2).
```

Proofs by Co-induction

- Use a tactic `cofix` to introduce a co-recursive value,
- Adds a new hypothesis in the context with the same type as the goal,
- The new hypothesis can only be used to fill a constructor's sub-component,
- Non-typed criterion, the correctness is checked using a `Guarded` command.

Example material

```
CoFixpoint lmap (A B:Set)(f:A -> B)(l:Llist A) :
Llist B :=
  match l with
    Lnil => Lnil B
  | Lcons a tl => Lcons (f a) (lmap A B f tl)
  end.
```

Example proof by co-induction

```
Theorem lmap_bi' :  forall (A:Set)(l:Llist A),
  bisimilar A (lmap A A (fun x => x) l) l.
cofix.
```

1 subgoal

lmap_bi' : forall (A : Set) (l : Llist A),
bisimilar A (lmap A A (fun x : A ⇒ x) l) l

forall (A : Set) (l : Llist A),
bisimilar A (lmap A A (fun x : A ⇒ x) l) l

Example proof by co-induction (continued)

```
intros A l; rewrite
    (Llist_dec_thm _ (lmap A A (fun x=>x) l)); simpl.
...
```

```
  ============================
   bisimilar A
     match
       match l with
       | Lcons a tl ⇒ Lcons a (lmap A A (fun x : A ⇒ x) tl)
       | Lnil ⇒ Lnil A
       end
     with
     | Lcons a tl ⇒ Lcons a tl
     | Lnil ⇒ Lnil A
     end l
```

Example proof by co-induction (continued)

```
case l.
...
  ============================
```

forall (a : A) (l0 : Llist A),
bisimilar A (Lcons a (lmap A A (fun x : A ⇒ x) l0)) (Lcons a l0)

subgoal 2 is:
bisimilar A (Lnil A) (Lnil A)

Example proof by co-induction (continued)

```
intros a k; apply bisim1.
```

```
...
  lmap_bi' : forall (A : Set) (l : Llist A),
             bisimilar A (lmap A A (fun x : A ⇒ x) l) l
...
  ============================
   bisimilar A (lmap A A (fun x : A ⇒ x) k) k
```

- A constructor was used, the recursive hypothesis can be used.

```
apply lmap_bi'.
apply bisim0.
Qed.
```

Minimal real arithmetics

- Represent the real numbers in [0,1] as infinite sequences of bits,
- add a third bit to make computation practical.

Redundant floating-point representations

- In usual represenation $1/2$ is both $0.01111\ldots$ and $0.1000\ldots$,
- Every number $p/2^n$ where p and n are integers has two representations,
- Other numbers have only one,
- A number whose prefix is $0.1010\ldots$ (but finite) is a number that can be bigger or smaller than $1/3$,
- When computing $1/3 + 1/6$ we can never decide what should be the first bit of the result.
- Problem solved by adding a third bit : Now `L`, `C`, or `R`.

Explaining redundancy

- A number of the form `L...` is in [0,1/2], (like a number of the form 0.0...),
  - A number of the form `R...` is in [1/2,1], (like a number of the form 0.1...),
  - A number of the form `C...` is in [1/4,3/4].
- Taking an infinite stream of bits and adding a `L` in front divides by 2,
  - Adding a `R` divides by 2 and adds 1/2,
  - Adding a `C` divides by 2 and adds 1/4.

Coq encoding

```
Inductive idigit :  Set := L | C | R.

CoInductive represents :  stream idigit ->
Rdefinitions.R -> Prop :=
  reprL : forall s r, represents s r ->
            (0 <= r <= 1)%R ->
            represents (Cons L s) (r/2)
| reprR : forall s r, represents s r ->
            (0 <= r <= 1)%R ->
            represents (Cons R s) ((r+1)/2)
| reprC : forall s r, represents s r ->
            (0 <= r <= 1)%R ->
            represents (Cons C s) ((2*r+1)/4).
```

Encoding rational numbers

```
CoFixpoint rat_to_stream (a b:Z) : stream idigit :=
  if Z_le_gt_dec (2*a) b then
    Cons L (rat_to_stream (2*a) b)
  else
    Cons R (rat_to_stream (2*a-b) b).
```

Affine combination of redundant digit streams

- compute the representation of

$$\frac{a}{a'}x + \frac{b}{b'}y + \frac{c}{c'},$$

where x and y are real numbers in [0,1] given by redundant digit streams, and $a \cdots c'$ are positive integers (non-zero when relevant).

- if $2c > c'$ then the result has the form $\mathtt{R}z$ where z is

$$\frac{2a}{a'}x + \frac{2b}{b'}y + \frac{2c - c'}{c'}$$

.

Computation of other digits

- Similar sufficient condition to decide on $\mathtt{C}z$ and $\mathtt{L}z$, for suitable values of z:
  - $$\frac{a}{a'} + \frac{b}{b'} + \frac{c}{c'} \leq \frac{1}{2} \text{ produce } \mathtt{L}$$
  - $$\frac{c}{c'} \geq \frac{1}{4} \text{and} \frac{a}{a'} + \frac{b}{b'} + \frac{c}{c'} \leq 3/4 \text{ produce } \mathtt{C}$$
- if $\frac{a}{a'} + \frac{b}{b'}$ is small enough, you can produce a digit,
- But sometimes necessary to observe x and y.

Consuming input

- if x and y are $\mathtt{L}x'$ and $\mathtt{L}y'$, then

$$\frac{a}{a'}x + \frac{b}{b'}y + \frac{c}{c'}$$

  is also

$$\frac{a}{2a'}x' + \frac{b}{2b'}y' + \frac{c}{c'}$$

- Condition for outputting a digit may still not be ensured, but

$$\frac{a}{2a'} + \frac{b}{2b'} = \frac{1}{2}(\frac{a}{a'} + \frac{b}{b'})$$

- Similar for other possible forms of x and y.

Coq encoding

- Use a well-founded recursive function to consume from x and y until the condition is ensured to produce a digit,
- Produce a digit and perform a co-recursive call,
- This style of decomposition between well-founded part and co-recursive is quite powerful (not documented in Coq'Art, though).

Introduction to the Why tool

Jean-Christophe Filliâtre

CNRS – Université Paris Sud

TYPES summer school – August 25th, 2005

Provers based on HOL are suitable tools to verify purely functional programs (see Tuesday's lecture)

What if you want to verify an imperative program with your favorite prover?

Provers based on HOL are suitable tools to verify purely functional programs (see Tuesday's lecture)

What if you want to verify an imperative program with your favorite prover?

Usual methods

- Floyd-Hoare logic
- Dijkstra's weakest preconditions

- could be formalized in the prover (deep embedding)
- could be applied by a tactic (shallow embedding)

$\Rightarrow$ would be specific to this prover

Usual methods

- Floyd-Hoare logic
- Dijkstra's weakest preconditions

- could be formalized in the prover (deep embedding)
- could be applied by a tactic (shallow embedding)

$\Rightarrow$ would be specific to this prover

Usual methods

- Floyd-Hoare logic
- Dijkstra's weakest preconditions

- could be formalized in the prover (deep embedding)
- could be applied by a tactic (shallow embedding)

$\Rightarrow$ would be specific to this prover

Which programming language?

a realistic existing programming language such as C or Java?

- many constructs $\Rightarrow$ many rules
- would be specific to this language

Which programming language?

a realistic existing programming language such as C or Java?

- many constructs $\Rightarrow$ many rules
- would be specific to this language

Which programming language?

a realistic existing programming language such as C or Java?

- many constructs $\Rightarrow$ many rules
- would be specific to this language

The Why tool

makes program verification

- prover-independent but prover-aware
- language-independent

so that we can use it to verify C, Java, etc. programs with HOL provers but also with FO decision procedures

The Why tool

makes program verification

- prover-independent but prover-aware
- language-independent

so that we can use it to verify C, Java, etc. programs with HOL provers but also with FO decision procedures

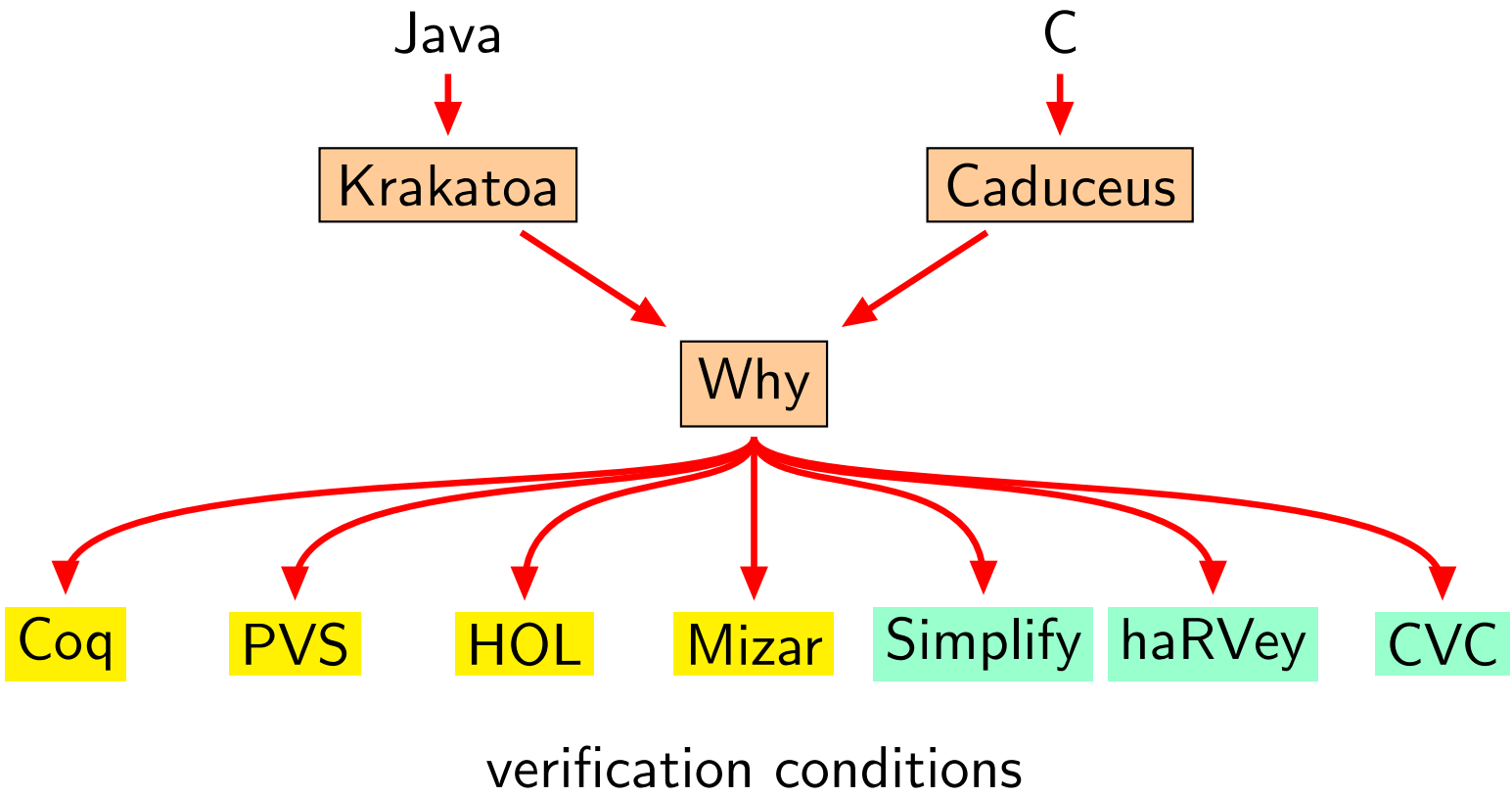
The Why tool

makes program verification

- prover-independent but prover-aware
- language-independent

so that we can use it to verify C, Java, etc. programs with HOL provers but also with FO decision procedures

An intermediate language



Outline

1. A language for program verification
   - syntax
   - typing
   - semantics
2. Proof rules
3. The WHY tool
   - Dijkstra's Dutch flag
4. Verification of C programs

Outline

1. A language for program verification
   - syntax
   - typing
   - semantics
2. Proof rules
3. The WHY tool
   - Dijkstra's Dutch flag
4. Verification of C programs

Outline

1. A language for program verification
   - syntax
   - typing
   - semantics
2. Proof rules
3. The WHY tool
   - Dijkstra's Dutch flag
4. Verification of C programs

Outline

1. A language for program verification
   - syntax
   - typing
   - semantics
2. Proof rules
3. The WHY tool
   - Dijkstra's Dutch flag
4. Verification of C programs

Part I

A language for program verification

The essence of Hoare logic assignment rule

$$\{\ P[x \leftarrow E]\ \}\ x\ :=\ E\ \{\ P\ \}$$

1. absence of aliasing
2. side-effects free E shared between program and logic

The essence of Hoare logic assignment rule

$$\{\ P[x \leftarrow E]\ \}\ x\ :=\ E\ \{\ P\ \}$$

1. absence of aliasing
2. side-effects free E shared between program and logic

The essence of Hoare logic assignment rule

$$\{\ P[x \leftarrow E]\ \}\ x\ :=\ E\ \{\ P\ \}$$

1. absence of aliasing
2. side-effects free E shared between program and logic

Data types

Any purely applicative data type from the logic can be used in programs

Example: a data type `int` for integers with constants 0, 1, etc. and operations +, *, etc.
The pure expression 1+2 belongs to both programs and logic

A single data structure: the reference (mutable variable) containing only pure values, with no possible alias between two different references

Data types

Any purely applicative data type from the logic can be used in programs

Example: a data type `int` for integers with constants `0`, `1`, etc. and operations `+`, `*`, etc.
The pure expression `1+2` belongs to both programs and logic

A single data structure: the reference (mutable variable) containing only pure values, with no possible alias between two different references

Data types

Any purely applicative data type from the logic can be used in programs

Example: a data type `int` for integers with constants `0`, `1`, etc. and operations `+`, `*`, etc.
The pure expression `1+2` belongs to both programs and logic

A single data structure: the reference (mutable variable) containing only pure values, with no possible alias between two different references

Data types

Any purely applicative data type from the logic can be used in programs

Example: a data type `int` for integers with constants `0`, `1`, etc. and operations `+`, `*`, etc.
The pure expression `1+2` belongs to both programs and logic

A single data structure: the reference (mutable variable) containing only pure values, with no possible alias between two different references

ML syntax

No distinction between expressions and statements

$\Rightarrow$ less constructs

$\Rightarrow$ less rules

dereference	`!`x
assignment	x `:=` e
local variable	`let` x `=` e_1 `in` e_2
local reference	`let` x `= ref` e_1 `in` e_2
conditional	`if` e_1 `then` e_2 `else` e_3
loop	`while` e_1 `do` e_2 `done`

sequence e_1 `;` e_2 $\equiv$ `let _ =` e_1 `in` e_2

ML syntax

No distinction between expressions and statements
$\Rightarrow$ less constructs
$\Rightarrow$ less rules

dereference	`!`x
assignment	x `:=` e
local variable	`let` x `=` e_1 `in` e_2
local reference	`let` x `= ref` e_1 `in` e_2
conditional	`if` e_1 `then` e_2 `else` e_3
loop	`while` e_1 `do` e_2 `done`
sequence e_1 `;` e_2	$\equiv$ `let _ =` e_1 `in` e_2

ML syntax

No distinction between expressions and statements
$\Rightarrow$ less constructs
$\Rightarrow$ less rules

dereference	`!`x
assignment	x `:=` e
local variable	`let` x `=` e_1 `in` e_2
local reference	`let` x `=` `ref` e_1 `in` e_2
conditional	`if` e_1 `then` e_2 `else` e_3
loop	`while` e_1 `do` e_2 `done`
sequence e_1 `;` e_2	$\equiv$ `let _ =` e_1 `in` e_2

ML syntax

No distinction between expressions and statements
$\Rightarrow$ less constructs
$\Rightarrow$ less rules

dereference	`!`x
assignment	x `:=` e
local variable	`let` x `=` e_1 `in` e_2
local reference	`let` x `= ref` e_1 `in` e_2
conditional	`if` e_1 `then` e_2 `else` e_3
loop	`while` e_1 `do` e_2 `done`
sequence e_1`;` e_2	$\equiv$ `let _ =` e_1 `in` e_2

ML syntax

No distinction between expressions and statements
$\Rightarrow$ less constructs
$\Rightarrow$ less rules

dereference	`!`x
assignment	x `:=` e
local variable	`let` x `=` e_1 `in` e_2
local reference	`let` x `= ref` e_1 `in` e_2
conditional	`if` e_1 `then` e_2 `else` e_3
loop	`while` e_1 `do` e_2 `done`

sequence e_1 `;` e_2 $\equiv$ `let _ =` e_1 `in` e_2

ML syntax

No distinction between expressions and statements
$\Rightarrow$ less constructs
$\Rightarrow$ less rules

dereference	`!`x
assignment	x `:=` e
local variable	`let` x `=` e_1 `in` e_2
local reference	`let` x `= ref` e_1 `in` e_2
conditional	`if` e_1 `then` e_2 `else` e_3
loop	`while` e_1 `do` e_2 `done`

sequence e_1 `;` e_2 $\equiv$ `let _ =` e_1 `in` e_2

ML syntax

No distinction between expressions and statements
$\Rightarrow$ less constructs
$\Rightarrow$ less rules

dereference	`!`x
assignment	x `:=` e
local variable	`let` x `=` e_1 `in` e_2
local reference	`let` x `= ref` e_1 `in` e_2
conditional	`if` e_1 `then` e_2 `else` e_3
loop	`while` e_1 `do` e_2 `done`

sequence e_1 `;` e_2 $\equiv$ `let _ =` e_1 `in` e_2

ML syntax

No distinction between expressions and statements
$\Rightarrow$ less constructs
$\Rightarrow$ less rules

dereference	`!`x
assignment	x `:=` e
local variable	`let` x `=` e_1 `in` e_2
local reference	`let` x `= ref` e_1 `in` e_2
conditional	`if` e_1 `then` e_2 `else` e_3
loop	`while` e_1 `do` e_2 `done`

sequence e_1 `;` e_2 $\equiv$ `let _ =` e_1 `in` e_2

Annotations

- `assert` $\{p\}$; e
- e $\{p\}$

Examples:

- `assert` $\{x > 0\}$; $1/x$
- x := 0 $\{!x = 0\}$
- `if` $!x > !y$ `then` $!x$ `else` $!y$ $\{\mathit{result} \geq !x \wedge \mathit{result} \geq !y\}$
- x := $!x + 1$ $\{!x > \texttt{old}(!x)\}$

Annotations

- $\texttt{assert}\ \{p\};\ e$
- $e\ \{p\}$

Examples:

- $\texttt{assert}\ \{x > 0\};\ 1/x$
- $x := 0\ \{!x = 0\}$
- $\texttt{if}\ !x > !y\ \texttt{then}\ !x\ \texttt{else}\ !y\ \{result \geq !x \wedge result \geq !y\}$
- $x := !x + 1\ \{!x > \texttt{old}(!x)\}$

Annotations

- $\texttt{assert}\ \{p\};\ e$
- $e\ \{p\}$

Examples:

- $\texttt{assert}\ \{x > 0\};\ 1/x$
- $x := 0\ \{!x = 0\}$
- $\texttt{if}\ !x > !y\ \texttt{then}\ !x\ \texttt{else}\ !y\ \{result \geq !x \wedge result \geq !y\}$
- $x := !x + 1\ \{!x > \texttt{old}(!x)\}$

Annotations

- $\texttt{assert}\ \{p\};\ e$
- $e\ \{p\}$

Examples:

- $\texttt{assert}\ \{x > 0\};\ 1/x$
- $x\ \texttt{:=}\ 0\ \{!x = 0\}$
- $\texttt{if}\ !x > !y\ \texttt{then}\ !x\ \texttt{else}\ !y\ \{\mathit{result} \geq !x \wedge \mathit{result} \geq !y\}$
- $x\ \texttt{:=}\ !x + 1\ \{!x > \texttt{old}(!x)\}$

Annotations (cont'd)

Loop invariant and variant

- `while` e_1 `do` $\{$`invariant` p `variant` $t\}$ e_2 `done`

Example:

```
while !x < N do
  { invariant !x ≤ N variant N − !x }
  x := !x + 1
done
```

Annotations (cont'd)

Loop invariant and variant

- `while` e_1 `do` $\{$`invariant` p `variant` $t\}$ e_2 `done`

Example:

```
while !x < N do
  { invariant !x ≤ N variant N − !x }
  x := !x + 1
done
```

Functions

A function declaration introduces a precondition

- $\texttt{fun}\ (x : \tau) \rightarrow \{p\}\ e$
- $\texttt{rec}\ f\ (x_1 : \tau_1) \ldots (x_n : \tau_n) : \beta\ \{\texttt{variant}\ t\} = \{p\}\ e$

Example:

$$\texttt{fun}\ (x : \texttt{int ref}) \rightarrow \{!x > 0\}\ x := !x - 1\ \{!x \geq 0\}$$

Functions

A function declaration introduces a precondition

- $\texttt{fun}\ (x : \tau) \rightarrow \{p\}\ e$
- $\texttt{rec}\ f\ (x_1 : \tau_1) \ldots (x_n : \tau_n) : \beta\ \{\texttt{variant}\ t\} = \{p\}\ e$

Example:

$$\texttt{fun}\ (x : \texttt{int ref}) \rightarrow \{!x > 0\}\ x := !x - 1\ \{!x \geq 0\}$$

Functions

A function declaration introduces a precondition

- $\texttt{fun}\ (x : \tau) \rightarrow \{p\}\ e$
- $\texttt{rec}\ f\ (x_1 : \tau_1) \ldots (x_n : \tau_n) : \beta\ \{\texttt{variant}\ t\} = \{p\}\ e$

Example:

$$\texttt{fun}\ (x : \texttt{int ref}) \rightarrow \{!x > 0\}\ x\ \texttt{:=}\ !x - 1\ \{!x \geq 0\}$$

Modularity

A function declaration extends the ML function type with a precondition, an effect and a postcondition

$$x : \tau_1 \to \{p\}\, \tau_2\ \texttt{reads}\, x_1, \ldots, x_n\ \texttt{writes}\, y_1, \ldots, y_m\, \{q\}$$

Example:

$$\mathit{swap} : \ x : \texttt{int ref} \to y : \texttt{int ref} \to \{\}\, \texttt{unit writes}\, x, y\, \{\texttt{!}x = \texttt{old(!}y\texttt{)} \wedge \texttt{!}y = \texttt{old(!}x\texttt{)}\}$$

Modularity

A function declaration extends the ML function type with a precondition, an effect and a postcondition

$$x : \tau_1 \to \{p\}\, \tau_2\ \texttt{reads}\, x_1, \ldots, x_n\ \texttt{writes}\, y_1, \ldots, y_m\, \{q\}$$

Example:

$$\mathit{swap} :\ x : \texttt{int ref} \to y : \texttt{int ref} \to \{\}\, \texttt{unit writes}\, x, y\, \{\texttt{!}x = \texttt{old}(\texttt{!}y) \wedge \texttt{!}y = \texttt{old}(\texttt{!}x)\}$$

Auxiliary variables

Used to denote the intermediate values of variables

Example: $\ldots\ \{!x = X\}\ \ldots\ \{!x > X\}\ \ldots$

We will use labels instead

- new construct $L : e$
- new annotation $\mathtt{at}(t, L)$

Example:

$$\begin{array}{l} \vdots \\ L : \mathtt{while}\ \ldots\ \mathtt{do}\ \{\ \mathtt{invariant}\ !x \geq \mathtt{at}(!x, L)\ \ldots\ \} \\ \quad \ldots \\ \mathtt{done} \end{array}$$

Auxiliary variables

Used to denote the intermediate values of variables

Example: $\ldots\ \{!x = X\} \ldots \{!x > X\}\ \ldots$

We will use labels instead

- new construct $L : e$
- new annotation $\texttt{at}(t, L)$

Example:

$$\begin{array}{l} \vdots \\ L : \texttt{while}\ \ldots\ \texttt{do}\ \{\ \texttt{invariant}\ !x \geq \texttt{at}(!x, L)\ \ldots\ \} \\ \quad \ldots \\ \texttt{done} \end{array}$$

Auxiliary variables

Used to denote the intermediate values of variables

Example: $\ldots\ \{!x = X\} \ldots \{!x > X\}\ \ldots$

We will use labels instead

- new construct $L : e$
- new annotation $\mathtt{at}(t, L)$

Example:

$$\begin{array}{l} \vdots \\ L : \mathtt{while}\ \ldots\ \mathtt{do}\ \{\ \mathtt{invariant}\ !x \geq \mathtt{at}(!x, L)\ \ldots\ \} \\ \quad \ldots \\ \mathtt{done} \end{array}$$

Auxiliary variables

Used to denote the intermediate values of variables

Example: $\ldots\ \{!x = X\} \ldots \{!x > X\}\ \ldots$

We will use labels instead

- new construct $L : e$
- new annotation $\texttt{at}(t, L)$

Example:

$$\begin{array}{l} \vdots \\ L : \texttt{while } \ldots \texttt{ do } \{ \texttt{ invariant } !x \geq \texttt{at}(!x, L) \ \ldots \ \} \\ \quad \ldots \\ \texttt{done} \end{array}$$

Exceptions

Finally, we introduce exceptions in our language

- a more realistic ML fragment
- to interpret abrupt statements like `return`, `break` or `continue`

new constructs

- `raise` $(E\ e) : \tau$
- `try` e_1 `with` $E\ x \longrightarrow e_2$ `end`

Exceptions

Finally, we introduce exceptions in our language

- a more realistic ML fragment
- to interpret abrupt statements like `return`, `break` or `continue`

new constructs

- `raise` $(E\ e) : \tau$
- `try` e_1 `with` $E\ x \longrightarrow e_2$ `end`

Exceptions

Finally, we introduce exceptions in our language

- a more realistic ML fragment
- to interpret abrupt statements like `return`, `break` or `continue`

new constructs

- `raise` $(E\ e) : \tau$
- `try` e_1 `with` $E\ x \longrightarrow e_2$ `end`

Exceptions

The notion of postcondition is extended

$$\texttt{if } x < 0 \texttt{ then raise Negative else } \mathit{sqrt}\ x$$
$$\{\ \mathit{result} \geq 0 \mid \text{Negative} \Rightarrow x < 0\ \}$$

So is the notion of effect

$$\mathit{div} : x : \texttt{int} \to y : \texttt{int} \to \{\ldots\}\,\texttt{int raises Negative}\,\{\ldots\}$$

Exceptions

The notion of postcondition is extended

$$\texttt{if } x < 0 \texttt{ then raise Negative else } \mathit{sqrt}\ x$$
$$\{\ \mathit{result} \geq 0 \mid \text{Negative} \Rightarrow x < 0\ \}$$

So is the notion of effect

$$\mathit{div} : x : \texttt{int} \to y : \texttt{int} \to \{\ldots\}\,\texttt{int raises Negative}\,\{\ldots\}$$

Loops and exceptions

We can replace the `while` loop by an infinite loop

- `loop` e `{invariant` p `variant` t`}`

and simulate the `while` loop using an exception

```
while e1 do {invariant p variant t} e2 done ≡
  try
    loop if e1 then e2 else raise Exit
    {invariant p variant t}
  with Exit _ -> void end
```

simpler constructs ⇒ simpler typing and proof rules

Loops and exceptions

We can replace the `while` loop by an infinite loop

- `loop` e `{invariant` p `variant` t`}`

and simulate the `while` loop using an exception

```
while e1 do {invariant p variant t} e2 done ≡
  try
    loop if e1 then e2 else raise Exit
    {invariant p variant t}
  with Exit _ -> void end
```

simpler constructs ⇒ simpler typing and proof rules

Loops and exceptions

We can replace the `while` loop by an infinite loop

- `loop` e `{invariant` p `variant` t`}`

and simulate the `while` loop using an exception

```
while e1 do {invariant p variant t} e2 done ≡
  try
    loop if e1 then e2 else raise Exit
    {invariant p variant t}
  with Exit _ -> void end
```

simpler constructs $\Rightarrow$ simpler typing and proof rules

Summary

Types

$$
\begin{array}{lcl}
\tau & ::= & \beta \mid \beta \text{ ref} \mid (x : \tau) \rightarrow \kappa \\
\kappa & ::= & \{p\}\, \tau\, \epsilon\, \{q\} \\
q & ::= & p; E \Rightarrow p; \ldots; E \Rightarrow p \\
\epsilon & ::= & \texttt{reads } x, \ldots, x \texttt{ writes } x, \ldots, x \texttt{ raises } E, \ldots, E
\end{array}
$$

Annotations

$$
\begin{array}{lcl}
t & ::= & c \mid x \mid \texttt{!}x \mid \phi(t, \ldots, t) \mid \texttt{old}(t) \mid \texttt{at}(t, L) \\
p & ::= & \text{True} \mid \text{False} \mid P(t, \ldots, t) \\
 & \mid & p \Rightarrow p \mid p \wedge p \mid p \vee p \mid \neg\, p \mid \forall x : \beta.p \mid \exists x : \beta.p
\end{array}
$$

Summary

Types

$$
\begin{array}{lcl}
\tau & ::= & \beta \mid \beta \text{ ref} \mid (x : \tau) \rightarrow \kappa \\
\kappa & ::= & \{p\}\, \tau\, \epsilon\, \{q\} \\
q & ::= & p; E \Rightarrow p; \ldots; E \Rightarrow p \\
\epsilon & ::= & \texttt{reads } x, \ldots, x \texttt{ writes } x, \ldots, x \texttt{ raises } E, \ldots, E
\end{array}
$$

Annotations

$$
\begin{array}{lcl}
t & ::= & c \mid x \mid \,!x \mid \phi(t, \ldots, t) \mid \texttt{old}(t) \mid \texttt{at}(t, L) \\
p & ::= & \text{True} \mid \text{False} \mid P(t, \ldots, t) \\
 & \mid & p \Rightarrow p \mid p \wedge p \mid p \vee p \mid \neg\, p \mid \forall x : \beta.p \mid \exists x : \beta.p
\end{array}
$$

Programs

$$
\begin{array}{lcl}
u & ::= & c \mid x \mid \,!x \mid \phi(u, \ldots, u) \\
e & ::= & u \\
& \mid & x := e \\
& \mid & \texttt{let } x = e \texttt{ in } e \\
& \mid & \texttt{let } x = \texttt{ref } e \texttt{ in } e \\
& \mid & \texttt{if } e \texttt{ then } e \texttt{ else } e \\
& \mid & \texttt{loop } e \,\{\texttt{invariant } p \texttt{ variant } t\} \\
& \mid & L : e \\
& \mid & \texttt{raise } (E\ e) : \tau \\
& \mid & \texttt{try } e \texttt{ with } E\ x \rightarrow e \texttt{ end} \\
& \mid & \texttt{assert } \{p\};\ e \\
& \mid & e\ \{q\} \\
& \mid & \texttt{fun } (x : \tau) \rightarrow \{p\}\ e \\
& \mid & \texttt{rec } x\ (x : \tau) \ldots (x : \tau) : \beta\ \{\texttt{variant } t\} = \{p\}\ e \\
& \mid & e\ e
\end{array}
$$

Typing

A typing judgment

$$\Gamma \vdash e : (\tau, \epsilon)$$

Rules given in the notes (page 24)

The main purpose is to exclude aliases
In particular, references can't escape their scopes

Typing

A typing judgment

$$\Gamma \vdash e : (\tau, \epsilon)$$

Rules given in the notes (page 24)

The main purpose is to exclude aliases
In particular, references can't escape their scopes

Semantics

Call-by-value semantics, with left to right evalutation

Big-step operational semantics given in the notes (page 26)

Part II

Proof rules

Weakest preconditions

We define the predicate $wp(e, q)$, called the weakest precondition for program e and postcondition q

Property: If $wp(e, q)$ holds, then e terminates and q holds at the end of execution (and all inner annotations are verified)

The converse holds for the fragment without loops and functions

The correctness of an annotated program e is thus $wp(e, \mathrm{True})$

Weakest preconditions

We define the predicate $wp(e, q)$, called the weakest precondition for program e and postcondition q

Property: If $wp(e, q)$ holds, then e terminates and q holds at the end of execution (and all inner annotations are verified)

The converse holds for the fragment without loops and functions

The correctness of an annotated program e is thus $wp(e, \mathrm{True})$

Weakest preconditions

We define the predicate $wp(e, q)$, called the weakest precondition for program e and postcondition q

Property: If $wp(e, q)$ holds, then e terminates and q holds at the end of execution (and all inner annotations are verified)

The converse holds for the fragment without loops and functions

The correctness of an annotated program e is thus $wp(e, \mathrm{True})$

Weakest preconditions

We define the predicate $wp(e, q)$, called the weakest precondition for program e and postcondition q

Property: If $wp(e, q)$ holds, then e terminates and q holds at the end of execution (and all inner annotations are verified)

The converse holds for the fragment without loops and functions

The correctness of an annotated program e is thus $wp(e, \mathrm{True})$

Definition of $wp(e, q)$

We actually define $wp(e, q; r)$ where

- q is the "normal" postcondition
- $r \equiv E_1 \Rightarrow q_1; \ldots; E_n \Rightarrow q_n$ is the set of "exceptional" post.

Basic constructs

$$wp(u, q; r) = q[result \leftarrow u]$$

$$wp(x := e, q; r) = wp(e, q[result \leftarrow \texttt{void}; x \leftarrow result]; r)$$

$$wp(\texttt{let } x = e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[x \leftarrow result]; r)$$

$$wp(\texttt{let } x = \texttt{ref } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[!x \leftarrow result]; r)$$

$$wp(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, q; r) = wp(e_1, \texttt{if } result \texttt{ then } wp(e_2, q; r) \texttt{ else } wp(e_3, q; r); r)$$

$$wp(L : e, q; r) = wp(e, q; r)[\texttt{at}(x, L) \leftarrow x]$$

Basic constructs

$$wp(u, q; r) = q[result \leftarrow u]$$

$$wp(x := e, q; r) = wp(e, q[result \leftarrow \texttt{void}; x \leftarrow result]; r)$$

$$wp(\texttt{let } x = e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[x \leftarrow result]; r)$$

$$wp(\texttt{let } x = \texttt{ref } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[!x \leftarrow result]; r)$$

$$wp(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, q; r) = wp(e_1, \texttt{if } result \texttt{ then } wp(e_2, q; r) \texttt{ else } wp(e_3, q; r); r)$$

$$wp(L : e, q; r) = wp(e, q; r)[\texttt{at}(x, L) \leftarrow x]$$

Basic constructs

$$wp(u, q; r) = q[result \leftarrow u]$$

$$wp(x \texttt{ := } e, q; r) = wp(e, q[result \leftarrow \texttt{void}; x \leftarrow result]; r)$$

$$wp(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[x \leftarrow result]; r)$$

$$wp(\texttt{let } x \texttt{ = ref } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[\texttt{!}x \leftarrow result]; r)$$

$$wp(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, q; r) = wp(e_1, \texttt{if } result \texttt{ then } wp(e_2, q; r) \texttt{ else } wp(e_3, q; r); r)$$

$$wp(L : e, q; r) = wp(e, q; r)[\texttt{at}(x, L) \leftarrow x]$$

Basic constructs

$$wp(u, q; r) = q[result \leftarrow u]$$

$$wp(x \texttt{ := } e, q; r) = wp(e, q[result \leftarrow \texttt{void}; x \leftarrow result]; r)$$

$$wp(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[x \leftarrow result]; r)$$

$$wp(\texttt{let } x \texttt{ = ref } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[\texttt{!}x \leftarrow result]; r)$$

$$wp(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, q; r) = wp(e_1, \texttt{if } result \texttt{ then } wp(e_2, q; r) \texttt{ else } wp(e_3, q; r); r)$$

$$wp(L : e, q; r) = wp(e, q; r)[\texttt{at}(x, L) \leftarrow x]$$

Basic constructs

$$wp(u, q; r) = q[result \leftarrow u]$$

$$wp(x \texttt{ := } e, q; r) = wp(e, q[result \leftarrow \texttt{void}; x \leftarrow result]; r)$$

$$wp(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[x \leftarrow result]; r)$$

$$wp(\texttt{let } x \texttt{ = ref } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[\texttt{!}x \leftarrow result]; r)$$

$$wp(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, q; r) = wp(e_1, \texttt{if } result \texttt{ then } wp(e_2, q; r) \texttt{ else } wp(e_3, q; r); r)$$

$$wp(L : e, q; r) = wp(e, q; r)[\texttt{at}(x, L) \leftarrow x]$$

Basic constructs

$$wp(u, q; r) = q[result \leftarrow u]$$

$$wp(x \texttt{ := } e, q; r) = wp(e, q[result \leftarrow \texttt{void}; x \leftarrow result]; r)$$

$$wp(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[x \leftarrow result]; r)$$

$$wp(\texttt{let } x \texttt{ = ref } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[\texttt{!}x \leftarrow result]; r)$$

$$wp(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, q; r) = wp(e_1, \texttt{if } result \texttt{ then } wp(e_2, q; r) \texttt{ else } wp(e_3, q; r); r)$$

$$wp(L : e, q; r) = wp(e, q; r)[\texttt{at}(x, L) \leftarrow x]$$

Traditional rules

Assignment of a side-effects free expression

$$wp(x := u, q) = q[x \leftarrow u]$$

Exception-free sequence

$$wp(e_1 ; e_2, q) = wp(e_1, wp(e_2, q))$$

Traditional rules

Assignment of a side-effects free expression

$$wp(x := u, q) = q[x \leftarrow u]$$

Exception-free sequence

$$wp(e_1 ; e_2, q) = wp(e_1, wp(e_2, q))$$

Exceptions

$$wp(\texttt{raise}\ (E\ e) : \tau, q; r) = wp(e, r(E); r)$$

$$wp(\texttt{try}\ e_1\ \texttt{with}\ E\ x \rightarrow e_2\ \texttt{end}, q; r) = wp(e_1, q; E \Rightarrow wp(e_2, q; r)[x \leftarrow result]; r)$$

Exceptions

$$wp(\texttt{raise}\ (E\ e) : \tau, q; r) = wp(e, r(E); r)$$

$$wp(\texttt{try}\ e_1\ \texttt{with}\ E\ x \rightarrow e_2\ \texttt{end}, q; r) = \\ wp(e_1, q; E \Rightarrow wp(e_2, q; r)[x \leftarrow \mathit{result}]; r)$$

Annotations

$$wp(\texttt{assert } \{p\};\ e, q; r) \quad = p \wedge wp(e, q; r)$$

$$wp(e\ \{q'; r'\}, q; r) \quad = wp(e, q' \wedge q; r' \wedge r)$$

Annotations

$$wp(\texttt{assert } \{p\};\ e, q; r) \quad = p \wedge wp(e, q; r)$$

$$wp(e\ \{q'; r'\}, q; r) \quad = wp(e, q' \wedge q; r' \wedge r)$$

Loops

$$wp(\texttt{loop}\ e\ \{\texttt{invariant}\ p\ \texttt{variant}\ t\}, q; r) =$$
$$p\ \wedge\ \forall \omega.\ p \Rightarrow wp(L\texttt{:}e, p \wedge t < \texttt{at}(t, L); r)$$

where $\omega =$ the variables (possibly) modified by e

Usual while loop

$$wp(\texttt{while}\ e_1\ \texttt{do}\ \{\texttt{invariant}\ p\ \texttt{variant}\ t\}\ e_2\ \texttt{done}, q; r)$$
$$= p\ \wedge\ \forall \omega.\ p \Rightarrow$$
$$wp(L\texttt{:if}\ e_1\ \texttt{then}\ e_2\ \texttt{else raise}\ E, p \wedge t < \texttt{at}(t, L), E \Rightarrow q; r)$$
$$= p\ \wedge\ \forall \omega.\ p \Rightarrow$$
$$wp(e_1, \texttt{if}\ \mathit{result}\ \texttt{then}\ wp(e_2, p \wedge t < \texttt{at}(t, L))\ \texttt{else}\ q, r)[\texttt{at}(x, L) \leftarrow x]$$

Loops

$$wp(\texttt{loop } e\ \{\texttt{invariant } p\ \texttt{variant } t\}, q; r) =$$
$$p\ \wedge\ \forall \omega.\ p \Rightarrow wp(L\texttt{:}e, p \wedge t < \texttt{at}(t, L); r)$$

where ω = the variables (possibly) modified by e

Usual while loop

$$wp(\texttt{while } e_1 \texttt{ do } \{\texttt{invariant } p\ \texttt{variant } t\}\ e_2\ \texttt{done}, q; r)$$
$$= p\ \wedge\ \forall \omega.\ p \Rightarrow$$
$$wp(L\texttt{:if } e_1 \texttt{ then } e_2 \texttt{ else raise } E, p \wedge t < \texttt{at}(t, L), E \Rightarrow q; r)$$
$$= p\ \wedge\ \forall \omega.\ p \Rightarrow$$
$$wp(e_1, \texttt{if } \mathit{result} \texttt{ then } wp(e_2, p \wedge t < \texttt{at}(t, L)) \texttt{ else } q, r)[\texttt{at}(x, L) \leftarrow x]$$

Loops

$$wp(\texttt{loop } e\ \{\texttt{invariant } p\ \texttt{variant } t\}, q; r) =$$
$$p\ \wedge\ \forall \omega.\ p \Rightarrow wp(L\texttt{:}e, p \wedge t < \texttt{at}(t, L); r)$$

where ω = the variables (possibly) modified by e

Usual while loop

$$wp(\texttt{while } e_1 \texttt{ do } \{\texttt{invariant } p\ \texttt{variant } t\}\ e_2\ \texttt{done}, q; r)$$
$$= p\ \wedge\ \forall \omega.\ p \Rightarrow$$
$$wp(L\texttt{:if } e_1 \texttt{ then } e_2 \texttt{ else raise } E, p \wedge t < \texttt{at}(t, L), E \Rightarrow q; r)$$
$$= p\ \wedge\ \forall \omega.\ p \Rightarrow$$
$$wp(e_1, \texttt{if } \mathit{result} \texttt{ then } wp(e_2, p \wedge t < \texttt{at}(t, L))\ \texttt{else}\ q, r)[\texttt{at}(x, L) \leftarrow x]$$

Loops

$$wp(\texttt{loop } e\ \{\texttt{invariant } p\ \texttt{variant } t\}, q; r) =$$
$$p\ \wedge\ \forall \omega.\ p \Rightarrow wp(L\texttt{:}e, p \wedge t < \texttt{at}(t, L); r)$$

where ω = the variables (possibly) modified by e

Usual while loop

$$wp(\texttt{while } e_1 \texttt{ do } \{\texttt{invariant } p\ \texttt{variant } t\}\ e_2\ \texttt{done}, q; r)$$
$$= p\ \wedge\ \forall \omega.\ p \Rightarrow$$
$$wp(L\texttt{:if } e_1 \texttt{ then } e_2 \texttt{ else raise } E, p \wedge t < \texttt{at}(t, L), E \Rightarrow q; r)$$
$$= p\ \wedge\ \forall \omega.\ p \Rightarrow$$
$$wp(e_1, \texttt{if } \mathit{result} \texttt{ then } wp(e_2, p \wedge t < \texttt{at}(t, L))\ \texttt{else}\ q, r)[\texttt{at}(x, L) \leftarrow x]$$

Functions

$$wp(\texttt{fun}\ (x:\tau) \rightarrow \{p\}\ e, q; r) \quad = q \ \wedge\ \forall x.\forall \rho.p \Rightarrow wp(e, \mathsf{True})$$

$$wp(\texttt{rec}\ f\ (x_1:\tau_1)\ldots(x_n:\tau_n):\tau\ \{\texttt{variant}\ t\} = \{p\}\ e, q; r)$$
$$= q \ \wedge\ \forall x_1.\ldots.\forall x_n.\forall \rho.p \Rightarrow wp(L:e, \mathsf{True})$$

when computing $wp(L:e, \mathsf{True})$, f is assumed to have type

$$(x_1:\tau_1) \rightarrow \cdots \rightarrow (x_n:\tau_n) \rightarrow \{p \wedge t < \mathtt{at}(t, L)\}\,\tau\ \epsilon\,\{q\}$$

Functions

$$wp(\texttt{fun}\ (x:\tau) \to \{p\}\ e, q; r) \quad = q\ \wedge\ \forall x.\forall \rho.p \Rightarrow wp(e, \mathsf{True})$$

$$wp(\texttt{rec}\ f\ (x_1:\tau_1)\ldots(x_n:\tau_n):\tau\ \{\texttt{variant}\ t\} = \{p\}\ e, q; r)$$
$$= q\ \wedge\ \forall x_1.\ldots.\forall x_n.\forall \rho.p \Rightarrow wp(L\!:\!e, \mathsf{True})$$

when computing $wp(L\!:\!e, \mathsf{True})$, f is assumed to have type

$$(x_1:\tau_1) \to \cdots \to (x_n:\tau_n) \to \{p \wedge t < \mathtt{at}(t, L)\}\,\tau\ \epsilon\,\{q\}$$

Function call

Simplified using

$$e_1\ e_2 \equiv \texttt{let}\ x_1 = e_1\ \texttt{in let}\ x_2 = e_2\ \texttt{in}\ x_1\ x_2$$

Assuming

$$x_1\ :\ (x:\tau) \rightarrow \{p'\}\,\tau'\ \epsilon\,\{q'\}$$

we define

$$wp(x_1\ x_2, q) = p'[x \leftarrow x_2] \wedge \forall\omega.\forall result.(q'[x \leftarrow x_2] \Rightarrow q)[\texttt{old}(t) \leftarrow t]$$

Function call

Simplified using

$$e_1\ e_2 \equiv \texttt{let}\ x_1 = e_1\ \texttt{in let}\ x_2 = e_2\ \texttt{in}\ x_1\ x_2$$

Assuming

$$x_1\ :\ (x:\tau) \rightarrow \{p'\}\,\tau'\ \epsilon\,\{q'\}$$

we define

$$wp(x_1\ x_2, q) = p'[x \leftarrow x_2] \wedge \forall\omega.\forall result.(q'[x \leftarrow x_2] \Rightarrow q)[\texttt{old}(t) \leftarrow t]$$

Part III

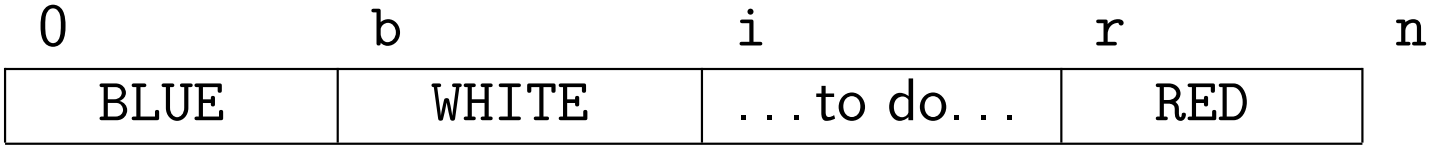
The WHY tool

Dijkstra's Dutch national flag

Goal: to sort an array where elements only have three different values (blue, white and red)

Dijkstra's Dutch national flag

Goal: to sort an array where elements only have three different values (blue, white and red)



A few lines of C code

```
typedef enum { BLUE, WHITE, RED } color;

void swap(int t[], int i, int j) {
  color c = t[i]; t[i] = t[j]; t[j] = c;
}

void flag(int t[], int n) {
  int b = 0, i = 0, r = n;
  while (i < r) {
    switch (t[i]) {
    case BLUE: swap(t, b++, i++); break;
    case WHITE: i++; break;
    case RED: swap(t, --r, i); break;
    }
  }
}
```

Modelization

We are not verifying the C code, but rather the algorithm

We model

- colors with an abstract datetype
- arrays using references containing functional arrays

Modelization

We are not verifying the C code, but rather the algorithm

We model

- colors with an abstract datetype
- arrays using references containing functional arrays

An abstract type for colors

```
type color

logic blue : color
logic white : color
logic red : color

predicate is_color(c:color) = c=blue or c=white or c=red

parameter eq_color  :
  c1:color → c2:color →
    {} bool { if result then c1=c2 else c1≠c2 }
```

Functional arrays

```
type color_array

logic acc : color_array, int → color
logic upd : color_array, int, color → color_array

axiom acc_upd_eq :
  ∀a:color_array. ∀i:int. ∀c:color.
    acc(upd(a,i,c),i) = c

axiom acc_upd_neq :
  ∀a:color_array. ∀i,j:int. ∀c:color.
    i ≠ j → acc(upd(a,j,c),i) = acc(a,i)
```

Array bounds

```
logic length : color_array → int

axiom length_update :
  ∀a:color_array. ∀i:int. ∀c:color.
    length(upd(a,i,c)) = length(a)

parameter get :
  t:color_array ref → i:int →
    { 0<=i<length(t) } color reads t { result=acc(t,i) }

parameter set :
  t:color_array ref → i:int → c:color →
    { 0<=i<length(t) } unit writes t { t=upd(t@,i,c) }
```

Array bounds

```
logic length : color_array → int

axiom length_update :
  ∀a:color_array. ∀i:int. ∀c:color.
    length(upd(a,i,c)) = length(a)

parameter get :
  t:color_array ref → i:int →
    { 0<=i<length(t) } color reads t { result=acc(t,i) }

parameter set :
  t:color_array ref → i:int → c:color →
    { 0<=i<length(t) } unit writes t { t=upd(t@,i,c) }
```

The swap function

```
let swap (t:color_array ref) (i:int) (j:int) =
  { 0 <= i < length(t) and 0 <= j < length(t) }
  let u = get t i in
  set t i (get t j);
  set t j u
  { t = upd(upd(t@,i,acc(t@,j)), j, acc(t@,i)) }
```

5 proofs obligations

- 3 automatically discharged by WHY
- 2 left to the user (and automatically discharged by Simplify)

The swap function

```
let swap (t:color_array ref) (i:int) (j:int) =
  { 0 <= i < length(t) and 0 <= j < length(t) }
  let u = get t i in
  set t i (get t j);
  set t j u
  { t = upd(upd(t@,i,acc(t@,j)), j, acc(t@,i)) }
```

5 proofs obligations

- 3 automatically discharged by WHY
- 2 left to the user (and automatically discharged by Simplify)

Function code

```
let dutch_flag (t:color_array ref) (n:int) =
  let b = ref 0 in
  let i = ref 0 in
  let r = ref n in
  while !i < !r do
     if eq_color (get t !i) blue then begin
       swap t !b !i;
       b := !b + 1;
       i := !i + 1
     end else if eq_color (get t !i) white then
       i := !i + 1
     else begin
       r := !r - 1;
       swap t !r !i
     end
  done
```

Function specification

```
predicate monochrome(t:color_array,i:int,j:int,c:color) =
  ∀k:int. i<=k<j → acc(t,k)=c

let dutch_flag (t:color_array ref) (n:int) =
  { 0 <= n and length(t) = n and
    ∀k:int. 0 <= k < n → is_color(acc(t,k)) }
  ⋮
  { ∃b:int. ∃r:int.
      monochrome(t,0,b,blue) and
      monochrome(t,b,r,white) and
      monochrome(t,r,n,red) }
```

Function specification

```
predicate monochrome(t:color_array,i:int,j:int,c:color) =
  ∀k:int. i<=k<j → acc(t,k)=c

let dutch_flag (t:color_array ref) (n:int) =
  { 0 <= n and length(t) = n and
    ∀k:int. 0 <= k < n → is_color(acc(t,k)) }
  ⋮
  { ∃b:int. ∃r:int.
      monochrome(t,0,b,blue) and
      monochrome(t,b,r,white) and
      monochrome(t,r,n,red) }
```

Loop invariant

```
  ⋮
while !i < !r do
   { invariant 0 <= b <= i and i <= r <= n and
               monochrome(t,0,b,blue) and
               monochrome(t,b,i,white) and
               monochrome(t,r,n,red) and
               length(t) = n and
               ∀k:int. 0 <= k < n → is_color(acc(t,k))
     variant r - i }
   ⋮
done
```

Proof obligations

11 proof obligations

- loop invariant holds initially
- loop invariant is preserved and variant decreases (3 cases)
- `swap` precondition (twice)
- array access within bounds (twice)
- postcondition holds at the end of function execution

All automatically discharged by Simplify!

Note: to be exhaustive, one has to show that the set of elements was not changed

Proof obligations

11 proof obligations

- loop invariant holds initially
- loop invariant is preserved and variant decreases (3 cases)
- `swap` precondition (twice)
- array access within bounds (twice)
- postcondition holds at the end of function execution

All automatically discharged by Simplify!

Note: to be exhaustive, one has to show that the set of elements was not changed

Proof obligations

11 proof obligations

- loop invariant holds initially
- loop invariant is preserved and variant decreases (3 cases)
- `swap` precondition (twice)
- array access within bounds (twice)
- postcondition holds at the end of function execution

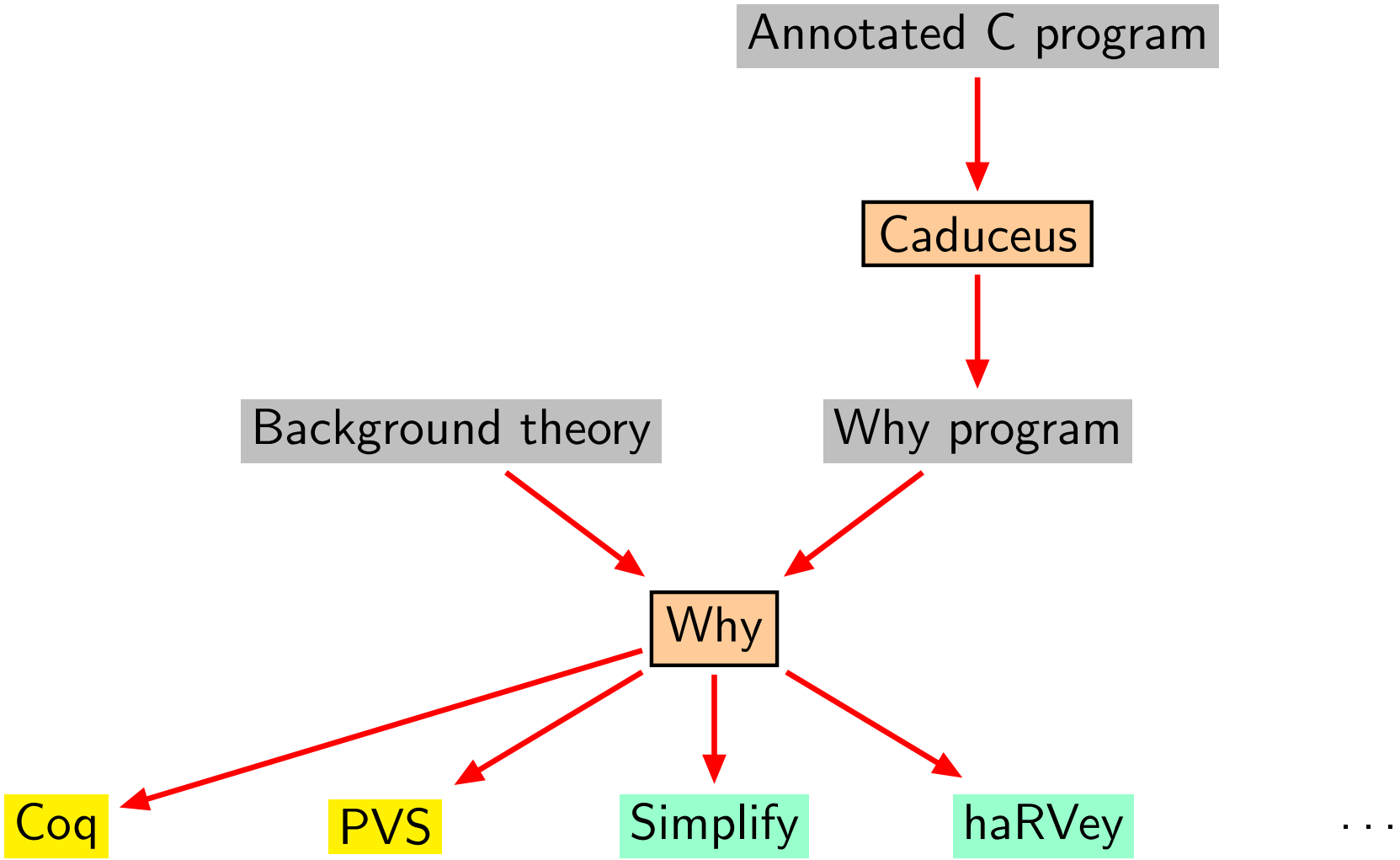
All automatically discharged by Simplify!

Note: to be exhaustive, one has to show that the set of elements was not changed

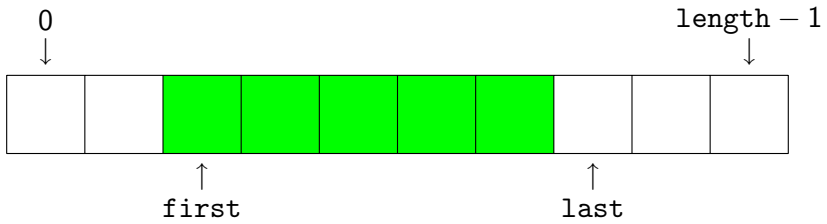
Part IV

Verification of C programs

Overview

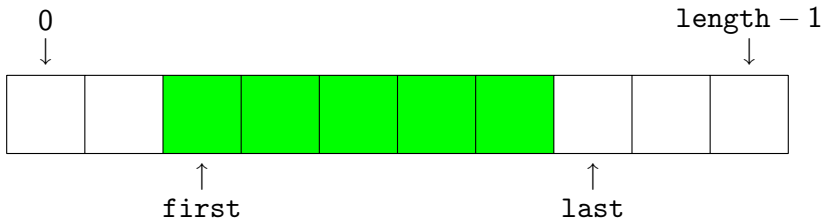

Annotated C program
Caduceus
Background theory
Why program
Why
Coq
PVS
Simplify
haRVey
...

Example: character queue as circular array




```
struct queue {
  char contents[];
  int length;
  int first, last;
  unsigned int empty, full :1;
} q;
/*@ invariant q_invariant :
  @   \valid_range(q.contents, 0, q.length-1) &&
  @   0 <= q.first < q.length &&
  @   0 <= q.last < q.length
  @*/
```

Example: character queue as circular array




```
struct queue {
  char contents[];
  int length;
  int first, last;
  unsigned int empty, full :1;
} q;
/*@ invariant q_invariant :
  @   \valid_range(q.contents, 0, q.length-1) &&
  @   0 <= q.first < q.length &&
  @   0 <= q.last < q.length
  @*/
```

Example continued: specifying functions

```
/*@ requires !q.full
  @ assigns  q.empty, q.full, q.last, q.contents[q.last]
  @ ensures  !q.empty && q.contents[\old(q.last)] == c
  @*/
void push(char c);

/*@ requires !q.empty
  @ assigns q.empty, q.full, q.first
  @ ensures !q.full && \result == q.contents[\old(q.first)]
  @*/
char pop();
```

Example continued: body for push function

```
/*@ requires !q.full
  @ assigns  q.empty, q.full, q.last, q.contents[q.last]
  @ ensures  !q.empty && q.contents[\old(q.last)] == c
  @*/
void push(char c) {
  q.contents[q.last++] = c;      // insert 'c' in the queue
  if (q.last == q.length)
        q.last = 0;              // wrap if needed
  q.empty = 0;                   // queue is not empty
  q.full = (q.first == q.last); // queue is full if
                                 // 'last' reaches 'first'
}
```

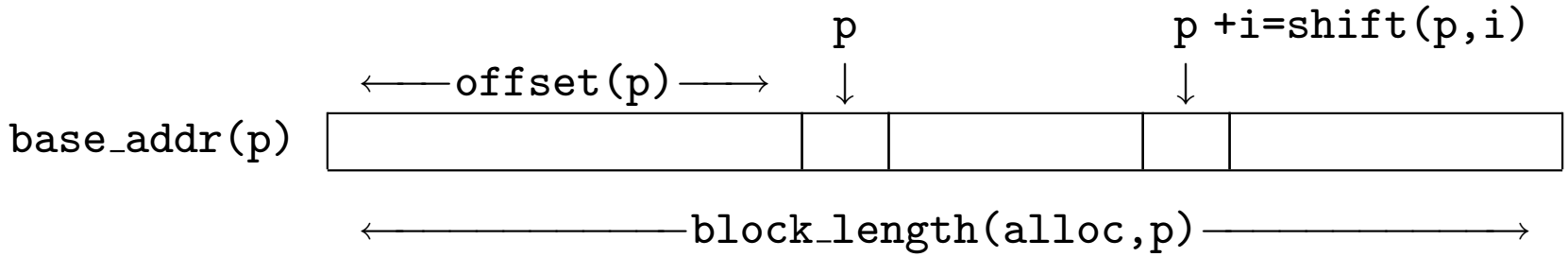
Modeling C memory heap

- Burstall-Bornat model: memory partition according to structure fields
- We extend this idea to handle C arrays and pointer arithmetic: a memory block is

- Each structure field is a map from addresses to memory blocks

Modeling C memory heap

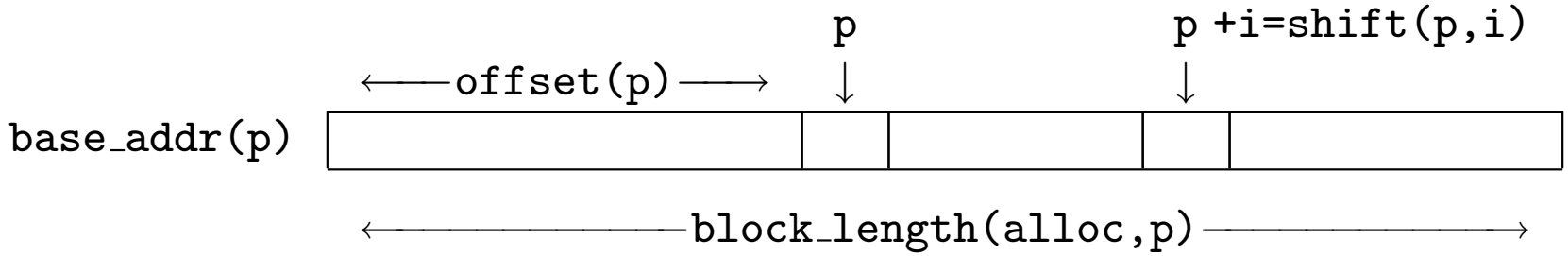
- Burstall-Bornat model: memory partition according to structure fields
- We extend this idea to handle C arrays and pointer arithmetic: a memory block is



- Each structure field is a map from addresses to memory blocks

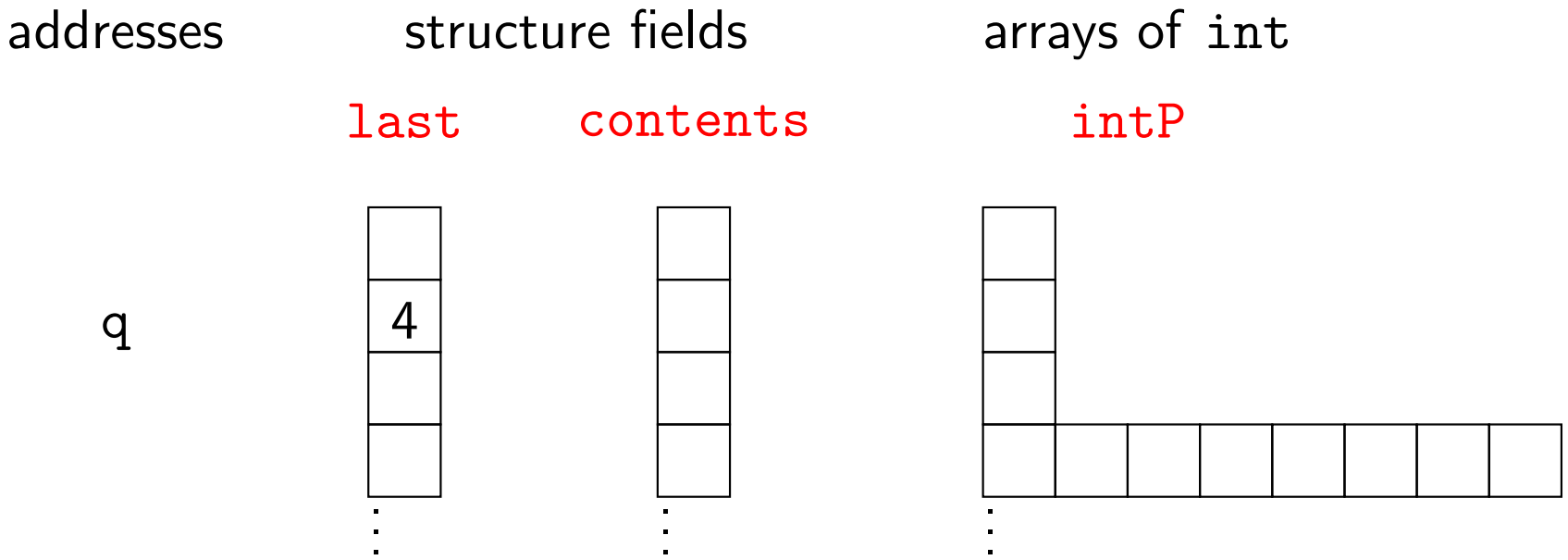
Modeling C memory heap

- Burstall-Bornat model: memory partition according to structure fields
- We extend this idea to handle C arrays and pointer arithmetic: a memory block is



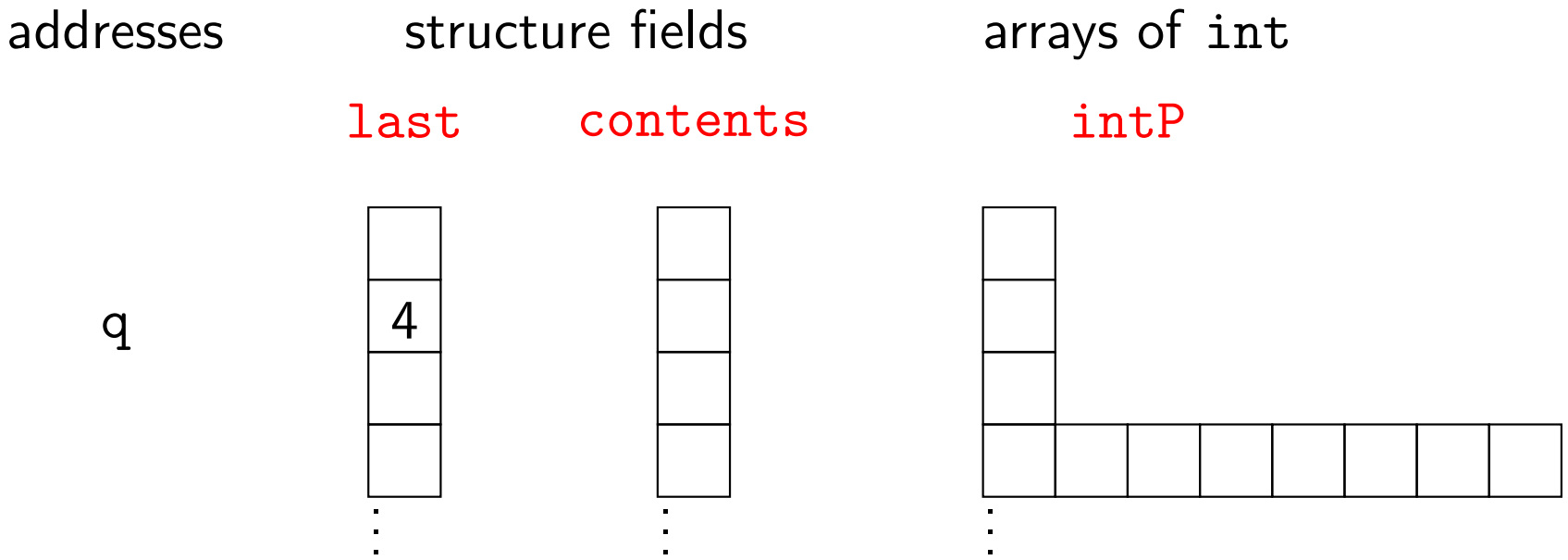
- Each structure field is a map from addresses to memory blocks

Burstall-Bornat model: example of character queue



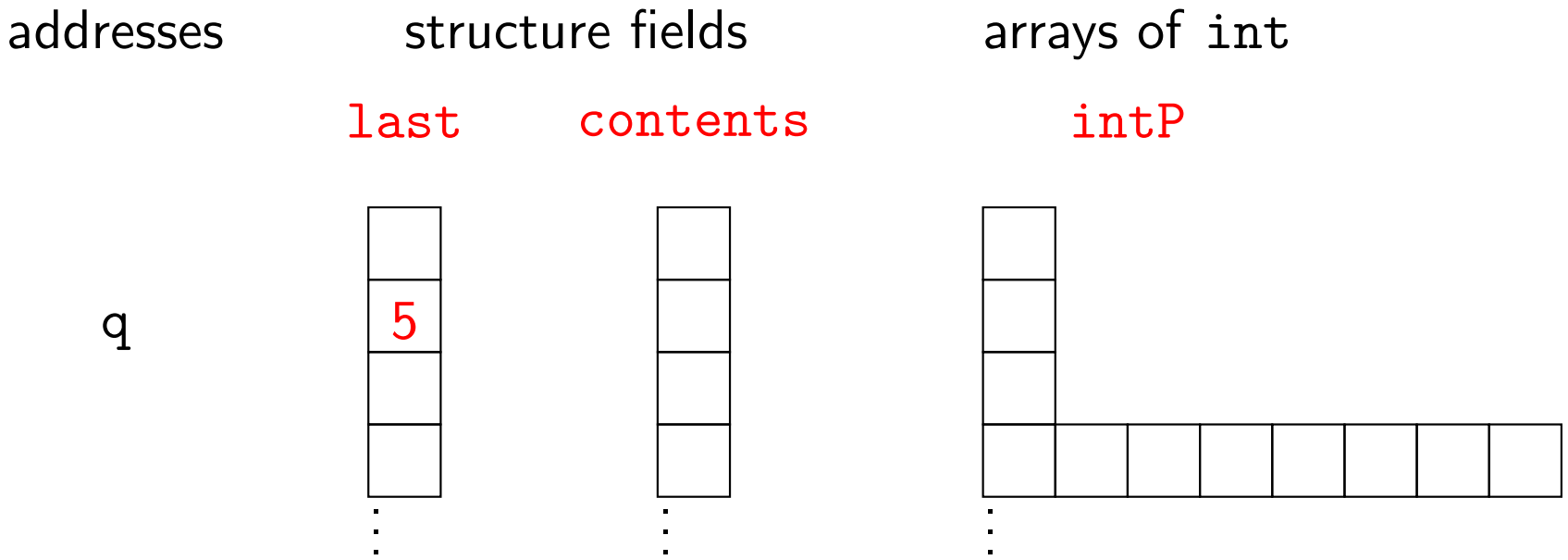

```
q.contents[q.last++] = c
    q.last <- 4 + 1
    *(q.contents+4) <- c
```

Burstall-Bornat model: example of character queue



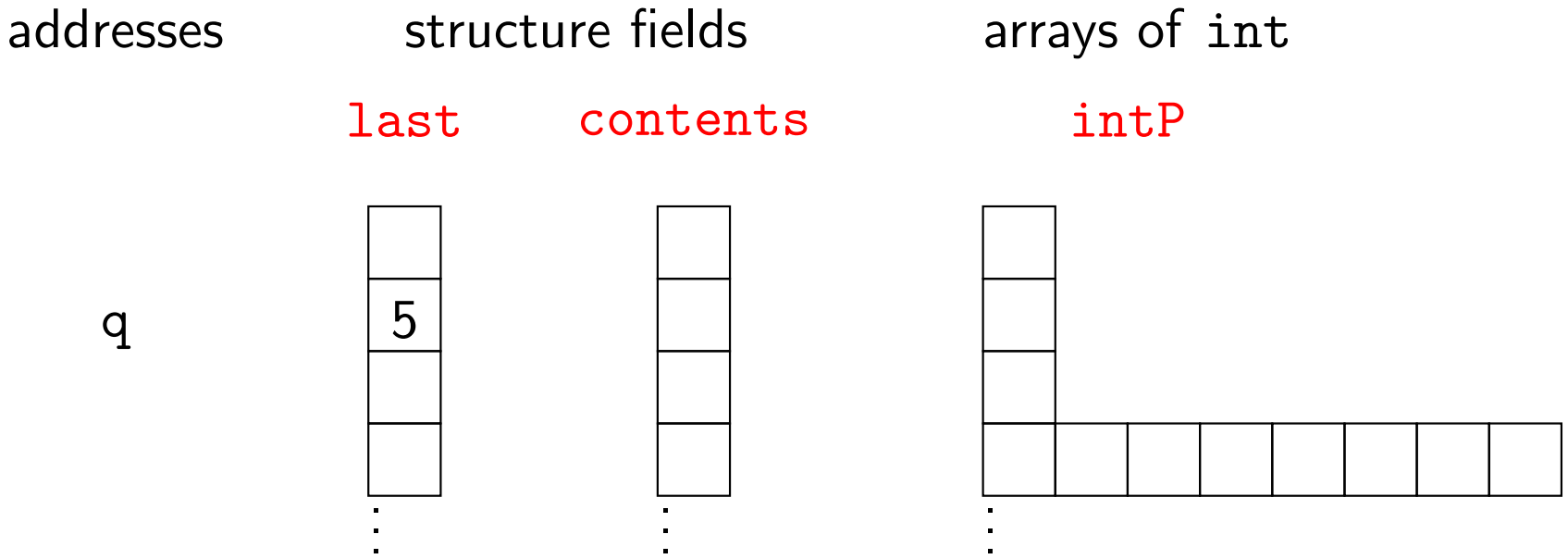

```
q.contents[q.last++] = c
    q.last <- 4 + 1
    *(q.contents+4) <- c
```

Burstall-Bornat model: example of character queue



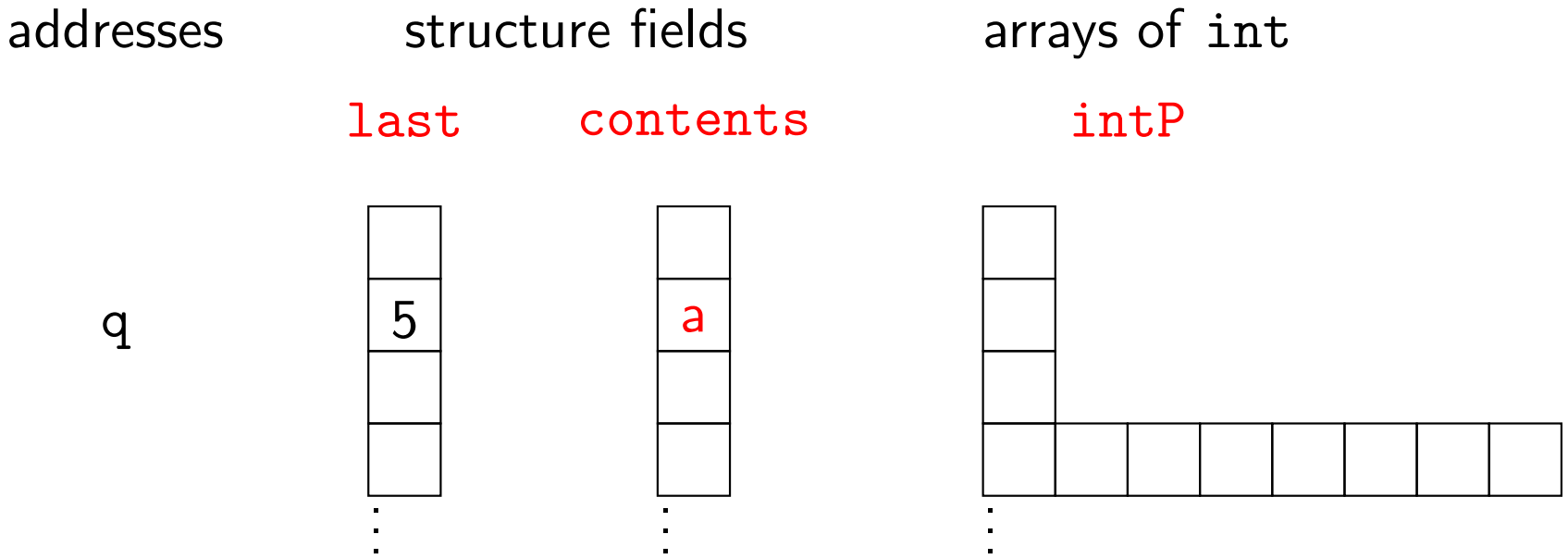

```
q.contents[q.last++] = c
    q.last <- 4 + 1
    *(q.contents+4) <- c
```

Burstall-Bornat model: example of character queue




```
q.contents[q.last++] = c
    q.last <- 4 + 1
    *(q.contents+4) <- c
```

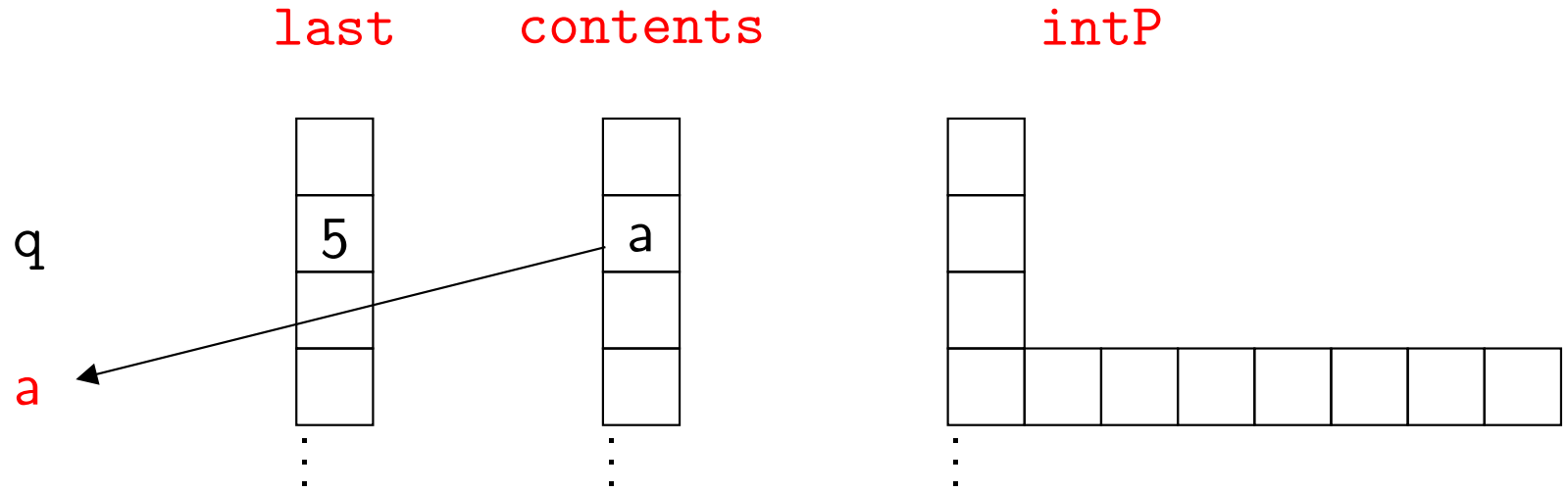
Burstall-Bornat model: example of character queue




```
q.contents[q.last++] = c
    q.last <- 4 + 1
    *(q.contents+4) <- c
```

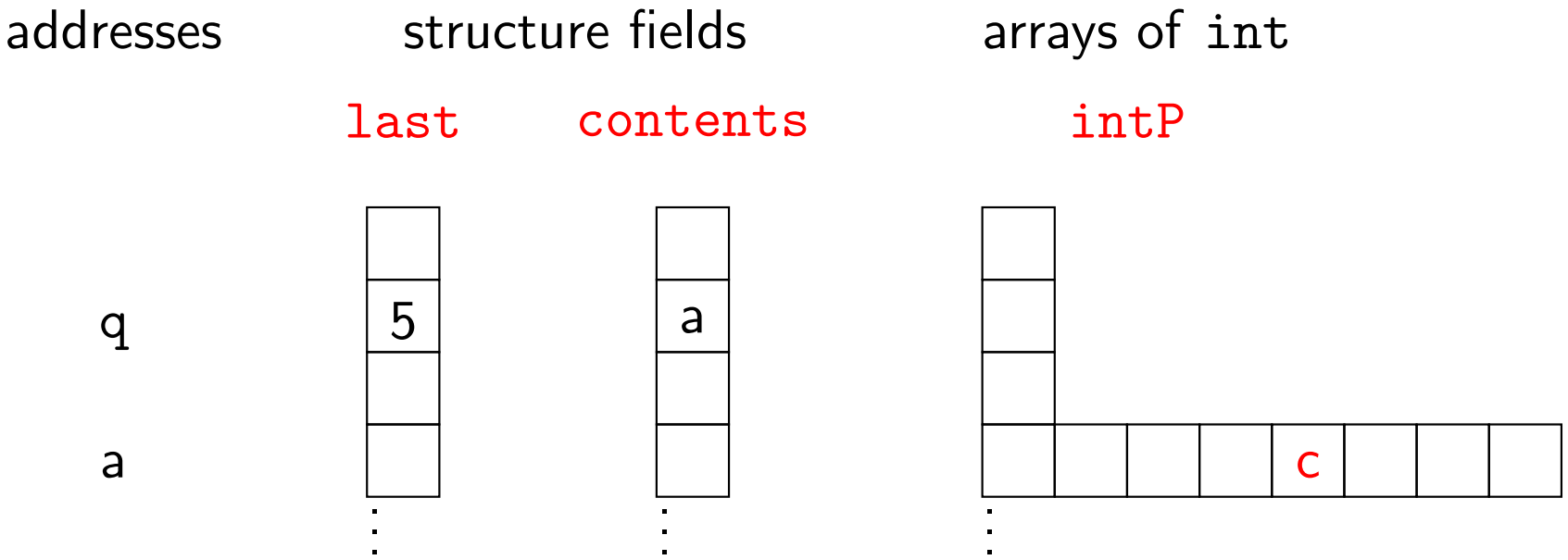
Burstall-Bornat model: example of character queue

addresses structure fields arrays of `int`



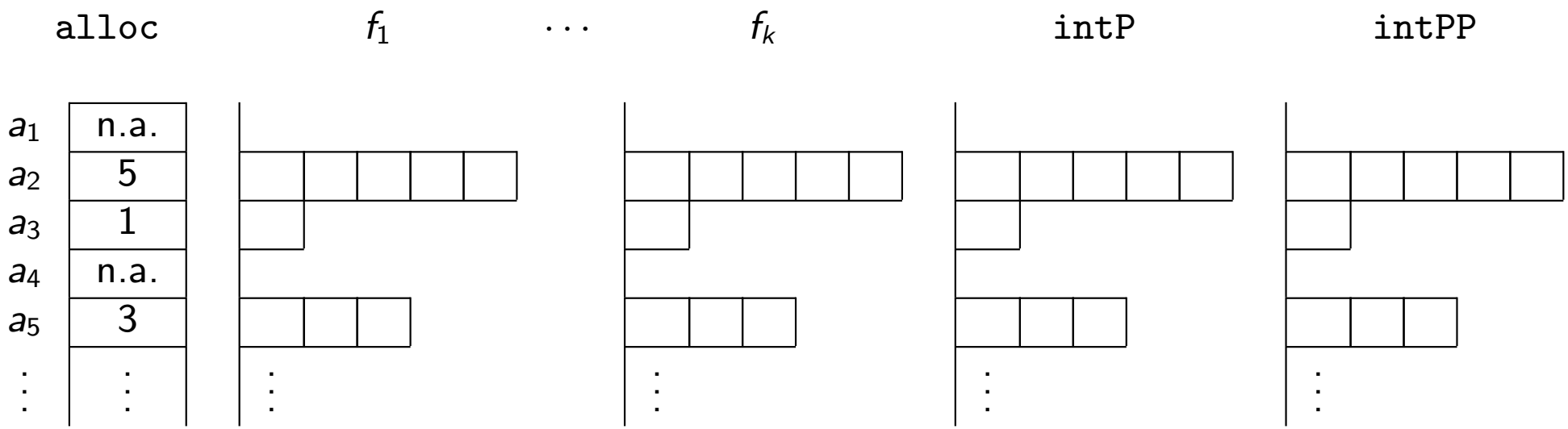

```
q.contents[q.last++] = c
    q.last <- 4 + 1
    *(q.contents+4) <- c
```

Burstall-Bornat model: example of character queue




```
q.contents[q.last++] = c
    q.last <- 4 + 1
    *(q.contents+4) <- c
```

General structure of C memory heap



Translation of C statements into Why

The C statement

```
q.contents[q.last++] = c
```

becomes in Why:

```
assert { valid(!alloc,q) };    // proof obligation
let tmp1 = acc(!last,q) in     // tmp1 <- q.last
last := upd(!last,q,tmp1+1);   // q.last <- tmp1+1
let tmp2 = shift(acc(!contents,q),tmp1) in
                               // tmp2 <- q.contents + tmp1
assert { valid(!alloc,tmp2) };// proof obligation
intP := upd(!intP,tmp2,c)      // *tmp2 <- c
```

Axiomatization

- The abstract Why functions `acc`, `upd`, `shift`, etc. are specified by axioms: the background theory
- Excerpt from this theory:

```
acc(upd(t,i,v),i) = v
i <> j -> acc(upd(t,i,v),j) = acc(t,j)
shift(p,0) = p
shift(shift(p,i),j) = shift(p,i+j)
...
```

- An important part of this theory is dedicated to assigns clauses

Example continued: certification of `push` function

Caduceus produces 3 verification conditions expressing that

- the code of `push` contains no unallocated pointer dereference (e.g. assignment of `q.contents[q.last++]` is valid)
- the postcondition and the `assigns` clause of `push` are established
- the invariant `q_invariant` is preserved by `push`

Proofs of these obligations

- with Simplify (100%) and CVC Lite (67%)
- with Coq (100%), very easy (6 lines of tactics)

Example continued: certification of `push` function

Caduceus produces 3 verification conditions expressing that

- the code of `push` contains no unallocated pointer dereference (e.g. assignment of `q.contents[q.last++]` is valid)
- the postcondition and the `assigns` clause of `push` are established
- the invariant `q_invariant` is preserved by `push`

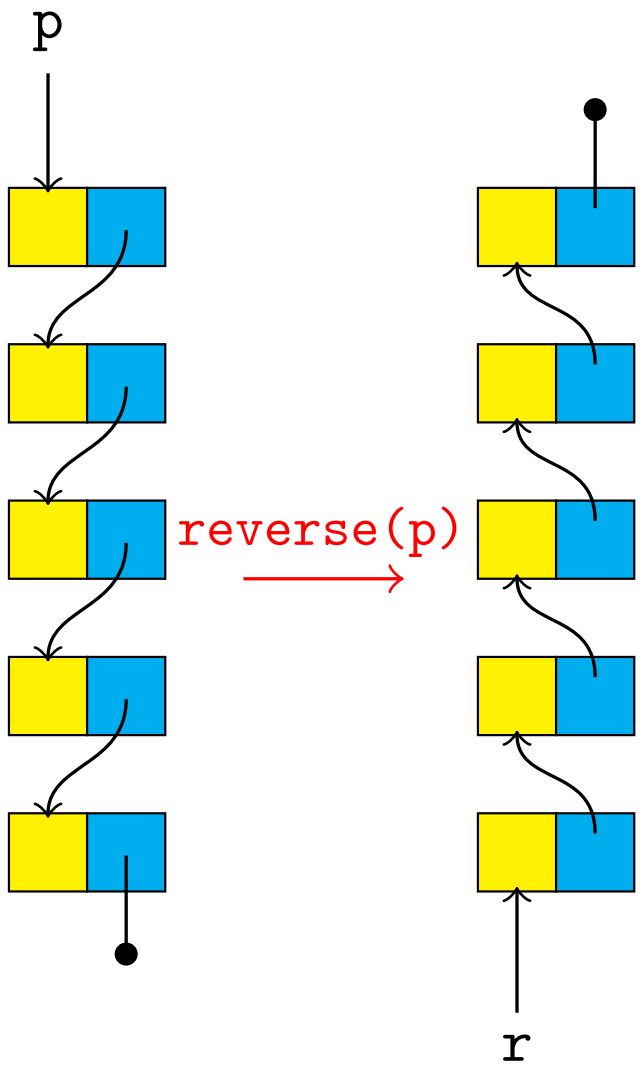
Proofs of these obligations

- with Simplify (100%) and CVC Lite (67%)
- with Coq (100%), very easy (6 lines of tactics)

Example: in-place list reversal

```
typedef struct struct_list {
  int hd;
  struct struct_list *tl;
} *list;

list reverse(list p) {
  list r = NULL;
  while (p != NULL) {
    list q = p ;
    p = p->tl;
    q->tl = r;
    r = q;
  }
  return r;
}
```



Introduction of new logical types and functions

- New predicates and functions can be introduced

```
// logical finite list of pointers
//@ logic plist nil()
//@ logic plist cons(list p, plist l)

// concatenation and reversal
//@ logic plist app(plist l1, plist l2)
//@ logic plist rev(plist pl)
```

- Axioms may be given, e.g.

```
//@ axiom app_nil : \forall plist l; app(nil(),l) == l
```

Introduction of new logical types and functions

- New predicates and functions can be introduced

```
// logical finite list of pointers
//@ logic plist nil()
//@ logic plist cons(list p, plist l)

// concatenation and reversal
//@ logic plist app(plist l1, plist l2)
//@ logic plist rev(plist pl)
```

- Axioms may be given, e.g.

```
//@ axiom app_nil : \forall plist l; app(nil(),l) == l
```

Specification of list reversal

```
/* llist(p,l) specifies that l is the list of pointers
   from p to NULL following tl fields */
//@ predicate llist(list p, plist l) reads p->tl

// is_list(p) specifies that p is finite
//@ predicate is_list(list p) { \exists plist l ; llist(p,l) }

/*@ requires is_list(p)
  @ ensures \forall plist l;
  @    \old(llist(p, l)) => llist(\result, rev(l))  */
list reverse(list p);
```

Specification of list reversal

```
/* llist(p,l) specifies that l is the list of pointers
   from p to NULL following tl fields */
//@ predicate llist(list p, plist l) reads p->tl

// is_list(p) specifies that p is finite
//@ predicate is_list(list p) { \exists plist l ; llist(p,l) }
```

```
/*@ requires is_list(p)
  @ ensures \forall plist l;
  @    \old(llist(p, l)) => llist(\result, rev(l))  */
list reverse(list p);
```

Specification of list reversal

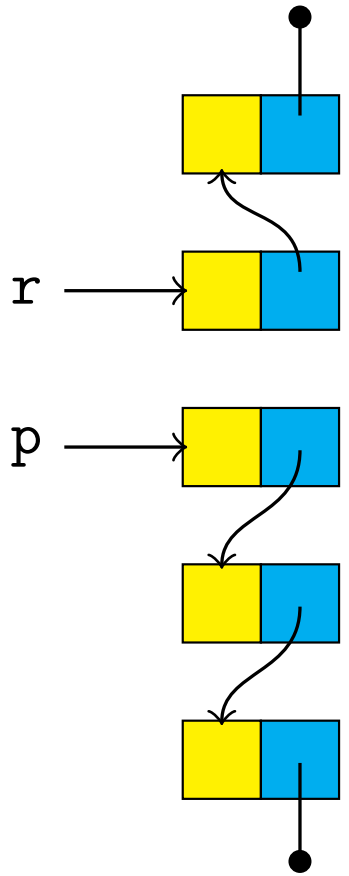
```
/* llist(p,l) specifies that l is the list of pointers
   from p to NULL following tl fields */
//@ predicate llist(list p, plist l) reads p->tl

// is_list(p) specifies that p is finite
//@ predicate is_list(list p) { \exists plist l ; llist(p,l) }

/*@ requires is_list(p)
  @ ensures \forall plist l;
  @    \old(llist(p, l)) => llist(\result, rev(l))  */
list reverse(list p);
```

Annotating the code of list reversal

```
list reverse(list p) {
  list r = NULL;
/*@ invariant
    \exists plist lp; \exists plist lr;
      llist(p, lp) && llist(r, lr) &&
      disjoint(lp, lr) &&
      \forall plist l; \old(llist(p, l)) =>
        app(rev(lp), lr) == rev(l)
  @ variant length(p) for length_order */
  while (p != NULL) {
    list q = p;
    p = p->tl; q->tl = r; r = q;
  }
  return r;
}
```



Certification of list reversal

- 7 verification conditions
- With Simplify: 71%
- With Coq: 100%, with 661 lines of tactics

Example: Schorr-Waite algorithm

- Graph marking algorithm
- Considered as a benchmark for the verification of pointer programs (Bornat, 1999, Jape system) (Nipkow-Mehta, 2003, Isabelle/HOL)
- 12 verification conditions
- With Simplify: 33%
- With Coq: 100%, with 2362 lines of tactics

Example: Schorr-Waite algorithm

- Graph marking algorithm
- Considered as a benchmark for the verification of pointer programs (Bornat, 1999, Jape system) (Nipkow-Mehta, 2003, Isabelle/HOL)
- 12 verification conditions
- With Simplify: 33%
- With Coq: 100%, with 2362 lines of tactics

Part V

Conclusion

Conclusion

- We are able to certify non trivial programs
- We support a large subset of ANSI C and Java/JML
- Tools freely available
  - http://why.lri.fr/
  - http://caduceus.lri.fr/
  - http://krakatoa.lri.fr/

But scaling up issues show up on large programs:

- Generated proof obligations can get large
- Clear need for assistance to write specifications
- Need for more automation of proofs, cooperation of provers

Conclusion

- We are able to certify non trivial programs
- We support a large subset of ANSI C and Java/JML
- Tools freely available
  - http://why.lri.fr/
  - http://caduceus.lri.fr/
  - http://krakatoa.lri.fr/

But scaling up issues show up on large programs:

- Generated proof obligations can get large
- Clear need for assistance to write specifications
- Need for more automation of proofs, cooperation of provers

Current limitations / work in progress

Limitations of the tools

- (mutually) recursive functions
- arithmetic overflow
- floating point arithmetic

Limitations of the C model

- pointer cast
- unions
- non ANSI (i.e. compiler dependent) features

Current limitations / work in progress

Limitations of the tools

- (mutually) recursive functions
- arithmetic overflow
- floating point arithmetic

Limitations of the C model

- pointer cast
- unions
- non ANSI (i.e. compiler dependent) features

Next challenge

Verification of ML programs (with side-effects)

Next challenge

Verification of ML programs (with side-effects)

Decision Procedures
1: Survey of decision procedures

John Harrison

Intel Corporation

TYPES summer school 2005, Göteborg

Fri 19th August 2005 (09:00 – 09:45)

Summary

- Interesting and uninteresting proofs
- Theory and practice
- Beyond our scope
- Logic and theories
- Pure logic
- Decidable theories

Interesting and uninteresting proofs

Much of this summer school emphasizes how interesting and useful proofs themselves are. But they aren't always!

$$
\begin{aligned}
&6(x_1^2 + x_2^2 + x_3^2 + x_4^2)^2 = \\
&(x_1 + x_2)^4 + (x_1 + x_3)^4 + (x_1 + x_4)^4 + \\
&(x_2 + x_3)^4 + (x_2 + x_4)^4 + (x_3 + x_4)^4 + \\
&(x_1 - x_2)^4 + (x_1 - x_3)^4 + (x_1 - x_4)^4 + \\
&(x_2 - x_3)^4 + (x_2 - x_4)^4 + (x_3 - x_4)^4
\end{aligned}
$$

We'd like to concentrate on interesting parts, automating parts with

- No interesting computational content
- No intellectual interest in the proof method

Theory and practice

We may ask what problems are decidable

- In principle
- In a feasible time bound
- On real problems of interest

Not always the same! Consider propositional logic.

- Trivial
- Infeasible
- Very useful

What we'll cover

We'll consider only theories in classical first-order logic.

- Key decidability results for first order theories
- Focus on pure logic and arithmetical theories

What we won't cover

We miss out several key related areas:

- Decision procedures for constructive/intuitionistic theories
- Decision procedures for fragments of higher-order logic
- Decision procedures for modal or other nonclassical logics.

For example:

- First-order validity semidecidable, but higher-order validity subsumes arithmetic truth, so not even semidecidable
- Example: first order theories of real and algebraically closed fields are decidable classically (Tarski 1930) but not intuitionistically (Gabbay 1973).

First-order logic

English	Standard	Other
false	$\bot$	0, F
true	$\top$	1, T
not p	$\neg p$	$\overline{p}$, $-p$, $\sim p$
p and q	$p \wedge q$	pq, $p\&q$, $p \cdot q$
p or q	$p \vee q$	$p + q$, $p \mid q$, $p\ or\ q$
p implies q	$p \Rightarrow q$	$p \leq q$, $p \rightarrow q$, $p \supset q$
p iff q	$p \Leftrightarrow q$	$p = q$, $p \equiv q$, $p \sim q$
For all x, p	$\forall x.\, p$	$(x)p$, Axp
Exists x s.t. p	$\exists x.\, p$	$(\exists x.\,)p$, Exp

Semantics

Key semantic notion is $A \models p$: in any model where all formulas in A hold, then p holds.

Crucial distinction between

- Logical validity — holds whatever the interpretation of symbols
- Truth in a particular theory

For example, $x + y = y + x$ holds in most arithmetical models, but not for *any* interpretation of '$+$', so $\not\models x + y = y + x$.

Theories

A theory is a set of formulas T closed under logical validity, i.e. $T \models p$ iff $p \in T$. A theory T is:

- *Consistent* if we never have $p \in T$ and $(\neg p) \in T$.
- *Complete* if for closed p we have $p \in T$ or $(\neg p) \in T$.
- *Decidable* if there's an algorithm to tell us whether a given closed p is in T

Note that a complete theory generated by an r.e. axiom set is also decidable.

Pure first-order logic

Not decidable but at least *semidecidable*: there is a complete proof search procedure to decide if $\models p$ for any given p.

- Can search for proofs in any of the standard calculi
- Tends to be easier using 'cut-free' systems like sequent calculus
- More convenient, though not necessary, to Skolemize first.
- Exploit unification to instantiate intelligently

A significant distinction

A significant characteristic is whether unifiers are global, applying everywhere, or just local:

- Top-down, global methods (tableaux, model elimination)
- Bottom-up, local methods (resolution, inverse method)

These proof methods tend to have corresponding characteristics.

Decidable problems

Although first order validity is undecidable, there are special cases where it is decidable, e.g.

- AE formulas: no function symbols, universal quantifiers before existentials in prenex form (so finite Herbrand base).
- Monadic formulas: no function symbols, only unary predicates

These are not particularly useful in practice, though they can be used to automate syllogistic reasoning.

> If all M are P, and all S are M, then all S are P

can be expressed as the monadic formula:

$$(\forall x.\, M(x) \Rightarrow P(x)) \wedge (\forall x.\, S(x) \Rightarrow M(x)) \Rightarrow (\forall x.\, S(x) \Rightarrow P(x))$$

The theory of equality

A simple but useful decidable theory is the universal theory of equality with function symbols, e.g.

$$\forall x.\ f(f(f(x)) = x \wedge f(f(f(f(f(x))))) = x \Rightarrow f(x) = x$$

after negating and Skolemizing we need to test a ground formula for satisfiability:

$$f(f(f(c)) = c \wedge f(f(f(f(f(c))))) = c \wedge \neg(f(c) = c)$$

Two well-known algorithms:

- Put the formula in DNF and test each disjunct using one of the classic 'congruence closure' algorithms.
- Reduce to SAT by introducing a propositional variable for each equation between subterms and adding constraints.

Decidable theories

More useful in practical applications are cases not of *pure* validity, but validity in special (classes of) models, or consequence from useful axioms, e.g.

- Does a formula hold over all rings (Boolean rings, non-nilpotent rings, integral domains, fields, algebraically closed fields, . . .)
- Does a formula hold in the natural numbers or the integers?
- Does a formula hold over the real numbers?
- Does a formula hold in all real-closed fields?
- . . .

Because arithmetic comes up in practice all the time, there's particular interest in theories of arithmetic.

Quantifier elimination

Often, a quantified formula is T-equivalent to a quantifier-free one:

- $\mathbb{C} \models (\exists x.\, x^2 + 1 = 0) \Leftrightarrow \top$
- $\mathbb{R} \models (\exists x. ax^2 + bx + c = 0) \Leftrightarrow a \neq 0 \wedge b^2 \geq 4ac \vee a = 0 \wedge (b \neq 0 \vee c = 0)$
- $\mathbb{Q} \models (\forall x.\, x < a \Rightarrow x < b) \Leftrightarrow a \leq b$
- $\mathbb{Z} \models (\exists k\ x\ y.\, ax = (5k + 2)y + 1) \Leftrightarrow \neg(a = 0)$

We say a theory T admits *quantifier elimination* if *every* formula has this property.

Assuming we can decide variable-free formulas, quantifier elimination implies completeness.

And then an *algorithm* for quantifier elimination gives a decision method.

Important arithmetical examples

- Presburger arithmetic: arithmetic equations and inequalities with addition but *not multiplication*, interpreted over $\mathbb{Z}$ or $\mathbb{N}$.
- Tarski arithmetic: arithmetic equations and inequalities with addition and multiplication, interpreted over $\mathbb{R}$ (or any real-closed field)
- General algebra: arithmetic equations with addition and multiplication interpreted over $\mathbb{C}$ (or other algebraically closed field).

However, arithmetic with multiplication over $\mathbb{Z}$ is not even semidecidable, by Gödel's theorem.

Nor is arithmetic over $\mathbb{Q}$ (Julia Robinson), nor just solvability of equations over $\mathbb{Z}$ (Matiyasevich). Equations over $\mathbb{Q}$ unknown.

Pick 'n mix

There are some known cases of quantifier elimination for combined theories

- BAPA — Boolean algebra of finite sets plus Presburger arithmetic (Feferman/Vaught, Kuncac/Nguyen/Rinard)
- Mixed real-integer linear arithmetic with floor function (Weispfenning)

In lecture 3 we'll examine more systemtic and modular ways of combining theories.

Summary

- We'd like to be able to automate boring routine proofs
- Well-established repertoire of decidable theories
- Theory/practice distinction can make a dramatic difference
- Many decision methods are based on more general quantifier elimination
- It is possible, but not routine, to find decidable mixtures.

Decision Procedures
2: Real quantifier elimination

John Harrison

Intel Corporation

TYPES summer school 2005, Göteborg

Fri 19th August 2005 (09:55 – 10:40)

Summary

- What we'll prove
- History
- Sign matrices
- The key recursion
- Parametrization
- Real-closed fields

What we'll prove

Take a first-order language:

- All rational constants p/q
- Operators of negation, addition, subtraction and multiplication
- Relations '$=$', '$<$', '$\leq$', '$>$', '$\geq$'

We'll prove that every formula in the language has a quantifier-free equivalent, and will give a systematic algorithm for finding it.

Applications

In principle, this method can be used to solve many non-trivial problems.

> Kissing problem: how many disjoint n-dimensional spheres can be packed into space so that they touch a given unit sphere?

Pretty much *any* geometrical assertion can be expressed in this theory.

If theorem holds for *complex* values of the coordinates, and then simpler methods are available (Gröbner bases, Wu-Ritt triangulation. . .).

History

- 1930: Tarski discovers quantifier elimination procedure for this theory.
- 1948: Tarski's algorithm published by RAND
- 1954: Seidenberg publishes simpler algorithm
- 1975: Collins develops and *implements* cylindrical algebraic decomposition (CAD) algorithm
- 1983: Hörmander publishes very simple algorithm based on ideas by Cohen.
- 1990: Vorobjov improves complexity bound to doubly exponential in number of quantifier *alternations*.

We'll present the Cohen-Hörmander algorithm.

Current implementations

There are quite a few simple versions of real quantifier elimination, even in computer algebra systems like Mathematica.

Among the more heavyweight implementations are:

- `qepcad` — `http://www.cs.usna.edu/~qepcad/B/QEPCAD.html`
- `REDLOG` — `http://www.fmi.uni-passau.de/~redlog/`

One quantifier at a time

For a general quantifier elimination procedure, we just need one for a formula

$$\exists x.\ P[a_1, \ldots, a_n, x]$$

where $P[a_1, \ldots, a_n, x]$ involves no other quantifiers but may involve other variables.

Then we can apply the procedure successively inside to outside, dealing with universal quantifiers via $(\forall x.\ P[x]) \Leftrightarrow (\neg \exists x.\ \neg P[x])$.

Forget parametrization for now

First we'll ignore the fact that the polynomials contain variables other than the one being eliminated.

This keeps the technicalities a bit simpler and shows the main ideas clearly.

The generalization to the parametrized case will then be very easy:

- Replace polynomial division by pseudo-division
- Perform case-splits to determine signs of coefficients

Sign matrices

Take a set of univariate polynomials $p_1(x)$, ..., $p_n(x)$.

A *sign matrix* for those polynomials is a division of the real line into alternating points and intervals:

$$(-\infty, x_1), x_1, (x_1, x_2), x_2, \ldots, x_{m-1}, (x_{m-1}, x_m), x_m, (x_m, +\infty)$$

and a matrix giving the sign of each polynomial on each interval:

- Positive ($+$)
- Negative ($-$)
- Zero (0)

Sign matrix example

The polynomials $p_1(x) = x^2 - 3x + 2$ and $p_2(x) = 2x - 3$ have the following sign matrix:

Point/Interval	p_1	p_2
$(-\infty, x_1)$	$+$	$-$
x_1	0	$-$
(x_1, x_2)	$-$	$-$
x_2	$-$	0
(x_2, x_3)	$-$	$+$
x_3	0	$+$
$(x_3, +\infty)$	$+$	$+$

Using the sign matrix

Using the sign matrix for all polynomials appearing in $P[x]$ we can answer any quantifier elimination problem: $\exists x.\ P[x]$

- Look to see if any row of the matrix satisfies the formula (hence dealing with existential)
- For each row, just see if the corresponding set of signs satisfies the formula.

We have replaced the quantifier elimination problem with sign matrix determination

Finding the sign matrix

For constant polynomials, the sign matrix is trivial (2 has sign '$+$' etc.)

To find a sign matrix for $p, p_1, \ldots, p_n$ it suffices to find one for $p', p_1, \ldots, p_n, r_0, r_1, \ldots, r_n$, where

- $p_0 \equiv p'$ is the derivative of p
- $r_i = \mathsf{rem}(p, p_i)$

(Remaindering means we have some q_i so $p = q_i \cdot p_i + r_i$.)

Taking p to be the polynomial of highest degree we get a simple recursive algorithm for sign matrix determination.

Details of recursive step

So, suppose we have a sign matrix for $p', p_1, \ldots, p_n, r_0, r_1, \ldots, r_n$.

We need to construct a sign matrix for $p, p_1, \ldots, p_n$.

- May need to add more points and hence intervals for roots of p
- Need to determine signs of $p_1, \ldots, p_n$ at the new points and intervals
- Need the sign of p itself everywhere.

Step 1

Split the given sign matrix into two parts, but keep all the points for now:

- M for $p', p_1, \ldots, p_n$
- M' for $r_0, r_1, \ldots, r_n$

We can infer the sign of p at all the 'significant' *points* of M as follows:

$$p = q_i p_i + r_i$$

and for each of our points, one of the p_i is zero, so $p = r_i$ there and we can read off p's sign from r_i's.

Step 2

Now we're done with M' and we can throw it away.

We also 'condense' M by eliminating points that are not roots of one of the $p', p_1, \ldots, p_n$.

Note that the sign of any of these polynomials is stable on the condensed intervals, since they have no roots there.

- We know the sign of p at all the points of this matrix.
- However, p itself may have additional roots, and we don't know anything about the intervals yet.

Step 3

There can be at most one root of p in each of the existing intervals, because otherwise p' would have a root there.

We can tell whether there is a root by checking the signs of p (determined in Step 1) at the two endpoints of the interval.

Insert a new point precisely if p has strictly opposite signs at the two endpoints (simple variant for the two end intervals).

None of the other polynomials change sign over the original interval, so just copy the values to the point and subintervals.

Throw away p' and we're done!

Multivariate generalization

In the multivariate context, we can't simply divide polynomials. Instead of

$$p = p_i \cdot q_i + r_i$$

we get

$$a^k p = p_i \cdot q_i + r_i$$

where a is the leading coefficient of p_i.

The same logic works, but we need case splits to fix the sign of a.

Real-closed fields

With more effort, all the 'analytical' facts can be deduced from the axioms for *real-closed fields*.

- Usual ordered field axioms
- Existence of square roots: $\forall x.\, x \geq 0 \Rightarrow \exists y.\, x = y^2$
- Solvability of odd-degree equations:
  $\forall a_0, \ldots, a_n.\, a_n \neq 0 \Rightarrow \exists x.\, a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = 0$

Examples include computable reals and algebraic reals. So this already gives a complete theory, without a stronger completeness axiom.

Summary

- Real quantifier elimination one of the most significant logical decidability results known.
- Original result due to Tarski, for general real closed fields.
- A half-century of research has resulted in simpler and more efficient algorithms (not always at the same time).
- The Cohen-Hörmander algorithm is remarkably simple (relatively speaking).
- The complexity, both theoretical and practical, is still bad, so there's limited success on non-trivial problems.

Decision Procedures
3: Combination and certification of decision procedures

John Harrison

Intel Corporation

TYPES summer school 2005, Göteborg

Sat 20th August 2005 (12:05 – 12:50)

Summary

- Need to combine multiple decision procedures
- Basics of Nelson-Oppen method
- Proof-producing decision procedures
- Separate certification
- LCF-style implementation and reflection

Need for combinations

In applications we often need to combine decision methods from different domains.

$$x - 1 < n \wedge \neg(x < n) \Rightarrow a[x] = a[n]$$

An arithmetic decision procedure could easily prove

$$x - 1 < n \wedge \neg(x < n) \Rightarrow x = n$$

but could not make the additional final step, even though it looks trivial.

Most combinations are undecidable

Adding almost any additions, especially uninterpreted, to the usual decidable arithmetic theories destroys decidability.

Some exceptions like BAPA ('Boolean algebra + Presburger arithmetic').

This formula over the reals constrains P to define the integers:

$$(\forall n.\, P(n+1) \Leftrightarrow P(n)) \wedge (\forall n.\, 0 \leq n \wedge n < 1 \Rightarrow (P(n) \Leftrightarrow n = 0))$$

and this one in Presburger arithmetic defines squaring:

$$(\forall n.\, f(-n) = f(n)) \wedge (f(0) = 0) \wedge$$
$$(\forall n.\, 0 \leq n \Rightarrow f(n+1) = f(n) + n + n + 1)$$

and so we can define multiplication.

Quantifier-free theories

However, if we stick to so-called ‘quantifier-free’ theories, i.e. deciding universal formulas, things are better.

Two well-known methods for combining such decision procedures:

- Nelson-Oppen
- Shostak

Nelson-Oppen is more general and conceptually simpler.

Shostak seems more efficient where it does work, and only recently has it really been understood.

Nelson-Oppen basics

Key idea is to combine theories $T_1, \ldots, T_n$ with *disjoint signatures*. For instance

- T_1: numerical constants, arithmetic operations
- T_2: list operations like cons, head and tail.
- T_3: other uninterpreted function symbols.

The only common function or relation symbol is '$=$'.

This means that we only need to share formulas built from equations among the component decision procedure, thanks to the *Craig interpolation theorem*.

The interpolation theorem

Several slightly different forms; we'll use this one (by compactness, generalizes to theories):

> If $\models \phi_1 \wedge \phi_2 \Rightarrow \bot$ then there is an 'interpolant' ψ, whose only free variables and function and predicate symbols are those occurring in *both* ϕ_1 and ϕ_2, such that $\models \phi_1 \Rightarrow \psi$ and $\models \phi_2 \Rightarrow \neg\psi$.

This is used to assure us that the Nelson-Oppen method is complete, though we don't need to produce general interpolants in the method.

In fact, interpolants can be found quite easily from proofs, including Herbrand-type proofs produced by resolution etc.

Nelson-Oppen I

Proof by example: refute the following formula in a mixture of Presburger arithmetic and uninterpreted functions:

$$f(v-1)-1 = v+1 \wedge f(u)+1 = u-1 \wedge u+1 = v$$

First step is to *homogenize*, i.e. get rid of atomic formulas involving a mix of signatures:

$$u+1 = v \wedge v_1+1 = u-1 \wedge v_2-1 = v+1 \wedge v_2 = f(v_3) \wedge v_1 = f(u) \wedge v_3 = v-1$$

so now we can split the conjuncts according to signature:

$$(u+1 = v \wedge v_1+1 = u-1 \wedge v_2-1 = v+1 \wedge v_3 = v-1) \wedge$$
$$(v_2 = f(v_3) \wedge v_1 = f(u))$$

Nelson-Oppen II

If the entire formula is contradictory, then there's an interpolant ψ such that in Presburger arithmetic:

$$\mathbb{Z} \models u + 1 = v \wedge v_1 + 1 = u - 1 \wedge v_2 - 1 = v + 1 \wedge v_3 = v - 1 \Rightarrow \psi$$

and in pure logic:

$$\models v_2 = f(v_3) \wedge v_1 = f(u) \wedge \psi \Rightarrow \bot$$

We can assume it only involves variables and equality, by the interpolant property and disjointness of signatures.

Subject to a technical condition about finite models, the pure equality theory admits quantifier elimination.

So we can assume ψ is a propositional combination of equations between variables.

Nelson-Oppen III

In our running example, $u = v_3 \wedge \neg(v_1 = v_2)$ is one suitable interpolant, so

$$\mathbb{Z} \models u + 1 = v \wedge v_1 + 1 = u - 1 \wedge v_2 - 1 = v + 1 \wedge v_3 = v - 1 \Rightarrow u = v_3 \wedge \neg(v_1 = v_2)$$

in Presburger arithmetic, and in pure logic:

$$\models v_2 = f(v_3) \wedge v_1 = f(u) \Rightarrow u = v_3 \wedge \neg(v_1 = v_2) \Rightarrow \bot$$

The component decision procedures can deal with those, and the result is proved.

Nelson-Oppen IV

Could enumerate all significanctly different potential interpolants.

Better: case-split the original problem over all possible equivalence relations between the variables (5 in our example).

$$T_1, \ldots, T_n \models \phi_1 \wedge \cdots \wedge \phi_n \wedge ar(P) \Rightarrow \bot$$

So by interpolation there's a C with

$$T_1 \models \phi_1 \wedge ar(P) \Rightarrow C$$
$$T_2, \ldots, T_n \models \phi_2 \wedge \cdots \wedge \phi_n \wedge ar(P) \Rightarrow \neg C$$

Since $ar(P) \Rightarrow C$ or $ar(P) \Rightarrow \neg C$, we must have one theory with $T_i \models \phi_i \wedge ar(P) \Rightarrow \bot$.

Nelson-Oppen V

Still, there are quite a lot of possible equivalence relations (bell$(5) = 52$), leading to large case-splits.

An alternative formulation is to repeatedly let each theory deduce new disjunctions of equations, and case-split over them.

$$T_i \models \phi_i \Rightarrow x_1 = y_1 \vee \cdots \vee x_n = y_n$$

This allows two imporant optimizations:

- If theories are *convex*, need only consider pure equations, no disjunctions.
- Component procedures can actually produce equational consequences rather than waiting passively for formulas to test.

Shostak's method

Can be seen as an optimization of Nelson-Oppen method for common special cases. Instead of just a decision method each component theory has a

- Canonizer — puts a term in a T-canonical form
- Solver — solves systems of equations

Shostak's original procedure worked well, but the theory was flawed on many levels. In general his procedure was incomplete and potentially nonterminating.

It's only recently that a full understanding has (apparently) been reached.

See ICS (`http://www.icansolve.com`) for one implementation.

Certification of decision procedures

We might want a decision procedure to produce a ‘proof’ or ‘certificate’

- Doubts over the correctness of the core decision method
- Desire to use the proof in other contexts

This arises in at least two real cases:

- Fully expansive (e.g. ‘LCF-style’) theorem proving.
- Proof-carrying code

Certifiable and non-certifiable

The most desirable situation is that a decision procedure should produce a short certificate that can be checked easily.

Factorization and primality is a good example:

- Certificate that a number is not prime: the factors! (Others are also possible.)
- Certificate that a number is prime: Pratt, Pocklington, Pomerance, ...

This means that primality checking is in NP $\cap$ co-NP (we now know it's in P).

Certifying universal formulas over $\mathbb{C}$

Use the (weak) *Hilbert Nullstellensatz*:

The polynomial equations $p_1(x_1, \ldots, x_n) = 0, \ldots, p_k(x_1, \ldots, x_n) = 0$ in an algebraically closed field have *no* common solution iff there are polynomials $q_1(x_1, \ldots, x_n), \ldots, q_k(x_1, \ldots, x_n)$ such that the following polynomial identity holds:

$$q_1(x_1, \ldots, x_n) \cdot p_1(x_1, \ldots, x_n) + \cdots + q_k(x_1, \ldots, x_n) \cdot p_k(x_1, \ldots, x_n) = 1$$

All we need to certify the result is the cofactors $q_i(x_1, \ldots, x_n)$, which we can find by an instrumented Gröbner basis algorithm.

The checking process involves just algebraic normalization (maybe still not totally trivial...)

Certifying universal formulas over $\mathbb{R}$

There is a similar but more complicated Nullstellensatz (and Positivstellensatz) over $\mathbb{R}$.

The general form is similar, but it's more complicated because of all the different orderings.

It inherently involves sums of squares (SOS), and the certificates can be found efficiently using semidefinite programming (Parillo . . .)

Example: easy to check

$$\forall a\ b\ c\ x.\ ax^2 + bx + c = 0 \Rightarrow b^2 - 4ac \geq 0$$

via the following SOS certificate:

$$b^2 - 4ac = (2ax + b)^2 - 4a(ax^2 + bx + c)$$

Less favourable cases

Unfortunately not all decision procedures seem to admit a nice separation of proof from checking.

Then if a proof is required, there seems no significantly easier way than generating proofs along each step of the algorithm.

Example: Cohen-Hörmander algorithm implemented in HOL Light by McLaughlin (CADE 2005).

Works well, useful for small problems, but about $1000\times$ slowdown relative to non-proof-producing implementation.

Summary

- There is a need for combinations of decision methods
- For general quantifier prefixes, relatively few useful results.
- Nelson-Oppen and Shostak give useful methods for universal formulas.
- We sometimes also want decision procedures to produce proofs
- Some procedures admit efficient separation of search and checking, others do not.
- Interesting research topic: new ways of compactly certifying decision methods.

KRAKATOA

Reasoning on Java Programs

Christine Paulin-Mohring (with Claude Marché)
INRIA Futurs & Université Paris Sud, Orsay, France

Proofs of Programs and Formalisation of Mathematics
TYPES Summer School 2005

Warning

- KRAKATOA is based on the WHY tool and uses a model in COQ.
- WHY and COQ will be presented next week ...

These lectures mainly focus on (an example of) applying type theory to programming language modeling and program verification

Outline

Lecture 1

Introduction

Motivations

Tools & methods which improve the quality of software development

Programs are :

- manipulated (compiled, executed) by a computer
- written and read by a human

We need :

- Less runtime errors
- Explicit link between documentation and code

Possible solutions

- Type-checking at compile time detects a certain class of errors and reduce the number of dynamic checks
- Many common errors are undecidable :
  - non-termination, division by zero ...

  Abstract interpretation can help detecting certain errors
- Many more properties can be interesting for the programmer
  - an array is sorted, a linked structure does not contain cycles ...

  Logical assertions to be proved.

How to prove programs ?

- Proving programs requires to analyse a mathematical model of the program and its specification.
- Find an apropriate model (many different semantics)
  - Denotational: mathematical functions on domains
  - Operational: execution steps
  - Axiomatic: relation between programs and properties of states
  - Monads: pure functional terms on complex data
- Proofs can be informal on paper or formal on computer

Formal proofs on computers

- Language for specifications
  - Understandable by both computers and humans
- A formal mathematical model for the specification language
- A formal correctness relation between programs and specifications
- Support for building the mathematical model of both program and specification and checking correctness

Which programming and specification language ?

- Most programming languages have complex syntax and semantics
- Semantics is not always abstractly defined but can be compiler dependent (requires a low level model of execution)
- Specification languages should be used during development and consequentely well accepted by the programmer

What about Type Theory ?

Type theory is definitely one solution:

- Programs are purely functional terms, with a natural mathematical model (strong termination)
- Dependent types are a natural specification language (can express directly properties of objects and programs)
- Curry-Howard : correctness is type-checking (of course with additional proof information)

More on this during Summer School !

The world is not yet ready to use Type Theory for programming!

What about JAVA ?

A high-level language designed for secure applications (mobile code executed on different platforms)

- garbage collection
- strong typing at compile time
- static checking of byte-code
- dynamic checking
  - security policies (sandbox, firewall)

JAVACARD

- A subset of JAVA designed for smartcards (sequential, no dynamic loading ...)
- Additional features for smartcards : (atomic transactions, persistent data, API ...)
- JAVACARD is a good target for verification
  - simple applets ...
  - evidence of security required (Common Criteria)
  - many smartcards based on JAVACARD or similar technologies

Lecture 1

Modeling JAVA (JAVACARD)

Outline

- More on strong typing
- Different approaches (deep versus shallow embedding)
- Our model of JAVA

Modeling JAVA

Strong typing

About strong typing

Type soundness :

ML a terminating program of type `list` evaluates to `nil` or `cons`

JAVA access to a field or a method of a non-null object always succeeds

Other dynamic errors may occur :

- access to fields or methods of a null object (raises NullPointerException)
- incorrect instantiation of arrays (raises ArrayStoreException)

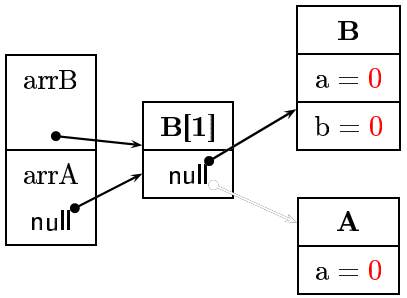
Instantiation of arrays : static view

Typing rule for arrays : $B \preceq A$ implies $B[] \preceq A[]$

```
class A { int a; }
class B extends A { int b; }
public static void main (String args[]) {
  A arrA[] ; B arrB[] = new B[1];
   arrB[0]=new B();
   arrA=arrB;
   arrA[0]=new A();
   System.out.println(arrB[0].b);
}
```


Instantiation of arrays : dynamic view

arrB[0]=new B(); raises ArrayStoreException



Modeling JAVA

Different approaches

Studying the JAVA or JAVACARD platforms

Type theory is a good framework to formally study the underlying definitions, algorithms and properties.

- Type soundness
- Operational and axiomatic semantics
- JAVA & JAVACARD virtual machines
- Byte-code verifiers
- Sandbox or Firewall mechanisms

References

Models of plaform components using proof assistants:

- Bali Project (T. Nipkow, Munich) using Isabelle/HOL
  `http://isabelle.in.tum.de/Bali/`
- Formavie project (Trusted Logic, Axalto) using Coq
  - certification at level EAL7
  - non-interference properties
- Certicartes (G. Barthe, Sophia-Antipolis) using Coq
  `http://www-sop.inria.fr/lemme/verificard/`
  Functional definition of semantics (Jakarta)

Applications

- Better understanding of semantics
- Useful for program verification
  - correct model of programs
  - identify properties valid from type-checking and properties which need logical verification
- Compilers, verifiers are programs that are likely to be written in a functional way

Proving a specific Java program

- Deep embedding : formalisation of the programming language (can reuse the work on platforms)
  - Abstract syntax tree formalised in the proof assistant
  - Translation from syntax to semantics done by an internal function
- Shallow embedding : direct representation of the program as a logical object
  - Programs constructions interpreted as notations
  - Translation from syntax to semantics done at the meta-level

Example

Concrete Syntax

$expr ::= \texttt{var} \mid \texttt{cte} \mid expr.\texttt{field} \mid expr\ \texttt{op}\ expr$

Semantics

Values are integers, null object or references in the heap

Example : deep embedding

Abstract syntax trees

```
type expr = Var of var | Cte of int |
            Acc of expr * field |
            Bin of  expr * op * expr
```

Values

```
type value = Int of int | Null | Ref of addr
```

Stack and heap

```
type env = var → value
type store = addr → (field → value)
```


Relational semantics

sem(s:env,h:store,e:expr,v:value) inductively defined

$$\frac{}{\texttt{sem}(s,h,\texttt{Var}(v),s(v))} \qquad \frac{}{\texttt{sem}(s,h,\texttt{Cte}(n),\texttt{Int}(n))}$$

$$\frac{\texttt{sem}(s,h,e,\texttt{Ref}(a))}{\texttt{sem}(s,h,\texttt{Acc}(e,f),h(a,f))}$$

$$\frac{\texttt{sem}(s,h,e1,\texttt{Int}(n1)) \quad \texttt{sem}(s,h,e2,\texttt{Int}(n2))}{\texttt{sem}(s,h,\texttt{Bin}(e1,\texttt{op},e2),\texttt{Int}(\texttt{semop}(n1,n2)))}$$

Functional semantics

sem(s:env,h:store,e:expr) :value option recursively defined

```
sem(s,h,Var(v)) = Some(s(v))
sem(s,h,Cte(n)) = Some(Int(n))
sem(s,h,Acc(e,f)) =  match sem(s,h,e) with
           None           ⇒ None
         | Some(Null)     ⇒ None
         | Some(Ref(a))   ⇒ Some(h(a,f))
         | Some(Int(n))   ⇒ None %Should not happen
sem(s,h,Bin(e1,op,e2)) =
   match sem(s,h,e1),sem(s,h,e2) with
    Some(Int(n1)),Some(Int(n2)) ⇒ Some(Int(semop(n1,n2)))
     | _                        ⇒ None
```


Shallow embedding

Can use static analysis for a more direct functional interpretation

- Expressions of static type *integer* are interpreted as logical integers
- *Objects* are interpreted as reference values

  ```
  type value = Null | Ref of addr
  ```

- Stack and heap are splitted in two parts

  ```
  type envo = var → value
  type envi = var → int
  type store = addr → (field→value) * (field→int)
  ```

Functional interpretation

$[e]^i_{si,so,h}$: `int option` $\quad$ $[e]^o_{si,so,h}$: `value option`

$[\mathtt{n}]^i_{si,so,h} = \mathtt{Some}(\mathtt{n})$

$[e_1\ \mathtt{op}\ e_2]^i_{si,so,h} = \mathsf{match}\ ([e_1]^i_{si,so,h}, [e_2]^i_{si,so,h})\ \mathsf{with}$

$\quad (\mathtt{Some}(n_1), \mathtt{Some}(n_2)) \Rightarrow \mathtt{Some}(\mathtt{semop}(n_1, n_2)) \mid \_ \Rightarrow \mathtt{None}$

$[\mathtt{v}]^i_{si,so,h} = \mathtt{Some}(si(\mathtt{v}))$ $\quad$ $[\mathtt{v}]^o_{si,so,h} = \mathtt{Some}(so(\mathtt{v}))$

$[\mathtt{e.f}]^i_{si,so,h} = \mathsf{match}\ ([e]^o_{si,so,h})\ \mathsf{with}$

$\quad \mathtt{Some}(\mathtt{Ref}(a)) \Rightarrow \mathsf{let}\ (\_, hi) = h(a)\ \mathsf{in}\ hi(f) \mid \_ \Rightarrow \mathtt{None}$

Remarks

- Shallow embedding takes advantage of static analyses; it avoids syntactic encodings
- Dependent types allows to attach static types to expression and avoid the `value` disjoint union in deep embedding

References

- A shallow embedding of JAVA in PVS has been done in the Loop project (B. Jacobs, Nijmegen)
  `http://www.sos.cs.ru.nl/research/loop/`

Modeling JAVA

Formalising JAVA programs

Basic model : types and values

Classes `classId`, `Object`:`classId`
simple inheritance : `super`:`classId` →`classId option`

Types primitive types : `int`, `bool`, `float` ...
reference types : arrays indexed by types, classes.

Primitive values represented by logical values of type boolean, integer, reals ...

Reference values represented by an address (type `addr`) in the heap or the null value (type `value`)

State

An implicit set of locations containing values :

Stack Local variables, parameters

Global variables corresponding to static fields

Heap One cell for an address of an object and a field, or for the address of an array and an index

Each allocated address is associated to a `tag` which gives dynamic type information: object (class) or array (size, type of elements).
A table of allocations (type `store`) contains a finite set of allocated addresses with corresponding tags.

Computation

- reads and writes state, returns a value
- possible exceptional behavior
  (still returns the exceptional value and a state)
  exceptions are also useful to model control flow
  (break, continue ...)

Idea

JAVA programs can be translated in a (CAML-like) language with functional values, references and exceptions.

This is what WHY provides and what is used in KRAKATOA.

Logical functions

Corresponding to primitive JAVA operations

- `arraylength` : `value` →`int`
  get information from the tag in the allocation table,
  0 as a default value
- `instanceof` : `value` →`javaType` →`bool`
  assume `super` does not generate infinite chains, uses the allocation table to look at the dynamic type of value
- `new_ref` : `value`
  `allocate` : `value` →`tag` →`unit`
  update the store

Examples with exceptions

```
try{ ...throw new Exci () ...}
catch(Exc1 e){ ...}
catch(Exc2 e){ ...}
```

```
exception JavaExc of value
try{ ...raise (JavaExc (Exci ())) ...}
with JavaExc e →
if instanceof e Exc1 then ...
else if instanceof e Exc2 then ...
else raise (JavaExc e)
```

```
while (test) {...break; ...}
code
```

```
try while test
do    ...raise Break ...done
with Break →();
code
```

More on the state

Functional interpretation of modifiable variables $x : \alpha$

$$x := a \mid \lambda(x : \alpha) \mapsto a$$

Proving $P(x)$ holds after executing program p

$$\forall x. P(\tilde{p}(x))$$

Alias problem

With different variables :

$$(x, y) := (a, b) \mid \lambda(x : \alpha)(y : \beta) \mapsto (a, b)$$

Correct when different variables correspond to different locations.
Proving $x \neq y$ after $(x, y) := (0, 1)$ is not just $0 \neq 1$

Possible solution

$$\lambda(s : \texttt{state}) \mapsto s\{x := a[s(\xi)/\xi], y := b[s(\xi)/\xi]\}$$

Reasoning on a variable z requires analysing $s\{\xi_i := e_i\}(z)$

Memory model in JAVA

- Different left-values (x, $e.f$, $e[i]$) can refer to the same location
- Variables are separate locations (call by value)
- No possible conversion between basic types and references
- Different fields correspond to different locations $a.f \neq b.g$
- $a.f$ only expression for the location corresponding to a field f

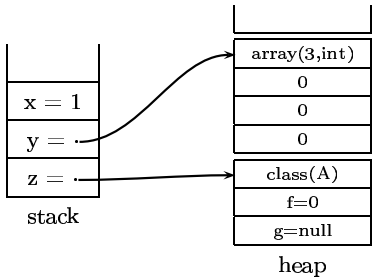
$$a.f \text{ interpreted as } \texttt{f}[\tilde{a}]$$

with `f` a new global state variable for each field f.
Following Burstall (see also Bornat, Nipkow...)

Example

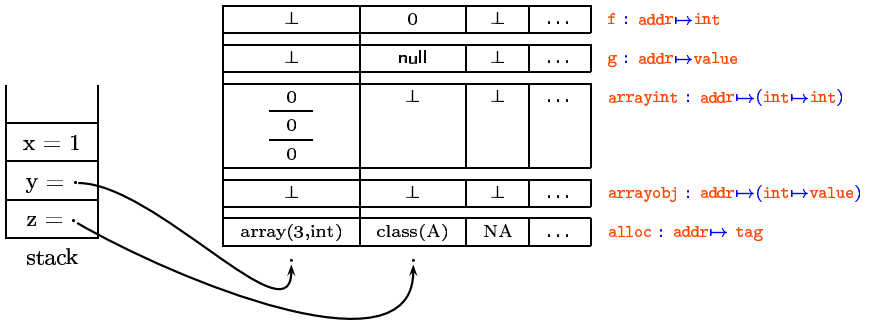
Standard JAVA memory model

```
class A = { int f; A g }
int x = 1;
int y[] = new int[3];
A z = new A;
```



Example: KRAKATOA memory model

The heap is structured in separate maps indexed by addresses, containing primitive values or references or arrays.



Outline

How to do proofs of JAVA programs ?

- JML presentation
- KRAKATOA architecture based on WHY
- Interpreting JAVA/JML programs in WHY
- Solving proof obligations

Lecture 2

KRAKATOA

KRAKATOA

JML presentation

JML : JAVA Modeling Language

http://www.jmlspecs.org

- Strongly related to the programming language:
  includes JAVA boolean expression without side effects
- Integrated to the source code : special comments, ignored by the JAVA compiler
- Different classes of specifications:
  pre and post conditions, class invariants, frame conditions, ghost variables ...
- Special additional operators (\forall, \old, \result ...)

JML example : an electronic purse

```
class Purse {

    //@ public invariant balance >= 0;
    int balance;
    /*@ public normal_behavior
      @   requires s >= 0;
      @   modifiable balance;
      @   ensures balance == \old(balance)+s;
      @*/
    public void credit(int s) {
    balance += s;
    }
```


Exceptional behavior

```
/*@ public behavior
      @  requires s >= 0;
      @  modifiable balance;
      @  ensures s <= \old(balance) && balance == \old(balance)-s;
      @  signals (NoCreditException)
      @         s > balance && balance == \old(balance);
      @*/
public void withdraw(int s) throws NoCreditException {
        if (balance >= s) { balance -= s; }
        else { throw new NoCreditException(); }
    }
```


Loops

```
public static int sqrt(int x) {
    int count = 0, sum = 1;
    /*@ loop_invariant
      @   count >= 0 && x >= count*count &&
      @   sum == (count+1)*(count+1);
      @ decreases x - sum;
      @*/
    while (sum <= x)  {
        count++;
        sum = sum + 2*count+1;
     }
    return count;
}
```

Tools using JML

Reference: *An overview of JML tools and applications*
Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. (STTT, 2005).

- Documentation (jmldoc), test (jmlunit)
- Dynamic checking (defensive code) (jmlc, jass)
- Partial automatic verification (ESC/Java(2), Chase)
- Total interactive verification (Loop, JIVE, Jack, Krakatoa)

Also JML specification of JavaCard API (E. Poll, Nijmegen)

Krakatoa

Architecture based on Why

The Why tool

A generic language for proving annotated programs
J.-C. Filliâtre, `http://why.lri.fr`

- Specification : multi-sorted predicate logic
- Body of programs : functions, references, exceptions, labels, assertions ...
- Signature of programs : extended with pre & post-conditions, + effects (read & written variables, exceptions)

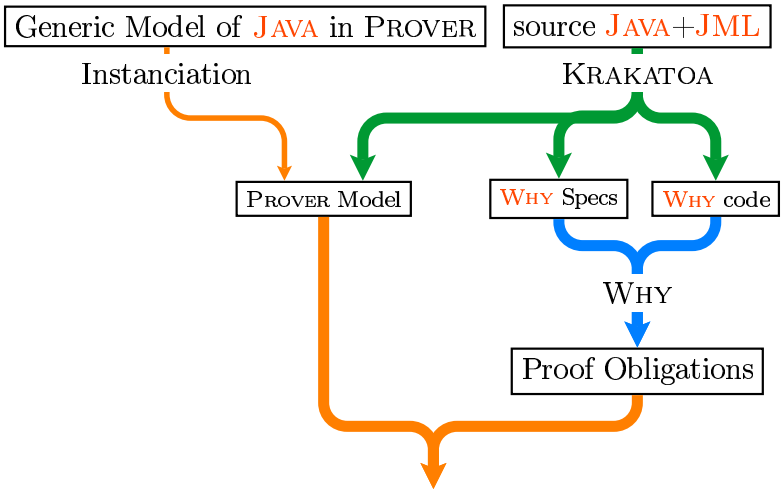
Why advantages

- A modular view of programs and specifications
- Generates sufficient proof obligations (pre, post, assertions)
- Proof obligations generated for interactive or automatic theorem provers : PVS, Coq, HOL, Mizar, Simplify, haRVey...

KRAKATOA approach

- Model the JAVA program (see before)
- Model the JML specification
- Translate JAVA/JML programs into WHY annotated programs (preserving semantics)
- Proof that the program meets its specification by generating proof obligations in WHY

KRAKATOA general architecture



KRAKATOA

WHY model of programs

WHY parametric theory

```
parameter alloc : store ref
parameter alloc_new_obj : (c:classId) → { }
   value reads alloc writes alloc
   { result≠ Null and fresh(alloc@, result)
     and typeof(alloc, result, ClassType(c))
     and store_extends(alloc@,alloc)}

external logic fresh : store, value → prop
external logic store_extends : store, store → prop
external logic Null : → value
external logic ClassType : classId → javaType
```

Body of programs

```
external parameter new_ref : store  → value
external parameter allocate : store → value → tag → store
external parameter Obj : classId → tag

let alloc_new_obj = fun (c:classId) →  { }
    let this = new_ref !alloc in
    begin alloc := allocate !alloc this (Obj c); this end
    { result≠ Null
      and fresh(alloc@, result)
      and typeof(alloc, result, ClassType(c))
      and store_extends(alloc@,alloc)
    }
```


Translation of expressions

Conditions to protect access and avoid runtime exceptions

e.f	{e≠Null} (acc !f e)
e.f=v	{e≠Null} f:=(update !f e v)
e[i]	{e≠Null ∧ 0≤i<(arraylength alloc e)}
	(array_acc !arrayint e i)
e[i]=v	{e≠Null ∧ 0≤i<(arraylength alloc e)
	∧ instanceof alloc v (arrayelemtype alloc e)}
	arrayobj:=(array_update !arrayobj e i v)

Handling methods

- Find a WHY specification for each JAVA method
  - Computes which variables are read or written (field variables, array variables, alloc ...)
  - Transforms the JML specification into pre/post conditions
- Keep a local and modular approach
- Handle partial correctness of recursive methods

WHY specification for methods

```
parameter Purse_credit_parameter :
 this:value → s:int →
   { s ≥ 0 ∧ this ≠ Null
     ∧ instanceof(alloc,this,ClassType(Purse))
     ∧ Purse_invariant(Purse_balance,this)}
   unit reads Purse_balance,alloc   writes Purse_balance
   { acc(Purse_balance,this)=acc(Purse_balance@,this)+s
      ∧ Purse_invariant(Purse_balance,this)
      ∧ modifiable(alloc@,Purse_balance@,Purse_balance,
                          value_loc(this))}
```

KRAKATOA

Solving proof obligations

The corresponding Coq theory

```
Inductive tag:Set := Obj: classId→ tag | Arr: N→ kind→ tag.
Definition store := (fmap.t tag).

Definition alive (h:store) (v:value) :=
  match v with  Null => True | Ref a => find h a ≠ None end.

Definition store_extends (h h':store) :=
  ∀ v:value, alive h v → tag_of h v = tag_of h' v.

Lemma typeof_extends_stable :
∀ (h h':store) (t:javaType) (v:value),
   typeof h v t → store_extends h h' → typeof h' v t.
```


Frame condition modeling

```
Variable A : Set
Definition memory := map.t value A.

Definition mod_loc := value → Prop.
Definition unchanged (ml:mod_loc) (v:value) := ml v.

Definition modifiable (h:store) (m m':memory) (ml:mod_loc) :=
  ∀v:value,alive h v → unchanged ml v → acc m v = acc m' v.

Definition value_loc (v: value) : mod_loc := fun w ⇒ v ≠ w.
Lemma value_loc_intro :
   ∀v1 v:value, v1 ≠ v -> unchanged (value_loc v1) v.
```


Coq theory generated for a particular program

```
Inductive classId : Set :=
  Object : classId | Math : classId | Purse : classId ...

Definition super (i:classId) : option classId :=
  match i with
  | Object => None  | Math => Some Object
  | Purse  => Some Object | ...
  end

Definition Purse_invariant (Purse_balance:memory Z) (this:value)
:= (acc Purse_balance this) >= 0.
```

Automatic proofs

- Extract an axiomatic first-order theory from the Coq model
- Use an automatic prover (mainly Simplify) in order to validate proof obligations

Good results on small programs (sorting, sets, purse ...)

Lecture 2

Conlusion

Related work

Tools with similar goals

- ESC/Java (Compaq) : only partial correctness, errors
- KeY (Chalmers, Karlsruhe) : UML specification, dynamic logic
- LOOP (Nijmegen) : shallow embedding in PVS
- JIVE (Hagen): ad-hoc axiomatic semantics, global memory, interface
- Jack (Gemplus, INRIA) : obligations originally for the B prover, nice interface

Remarks on Krakatoa

A good combination of known techniques

- A rigorous approach
- Specification and proofs are integrated in real programs
- Proofs are partly automated
- Experimented on two JavaCard applets

A very preliminary tool under development

- Many important features of Java are not (yet) covered
- The interface is not really user-friendly

Choice of architecture

- An open-source system
- Each step of translation is readable
- WHY language (functions, references and exceptions) is a powerful language for representing operational semantics
- The same architecture can be used for other input programming languages:
  CADUCEUS for C, J.-C. Filliâtre & C. Marché
- The best of each theorem provers can be used (even combined)

More on specifications

Writing apropriate specifications can be as hard as writting programs and proofs ...

The tool should help you in this process

How convenient are JML specifications ?

- Some relations are not easily defined by pure JAVA programs but would be naturally specified inductively.
  Example: A linked structure does not contains loops
- Global security properties :
  - Security automata : control the correct sequences of method calls
  - Non interference properties : we cannot infer secret information from looking at public variables

Can be checked using JAVA/JML technology
(Everest project, Sophia-Antipolis)

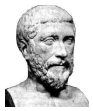
That is the end ...

See the demo on Saturday!

Formal Proofs for Computer Arithmetics

Laurent Théry
Marelle Project INRIA Sophia-Antipolis

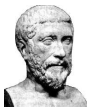
Computer arithmetics



Numbers are everywhere:

- Billing system
- Weather forecast
- Computer vision
- Cryptography
- ...

Computer arithmetics



Different flavours:

- Integer Arithmetic: $a \in \mathbb{N}$
- Rational Arithmetic: $a \in \mathbb{Q}$
- Floating-Point Arithmetic: $a \in \mathbb{F}$
- Interval Arithmetic: $[a, b] \in \mathbb{F} \times \mathbb{F}$
- Exact Arithmetic: $a \in \mathbb{R}$
- ...

Formal Proofs

Therac-25:

Arithmetic overflows could cause the software to bypass safety checks.

Patriot Missile:

The calculation of the time since boot was inacurate.

Pentium FDIV

Some division operations were wrong by a very small amount.

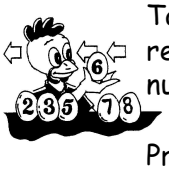
Ariane 5:

A conversion 64 bit floating-point number to 16 bit signed integer failed.

Outline

- A simple example that illustrates various aspects of formal verification.
- An overview of the formalisation of floating-point arithmetic.

An Example



Take a program that given a parameter n returns the list of the first n prime numbers

Prove it correct

Knuth: The Art of Computer Programming

Natural Numbers

```
Inductive ℕ:  Set :=
   O : ℕ
|  S : ℕ→ℕ.
```

O, S(O), S(S(O)), ...

$$P(\mathtt{O}) \wedge \forall n,\ P(n) \Rightarrow P(\mathtt{S}(n)) \Rightarrow \forall n,\ P(n)$$

Addition

```
Fixpoint + [a,b:ℕ] :  ℕ :=
match a with
      O => b
 | S a' => S(a' + b)
end.
```


Multiplication

```
Fixpoint * [a,b:ℕ] :  ℕ :=
match a with
      O => O
 | S a' => b + (a' * b)
end.
```


Divisibility

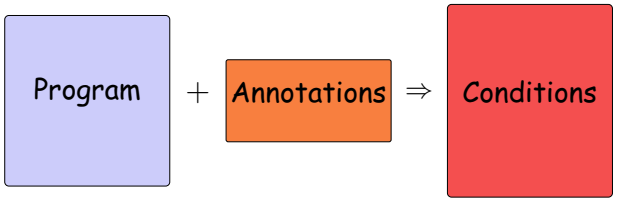
```
Definition a|b := ∃c,  b = c * a.
```


Primality

```
Definition prime(p) :=
      ∀n,  n|p ⇒ n = 1 ∨ n = p
   ∧
      p ≠ 1.
```


Program Correctness

Verification Condition Generator



Krakatoa: Java ⟶ Coq

Program Correctness

{ *Pre-conditions* }
P
{ *Post-conditions* }

Example

```
{ odd(x) }
 x = x + 2;
{ odd(x) }
```

Generated condition:

$$\forall x,\ \texttt{odd}(x) \Rightarrow \texttt{odd}(x+2)$$

Program Correctness

Loop:

```
while (C) {
{ invariant I
  variant  V
}
     P
}
```


Program Correctness

Example:

```
while (0 < i--) {
   { invariant odd(x)
     variant  i
   }
    x = x + 2
}
```

Generated conditions:

$$\forall x.\ \texttt{odd}(x) \Rightarrow \texttt{odd}(x+2)$$
$$\forall i.\ 0 \le i-1 < i$$

Knuth Algorithm

```
int[] firstPrimes(int n) {
  int[] res = new int[n];
```

{

$\forall k.\ 0 \le k < n \Rightarrow \texttt{prime}(res[k])$

$\wedge$

$\forall k, j.\ 0 \le k < j < n \Rightarrow res[k] < res[j]$

$\wedge$

$\forall k.\ \wedge \begin{matrix} 0 \le k \le res[n-1] \\ \texttt{prime}(k) \end{matrix} \Rightarrow \exists j.\ \wedge \begin{matrix} 0 \le j < n \\ res[j] = k \end{matrix}$

}

```
  return res;
}
```


Knuth Algorithm

```
int[] firstPrimes(int n) {
  int[] res = new int[n];
  int number = 2;
  boolean isPrime;
  for (int i = 0; i < n; i++) {
    while (true) {
      isPrime = true;
      for (int j = 2; j < number; j++) {
        if (number % j == 0) {
          isPrime = false;
          break;
        }
      }
      if (isPrime) break;
      number++;
    }
    res[i] = number;
    number++;
  }
  return res;
}
```

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11

$\forall p, \texttt{prime}(p) \Rightarrow 2 \le p$

$\forall n, \exists p, n < p \wedge \texttt{prime}(p)$

$\forall m\, n,\ m \mid n \wedge n \neq 0 \Rightarrow m < n$

Knuth Algorithm

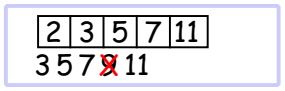
```
int[] firstPrimes(int n) {
  int[] res = new int[n];
  int number = 2;
  boolean isPrime;
  for (int i = 0; i < n; i++) {
    while (true) {
      isPrime = true;
      for (int j = 0; j < i; j++) {
        if (number % res[j] == 0) {
          isPrime = false;
          break;
        }
      }
      if (isPrime) break;
      number++;
    }
    res[i] = number;
    number++;
  }
  return res;
}
```

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11

$\forall n, 2 \le n \Rightarrow$
$(\forall p, \texttt{prime}(p) \wedge p < n \Rightarrow \neg(p \mid n)) \Rightarrow$
$\texttt{prime}(n)$

Knuth Algorithm

```
int[] firstPrimes(int n) {
  int[] res = new int[n];
  res[0] = 2;
  int number = 3;
  boolean isPrime;
  for (int i = 1; i < n; i++) {
    while (true) {
      isPrime = true;
      for (int j = 1; j < i; j++) {
        if (number % res[j] == 0) {
          isPrime = false;
          break;
        }
      }
      if (isPrime) break;
      number += 2;
    }
    res[i] = number;
    number += 2;
  }
  return res;
}
```

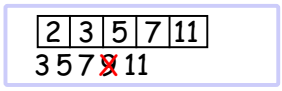


$prime(2)$

$\forall p,\ \text{prime}(p) \Rightarrow p = 2 \vee \text{odd}(p)$

Knuth Algorithm

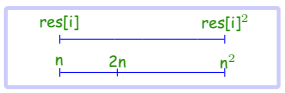
```
int[] firstPrimes(int n) {
  int[] res = new int[n];
  res[0] = 2;
  int number = 3, snum;
  boolean isPrime;
  for (int i = 1; i < n; i++) {
    while (true) {
      isPrime = true;
      snum = (int) Math.sqrt(number);
      for (int j = 1; j < i && res[j] <= snum; j++) {
        if (number % res[j] == 0) {
          isPrime = false;
          break;
        }
      }
      if (isPrime) break;
      number += 2;
    }
    res[i] = number;
    number += 2;
  }
  return res;
}
```



$\forall n\, p\, q,\ n = p * q \Rightarrow p \leq \sqrt{n} \vee q \leq \sqrt{n}$

Knuth Algorithm

```
int[] firstPrimes(int n) {
  int[] res = new int[n];
  res[0]=2;
  int number=3, snum;
  boolean isPrime;
  for (int i=1; i<n; i++) {
    while (true) {
      isPrime=true;
      snum = (int)Math.sqrt(number);
      for(int j=0; res[j]<= snum; j++) {
        if (number%res[j]==0) {
          isPrime=false;
          break;
        }
      }
      if (isPrime) break;
      number+=2;
    }
    res[i]=number;
    number+=2;
  }
  return res;
}
```



$\forall n,\ 2 \leq n \Rightarrow$
$\exists p,\ \text{prime}(p) \wedge n < p < 2 * n$

Bertrand Postulate

For n greater than 2, there is always at least one prime number strictly between n and $2n$.

Proof by Contradiction (Erdös)

Upper Bound: $\binom{2n}{n} < (2n)^{\sqrt{2n}/2-1} 4^{2n/3}$

Lower Bound: $4^n \leq 2n\binom{2n}{n}$

Necessary Condition: $4^{n/3} < (2n)^{\sqrt{2n}/2}$

Example of properties

Upper Bound on the Product of Prime Numbers

$$\prod_{p \leq n} p < 4^n$$

By Strong Induction on n

$2 < 4^2$

If n is odd, $\prod_{p \leq n+1} p = \prod_{p \leq n} p < 4^n < 4^{n+1}$

If n is even, $\prod_{p \leq 2m+1} p = (\prod_{p \leq m+1} p) \quad (\prod_{m+1 < p \leq 2m+1} p)$

$\prod_{p \leq 2m+1} p < 4^{m+1} \quad (\prod_{m+1 < p \leq 2m+1} p)$

$\prod_{p \leq 2m+1} p < 4^{m+1} \binom{2m+1}{m+1}$

$\prod_{p \leq 2m+1} p < 4^{m+1} 4^m$

$\prod_{p \leq 2m+1} p < 4^{2m+1}$

Necessary Condition

$$4^{n/3} < (2n)^{\sqrt{2n}/2}$$

Logarithmic Scale

$$\frac{n}{3} ln(4) < \frac{\sqrt{2n}}{2} ln(2n)$$

Simplification:

$$\sqrt{8n}\, ln(2) - 3\, ln(2n) < 0$$

Function Analysis

Inequality: $\sqrt{8n}\,ln(2) - 3\,ln(2n) < 0$

Function: $f(x) = \sqrt{8x}\,ln(2) - 3ln(2x)$

Evaluation: $f(2^7) = 2^5 ln(2) - 3.2^3 ln(2) > 0$

Derivative: $f'(x) = \dfrac{\sqrt{2x}\,ln(2) - 3}{x}$

Conclusion: For $n \geq 2^7$, ok.

Remaining cases

The theorem is true for $n < 2^7$

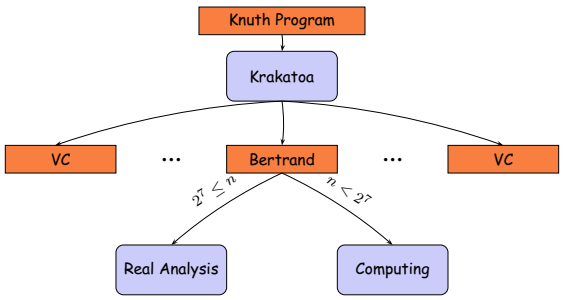
Computing inside Coq:

Write a program that checks the property

Prove it correct

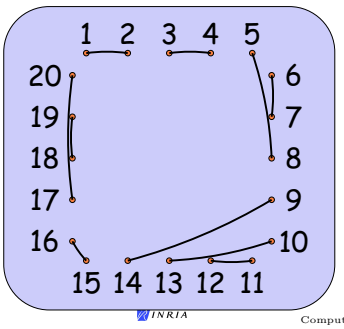
Run it

Proof Map



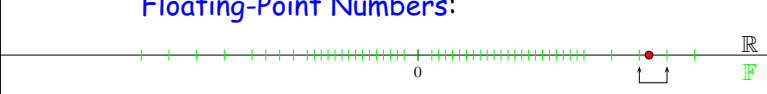
Little Problem

Sort the numbers from 1 to $2n$ in pairs (a_i, b_i) such that each $a_i + b_i$ is prime ?



Floating-Point Arithmetic

Floating-Point Numbers:



Correct Rounding (IEEE 754):

$$a \oplus b = \circ(a + b)$$

Floating-Point Numbers

sign exponent mantissa

Normal ($e > 0$): $(-1)^s \beta^{e-B}(1 + \sum_{i=1}^{p-1} f_i \beta^{-i})$

Subnormal ($e = 0$): $(-1)^s \beta^{1-B} \sum_{i=1}^{p-1} f_i \beta^{-i}$

$$x \ominus y = 0 \Longleftrightarrow x = y$$

Floating-Point Numbers

Number as pair:

Definition $\mathbb{F}$:= $\mathbb{Z} \times \mathbb{Z}$.

Projections:

Definition m(p) := let (x,_) = p in x.

Definition e(p) := let (_,y) = p in y.

Floating-Point Numbers

Value:

Definition v(p) := m(p) $*$ $\beta^{\mathtt{e}(p)}$.

Equivalence:

Definition $p \simeq q$:= v(p) = v(q).

Bound

Bound as pair:

Definition $\mathbb{B}$:= $\mathbb{N} \times \mathbb{N}$.

Projections:

Definition M(b) := let (x,_) = b in x.

Definition E(b) := let (_,y) = b in y.

Bound

Bounded number:

Definition $\mathcal{B}_b(p)$:= $|\mathtt{m}(p)| \leq \mathtt{M}(b) \wedge -\mathtt{E}(b) \leq \mathtt{e}(p)$.

Rounding

Rounded Mode:
Toward +∞, Toward -∞, Toward 0, Closest.

Rounded as a Predicate:

$\mathcal{R}$: $\mathbb{R} \rightarrow \mathbb{F} \rightarrow$ Prop

Rounding: Min Max

Elements on the left and on the right:

Definition isMin(r,p) :=
$\mathcal{B}_b(p) \wedge p \leq r \wedge \forall q,\, \mathcal{B}_b(q) \wedge q \leq r \Rightarrow q \leq p$.

Definition isMax(r,p) :=
$\mathcal{B}_b(p) \wedge r \leq p \wedge \forall q,\, \mathcal{B}_b(q) \wedge r \leq q \Rightarrow p \leq q$.

Rounding chooses one of those:

Definition MinOrMax($\mathcal{R}$) :=
$\forall p\, r,\, \mathcal{R}(r,p) \Rightarrow \mathtt{isMin}(r,p) \vee \mathtt{isMax}(r,p)$.

Rounding: Monotone

Rounding is non decreasing:

`Definition Monotone(`$\mathcal{R}$`) :=`
$\forall p_1\, p_2\, r_1\, r_2,\ \mathcal{R}(r_1, p_1) \wedge \mathcal{R}(r_2, p_2) \wedge r_1 < r_2 \Rightarrow p_1 \leq p_2.$

Rounding is total:

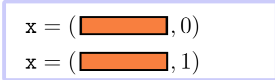
`Definition Total(`$\mathcal{R}$`) :=` $\forall r,\ \exists p,\ \mathcal{R}(r, p).$

Rounding is compatible:

`Definition Compatible(`$\mathcal{R}$`) :=`
$\forall p_1\, p_2\, r,\ \mathcal{R}(r, p_1) \wedge \mathcal{B}_p(p_2) \wedge p_1 \simeq p_2 \Rightarrow \mathcal{R}(r, p_2).$

Example: Malcolm Test

```
float malcolmTest() {
  float x = 1;
  float y = 1;
  while (((x + 1) - x) == 1) {
    x = 2 * x;
  }
  while (((x + y) - x) != y) {
    y = y + 1;
  }
  return y;
}
```



$1 = (1, 0) \qquad \beta = (1, 1)$
$(m + 1, e) \ominus (m, e) = (1, e)$
$(m, e) \oplus (1, e) \simeq (m + 1, e)$
$(m_1, e) \ominus (m_2, e) = (m_1 - m_2, e)$

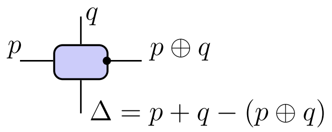
Example: Expansion

Ordered list of non-overlapping floats:

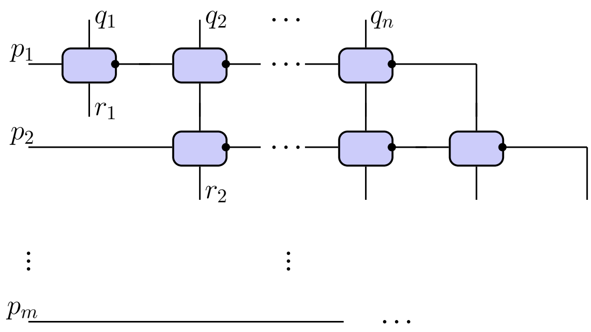
110110001110000000000011111110000010

(11011,29); (11100,21); (11111,9); (11000,4); (10000,-3)

Addition:



Example: Expansion

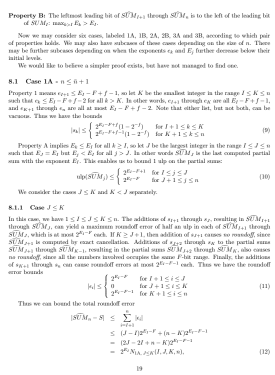


Pencil and Paper Proof

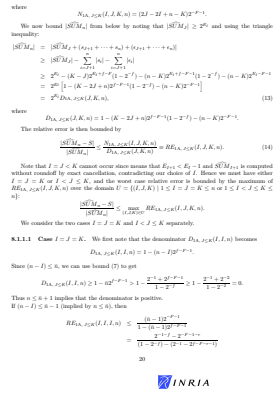
J. Demmel and Y. Hida,
Accurate floating point summation

19 page proof

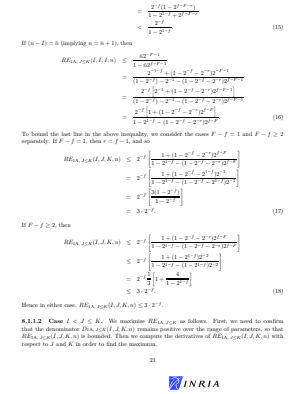
Pencil and Paper Proof



Pencil and Paper Proof



Pencil and Paper Proof



Benefits

Finding bugs in proof

typos

missing cases

missing side-conditions

large versus strict inequalities

...

$\longrightarrow$ Effective?

Benefits

Library of validated facts

statements as general as possible

explicit side-conditions

Example (Sterbenz)

if $\frac{1}{2}y \leq x \leq 2y$ then $x \ominus y = x - y$

Benefits

Improved Results

Example (Fast Two Sum)

$$b \ominus ((a \oplus b) \ominus a) = (a + b) - (a \oplus b)$$

if $|a| \leq |b|$

if $e_a \leq e_b$ where $a = \overline{m}_a 2^{e_a}$ and $b = \overline{m}_b 2^{e_b}$

if $e_a \leq e_b$ where $a = m_a 2^{e_a}$ and $b = m_b 2^{e_b}$

Benefits

Alternative Proofs

Different metric

Completly new proofs

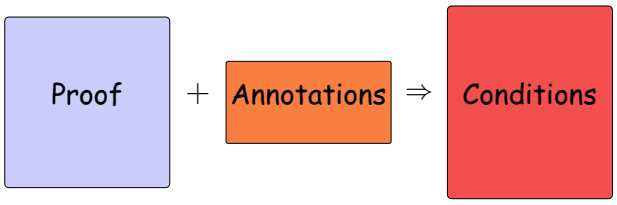
Challenges

Pencil and Paper Proof $+$ Formal Proof

Single Program $\rightsquigarrow$ Library

Proof Checking $\rightsquigarrow$ Computer Aided Proof

Pencil and Paper Proof



CYP tool: text $\longrightarrow$ Coq

Colouring Proof

Theorem **Sterbenz**: Let p and q such that $bounded(p)$ and $bounded(q)$, if $q/2 \leq p \leq 2 * q$ then $bounded(p-q)$.

The proof proceeds like this. First of all, we restrict ourselves to the case $q \leq p \leq 2 * q$ because of the symmetry of the problem. For the exponent, by definition of the substraction, $e(p-q) = min(e(p), e(q))$, so $e(p-q) \geq -E(bound)$ since both p and q are bounded. For the mantissa, we do a case analysis on the value of $\min(e(p), e(q))$. If $min(e(p), e(q)) = e(q)$, the initial equation can be rewritten as $0 \leq p - q \leq q$ and since $e(p-q) = e(q)$, we obtain $0 \leq m(p-q) \leq m(q)$. As $bounded(q)$, we have $0 \leq m(p-q) < M(bound)$. Similarly if $min(e(p), e(q)) = e(p)$, we can rewrite the initial equation as $0 \leq p - q \leq q \leq p$ and since $e(p-q) = e(p)$ we have $0 \leq m(p-q) \leq m(p)$. In both cases we have $0 \leq m(p-q) < M(bound)$. The mantissa and the exponent are then bounded, so we have $bounded(p-q)$.

Related Works

Floating point arithmetic

Barrett (Z), Miner (Pvs), Russinoff (Acl2), Harrison (Hol), Boldo & al (Coq)

Interval arithmetic

Melquiond (Coq)

Multiprecision arithmetic

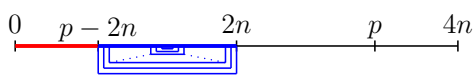
Bertot & al (Coq), Bondyfalat (Coq)

Exact arithmetic

Ciaffaglione & al (Coq), Lester & al (Pvs), Niqui (Coq)

Little problem

Sort the numbers from 1 to $2n$ in pairs (a_i, b_i) such that each $a_i + b_i$ is prime ?



p is prime

$p - 2n$ is odd

$p - 2n - 1$ is even

Dependently Typed Programming in Cayenne

or

Why does Agda look so strange?

Lennart Augustsson

lennart@augustsson.net

www.dependent-types.org

An example

We want to write a small program that does bracket abstraction for λ-calculus.

```
data Exp = App Exp Exp
         | Lam Sym Exp
         | Var Sym
type Sym = String
```

The function we want will remove all λ-expressions and replace them with the S, K, and I combinators. We could give it this type:

```
abstractVars :: Exp -> Exp
```

This does not reflect that all `Lam` constructors are gone.

Bracket abstraction

Remove all λ-expressions by using combinators.

$$I\ x = x$$

$$K\ x\ y = x$$

$$S\ f\ g\ x = (f\ x)\ (g\ x)$$

Every lambda term is replaced by its bracket abstraction:

$$\lambda\ x.e = [x]e$$

$$[x]x = I$$

$$[x]y = K\ y$$

$$[x](f\ e) = S\ ([x]f)\ ([x]e)$$

An example

Use a different result type.

```
abstractVars :: Exp -> LamFreeExp
```

Use a "subtype"

```
type LamFreeExp =
    sig exp :: Exp
        lf  :: LamFree exp
```

An example, a little logic

```
data Absurd =

absurd :: (a :: #) |-> Absurd -> a
absurd i = case i of { }

data Truth = truth

data (/\) a b = (&) a b
```

An example

Describe what it means to be `LamFree`:

```
LamFree :: Exp -> #
LamFree (App f a) = LamFree f /\ LamFree a
LamFree (Lam _ _) = Absurd
LamFree (Var _) = Truth
```

An example

We are all set, just proceed as usual:

```
abstractVars :: Exp -> LamFreeExp
abstractVars e@(Var _) =  struct { exp = e; lf = truth }
abstractVars (App f a) =
    let f' = abstractVars f
        a' = abstractVars a
    in  struct exp = App f'.exp a'.exp
               lf  = f'.lf & a'.lf
abstractVars (Lam x e) =
    let e' = abstractVars e
    in  abstractVar x e'.exp e'.lf
```

An example

```
abstractVar :: Sym -> (e :: Exp) -> LamFree e -> LamFreeExp
abstractVar s (App f a) (lf & la) =
    let f' = abstractVar s f lf
        a' = abstractVar s a la
    in  struct exp = App (App S f'.exp) a'.exp
               lf  = (truth & f'.lf) & a'.lf
abstractVar s (Lam _ _) l = absurd l
abstractVar s e@(Var x) l =
    if (s == x)
       (struct {exp = I; lf :: LamFree exp = truth})
       (struct {exp = App K e; lf :: LamFree exp = truth & l})

S = Var "S"
K = Var "K"
I = Var "I"
```

Cayenne design goals

- A **programming** language with dependent types.
- "First class" types.
- Few basic concepts.
- No *top level*.
- "Pure", i.e., the β-rule is valid.
- Uniform way to define and name things.
- Staged execution, i.e., compiled.
- All used variables must be explicitly bound.

Cayenne design goals

Lesser goals:

- No silly case restrictions on names.
- Compiled with same efficiency as Haskell.
- Proofs possible.
- Haskell like.

No top level

Many languages have a *top level* that is different. E.g., C only allows function definitions on the top level, Haskell only allows type definitions on the top level.

I want to take any program fragment and move it to where it belongs.

Example:

```
data BT a = Leaf | Node (BT a) a (BT a)
sortBy :: (a -> a -> Ordering) -> List a -> List a
sortBy cmp xs = ...
```

If the binary tree type is only used in `sortBy` it should be like this.

```
sortBy :: (a -> a -> Ordering) -> List a -> List a
sortBy cmp xs =
    let data BT a = Leaf | Node (BT a) a (BT a)
    in  ...
```

Staged execution

I want a phase distinction; execution has two phases:

- Compile time: type checking and maybe more.
- Run time: actual program execution.

Any (closed) expression should be possible to compile.

The type of an expression, A, should be the only thing needed to compile an expression, B, that is using A.

The function type

The function type is easy, we only need some syntax for dependent functions. Most of the syntax comes from Haskell.

```
\ (x :: t) -> e  ::  (x :: t) -> s
```

Application looks as usual.

```
f e  ::  s
```

If the function is not actually dependent on `x` we can use the usual Haskell syntax.

```
\ (x :: t) -> e  ::  t -> s
```

We can also usually leave out the type in the term.

```
\ x -> e  ::  t -> s
```

The function type, hidden arguments

Cayenne has the ability to "hide" arguments. This means that they need not be given when a function is applied, if the type checker can deduce them.

```
id :: (a :: #) |-> a -> a
id |a x = x

... id 5 ...
```

The sum type

We need sum types. Can we take Haskell's data type definitions?

NO

```
data T = A | B | C
```

Haskell's data type definition forces the type to have a name.

Naming things should be uniform, so if a type has a name it should be given in the same way as for anything else, possibly no name at all.

```
T :: #
T = data A | B | C
```

The sum type, constructors

Constructors are written in a peculiar way:

```
C@t  ::  t
```

Example:

```
True@(data False | True)
```

Contructors do not have any scope (just like record labels), they are only meaningful with an `@`.

The sum type, weird stuff

What's the type of this expression?

```
let T = data A | B | C
in  A@T
```

You could give a type in terms of T, but it's not in scope. I find that bizarre.

The Cayenne answer to the question is:

```
let T = data A | B | C
in  A@T                    ::  data A | B | C
```

Structural equivalence

Types are compared with structural equivalence rather than name equivalence (unlike, e.g., Haskell).

Rationale: (A) Types do not have to have names. (B) Name equivalence does not work with the β-rule.

Example:

```
List a = data Nil | Cons a (List a)
```

Unfold `List`

```
List a = data Nil | Cons a ( data Nil | Cons a (List a))
```

According to the β-rule principle these must be considered equal.

Btw, this is also equivalent:

```
List a = Fix (\ l ->  data Nil | Cons a l)
```

The sum type, case

The case construct looks mostly familiar:

```
case xs of
(Nil) -> ...
(x : xs) -> ...
```

Contructor patterns must have parenthesis around them. This is to distinguish them from variable patterns. (There is no case distinction like in Haskell.)

The dependent type system shows up in that the case arms can have different types.

```
case b of                    case b of
(False) -> 1           ::    (False) -> Int
(True) -> "Hello"            (True) -> String
```

The sum type, with plenty of sugar

To make life simpler, Cayenne allows

```
data T = C1 | C2 ...
```

which is equivalent to

```
T = data C1 | C2 ...
C1 = C1@T
C2 = C2@T
...
```

Furthermore, function definitions can be written with pattern matching (like in Haskell) instead of λ and case.

The type of types

The type of types is named `#` (because `*` is used for multiplication).

```
# :: #1 :: #2 :: ...
```

This isn't the whole story...

The record type

Records with named fields are very, very useful in programming. Their omission from the original Haskell definition is something of a mystery.

```
struct              sig
  x1 = e1     ::      x1 :: t1
  ...                 ...
  xn = en             xn :: tn
```

Record selection uses the ordinary `.´ notation.

```
s.xn  ::  tn
```

The record type

Should the record type be dependent in some way?

YES!

Consider the type theory type:

∃ x ε A. P(x)

which has elements of the form:

(e, P(e))

We need this in Cayenne records too.

```
sig
  x :: A
  p :: P(x)
```

Generalize: Let all labels be in scope in all types.

The record type

Since the labels have to be bound in the `sig` it's natural to have it the same way in a `struct`.

Example:

```
struct
  x = 5
  y = x + 2     -- i.e., y = 7
```

This interacts well with types too.

```
struct
  Coord = sig { x :: Int; y :: Int }
  origin = struct { x = 0; y = 0 }
  ...
```

let expressions

The `struct` expression is similar to the definition part of `let` expressions in, e.g., Haskell.

Cayenne defines `let` in terms of `struct`. (The label `r` should be fresh.)

```
let                          (struct
    x1 = e1                      x1 = e1
    ...              =           ...
    xn = en                      xn = en
in  e                            r = e
                             ).r
```

open expressions

A very convenient feature of Pascal (and other languages) is to "open" a record and bring its labels into scope. Cayenne defines syntactic sugar for this too. (The variable `r` should be fresh.)

```
                                        let r = e
                                            x1 = r.x1
open e use x1, ... xn in e'     =           ...
                                            xn = r.xn
                                        in  e'
```

Example:

```
open coord use x, y, z
in sqrt(x^2 + y^2 + z^2)
```

Modules

In Cayenne the record type has all the power of modules in most languages. The `sig` is used for module signatures, and `struct` for module implementation. Furthermore, ordinary functions can be used instead of (ML) functors.

```
STACK = sig
  Stack   :: # -> #
  empty   :: (a :: #) |-> Stack a
  push    :: (a :: #) |-> a -> Stack a -> Stack a
  pop     :: (a :: #) |-> Stack a -> Stack a
  top     :: (a :: #) |-> Stack a -> a
  isEmpty :: (a :: #) |-> Stack a -> Bool
```

BUT, this doesn't always work as intended...

Modules, abstract and concrete

Consider the following module for booleans.

```
struct
  Bool = data False | True
  not = ...
  ...
```

This module would have the signature

```
sig
  Bool :: #
  not :: Bool -> Bool
  ...
```

That's not right. Where are the constructors?

Modules, abstract and concrete

We can export values for them.

```
struct
  Bool = data False | True
  False = False@Bool
  True = True@Bool
  not = ...
  ...
```

This module would have the signature

```
sig
  Bool :: #
  False :: Bool
  True :: Bool
  not :: Bool -> Bool
  ...
```

That's still not right. Pattern matching would not work since there is no indication that `Bool` is actually a data type.

Modules, abstract and concrete

We need something more, we need the signature to actually tell us the *definition* of `Bool`.

Enter `concrete` and `abstract`!

```
struct
  concrete Bool = data False | True
  abstract not = ...
  ...
```

This module would have the signature

```
sig
  Bool :: # = data False | True
  not :: Bool -> Bool
  ...
```

The `concrete` and `abstract` qualifiers can be applied to any kind of fields in a record. Sensible defaults are used if they are not given.

Modules, public and private

When making modules you often need auxilliary definitions that should not be part of the visible interface of the module.

So we extend the record syntax even more, with `public` and `private`.

```
struct
  private x = 12
  public y = x * 2
```

This module would have the signature

```
sig
  y :: Integer
```

As usual, sensible defaults are provided.

Modules

Recall

```
STACK = sig
  Stack   :: # -> #
  empty   :: (a :: #) |-> Stack a
  push    :: (a :: #) |-> a -> Stack a -> Stack a
  pop     :: (a :: #) |-> Stack a -> Stack a
  top     :: (a :: #) |-> Stack a -> a
  isEmpty :: (a :: #) |-> Stack a -> Bool
```

We can now give an implementation

```
ListStack :: STACK
ListStack = struct
    abstract Stack = List
    empty = Nil
    push = (:)
    pop = tail
    top = head
    isEmpty = null
```

Modules, functors

A signature for queues

```
QUEUE = sig
  Queue   :: # -> #
  empty   :: (a :: #) |-> Queue a
  enqueue :: (a :: #) |-> a -> Queue a -> Queue a
  dequeue :: (a :: #) |-> Queue a -> Queue a
  first   :: (a :: #) |-> Queue a -> a
```

A "functor" to turn stacks into queues (very badly).

```
SQ :: STACK -> QUEUE
SQ s = struct
  open s use Stack, empty, push, pop, top, isEmpty

  abstract Queue = Stack
  empty = empty
  enqueue x xs = app xs x
  dequeue xs = pop xs
  first xs = top xs
  private
  app :: (a :: #) |-> Stack a -> a -> Stack a
  app xs y =
    if (isEmpty xs)
       (push y empty)
       (push (top xs) (app (pop xs) y))
```

Modules

A signature for stacks more in the style of, e.g., Oberon

```
Stack (a :: #) =  sig
  T       :: #
  empty   :: T
  push    :: a -> T -> T
  pop     :: T -> T
  top     :: T -> a
  isEmpty :: T -> Bool
```

```
mkListStack :: (a :: #) |-> Stack a
mkListStack |a =  struct
  T = List a
  empty = Nil
  push = (:)
  pop = tail
  top = head
  isEmpty = null
```

Modules in the world

To make modules reusable they need to have a name that is actually mapped to some external storage so they can be accessed by different programs.

Cayenne is similar to Java in how this is done.

```
module a$global$identifier = e
```

This defines a "module" in the global name space, named `a$global$identifier` .

Cayenne programs may contain free module identifiers. (They are checked at compile time, of course.)

```
... System$Integer.(+) 30 12 ...
```

A named module is a compilation unit. In fact, any kind of expression can be named, not just a `struct`.

A very simple evaluator

Consider a tiny language of typed expressions:

```
data Expr = EBool Bool | EInt Int
          | EAdd Expr Expr | EAnd Expr Expr | ELE Expr Expr
data Type = TBool | TInt
```

It has the usual typing rules:

$$\frac{}{i:\mathrm{Int}} \qquad \frac{}{b:\mathrm{Bool}} \qquad \frac{x:\mathrm{Int} \quad y:\mathrm{Int}}{x+y:\mathrm{Int}} \qquad \frac{x:\mathrm{Int} \quad y:\mathrm{Int}}{x<=y:\mathrm{Bool}} \qquad \frac{x:\mathrm{Bool} \quad y:\mathrm{Bool}}{x\&y:\mathrm{Bool}}$$

A very simple evaluator

In Haskell (without GADTs) we would have to write an evaluator like this:

```
data Value = VBool Bool | VInt Int
eval :: Expr -> Value
eval (EBool b) = VBool b
eval (EInt i) = VInt i
eval (EAdd x y) =
    case (eval x, eval y)  of
    (VInt x', VInt y') -> VInt (x' + y')
    _ -> error "eval"
...
```

The wrapping and unwrapping of the values is inefficient. We would like to write the following, but it's not well typed.

```
eval (EBool b) = b
eval (EInt i) = i
eval (EAdd x y) = (eval x) + (eval y)
...
```

A very simple evaluator

So we can try something better in Cayenne. How about?

```
eval :: (e :: Expr) -> TypeOf e
eval (EBool b) = b
eval (EInt i) = i
eval (EAdd x y) = (eval x) + (eval y)
...
```

Well, this doesn't work, because not all expressions are well typed. So we need to express the when an expression is well typed.

```
HasType :: Expr -> Type -> #
HasType (EBool _)     (TBool) = Truth
HasType (EInt _)      (TInt)  = Truth
HasType (EAdd e1 e2) (TInt)  = HasType e1 TInt  /\ HasType e2 TInt
HasType (EAnd e1 e2) (TBool) = HasType e1 TBool /\ HasType e2 TBool
HasType (ELE  e1 e2) (TBool) = HasType e1 TInt  /\ HasType e2 TInt
HasType _             _       = Absurd
```

A very simple evaluator

Now we can write an evaluator, given a proof that the term is well typed.

```
eval :: (e :: Expr) -> (t :: Type) -> HasType e t -> Decode t
eval (EBool b)    (TBool) p          = b
eval (EInt  i)    (TInt)  p          = i
eval (EAdd e1 e2) (TInt)  (p1 & p2) = eval e1 TInt  p1 +  eval e2 TInt  p2
eval (EAnd e1 e2) (TBool) (p1 & p2) = eval e1 TBool p1 && eval e2 TBool p2
eval (ELE  e1 e2) (TBool) (p1 & p2) = eval e1 TInt  p1 <= eval e2 TInt  p2
eval _            _       p          = absurd p

Decode :: Type -> #
Decode (TBool) = Bool
Decode (TInt)  = Int
```

Where do we get the proof? Well, from a type checker, of course.

This can be extended to deal with variables.

A small equality proof

Cayenne has special syntax for equality proofs.

```
(++) :: (a :: #) |-> List a -> List a -> List a
(++) (Nil) ys = ys
(++) (x : xs) ys = x : (xs ++ ys)

appendNilP :: (a :: #) |-> (xs :: List a) ->
        xs ++ Nil === xs
appendNilP (Nil) =
        Nil ++ Nil                          ={ DEF }=
        Nil
appendNilP (x : xs) =
        (x:xs) ++ Nil                       ={ DEF }=
        x:(xs ++ Nil)                       ={ appendNilP xs }=
        x:xs                                ={ DEF }=
        Nil ++ (x:xs)
```

An example with no proofs

In C there is a very useful function, printf, which takes a varying number of arguments of varying types. I want it!

```
-- Haskell version  WRONG
printf fmt = pr fmt "" where
 pr ""           res = res
 pr ('%':'d':s) res = \i -> pr s (res ++ show i)
 pr ('%':'s':s) res = \s -> pr s (res ++ s)
 pr ('%': c :s) res =       pr s (res ++ [c])
 pr (     c :s) res =       pr s (res ++ [c])
```

Using it:

```
printf "%d(%d)" :: Int -> Int -> String
printf "hello %s!" :: String -> String
```

An example with no proofs

```
PrintfType :: String -> #
PrintfType ""            = String
PrintfType ('%':'d':cs) = Int    -> PrintfType cs
PrintfType ('%':'s':cs) = String -> PrintfType cs
PrintfType ('%': _ :cs) =           PrintfType cs
PrintfType ( _ :cs)     =           PrintfType cs

printf :: (fmt::String) -> PrintfType fmt
printf fmt = pr fmt ""

pr :: (fmt::String) -> String -> PrintfType fmt
pr ""           res = res
pr ('%':'d':cs) res = \ i -> pr cs (res ++ show i)
pr ('%':'s':cs) res = \ s -> pr cs (res ++ s)
pr ('%': c :cs) res =        pr cs (res ++ (c : Nil))
pr (c:cs)       res =        pr cs (res ++ (c : Nil))
```

Conclusions

- Cayenne was reasonable successful design (in my opinion).
- It needs more work to become a useful programming language.
- Things I would do differently next time:
  - The language should have Agda's `idata`.
  - Stratified universes are just a pain for programming, use `#:: #`. (And use global data flow analysis to get rid of types and proofs.)
  - Type error messages need to be much better.
- I want to see more examples that are totally proof free, but uses dependent types in an essential way (like printf). There are many examples of dependent vector sizes, but they usually require a complicated constraint solver.

Try it!

Cayenne can be found at

www.dependent-types.org

All you need is GHC to compile and run programs.

Types Summer School
Gothenburg Sweden August 2005

Dependently Typed Programming

Benjamin Grégoire
INRIA Sophia Antipolis, France

Lecture 1:
Conversion Rule

How to do formal proofs:

- A nice theory (Type Theory)
- A nice implementation:
  - A proof checker (type checker)
  - A proof assistant
- A nice user

Subject: Conversion rule

1. User point of view:
   Why we need it and how we can use it?
2. Implementor point of view:
   - Type inference Algorithm with conversion rule
   - How to get an efficient conversion test

Why we need it?

Because we do not want to do large and boring proofs

$2 + 2 = 4$ in a deduction style:

$$\dfrac{\dfrac{\overline{\text{eqTrans}} \quad \overline{2+2 = S(1+2)}}{S(1+2) = 4 \Rightarrow 2+2 = 4} \qquad \dfrac{\overline{\text{eqS}} \qquad \dfrac{\dfrac{\overline{\text{eqTrans}} \quad \overline{1+2 = S(0+2)}}{S(0+2) = 3 \Rightarrow 1+2 = 3} \qquad \dfrac{\overline{\text{eqS}} \quad \overline{0+2 = 2}}{S(0+2) = 3}}{1+2 = 3}}{S(1+2) = 4}}{2+2 = 4}$$

eqS : $x = y \Rightarrow Sx = Sy$

eqTrans : $x = y \Rightarrow y = z \Rightarrow x = z$

How to prove $2+2=4$

Computational style: Replace the axioms on addition by rewriting rules:

$$\begin{array}{rcl} 0+m & \longrightarrow & m \\ Sn+m & \longrightarrow & S(n+m) \end{array}$$

$$2+2 \longrightarrow S(1+2) \longrightarrow SS(0+2) \longrightarrow SS(2)$$

Reason modulo rewriting rules:

$$\frac{4=4 \quad 2+2 \stackrel{*}{\longrightarrow} 4}{2+2=4}$$

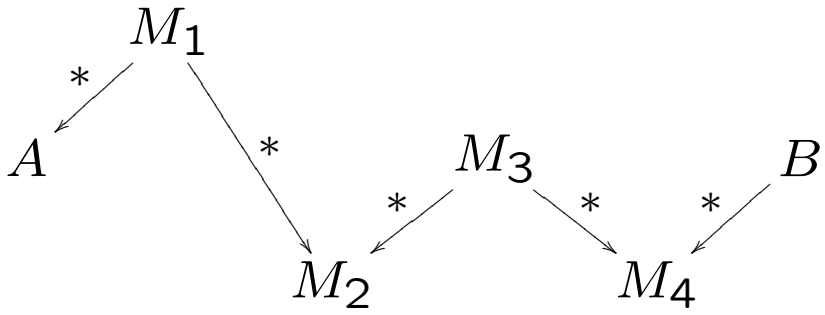
Conversion rule in PTS

A simple set of rewriting rules: reduction rules on terms (functional programs)

$$(\lambda x{:}T.\ M)\ N \xrightarrow{\beta} M[x := N]$$

With inductive definitions: reduction rule for pattern matching and fixpoint

$\approx$ ::= reflexive, symmetric and transitive closure of the reduction rules (β, ι, ...)



$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A \approx B}{\Gamma \vdash M : B}\,[\text{Conv}]$$

Encoding rewriting rule of addition

$$\begin{aligned} 0 + m &\longrightarrow m \\ Sn + m &\longrightarrow S(n + m) \end{aligned}$$

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

Fixpoint plus (n m : nat) {struct n} : nat :=
  match n with
  | O => m
  | S n1 => S (plus n1 m)
  end.
```

$$\begin{aligned} \texttt{plus O m} &\xrightarrow{*} \texttt{m} \\ \texttt{plus (S n) m} &\xrightarrow{*} \texttt{S(plus n m)} \end{aligned}$$

$2 + 2 = 4$ **again!**

$$\dfrac{\dfrac{\dfrac{}{\vdash \texttt{refl\_eq} : \forall x{:}\texttt{nat}.\ x = x} \quad \dfrac{}{\vdash 4 : \texttt{nat}}}{\vdash \texttt{refl\_eq}\ 4 : 4 = 4} \qquad 4 = 4 \approx 2 + 2 = 4}{\vdash \texttt{refl\_eq}\ 4 : 2 + 2 = 4}\,[\text{Conv}]$$

- The reduction steps do not appear in the proof
  $\Rightarrow$ The proof is small : `refl_eq 4`

- Reduction steps appear in the type checking
  $\Rightarrow$ Cost remains in proof checking (in the conversion test)

If we reduce this cost, we get:

- Small proofs

- Quickly type checked

Reflexivity

- A predicate $P : T \to \texttt{Prop}$
- A decision procedure $f : T \to \texttt{bool}$
- A correctness lemma $C : \forall x{:}T.\ f\ x = \texttt{true} \to P\ x$

Assume $f\ a$ reduce to $\texttt{true}$, a proof of $P\ a$ is $C\ a\ (\texttt{refl\_eq true})$

$$\dfrac{\dfrac{\vdots}{\vdash C\ a : f\ a = \texttt{true} \to P\ a} \qquad \dfrac{\vdash \texttt{refl\_eq true} : \texttt{true} = \texttt{true} \qquad \texttt{true} = \texttt{true} \approx f\ a = \texttt{true}}{\vdash \texttt{refl\_eq true} : f\ a = \texttt{true}}}{\vdash C\ a\ (\texttt{refl\_eq true}) : P\ a}$$

Example: primality proof

Pocklington criteria:

Let n be a positive integer, if

- $n - 1 = q.p_1 \dots p_t$ where p_i are prime numbers
- there exits a such that $\begin{cases} a^{n-1} = 1 (\text{mod } n) \\ \gcd(a^{\frac{n-1}{p_i}} - 1, n) = 1 \text{ for } i = 1 \dots t \end{cases}$
- $p_1.p_2 \dots p_t \geq \sqrt{n}$

then n is prime

Formal proof by Martin Oostdijk and Olga Caprotti

Using deduction style the proof of

20988936657440586486151264256610222593863921

take 18509 lines.

Can we use reflexivity?

18th Mersenne number: $2^{3217} - 1$

2591170860132026277624676792244153094181887553125
4273039749231618740192665863620862012095168004834 06
5506952417331941774416895092388070174103777095975 12
0423130666240829163535179523111861548622656045476 91
1275958487756105687579311910177114088262521538490 35
8304011850721164247474618230314713983402292880745 45
6779079410372882358207058923510684338829868886166 58
6502809276920803396058693087905004095037098759021 19
0183719916209940025689351131365488297391126567973 03
2419865172501164127035097054277734779723498216764 43
4466683831193225400996489940517902416240565190544 83
6908096160616257430423617218633394158524264312087 37
2665919620617535357488928945996291951830826218608 53
4009379328394202618665861425032514507730962742353 76
8229386494071277008460771242118230808041392980870 57
5047138252645714483793711250320818261265664908425 1
6994539518877896136502484057393785945994443352311 88
2801236604062624686092121503499375847822922371443 39
6288584859382157388212323936870461606773629093150 71

A 969 digits number

The proof is 8461 chars!

Others examples

Proof automation (useful for the user):

- Ring or field equalities
- Presburger arithmetic
- Decide polynomial inequalities in $\mathbb{R}$ (CAD)

Exotic theorems:

- 4-colors theorem (Gonthier, Werner)
- Kepler conjecture (Hales)

Conversion rule is very useful and convenient

How to implement a type checker with the conversion rule?

Calculus of Constructions

$$\begin{array}{lrcl} \text{Terms:} & T & ::= & s \mid x \mid \forall x{:}T.\ T \mid \lambda x{:}T.\ T \mid T\ T \\ \text{Contexts:} & \Gamma & ::= & x_1{:}T_1, x_2{:}T_2, \ldots, x_n{:}T_n \end{array}$$

One reduction rule:

$$(\lambda x{:}T.\ M)\ N \xrightarrow{\beta} M[x := N]$$

Type judgment:

$$\Gamma \vdash M : T$$

Type inference: $\Gamma \vdash M \rightsquigarrow T$ such that $\Gamma \vdash M : T$

Typing rules

$$\frac{}{\mathsf{WF}(\emptyset)} \qquad \frac{\mathsf{WF}(\Gamma) \quad \Gamma \vdash T : s}{\mathsf{WF}(\Gamma, x:T)}$$

$$\frac{\mathsf{WF}(\Gamma)}{\Gamma \vdash \mathtt{Set} : \mathtt{Type}} \qquad \frac{\mathsf{WF}(\Gamma) \quad (x:T) \in \Gamma}{\Gamma \vdash x : T}$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x:A \vdash B : s_2 \quad (s_1, s_2, s_3) \in \mathsf{Rules}}{\Gamma \vdash \forall x:A.\ B : s_3}$$

$$\frac{\Gamma \vdash \forall x:A.\ B : s \quad \Gamma, x:A \vdash M : B}{\Gamma \vdash \lambda x:A.\ M : \forall x:A.\ B}$$

$$\frac{\Gamma \vdash M : \forall x:A.\ B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B[x := N]}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s \quad A \approx B}{\Gamma \vdash M : B}$$

Definitions on reduction

Using only β-rule:

$$(\lambda x{:}T.\ M)\ N \xrightarrow{\beta} M[x := N]$$

The term $((\lambda x{:}T.\ M)\ N)\ P$ does not reduce

Context rule: $$\frac{t \xrightarrow{\beta} t'}{C(t) \xrightarrow{\beta} C(t')}$$

Weak reduction

$$\begin{array}{rcl} C & ::= & []\ N \mid M\ [] \\ & \mid & \lambda x{:}[].\ M \mid \forall x{:}[].\ B \end{array}$$

Strong reduction

$$\begin{array}{rcl} C & ::= & []\ N \mid M\ [] \\ & \mid & \lambda x{:}[].\ M \mid \forall x{:}[].\ B \\ & \mid & \lambda x{:}T.\ [] \mid \forall x{:}A.\ [] \end{array}$$

NF: Normal form using strong reduction

WNF: Normal form using weak reduction

WHNF: Normal form using $C ::= []\ N$

Meta-theory of CC

Subject reduction: $\Gamma \vdash M : T \Rightarrow M \xrightarrow{\beta} M' \Rightarrow \Gamma \vdash M' : T$

Strong Normalization: $\Gamma \vdash M : T \Rightarrow M \in \mathsf{SN}$

Uniqueness of typing: $\Gamma \vdash M : T_1 \Rightarrow \Gamma \vdash M : T_2 \Rightarrow T_1 \approx T_2$

Correctness of type:

$$\Gamma \vdash M : T \;\Rightarrow\; T = \mathtt{Type} \vee \Gamma \vdash T : s$$

Corollary: $\Gamma \vdash M : T \Rightarrow \Gamma \vdash M : \mathsf{WHNF}(T)$

$$\frac{\Gamma \vdash M : T \quad \Gamma \vdash \mathsf{WHNF}(T) : s \quad T \approx \mathsf{WHNF}(T)}{\Gamma \vdash M : \mathsf{WHNF}(T)}$$

Inference Algorithm

Type inference: $\Gamma \vdash M \rightsquigarrow T$ such that $\mathsf{WF}(\Gamma) \Rightarrow \Gamma \vdash M : T$

$$\frac{}{\Gamma \vdash \texttt{Set} \rightsquigarrow \texttt{Type}} \qquad \frac{(x:T) \in \Gamma}{\Gamma \vdash x \rightsquigarrow T}$$

$$\frac{\Gamma \vdash A \rightsquigarrow T_1 \quad \mathsf{WHNF}(T_1) = s_1 \quad \Gamma, x:A \vdash B \rightsquigarrow T_2 \quad \mathsf{WHNF}(T_2) = s_2}{\Gamma \vdash \forall x:A.\ B \rightsquigarrow s_3}$$

$$\frac{\Gamma \vdash A \rightsquigarrow T_1 \quad \mathsf{WHNF}(T_1) = s_1 \quad \Gamma, x:A \vdash M \rightsquigarrow B \quad B \neq \texttt{Type}}{\Gamma \vdash \lambda x:A.\ M \rightsquigarrow \forall x:A.\ B}$$

$$\frac{\Gamma \vdash M \rightsquigarrow T_1 \quad \mathsf{WHNF}(T_1) = \forall x:A.\ B \quad \Gamma \vdash N \rightsquigarrow T_2 \quad T_2 \approx A}{\Gamma \vdash M\ N \rightsquigarrow B[x := N]}$$

Soundness

Soundness: $\mathsf{WF}(\Gamma) \Rightarrow \Gamma \vdash M \rightsquigarrow T \Rightarrow \Gamma \vdash M : T$

Proof: by induction on M

$$\frac{\Gamma \vdash M \rightsquigarrow T_1 \quad \mathsf{WHNF}(T_1) = \forall x : A.\ B \quad \Gamma \vdash N \rightsquigarrow T_2 \quad T_2 \approx A}{\Gamma \vdash M\ N \rightsquigarrow B[x := N]}$$

$$\frac{\Gamma \vdash M : \forall x : A.\ B \quad \dfrac{\Gamma \vdash N : T_2 \quad \Gamma \vdash A : s \quad T_2 \approx A}{\Gamma \vdash N : A}}{\Gamma \vdash M\ N : B[x := N]}$$

Completeness

Completeness: $\Gamma \vdash M : T_1 \Rightarrow \Gamma \vdash M \rightsquigarrow T_2$

Proof: by induction on $\Gamma \vdash M : T_1$

$$\frac{\Gamma \vdash M : \forall x : A.\ B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B[x := N]}$$

$$\frac{\Gamma \vdash M \rightsquigarrow T_M \quad \mathsf{WHNF}(T_M) = \forall x : A'.\ B' \quad \Gamma \vdash N \rightsquigarrow T_N \quad T_N \approx A'}{\Gamma \vdash M\ N \rightsquigarrow B[x := N]}$$

(Geuvers): $T \approx \forall x : A.\ B \ \Rightarrow\ \exists A', B', \mathsf{WHNF}(T) = \forall x : A'.\ B'$

(Generation lemma):

$$\Gamma \vdash \forall x : A.\ B : T \ \Rightarrow\ \exists s_1, s_2, s_3, \begin{cases} \Gamma \vdash A : s_1 \wedge \Gamma, x : A \vdash B : s_2 \\ T \approx s_3 \end{cases}$$

So

We want a proof assistant

$\Rightarrow$ We develop a proof language

But it is also a nice functional programing language

- We have type checker
- We should now develop compiler

Types Summer School
Gothenburg Sweden August 2005

Dependently Typed Programming

Benjamin Grégoire
INRIA Sophia Antipolis, France

Lecture 2:
Conversion test, compilation

Summary of the problem

Proof / type checkers based on dependent types work up to conversion:

$$\frac{\Gamma \vdash M : A \quad A \approx B}{\Gamma \vdash M : B}$$

It is very convenient: allows small proofs and automation (using reflexive proofs)

If we have a algorithm for testing convertibility, we get a type checker

Testing convertibility require strong β-reduction (under λ abstractions)

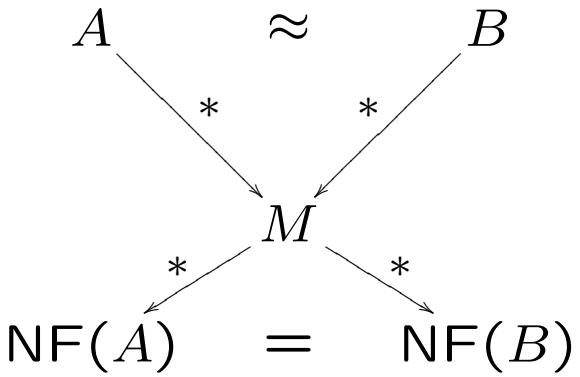
For most proofs, the amount of reduction is small (simple interpreter suffice)

But proofs based on reflection require large amounts of reductions. The speed of the reducer becomes the limiting factor

Convertibility is decidable

Testing convertibility of two terms is decidable if the reduction rules are

- Confluents $\Rightarrow$ Church-Rosser, uniqueness of normal forms
- Strongly normalizing



λ-calculus

$$\begin{array}{lrcl} \text{terms} & t & ::= & x \mid \lambda x.t \mid t\ t \\ \text{values(WNF)} & v & ::= & \lambda x.t \mid x\ v_1 \ldots v_n \end{array}$$

Conversion algorithm:

$$\frac{t_1 = t_2}{t_1 \approx t_2} \qquad \frac{\mathsf{WNF}(t_1) \approx \mathsf{WNF}(t_2)}{t_1 \approx t_2}$$

$$\frac{v_1 = v_2}{v_1 \approx v_2} \qquad \frac{x = y \quad v_i \approx w_i}{x\ v_1 \ldots v_n \approx y\ w_1 \ldots w_n}$$

$$\frac{\mathsf{WNF}(\lambda x.M\ z) \approx \mathsf{WNF}(\lambda y.M'\ z) \quad z \text{ fresh}}{\lambda x.M \approx \lambda y.M'}$$

Computing the WNF

```
type term = Var of var | Abs of var*term | App of term*term

let rec wnf t =
   match t with
   | Var _ | Abs _ -> t

   | App(t1, t2) ->
      let v1 = wnf t1 in
      let v2 = wnf t2 in
      match v1 with
      | Abs(x,u) -> wnf (subst u x v2)
      | _ -> App(v1,v2)
```

WNF by compilation

WNF : execution of ML-like program

$$\lambda\text{-term} \xrightarrow{\text{Compilation}} \text{bytecode} \xrightarrow[\text{Abs. Machine}]{\text{Execution}} \text{value}$$

bytecode : sequence of instructions

Problem: usual compilation techniques work only for closed terms

$$\text{WNF}(\lambda x.Mz)$$

ZINC abstract machine

ZINC : a stack based abstract machine in call by value

Instructions : Acc, Closure, Grab, Pushra, Apply, Return

Representation of values v (closures): $[c, e]$

Environment e : $[v_1; \ldots; v_n]$

Components of the machine:

- c code pointer
- e environment (values associate to variables)
- s stack (arguments + intermediate results + return address)
- n number of available arguments on the top of s

Compilation and execution of variables

Compilation scheme: $[\![t]\!]k \rightsquigarrow c$

The resulting code c compute the value corresponding to t, push it on top of the stack, then restart the execution of k

$$[\![x]\!]k = \mathsf{Acc}(i);\ k$$

where i = deBruijn index of x

Code	Env	Stack	#args
$\mathsf{Acc}(i); k$	e	s	n
k	e	$e(i).s$	n

Compilation and execution of applications

$$[\![f\ a_1\ \ldots\ a_i]\!]k \;=\; \mathsf{Pushra}(k);\ [\![a_i]\!]\ \ldots\ [\![a_1]\!]\ [\![f]\!]\ \mathsf{Apply}(i)$$

Code	Env	Stack	#args
$\mathsf{Pushra}(k); c$	e	s	n
c	e	$\langle k, e, n\rangle.s$	n
$\mathsf{Apply}(i)$	e	$[c, e'].v_1 \ldots v_i.\langle k, e, n\rangle.s$	n
c	e'	$v_1 \ldots v_i.\langle k, e, n\rangle.s$	i

Compilation and execution of functions

$$[\![\lambda x_1 . \ldots . \lambda x_n . t]\!] k = \mathsf{Closure}(c); k$$
$$c = \underbrace{\mathsf{Grab}; \ldots ; \mathsf{Grab}}_{n \text{ times}}; [\![t]\!]\mathsf{Return}$$

Code	Env	Stack	#args
Closure$(c); k$	e	s	n
k	e	$[c, e].s$	n
Grab$; k$	e	$v.s$	$n + 1$
k	$v.e$	s	n
Return	e	$v.\langle k, e', n\rangle.s$	0
k	e'	$v.s$	n

Under or over application

Under application:

Code	Env	Stack	#args
Grab; c	e	$\langle k, e', n\rangle.s$	0
k	e'	$[(\text{Grab}; c), e].s$	n

Over application:

Code	Env	Stack	#args
Return	e	$[c, e'].s$	$n > 0$
c	e'	s	n

Compilation with free variables

Code	Env	Stack	#args
Acc$(i); k$	e	s	n
k	e	$e(i).s$	n

Free variables have no associated value in the environment

$\Rightarrow$ add values for free variables

What should be the value associated to a free variables?

What happens when this value is applied?

What is the computational behavior of a free variable?

Symbolic calculus:

$$\begin{array}{lrcl} \text{Terms} & t & ::= & x \mid t\ t \mid v \\ \text{Values} & v & ::= & \lambda x.t \mid [\widetilde{x}] \end{array}$$

Reduction rules:

$$\begin{array}{rcl} (\lambda x.t)\ v & \longrightarrow & t[x := v] \\ [\widetilde{x}]\ v & \longrightarrow & [\widetilde{x}\ v] \end{array}$$

Computational behavior of a free variable

Symbolic calculus:

$$
\begin{array}{lrcl}
\text{Terms} & t & ::= & x \mid t\ t \mid v \\
\text{Values} & v & ::= & \lambda x.t \mid [k] \\
\text{Accumulators} & k & ::= & \tilde{x} \mid k\ v
\end{array}
$$

Reduction rules:

$$
\begin{array}{rcl}
(\lambda x.t)\ v & \longrightarrow & t[x := v] \\
[k]\ v & \longrightarrow & [k\ v]
\end{array}
$$

The value associate to a free variable is a function that accumulate its arguments

Encoding accumulator

Code	Env	Stack	#args
Apply(n)	_	$[c, e].v_1 \ldots v_n.\langle c', e', n' \rangle.s$	_
c	e	$v_1 \ldots v_n.\langle c', e', n' \rangle.s$	n'

The top value can now be a accumulator, encoding of accumulator should be compatible with the one of closure

$$[\text{Accumulate}, \widehat{k}]$$

where $\widehat{k}$ is the machine-level encoding of k: $\widehat{k} = [\widetilde{x}; v_1; \ldots; v_n]$

This suffices to trick function application:

Code	Env	Stack	#args
Apply(n)	e	$[\text{Accumulate}, \widehat{k}].v_1 \ldots v_n.\langle c', e', n' \rangle.s$	_
Accumulate	$\widehat{k}$	$v_1 \ldots v_n.\langle c', e', n' \rangle.s$	n
c'	e'	$[\text{Accumulate}, (\widehat{k}.v_1 \ldots v_n)].s$	n'

The move from $[\text{Accumulate}, \widehat{k}]$ to $[\text{Accumulate}, (\widehat{k}.v_1 \ldots v_n)]$ implements exactly the symbolic reduction

$[k]\ v_1\ \ldots\ v_n \longrightarrow [k\ v_1\ \ldots\ v_n]$

Distinguishing feature of this encoding

The representation of $[k]$ looks like a function

$\Rightarrow$ No need to test at application time whether the function is a closure or an accumulator

$\Rightarrow$ No overhead on evaluation of closed terms

Similarly, we arrange that the representation of $[k]$ looks like the representation of inductive constructors

$\Rightarrow$ No overhead for ι-reduction

Experimental results

4-colors theorem

Perimeter	Coq	Coq-vm	OCaml bytecode	OCaml natif
11	56.7s	1.68s	1.18s	0.30s
12	259s	6.50s	6.18s	1.92s
13	680s	14.8s	15.5s	4.11s

Prime numbers

	Size	time
	1234567891 (10)	
Deductive :	3099	13.26 s
Reflexive :	58	0.59 s
209889366574405864861512642566102225938639	21 (44)	
Deductive :	18509	1862.52 s
Reflexive :	95	21.30 s

Conclusion

Conversion is very convenient: allows small proofs and automation (using reflexive proofs)

The use of a compiler and an abstract machine for testing convertibility leads to an efficient algorithm

So reflexive proofs can be efficiently type checked

A formally verified proof of the prime number theorem (draft)

Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff

August 19, 2005

Abstract

The prime number theorem, established by Hadamard and de la Vallée Poussin independently in 1896, asserts that the density of primes in the positive integers is asymptotic to $1/\ln x$. Whereas their proofs made serious use of the methods of complex analysis, elementary proofs were provided by Selberg and Erdös in 1948. We describe a formally verified version of Selberg's proof, obtained using the Isabelle proof assistant.

1 Introduction

For each positive integer x, let $\pi(x)$ denote the number of primes less than or equal to x. The prime number theorem asserts that the density of primes $\pi(x)/x$ in the positive integers is asymptotic to $1/\ln x$, i.e. that $\lim_{x\to\infty} \pi(x) \ln x/x = 1$. This was conjectured by Gauss and Legendre around the turn of the nineteenth century, and posed a challenge to the mathematical community for almost a hundred years, until Hadamard and de la Vallée Poussin proved it independently in 1896.

On September 6, 2004, the first author of this article verified the following statement, using the Isabelle proof assistant:

```
(λx. pi x * ln (real x) / (real x)) ----> 1
```

The system thereby confirmed that the prime number theorem is a consequence of the axioms of higher-order logic, together with an axiom asserting the existence of an infinite set.

One reason the formalization is interesting is simply that it is a landmark, showing that today's proof assistants have achieved a level of usability that makes it possible to formalize substantial theorems of mathematics. Similar achievements in the past year include George Gonthier's verification of the four color theorem using Coq, and Thomas Hales's verification of the Jordan curve theorem using HOL-light (see the introduction to [19]). As contemporary mathematical proofs become increasingly complex, the need for formal verification becomes pressing. Formal verification can also help guarantee correctness when, as is becoming increasingly common, proofs rely on computations that are too long to check by hand. Hales's ambitious Flyspeck project [10], which aims for

a fully verified form of his proof of the Kepler conjecture, is a response to both of these concerns. Here, we will provide some information as to the time and effort that went into our formalization, which should help gauge the feasibility of such verification efforts.

More interesting, of course, are the lessons that can be learned. This, however, puts us on less certain terrain. Our efforts certainly provide some indications as to how to improve libraries and systems for verifying mathematics, but we believe that right now the work is best viewed as raw data. Here, therefore, we simply offer some initial thoughts and observations.

The outline of this paper is as follows. In Section 2, we provide some background on the prime number theorem and the Isabelle proof assistant. In Section 3, we provide an overview of Selberg's proof, our formalization, and the effort involved. Finally, in Section 4, we discuss some interesting aspects of the formalization: the use of asymptotic reasoning; calculations with real numbers; casts between natural numbers, integers, and real numbers; combinatorial reasoning in number theory; and the use of elementary methods.

Our formalization of the prime number theorem was a collaborative effort on the part of Avigad, Donnelly, Gray, and Raff, building, of course, on the efforts of the entire Isabelle development team. This article was, however, written by Avigad, so opinions and speculation contained herein should be attributed to him.

2 Background

2.1 The prime number theorem

The statement of the prime number theorem was conjectured by both Gauss and Legendre, on the basis of computation, around the turn of the nineteenth century. In a pair of papers published in 1851 and 1852, Chebyshev made significant advances towards proving it. Note that we can write

$$\pi(x) = \sum_{p \leq x} 1,$$

where p ranges over the prime numbers. Contrary to our notation above, x is usually treated as a real variable, making π a step function on the reals. Chebyshev defined, in addition, the functions

$$\theta(x) = \sum_{p \leq x} \ln p$$

and

$$\psi(x) = \sum_{p^a \leq x} \ln p = \sum_{n \leq x} \Lambda(n),$$

where

$$\Lambda(n) = \begin{cases} \ln p & \text{if } n = p^a, \text{ for some } a \geq 1 \\ 0 & \text{otherwise.} \end{cases}$$

The functions θ and ψ are more sensitive to the presence of primes less than x, and have nicer analytic properties. Chebyshev showed that the prime number

theorem is equivalent to the assertion $\lim_{x\to\infty} \theta(x)/x = 1$, as well as to the assertion $\lim_{x\to\infty} \psi(x)/x = 1$. He also provided bounds

$$B < \pi(x) \ln x/x < 6B/5$$

for sufficiently large x, where

$$B = \ln 2/2 + \ln 3/3 + \ln 5/5 - \ln 30/30 > 0.92$$

and $6B/5 < 1.11$. So, as x approaches infinity, $\pi(x) \ln x/x$, at worst, oscillates between these two values.

In a landmark work of 1859, Riemann introduced the complex-valued function ζ into the study of number theory. It was not until 1894, however, that von Mangoldt provided an expression for ψ that reduced the prime number theorem, essentially, to showing that ζ has no roots with real part equal to 1. This last step was achieved by Hadamard and de la Vallée Poussin, independently, in 1896. The resulting proofs make strong use of the theory of complex functions. In 1921, Hardy expressed strong doubts as to whether a proof of the theorem was possible which did not depend, fundamentally, on these ideas. In 1948, however, Selberg and Erdös found elementary proofs based on a "symmetry formula" due to Selberg. (The nature of the interactions between Selberg and Erdös at the time and the influence of ideas is a subtle one, and was the source of tensions between the two for years to come.) Since the libraries we had to work with had only a minimal theory of the complex numbers and a limited real analysis library, we chose to formalize the Selberg proof.

There are a number of good introductions to analytic number theory (for example, [1, 12]). Edwards's *Riemann's zeta function* [9] is an excellent source of both historical and mathematical information. A number of textbooks present the Selberg's proof in particular, including those by Nathanson [14], Shapiro [16], and Hardy and Wright [11]. We followed Shapiro's excellent presentation quite closely, though we made good use of Nathanson's book as well.

We also had help from another source. Cornaros and Dimitricopoulis [8] have shown that the prime number theorem is provable in a weak fragment of arithmetic, by showing how to formalize Selberg's proof (based on Shapiro's presentation) in that fragment.[1] Their concerns were different from ours: by relying on a formalization of higher-order logic, we were allowing ourselves a logically stronger theory; on the other hand, Cornaros and Dimitricopoulis were concerned solely with axiomatic provability and not ease of formalization. Their work was, however, quite helpful in stripping the proof down to its bare essentials. Also, since, our libraries did not have a good theory of integration, we had to take some care to avoid the mild uses of analysis in the textbook presentations. Cornaros and Dimtricopoulis's work was again often helpful in that respect.

2.2 Isabelle

Isabelle [20] is a generic proof assistant developed under the direction of Larry Paulson at Cambridge University and Tobias Nipkow at TU Munich. The HOL

[1] For issues relating to the formalization of mathematics, and number theory in particular, in weak theories of arithmetic, see [3].

instantiation [15] provides a formal framework that is a conservative extension of Church's simple type theory with an infinite type (from which the natural numbers are constructed), extensionality, and the axiom of choice. Specifically, HOL extends ordinary type theory with set types, and a schema for polymorphic axiomatic type classes designed by Nipkow and implemented by Marcus Wenzel [17]. It also includes a definite description operator ("THE"), and an indefinite description operator ("SOME").[2]

Isabelle offers good automated support, including a term simplifier, an automated reasoner (which combines tableau search with rewriting), and decision procedures for linear and Presburger arithmetic. It is an LCF-style theorem prover, which is to say, correctness is guaranteed by the use of a small number of constructors, in an underlying typed programming language, to build proofs. Using the Proof General interface [21], one can construct proofs interactively by repeatedly applying "tactics" that reduce a current subgoal to simpler ones. But Isabelle also allows one to take advantage of a higher-level proof language, called Isar, implemented by Wenzel [18]. These two styles of interaction can, furthermore, be combined within a proof. We found Isar to be extremely helpful in structuring complex proofs, whereas we typically resorted to tactic-application for filling in low-level inferences. Occasionally, we also made mild use of Isabelle's support for locales [7]. For more information on Isabelle, one should consult the tutorial [15] and other online documentation [20].

Our formalization made use of the basic HOL library, as well as those parts of the HOL-Complex library, developed primarily by Jacques Fleuriot, that deal with the real numbers. Some of our earlier definitions, lemmas, and theorems made their way into the 2004 release of Isabelle, in which the formalization described here took place. Some additional theorems in our basic libraries will be part of the 2005 release.

3 Overview

3.1 The Selberg proof

The prime number theorem describes the asymptotic behavior of a function from the natural numbers to the reals. Analytic number theory works by extending the domain of such functions to the real numbers, and then providing a toolbox for reasoning about such functions. One is typically concerned with rough characterizations of a function's rate of growth; thus $f = O(g)$ expresses the fact that for some constant C, $|f(x)| \leq C|g(x)|$ for every x. (Sometimes, when writing $f = O(g)$, one really means that the inequality holds except for some initial values of x, where g is 0 or one of the functions is undefined; or that the inequality holds when x is large enough.)

[2] The extension by set types is mild, since they are easily interpretable in terms of predicate types $\sigma \to \mathit{bool}$. Similarly, the definite description operator can be eliminated, at least in principle, using Russell's well-known interpretation. It is the indefinite description operator, essentially a version of Hilbert's epsilon operator, that gives rise to the axiom of choice. Though we occasionally used the indefinite description operator for convenience, these uses could easily be replaced by the definition description operator, and it is likely that uses of the axiom of choice can be dispensed with in the libraries as well. In any event, it is a folklore result that Gödel's methods transfer to higher-order logic to show that the axiom of choice is a conservative extension for a fragment the includes the prime number theorem.

For example, all of the following identities can be obtained using elementary calculus:

$$\begin{aligned}
\ln(1 + 1/n) &= 1/n + O(1/n^2) \\
\sum_{n \le x} 1/n &= \ln x + O(1) \\
\sum_{n \le x} \ln n &= x \ln x - x + O(\ln x) \\
\sum_{n \le x} \ln n/n &= \ln^2 x/2 + O(1)
\end{aligned}$$

In all of these, n ranges over positive integers. The last three inequalities hold whether one takes x to be an integer or a real number greater than or equal to 1. The second identity reflects the fact that the integral of $1/x$ is $\ln x$, and the third reflects the fact that the integral of $\ln x$ is $x \ln x - x$. A list of identities like these form one part of the requisite background to the Selberg proof.

Some of Chebyshev's results form another. Rate-of-growth comparisons between θ, ψ, and π sufficient to show the equivalence of the various statements of the prime number theorem can be obtained by fairly direct calculations. Obtaining any of the upper bounds equivalent to $\psi(x) = O(x)$ requires more work. A nice way of doing this, using binomial coefficients, can be found in [14].

Number theory depends crucially on having different ways of counting things, and rudimentary combinatorial methods form a third prerequisite to the Selberg proof. For example, consider the set of (positive) divisors d of a positive natural number n. Since the function $d \mapsto n/d$ is a permutation of that set, we have the following identity:

$$\sum_{d|n} f(d) = \sum_{d|n} f(n/d).$$

For a more complicated example, suppose n is a positive integer, and consider the set of pairs d, d' of positive integers such that $dd' \le n$. There are two ways to enumerate these pairs: for each value of d between 1 and n, we can enumerate all the values d' such that $d' \le n/d$; or for each product c less than n, we can enumerate all pairs $d, c/d$ whose product is c. Thus we have

$$\begin{aligned}
\sum_{d \le n} \sum_{d' \le n/d} f(d, d') &= \sum_{dd' \le n} f(d, d') \\
&= \sum_{c \le n} \sum_{d|c} f(d, c/d).
\end{aligned} \tag{1}$$

A similar argument yields

$$\begin{aligned}
\sum_{d|n} \sum_{d'|(n/d)} f(d, d') &= \sum_{dd'|n} f(d, d') \\
&= \sum_{c|n} \sum_{d|c} f(d, c/d).
\end{aligned} \tag{2}$$

Yet another important combinatorial identity is given by the partial summation formula, which, in one formulation, is as follows: if $a \le b$, $F(n) = \sum_{i=1}^{n} f(i)$,

and G is any function, then

$$\sum_{n=a}^{b} f(n+1)G(n+1) = F(b+1)G(b+1) - F(a)G(a+1) - \sum_{n=a}^{b-1} F(n+1)(G(n+2) - G(n+1)).$$

This can be viewed as a discrete analogue of integration by parts, and can be verified by induction.

An important use of (2) occurs in the proof of the Möbius inversion formula, which we now describe. A positive natural number n is said to be *square free* if no prime in its factorization occurs with multiplicity greater than 1; in other words, $n = p_1 p_2 \cdots p_s$ where the p_i's are distinct primes (and s may be 0). Euler's function μ is defined by

$$\mu(n) = \begin{cases} (-1)^s & \text{if } n \text{ is squarefree and } s \text{ is as above} \\ 0 & \text{otherwise.} \end{cases}$$

A remarkably useful fact regarding μ is that for $n > 0$,

$$\sum_{d|n} \mu(d) = \begin{cases} 1 & \text{if } n = 1 \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

To see this, define the *radical* of a number n, denoted $rad(n)$, to be the greatest squarefree number dividing n. It is not hard to see that if n has prime factorization $p_1^{j_1} p_2^{j_2} \cdots p_s^{j_s}$, then $rad(n)$ is given by $p_1 p_2 \cdots p_s$. Then $\sum_{d|n} \mu(d) = \sum_{d|rad(n)} \mu(d)$, since divisors of n that are not divisors of $rad(n)$ are not squarefree and hence contribute 0 to the sum. If $n = 1$, equation (3) is clear. Otherwise, write $rad(n) = p_1 p_2 \cdots p_s$, write

$$\sum_{d|rad(n)} \mu(d) = \sum_{d|rad(n), p_1|d} \mu(d) + \sum_{d|rad(n), p_1 \nmid d} \mu(d),$$

and note that each term in the first sum is canceled by a corresponding one in the second.

Now, suppose g is any function from $\mathbb{N}$ to $\mathbb{R}$, and define f by $f(n) = \sum_{d|n} g(d)$. The Möbius inversion formula provides a way of "inverting" the definition to obtain an expression for g in terms of f. Using (2) for the third equality below and (3) for the last, we have, somewhat miraculously,

$$\begin{aligned} \sum_{d|n} \mu(d) f(n/d) &= \sum_{d|n} \mu(d) \sum_{d'|(n/d)} g((n/d)/d') \\ &= \sum_{d|n} \sum_{d'|(n/d)} \mu(d) g((n/d)/d') \\ &= \sum_{c|n} \sum_{d|c} \mu(d) g(n/c) \\ &= \sum_{c|n} g(n/c) \sum_{d|c} \mu(d) \\ &= g(n), \end{aligned}$$

since the inner sum on the second-to-last line is 0 except when c is equal to 1.

All the pieces just described come together to yield additional identities involving sums, ln, and μ, as well as Mertens's theorem:

$$\sum_{n \leq x} \Lambda(n)/n = \ln x + O(1).$$

These, in turn, are used to derive Selberg's elegant "symmetry formula," which is the central component in the proof. One formulation of the symmetry formula is as follows:

$$\sum_{n \leq x} \Lambda(n) \ln n + \sum_{n \leq x} \sum_{d|n} \Lambda(d)\Lambda(n/d) = 2x \ln x + O(x).$$

There are, however, many variants of this identity, involving Λ, ψ, and θ. These crop up in profusion because one can always unpack definitions of the various functions, apply the types of combinatorial manipulations described above, and use identities and approximations to simplify expressions.

What makes the Selberg symmetry formula so powerful is that there are two terms in the sum on the left, each sensitive to the presence of primes in different ways. The formula above implies there have to be some primes — to make left-hand side nonzero — but there can't be too many. Selberg's proof involves cleverly balancing the two terms off each other, to show that in the long run, the density of the primes has the appropriate asymptotic behavior.

Specifically, let $R(x) = \psi(x) - x$ denote the "error term," and note that by Chebyshev's equivalences the prime number theorem amounts to the assertion $\lim_{x \to \infty} R(x)/x = 0$. With some delicate calculation, one can use the symmetry formula to obtain a bound on $|R(x)|$:

$$|R(x)| \ln^2 x \leq 2 \sum_{n \leq x} |R(x/n)| \ln n + O(x \ln x). \tag{4}$$

Now, suppose we have a bound $|R(x)| \leq ax$ for sufficiently large x. Substituting this into the right side of (4) and using an approximation for $\sum_{n \leq x} \ln n/n$ we get

$$|R(x)| \leq ax + O(x/\ln x),$$

which is not an improvement on the original bound. Selberg's method involves showing that in fact there are always sufficiently many intervals on which one can obtain a stronger bound on $R(x)$, so that for some positive constant k, assuming we have a bound $|R(x)| \leq ax$ that valid for $x \geq c_1$, we can obtain a c_2 and a better bound $|R(x)| \leq (a - ka^3)$, valid for $x \geq c_2$. The constant k depends on a, but the same constant also works for any $a' < a$.

By Chebyshev's theorem, we know that there is a constant a_1 such that $|R(x)| \leq a_1 x$ for every x. Choosing k appropriate for a_1 and then setting $a_{n+1} = a_n - ka_n^3$, we have that for every n, there is a c large enough so that $|R(x)|/x \leq a_n$ for every $x \geq c$. But it is not hard to verify that the sequence $a_1, a_2, \ldots$ approaches 0, which implies that $R(x)/x$ approaches 0 as x approaches infinity, as required.

3.2 Our formalization

All told, our number theory session, including the proof of the prime number theorem and supporting libraries, constitutes 673 pages of proof scripts, or roughly 30,000 lines. This count includes about 65 pages of elementary number theory that we had at the outset, developed by Larry Paulson and others; also about 50 pages devoted to a proof of the law of quadratic reciprocity and properties of Euler's φ function, neither of which are used in the proof of the prime number theorem. The page count does not include the basic HOL library, or properties of the real numbers that we obtained from the HOL-Complex library.

The overview provided in the last section should provide a general sense of the components that are needed for the formalization. To start with, one needs good supporting libraries:

- a theory of the natural numbers and integers, including properties of primes and divisibility, and the fundamental theorem of arithmetic
- a library for reasoning about finite sets, sums, and products
- a library for the real numbers, including properties of ln

The basic Isabelle libraries provided a good starting point, though we had to augment these considerably as we went along. More specific supporting libraries include:

- properties of the μ function, combinatorial identities, and the Möbius inversion formula
- a library for asymptotic "big O" calculations
- a number of basic identities involving sums and ln
- Chebyshev's theorems

Finally, the specific components of the Selberg proof are:

- the Selberg symmetry formula
- the inequality involving $R(n)$
- a long calculation to show $R(n)$ approaches 0

This general outline is clearly discernible in the list of theory files, which can be viewed online [2]. Keep in mind that the files described here have not been modified since the original proof was completed, and many of the proofs were written while various participants in the project were still learning how to use Isabelle. Since then, some of the basic libraries have been revised and incorporated into Isabelle, but Avigad intends to revise the number theory libraries substantially before cleaning up the rest of the proof.

There are three reasons that it would not be interesting to give a play-by-play description of the formalization. The first is that our formal proof follows Shapiro's presentation quite closely, though for some parts we followed Nathanson instead. A detailed description of our proof would therefore be little more than a step-by-step narrative of (one of the various paths through)

Selberg's proof, with page correspondences in texts we followed. For example, one of our formulations of the Möbius inversion is as follows:

```
lemma mu_inversion_nat1a: "ALL n. (0 < n ⟶
  f n = (∑ d | d dvd n. g(n div d))) ⟹ 0 < (n::nat) ⟹
  g n = (∑ d | d dvd n. of_int(mu(int(d))) * f (n div d))"
```

This appears on page 64 of Shapiro's book, and on page 218 of Nathanson's book. We formalized a version of the fourth identity listed in Section 3.2 as follows:

```
lemma identity_four_real_b: "(λx. ∑i=1..natfloor(abs x).
    ln (real i) / (real i)) =o
      (λx. ln(abs x + 1)^2 / 2) +o O(λx. 1)"
```

In fact, stronger assertions can be found on page 93 of Shapiro's book, and on page 209 of Nathanson's book. Here is one of our formulations of the Selberg symmetry principle:

```
lemma Selberg3: "(λx. ∑n = 1..natfloor (abs x) + 1.
    Lambda n * ln (real n)) + (λx. ∑n=1..natfloor (abs x) + 1.
     (∑ u | u dvd n. Lambda u * Lambda (n div u)))
      =o (λx. 2 * (abs x + 1) * ln (abs x + 1)) +o O(λx. abs x + 1)"
```

This is given on page 419 of Shapiro's book, and on page 293 of Nathanson's book. The error estimate given in the previous section, taken from 431 of Shapiro's book, takes the following form:

```
lemma error7: "(λx. abs (R (abs x + 1)) * ln (abs x + 1) ^ 2) <o
  (λx. 2 * (∑ n = 1..natfloor (abs x) + 1.
     abs (R ((abs x + 1) / real n)) * ln (real n))) =o
    O(λx. (abs x + 1) * (1 + ln (abs x + 1)))"
```

We *will* have more to say, below, about handling of asymptotic notation, the type casts, and the various occurrences of *abs* and +1 that make the formal presentation differ from ordinary mathematical notation. But aside from calling attention to differences like these, a detailed outline of the formal proof would be, in large part, nothing more than a detailed outline of the ordinary mathematical one.

The second reason that it does not pay to focus too much attention on the proof scripts is that they are not particularly nice. Our efforts were designed to get us to the prime number theorem as quickly as possible rather than as cleanly as possible, and, in retrospect, there are many ways in which we could make the proofs more readable. For example, long after deriving some of the basic identities involving ln, we realized that we needed either stronger or slightly different versions, so we later incorporated a number of ad-hoc reworkings and fixes. A couple of months after completing the formalization, Avigad was dismayed to discover that his definition of the constant γ really described the *negation* of the constant, defined by Euler, that goes under the same name. This results in infelicitous differences between the statements of a few of our theorems and the ordinary mathematical versions. Our proofs also make use of two different summation operators that were in the libraries that we used, which will, fortunately, be subsumed by the more general one in future releases of the Isabelle libraries. Even the presentation of the theorems displayed above could easily be improved using Isabelle's various translation and output facilities.

This points to a final reason for not delving into too much detail: we know that our formalization is not optimal. It hardly makes sense for us to describe exactly how we went about proving the Möbius inversion formula until we are convinced that we have done it right; that is, until we are convinced the we have made the supporting libraries as generally useful as possible, and configured the automated tools in such a way to make the formalization as smooth as possible. We therefore intend to invest more time in improving the various parts of the formalization and report on these when it is clear what we have learned from the efforts.

In the meanwhile, we will devote the rest of this report to conveying two types of information. First, to help gauge the usability of the current technology, we will try to provide a sense of the amount of time required to seeing the project through to its completion. Second, we will provide some initial reflections on the project, and on the strengths and weaknesses of contemporary proof assistants. In particular, we will discuss what we take to be some of the novel aspects of the formalization, and indicate where we believe better automated support would have been especially helpful.

3.3 The effort involved

As we have noted in the introduction, one of the most interesting features of our formalization of the prime number theorem is simply its existence, which shows that current technology makes it possible to treat a proof of this complexity. The question naturally arises as to how long the formalization took.

This is a question that it hard to answer with any precision. Avigad first decided to undertake the project in March of 2003, having learned how to use Isabelle and proved Gauss's law of quadratic reciprocity with Gray and Adam Kramer the preceding summer and fall. But this was a side project for everyone involved, and time associated it includes time spent learning to use Isabelle, time spent learning the requisite number theory, and so on. Gray developed a substantial part of the number theory library, including basic facts about primes and multiplicity, the μ function, and the identity (2), working a few hours per week in the summer of 2003, before his thesis work in ethics took over. Donnelly and Avigad developed the library to support big O calculations [5] while Donnelly worked half-time during the summer of 2003, just after he completed his junior year at Carnegie Mellon. During that summer, and working part time the following year, Donnelly also derived some of the basic identities involving ln. Raff started working on the project in the 2003-2004 academic year, but most of his contributions came working roughly half-time in the summer of 2004, just after he obtained his undergraduate degree. During that time, he proved Chebyshev's theorem to the effect that $\psi(x) = O(x)$, and also did most of the work needed to prove the equivalence of statements of the prime number theorem in terms of the functions π, θ, and ψ. Though Avigad's involvement was more constant, he rarely put in more than a few hours per week before the summer of 2004, and set the project aside for long stretches of time. The bulk of his proof scripts were written during the summer of 2004, when he worked roughly half-time on the project from the middle of June to the end of August.

Some specific benchmarks may be more informative. Proving most of the inversion theorems we needed, starting from (2) and the relevant properties of μ, took Avigad about a day. (For a "day" read eight hours of dedicated

formalization. Though he could put in work-days like that for small stretches, in some of the estimates below, the work was spread out over longer periods of time.) Proving the first version of the Selberg symmetry formula using the requisite identities took another day. Along the way, he was often sidetracked by the need to prove elementary facts about things like primes and divisibility, or the floor function on the real numbers. This process stabilized, however, and towards the end he found that he could formalize about a page of Shapiro's text per day. Thus, the derivation of the error estimate described above, taken from pages 428–431 in Shapiro's book, took about three-and-a-half days to formalize; and the remainder of the proof, corresponding to 432–437 in Shapiro's book, took about five days.

In many cases, the increase in length is dramatic: the three-and-a-half pages of text associated with the proof of the error estimate translate to about than 1,600 lines, or 37 pages, of proof scripts, and the five pages of text associated with the final part of the proof translate to about 4,000 lines, or 89 pages, of proof scripts. These ratios are abnormally high, however, for reasons discussed in Section 4.2. The five-line derivation of the Möbius inversion formula in Section 3.1 translates to about 40 lines, and the proof of the form of the Selberg symmetry formula discussed there, carried out in about two-and-a-half pages in Shapiro's book, takes up about 600 lines, or 13 pages. These ratios are more typical.

We suspect that over the coming years both the time it takes to carry out such formalizations, as well as the lengths of the formal proof scripts, will drop significantly. Much of the effort involved in the project was spent on the following:

- Defining fundamental concepts and gathering basic libraries of easy facts.
- Proving trivial lemmas and spelling out "straightforward" inferences.
- Finding the right lemmas and theorems to apply.
- Entering long formulas and expressions correctly, and adapting ordinary mathematical notation to the formal notation in Isabelle.

Gradually, all these requirements will be ameliorated, as better libraries, automated tools, and interfaces are developed. On a personal note, we are entirely convinced that, although there is a long road ahead, formal verification of mathematics will inevitably become commonplace. Getting to that point will require both theoretical and practical ingenuity, but we do not see any conceptual hurdles.[3]

4 Reflection

In this section, we will discuss features of the formalization that we feel are worthy of discussion, either because they represent novel and successful solutions to general problems, or (more commonly) because they indicate aspects of formal mathematical verification where better support is possible.

[3]For further speculation along these lines, see the preliminary notes [4].

4.1 Asymptotics

One of our earliest tasks in the formalization was to develop a library to support the requisite calculations with big O expressions. To that end, we gave the expression $f = O(g)$ the strict reading $\exists C\ \forall x\ (|f(x)| \leq C|g(x)|)$, and followed the common practice of taking $O(g)$ to be the set of all functions with the requisite rate of growth, i.e.

$$O(g) = \{f \mid \exists C\ \forall x\ (|f(x)| \leq C|g(x)|)\}.$$

We then read the "equality" in $f = O(g)$ as the element-of relation, $\in$.

Note that these expressions make sense for any function type for which the codomain is an ordered ring. Isabelle's axiomatic type classes made it possible to develop the library fully generally. We could lift operations like addition and multiplication to such types, defining $f + g$ to denote the pointwise sum, $\lambda x.(f(x) + g(x))$. Similarly, given a set B of elements of a type that supports addition, we could define

$$a +_o B = \{c \mid \exists b \in B\ (c = a + b)\}.$$

We also defined $a =_o B$ to be alternative input syntax for $a \in B$. This gave expressions like $f =_o g +_o O(h)$ the intended meaning. In mathematical texts, convention dictates that in an expression like $x^2 + 3x = x^2 + O(x)$, the terms are to be interpreted as functions of x; in Isabelle we he had to use lambda notation to make this explicit. Thus, the expression above would be entered

```
(λx. x^2 + 3 * x) =o (λx. x^2) +o O(λx. x)
```

This should help the reader make sense of sense of the formalizations presented in Section 3.2.

An early version of our big O library is described in detail in [5]. That version is nonetheless fairly close to the version used in the proof of the prime number theorem described here, as well as a version that is scheduled for the 2005 release of Isabelle. The main differences between the latter and the version described in [5] are as follows:

1. In the version described in [5], we support reasoning about O applied to sets, $O(S)$, as well as to functions, $O(f)$. It now seems that uses of the former can easily be eliminated in terms of uses of the latter, and having both led to annoying type ambiguities. The most recent library only defines $O(f)$.

2. In [5], we advocated using $f + O(g)$ as output syntax for $f +_o O(g)$. We no longer think this is a good idea: the greater clarity in keeping the "o" outweighs the slight divergence from ordinary mathematical notation.

3. The more recent libraries have theorems to handle composition of functions in big O equations.

4. The more recent libraries have better and more general theorems for summations. (In the most recent library, the function "sumr" is entirely eliminated in favor of Isabelle's "setsum.")

5. The more recent libraries support reasoning about asymptotic inequalities, $f \leq g + O(h)$. This is entered as `f <=o g +o O(h)`, which is a hack, but an effective one.

There is one feature of our library that seems to be less than optimal, and resulted in a good deal of tedium. With our definition, a statement like $\lambda x.\ x + 1 = O(\lambda x.\ x^2)$ is false when the variables range over the natural numbers, since x^2 is equal to 0 when x is 0. Often one wants to restrict one's attention to strictly positive natural numbers, or nonnegative real numbers. There are four ways one can do this:

- Define new types for the strictly positive natural numbers, or nonnegative real numbers, and state the identities for those types.
- Formalize the notion "$f = O(g)$ on S."
- Formalize the notion "$f = O(g)$ eventually."
- Replace x by $x + 1$ in the first case, and by $|x|$ in the second case, to make the identities correct. For example, "$f(|x|) = O(|x|^3)$" expresses that $f(x) = O(x^3)$ on the nonnegative reals. Various similar tinkerings are effective; for example, the relationship intended in the example above is probably best expressed as $\lambda x.\ x + 1 = O(\lambda x.\ x^2 + 1)$.

These various options are discussed in [5], and all come at a cost. For example, the first requires annoying casts, say, between positive natural numbers, and natural numbers. The second requires carrying around a set S in every formula, and both the second and third require additional work when composing expressions or reasoning about sums (roughly, one has to make sure that the range of a function lies in the domain where an asymptotic estimate is valid).

In our formalization, we chose the fourth route, which explains the numerous occurrences of $+1$ and *abs* in the statements in Section 3.2. This often made some of the more complex calculations painfully tedious, forcing us, for example, the following "helper" lemma in Selberg:

```
lemma aux: "1 <= z ⟹ natfloor(abs(z - 1)) + 1 = natfloor z"
```

We still do not know, however, whether following any of the alternative options would have made much of a difference.

Donnelly and Avigad have designed a decision procedure for entailments between linear big O equations, and have obtained a prototype implementation (though we have not incorporated it into the Isabelle framework). This would eliminate the need for helper lemmas like the following:

```
lemma aux5: "f + g =o h +o O(k::'a=>('b::ordered_ring)) ⟹
  g + l =o h +o O(k) ⟹ f =o l +o O(k)"
```

We believe calculations going beyond the linear fragment would also benefit from a better handling of monotonicity, just as is needed to support ordinary calculations with inequalities, as described in the next section.

4.2 Calculations with real numbers

One salient feature of the Selberg proof is the amount of calculation involved. The dramatic increase in the length of the formalization of the final part of the

proof (5 pages in Shapiro, compared to 89 or so in the formal version) is directly attributable to the need to spell out calculations involving field operations, logarithms and exponentiation, the greatest and least integer functions ("ceiling" and "floor"), and so on. The textbook calculations themselves were complex; but then each textbook inference had to be expanded, by hand, to what was often a long sequence of entirely straightforward inferences.

Of course, Isabelle does provide some automated support. For example, the simplifier employs a form of ordered rewriting for operations, like addition and multiplication, that are associative and commutative. This puts terms involving these operations into canonical normal forms, thereby making it easy to verify equality of terms that differ up to such rewriting. More complex equalities can similarly be obtained by simplifying with appropriate rewrite rules, such as various forms of distributivity in a ring or identities for logarithms and exponents.

Much of the work in the final stages of the proof, however, involved verifying *inequalities* between expressions. Isabelle's linear arithmetic package is complete for reasoning about inequalities between linear expressions in the integers and reals, i.e. validities that depend only on the linear fragment of these theories. But, many of the calculations went just beyond that, at which point we were stuck manipulating expressions by hand and applying low-level inferences.

As a simple example, part of one of the long proofs in PrimeNumberTheorem required verifying that

$$(1 + \frac{\varepsilon}{3(C^* + 3)}) \cdot \mathit{real}(n) < Kx$$

using the following hypotheses:

$$\begin{aligned} &\mathit{real}(n) \leq (K/2)x \\ &0 < C^* \\ &0 < \varepsilon < 1 \end{aligned}$$

The conclusion is easily obtained by noting that $1 + \frac{\varepsilon}{3(C^*+3)}$ is strictly less than 2, and so the product with $\mathit{real}(n)$ is strictly less than $2(K/2)x = Kx$. But spelling out the details requires, for one thing, invoking the relevant monotonicity rules for addition, multiplication, and division. The last two, in turn, require verifying that the relevant terms are positive. Furthermore, getting the calculation to go through can require explicitly specifying terms like $2(K/2)x$ (which can be simplified to Kx), or, in other contexts, using rules like associativity or commutativity to manipulate terms into the the forms required by the rules.

The file PrimeNumberTheorem consists of a litany of such calculations. This required us to have names like "mult-left-mono" "add-pos-nonneg," "order-le-less-trans," "exp-less-cancel-iff," "pos-divide-le-eq" at our fingertips, or to search for them when they were needed. Furthermore, sign calculations had a way of coming back to haunt us. For example, verifying an inequality like $1/(1 + st) < 1/(1 + su)$ might require showing that the denominators are positive, which, in turns, might require verifying that s, t, and u are nonnegative; but then showing $st > su$ may again require verifying that s is positive. Since s can be carried along in a chain of inequalities, such queries for sign information can keep coming back. Isar made it easy to break out such facts, name them, and reuse them as needed. But since we were usually working in a context where

obtaining the sign information was entirely straightforward, these concerns always felt like an annoying distraction from the interesting and truly difficult parts of the calculations.

In short, inferences like the ones we have just described are commonly treated as "obvious" in ordinary mathematical texts, and it would be nice if mechanized proof assistants could recognize them as such. Decision procedures that are stronger than linear arithmetic are available; for example, a proof-producing decision procedure for real-closed fields has recently been implemented in HOL-light [13]. But for calculations like the one above, computing sequences of partial derivatives, as decision procedures for the real closed fields are required to do, is arguably unnecessary and inefficient. Furthermore, decision procedures for real closed fields cannot be extended, say, to handle exponentiation and logarithms; and adding a generic monotone function, or trigonometric functions, or the floor function, renders the full theory undecidable.

Thus, in contexts similar to ours, we expect that principled heuristic procedures will be most effective. Roughly, one simply needs to chain backwards through the obvious rules in a sensible way. There are stumbling blocks, however. For one thing, excessive case splits can lead to exponential blowup; e.g. one can show $st > 0$ by showing that s and t are either both strictly positive or strictly negative. Other inferences are similarly nondeterministic: one can show $r + s + t > 0$ by showing that two of the terms are nonnegative and the third is strictly positive, and one can show $r + s < t + u + v + w$, say, by showing $r < u$, $s \leq t + v$, and $0 \leq w$.

As far as case splits are concerned, we suspect that they are rarely needed to establishing "obvious" facts; for example, in straightforward calculations, the necessary sign information is typically available. As far as the second sort of nondeterminism is concerned, notice that the procedures for linear arithmetic are effective in drawing the requisite conclusions from available hypotheses; this is a reflection that of the fact that the theory of the real numbers with addition (and, say, multiplication by rational constants) is decidable.

The analogous theory of the reals with multiplication is also decidable. To see this, observe that the structure consisting of the strictly positive real numbers with multiplication is isomorphic to the structure of the real numbers with addition, and so the usual procedures for linear arithmetic carry over. More generally, by introducing case splits on the signs of the basic terms, one can reduce the multiplicative fragment of the reals to the previous case.

In short, when the signs of the relevant terms are known, there are straightforward and effective methods of deriving inequalities in the additive and multiplicative fragments. This suggests that what is really needed is a principled method of amalgamating such "local" procedures, together with, say, procedures that make use of monotonicity and sign properties of logarithms and exponentiation. The well-known Nelson-Oppen procedure provides a method of amalgamating decision procedures for disjoint theories that share only the equality symbol in their common language; but these methods fail for theories that share an inequality symbol when one adds, say, rational constants to the language, which is necessary to render such combinations nontrivial. We believe that there are principled ways, however, of extending the Nelson-Oppen framework to obtain useful heuristic procedures. This possibility is explored by Avigad and Harvey Friedman in [6].

4.3 Casting between domains

In our formalization, we found that the most natural way to establish basic properties of the functions θ, ψ, and π, as well as Chebyshev's theorems, was to treat them as functions from the natural numbers to the reals, rather them as functions from the reals to the reals. Either way, however, it is clear that the relevant proofs have to use the embedding of the natural numbers into the reals in an essential way. Since the μ function takes positive and negative values, we were also forced to deal with integers as soon as μ came into play. In short, our proof of the prime number theorem inevitably involved combining reasoning about the natural numbers, integers, and real numbers effectively; and this, in turn, involved frequent casting between the various domains.

We tended to address such needs as they arose, in an ad-hoc way. For example, the version of the fundamental theorem of arithmetic that we inherited from prior Isabelle distributions asserts that every positive natural number can be written uniquely as the product of an increasing list of primes. Developing properties of the radical function required being able to express the unique factorization theorem in the more natural form that every positive number is the product of the primes that divide it, raised to the appropriate multiplicity; i.e. the fact that for every $n > 0$,

$$n = \prod_{p|n} p^{mult_p(n)},$$

where $mult_p(n)$ denotes the multiplicity of p in n. We also needed, at our disposal, things like the fact that n divides m if and only if for every prime number p, the multiplicity of p in n is less than or equal to the multiplicity of p in m. Thus, early on, we faced the dual tasks of translating the unique factorization theorem from a statement about positive natural numbers to positive integers, and developing a good theory of multiplicity in that setting. Later, when proving Chebyshev's theorems, we found that we needed to recast some of the facts about multiplicity to statements about natural numbers.

We faced similar headaches when we began serious calculations involving natural numbers and the reals. In particular, as we proceeded we were forced to develop a substantial theory of the floor and ceiling functions, including a theory of their behavior vis-a-vis the various field operations. In calculations, expressions sometimes involved objects of all three types, and we often had to explicitly transport operations in or out of casts in order to apply a relevant lemma.

When one extends a domain like the natural numbers to the integers, or the integers to the real numbers, some operations are simply extended. For example, properties of addition and multiplication of natural numbers carry all the way through to the reals. On the other hand, one has new operations, like subtraction on the integers and division in the real numbers, that are mirrored imperfectly in the smaller domains. For example, subtraction on the integers extends truncated subtraction $x \dot{-} y$ on the natural numbers only when $x \geq y$, and division in the reals extends the function $x \mathit{div} y$ on the integers or natural numbers only when y divides x. Finally, there are facts the depend on the choice of a left inverse to the embedding: for example, if n is an integer, x is a real number, *real* is the embedding of the integers into the reals, and $\lfloor \cdot \rfloor$ denotes the

floor function from the reals to the integers, we have

$$(n \leq \lfloor x \rfloor) \equiv (real(n) \leq x).$$

This is an example of what mathematicians call a Galois correspondence, and category theorists call an adjunction, between the integers and the real numbers with the ordering relation.

Our formalization of the prime number theorem involved a good deal of manipulation of expressions, by hand, using the three types of facts just described. Many of these inferences should be handled automatically. After all, such issues are transparent in mathematical texts; we carry out the necessary inferences smoothly and unconsciously whenever we read an ordinary proof. The guiding principle should be that anything that is transparent to us can be made transparent to a mechanized proof assistant: we simply need to reflect on *why* we are effectively able to combine domains in ordinary mathematical reasoning, and codify that knowledge appropriately.

4.4 Combinatorial reasoning with sums

As described in Section 3.2, formalizing the prime number theorem involved a good deal of combinatorial reasoning with sums and products. Thus, we had to develop some basic theorems to support such reasoning, many of which have since been moved into Isabelle's HOL library. These include, for example,

lemma *setsum_cartesian_product:*
`"(∑x∈A. (∑y∈B. f x y)) = (∑z∈A <*> B. f (fst z) (snd z))"`

which allows one to view a double summation as a sum over a cartesian product; as well as

lemma *setsum_reindex:*
`"inj_on f B ⟹ (∑x∈f‘B. h x) = (∑x∈B. (h ∘ f)(x))"`

which expresses that if f is an injective function on a set B, then summing h over the image of B under f is the same as summing $h \circ f$ over B. In particular, if f is a bijection from B to A, the second identity implies that summing h over A is the same as summing $h \circ f$ over B. This type of "reindexing" is often so transparent in mathematical arguments that when we first came across an instance where we needed it (long ago, when proving quadratic reciprocity), it took some thought to identify the relevant principle. It is needed, for example, to show

$$\sum_{d|n} h(n) = \sum_{d|n} h(n/d),$$

using the fact that $f(d) = n/d$ is a bijection from the set of divisors of n to itself; or, for example, to show

$$\sum_{dd'=c} h(d, d') = \sum_{d|c} h(d, c/d),$$

using the fact that $f(d) = \langle d, c/d \rangle$ is a bijection from the set of divisors of c to $\{\langle d, d' \rangle \mid dd' = c\}$.

In Isabelle, if σ is any type, one also has the type of all subsets of σ. The predicate "finite" is defined inductively for these subset types. Isabelle's summation operator takes a subset A of σ and a function f from σ to any type with

an appropriate notion of addition, and returns $\sum_{x \in A} f(x)$. This summation operator really only makes sense when A is a finite subset, so many identities have to be restricted accordingly. (An alternative would be to define a type of finite subsets of σ, with appropriate closure operations; but then work would be required to translate properties of arbitrary subsets to properties of finite subsets, or to mediate relationships between finite subsets and arbitrary subsets.) This has the net effect that applying an identity involving a sum or product often requires one to verify that the relevant sets are finite. This difficulty is ameliorated by defining $\sum_{x \in A} f(x)$ to be 0 when A is infinite, since it then turns out that a number of identities hold in the unrestricted form. But this fix is not universal, and so finiteness issues tend to pop up repeatedly when one carries out a long calculation.

In short, at present, carrying out combinatorial calculations often requires a number of straightforward verifications involving reindexing and finiteness. Once again, these are inferences that are nearly transparent in ordinary mathematical texts, and so, by our general principle, we should expect mechanized proof assistants to take care of them. As before, there are stumbling blocks; for example, when reindexing is needed, the appropriate injection f has to be pulled from the air. We expect, however, that in the types of inferences that are commonly viewed as obvious, there are natural candidates for f. So this is yet another domain where reflection and empirical work should allow us to make proof assistants more usable.

4.5 Devising elementary proofs

Anyone who has undertaken serious work in formal mathematical verification has faced the task of adapting an ordinary mathematical proof so that it can be carried out using the libraries and resources available. When a proof uses mathematical "machinery" that is unavailable, one is faced with the choice of expanding the background libraries to the point where one can take the original proof at face value, or finding workarounds, say, by replacing the original arguments with ones that are more elementary. The need to rewrite proofs in such a way can be frustrating, but the task can also be oddly enjoyable: it poses interesting puzzles, and enables one to better understand the relationship of the advanced mathematical methods to the elementary substitutes. As more powerful mathematical libraries are developed, the need for elementary workarounds will gradually fade, and with it, alas, one good reason for investing time in such exercises.

Our decision to use Selberg's proof rather than a complex-analytic one is an instance of this phenomenon. To this day, we do not have a sense of how long it would have taken to build up a complex-analysis library sufficient to formalize one of the more common proofs of the prime number theorem, nor how much easier a formal verification of the prime number theorem would have been in the presence of such a library.

But similar issues arose even with respect to the mild uses of analysis required by the Selberg proof. Isabelle's real library gave us a good theory of limits, series, derivatives, and the basic transcendental functions, but it had almost no theory of integration to speak of. Rather than develop such a theory, we found that we were able to work around the mild uses of integration needed

in the Selberg proof.[4] Often, we also had to search for quick patches to other gaps in the underlying library. For the reader's edification and entertainment, we describe a few such workarounds here.

Recall that one of the fundamental identities we needed asserts

$$\ln(1 + 1/n) = 1/n + O(1/n^2).$$

This follows from the fact that $\ln(1 + x)$ is well approximated by x when x is small, which, in turn, can be seen from the Maclaurin series for $\ln(1 + x)$, or even the fact that the derivative of $\ln(1 + x)$ is equal to 1 at 0. But these were among the few elementary properties of transcendental functions that were missing from the real library. How could we work around this?

To be more specific: Fleuriot's real library defined e^x by the power series $e^x = \sum_{n=0}^{\infty} x^n/n!$, and showed that e^x is strictly increasing, $e^0 = 1$, $e^{x+y} = e^x e^y$ for every x and y, and the range of e^x is exactly the set of positive reals. The library then defines ln to be a left inverse to e^x. The puzzle was to use these facts to show that $|\ln(1 + x) - x| \leq x^2$ when x is positive and small enough.

Here is the solution we hit upon. First, note that when $x \geq 0$, $e^x \geq 1 + x$, and so, $x \geq \ln(1 + x)$. Replacing x by x^2, we also have

$$e^{x^2} \geq 1 + x^2. \tag{5}$$

On the other hand, the definition of e^x can be used to show

$$e^x \leq 1 + x + x^2 \tag{6}$$

when $0 \leq x \leq 1/2$. From (5) and (6) we have

$$\begin{aligned} e^{x-x^2} &= e^x/e^{x^2} \\ &\leq (1 + x + x^2)/(1 + x^2) \\ &\leq 1 + x, \end{aligned}$$

where the last inequality is easily obtained by multiplying through. Taking logarithms of both sides, we have

$$x - x^2 \leq \ln(1 + x) \leq x$$

when $0 \leq x \leq 1/2$, as required. In fact, a similar calculation yields bounds on $\ln(1 + x)$ when x is negative and close to 0. This can be used to show that the derivative of $\ln x$ is $1/x$; the details are left to the reader.

For another example, consider the problem of showing that $\sum_{n=1}^{\infty} 1/n^2$ converges. This follows immediately from the integral test: $\sum_{n=1}^{\infty} 1/n^2 \leq \int_1^{\infty} 1/x^2 = 1$. How can it be obtained otherwise? Answer: simply write

$$\begin{aligned} \sum_{n=1}^{M} 1/n^2 &\leq 1 + \sum_{n=2}^{M} 1/n(n-1) \\ &= 1 + \sum_{n=2}^{M} (1/(n-1) - 1/n) \\ &= 1 + 1 - 1/M \\ &\leq 2, \end{aligned}$$

[4]Since the project began, Sebastian Skalberg managed to import the more extensive analysis library from the HOL theorem prover to Isabelle. By the time that happened though, we had already worked around most of the applications of analysis needed for the proof.

where the second equality relies on the fact that the preceding expression involves a telescoping sum. Having to stop frequently to work out puzzles like these helped us appreciate the immense power of the Newton-Leibniz calculus, which provides uniform and mechanical methods for solving such problems. The reader may wish to consider what can be done to show that the sum $\sum_{n=1}^{\infty} 1/x^a$ is convergent for general values of $a > 1$, or even for the special case $a = 3/2$. Fortunately, we did not need these facts.

Now consider the identity

$$\sum_{n \leq x} 1/n = \ln x + O(1).$$

To obtain this, note that when x is positive integer we can write $\ln x$ as a telescoping sum,

$$\begin{aligned} \ln x &= \sum_{n \leq x-1} (\ln(n+1) - \ln n) \\ &= \sum_{n \leq x-1} \ln(1 + 1/n) \\ &= \sum_{n \leq x-1} 1/n + O(\sum_{n \leq x} 1/n^2) \\ &= \sum_{n \leq x} 1/n + O(1). \end{aligned}$$

We learned this trick from [8]. In fact, a slight refinement of the argument shows

$$\sum_{n \leq x} 1/n = \ln x + C + O(1/x)$$

for some constant, C. This constant is commonly known as Euler's constant, denoted by γ.

One last puzzle: how can one show that $\ln x/x^a$ approaches 0, for any $a > 0$? Here is our solution. First, note that we have $\ln x \leq \ln(1 + x) \leq x$ for every positive x. Thus we have

$$a \ln x = \ln x^a \leq x^a,$$

for every positive x and a. Replacing a by $a/2$ and dividing both sides by $ax^a/2$, we obtain $\ln x/x^a \leq 2/(ax^{a/2})$. It is then easy to show that the right-hand-side approaches 0 as x approaches infinity.

References

[1] Tom M. Apostol. *Introduction to analytic number theory.* Springer-Verlag, New York, 1976.

[2] Jeremy Avigad. Mathematics in Isabelle. http://www.andrew.cmu.edu/user/avigad/isabelle/.

[3] Jeremy Avigad. Number theory and elementary arithmetic. *Philosophia Mathematica*, 11:257–284, 2003.

[4] Jeremy Avigad. Notes on a formalization of the prime number theorem. Technical Report CMU-PHIL-163, Carnegie Mellon University, 2004.

[5] Jeremy Avigad and Kevin Donnelly. Formalizing O notation in Isabelle/HOL. In David Basin and Michaël Rusinowitch, editors, *Automated Reasoning: second international joint conference, IJCAR 2004*. Springer-Verlag, 2005.

[6] Jeremy Avigad and Harvey Friedman. Combining decision procedures for theories of the real numbers with inequality. In preparation.

[7] Clemens Ballarin. Locales and locale expressions in Isabelle/Isar. http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/packages/Isabelle/doc/locales.pdf.

[8] C. Cornaros and C. Dimitracopoulos. The prime number theorem and fragments of PA. *Arch. Math. Logic*, 33:265–281, 1994.

[9] Harold M. Edwards. *Riemann's zeta function*. Dover Publications Inc., Mineola, NY, 2001. Reprint of the 1974 original [Academic Press, New York].

[10] Thomas Hales. The flyspeck project fact sheet. http://www.math.pitt.edu/∼thales/flyspeck/.

[11] G. H. Hardy and E. M. Wright. *An introduction to the theory of numbers*. Oxford, fifth edition, 1979.

[12] G. J. O. Jameson. *The prime number theorem*. Cambridge University Press, Cambridge, 2003.

[13] Sean McLaughlin and John Harrison. A proof producing decision procedure for real arithmetic. In Robert Nieuwenhuis, editor, *Automated deduction – CADE-20. 20th international conference on automated deduction*, Springer-Verlag, 2005.

[14] Melvyn B. Nathanson. *Elementary methods in number theory*. Springer-Verlag, New York, 2000.

[15] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL. A proof assistant for higher-order logic*. Springer-Verlag, Berlin, 2002.

[16] Harold N. Shapiro. *Introduction to the theory of numbers*. John Wiley & Sons Inc., New York, 1983.

[17] Markus Wenzel. Type classes and overloading in higher-order logic. In E. Gunter and A. Felty, editors, *Proceedings of the 10th international conference on theorem proving in higher order logics (TPHOLs'97)*, pages 307–322, Murray Hill, New Jersey, 1997.

[18] Markus Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002.

[19] Freek Wiedijk. *The seventeen provers of the world.* Springer-Verlag, to appear.

[20] The Isabelle theorem proving environment. Developed by Larry Paulson at Cambridge University and Tobias Nipkow at TU Munich. http://www.cl.cam.ac.uk/Research/HVG/Isabelle/index.html.

[21] Proof general. http://proofgeneral.inf.ed.ac.uk/.

A formally verified proof of the prime number theorem

Jeremy Avigad

Department of Philosophy

Carnegie Mellon University

http://www.andrew.cmu.edu/∼avigad

The prime number theorem

Let $\pi(x)$ denote the number of primes less than or equal to x.

The prime number theorem: $\pi(x)/x$ is asymptotic to $1/\ln x$, i.e.

$$\lim_{x \to \infty} \pi(x) \ln x / x = 1.$$

Conjectured by Gauss and Legendre, on the basis of computation, around 1800; proved by Hadamard and de la Vallée Poussin in 1896.

Kevin Donnelly, David Gray, Paul Raff, and I used Isabelle to verify:

$$(\lambda x.\ pi\ x * ln\ (real\ x)\ /\ (real\ x)) \longrightarrow 1$$

Outline

- Historical background
- Overview of the Selberg proof
- Overview of the formalization
- Interesting aspects of the formalization
  - Asymptotic reasoning
  - Calculations with reals
  - Casts between natural numbers, integers, and reals
  - Combinatorial reasoning with sums
  - Elementary workarounds
- Heuristic procedures for the reals

Chebyshev's advances (∼1850)

$$\theta(x) = \sum_{p \le x} \ln p$$

$$\psi(x) = \sum_{p^a \le x} \ln p = \sum_{n \le x} \Lambda(n) \quad \text{where}$$

$$\Lambda(n) = \begin{cases} \ln p & \text{if } n = p^a, \text{ for some } a \ge 1 \\ 0 & \text{otherwise.} \end{cases}$$

- The prime number is equivalent to the statements $\lim_{x\to\infty} \theta(x)/x = 1$ and $\lim_{x\to\infty} \psi(x)/x = 1$.
- For x large enough,

$$0.92 < \pi(x) \ln x/x < 1.11.$$

More history

In 1859, Riemann introduces the complex-valued function, ζ.

In 1894, von Mangoldt reduced the PNT to showing that ζ has no roots with real part equal to 1.

This was done by Hadamard and de la Vallée Poussin, independently, in 1896.

In 1921, Hardy expressed doubts that there is a proof that does not essentially use these ideas.

In 1948, Selberg and Erdös found elementary proofs based on Selberg's "symmetry formula."

Outline

- Historical background
- Overview of the Selberg proof
- Overview of the formalization
- Interesting aspects of the formalization
  - Asymptotic reasoning
  - Calculations with reals
  - Casts between natural numbers, integers, and reals
  - Combinatorial reasoning with sums
  - Elementary workarounds
- Heuristic procedures for the reals

Asymptotic reasoning

View $\pi(x)$ as a step function from $\mathbb{R}$ to $\mathbb{R}$.

Analytic number theory provides a toolbox for characterizing growth rates.

For example, $f = O(g)$ means: there is a constant, C, such that for every x,

$$|f(x)| \leq C|g(x)|.$$

Sometimes, one really means "for all but a few exceptional cases of x," or "for large enough x."

Examples

Here are some identities involving ln:

$$\ln(1+1/n) = 1/n + O(1/n^2)$$

$$\sum_{n \le x} 1/n = \ln x + O(1)$$

$$\sum_{n \le x} \ln n = x \ln x - x + O(\ln x)$$

$$\sum_{n \le x} \ln n/n = \ln^2 x/2 + O(1)$$

These, and a few others, form a starting point for the Selberg proof.

Chebyshev's results

Fairly direct calculations yield $\theta(x)/x \to 1$ and $\pi(x)\ln x/x \to 1$ from $\psi(x)/x \to 1$.

This allows us to prove the prime number theorem in the form $\psi(x)/x \to 1$.

Along the way, we need $\psi(x) = O(x)$.

There is a nice way to do this, using binomial coefficients.

Combinatorial tricks

Since $d \mapsto n/d$ permutes the set of divisors of n,

$$\sum_{d|n} f(d) = \sum_{d|n} f(n/d).$$

Enumerating pairs d, d' such that $dd' \leq n$ in two different ways yields

$$\sum_{d \leq n} \sum_{d' \leq n/d} f(d, d') = \sum_{dd' \leq n} f(d, d') = \sum_{c \leq n} \sum_{d|c} f(d, c/d).$$

A similar argument yields

$$\sum_{d|n} \sum_{d'|(n/d)} f(d, d') = \sum_{dd'|n} f(d, d') = \sum_{c|n} \sum_{d|c} f(d, c/d).$$

Combinatorial tricks

The following is a version of the "partial summation formula": if $a \leq b$, $F(n) = \sum_{i=1}^{n} f(i)$, and G is any function, then

$$\sum_{n=a}^{b} f(n+1)G(n+1) = F(b+1)G(b+1) - F(a)G(a+1) - \sum_{n=a}^{b-1} F(n+1)(G(n+2) - G(n+1)).$$

This is a discrete analogue of integration by parts.

It is easily verified by induction.

Euler's function μ

A positive natural number n is *square free* if $n = p_1 p_2 \cdots p_s$ with p_i's distinct.

$$\mu(n) = \begin{cases} (-1)^s & \text{if } n \text{ is squarefree and } s \text{ is as above} \\ 0 & \text{otherwise.} \end{cases}$$

A remarkably useful fact regarding μ is that for $n > 0$,

$$\sum_{d|n} \mu(d) = \begin{cases} 1 & \text{if } n = 1 \\ 0 & \text{otherwise.} \end{cases}$$

This is clear for $n = 1$.

Euler's function μ

If $n = p_1^{j_1} p_2^{j_2} \cdots p_s^{j_s}$, define the *radical* of n to be $p_1 p_2 \cdots p_s$.

Then

$$\sum_{d|n} \mu(d) = \sum_{d|rad(n)} \mu(d)$$

$$= \sum_{d|rad(n),\, p_1|d} \mu(d) + \sum_{d|rad(n),\, p_1 \nmid d} \mu(d),$$

and the two terms cancel.

Möbius inversion

Suppose $f(n) = \sum_{d|n} g(d)$. Then

$$\begin{aligned}
\sum_{d|n} \mu(d) f(n/d) &= \sum_{d|n} \mu(d) \sum_{d'|(n/d)} g((n/d)/d') \\
&= \sum_{d|n} \sum_{d'|(n/d)} \mu(d) g((n/d)/d') \\
&= \sum_{c|n} \sum_{d|c} \mu(d) g(n/c) \\
&= \sum_{c|n} g(n/c) \sum_{d|c} \mu(d) \\
&= g(n),
\end{aligned}$$

expresses g in terms of f.

Selberg's formula

All these pieces come together in the proof of Selberg's symmetry formula:

$$\sum_{n \le x} \Lambda(n) \ln n + \sum_{n \le x} \sum_{d|n} \Lambda(d)\Lambda(n/d) = x \ln x + O(x).$$

There are many variants of this identity.

The reason it is useful is that there are two terms in the sum on the left, each sensitive to the presence of primes in different ways.

Selberg's proof involves cleverly balancing the two terms off each other, to show that in the long run, the density of the primes has the appropriate asymptotic behavior.

The error term

Let $R(x) = \psi(x) - x$ denote the "error term."

By Chebyshev's equivalences the prime number theorem amounts to the assertion $\lim_{x\to\infty} R(x)/x = 0$.

With some delicate calculation, the symmetry formula yields:

$$|R(x)| \ln^2 x \leq 2 \sum_{n\leq x} |R(x/n)| \ln n + O(x \ln x). \tag{1}$$

Selberg used this to show that, given a bound $|R(x)| \leq ax$ for sufficiently large x, one can get a better bound, $|R(x)| \leq a'x$, for sufficiently large x.

These bounds approach 0.

Outline

- Historical background
- Overview of the Selberg proof
- Overview of the formalization
- Interesting aspects of the formalization
  - Asymptotic reasoning
  - Calculations with reals
  - Casts between natural numbers, integers, and reals
  - Combinatorial reasoning with sums
  - Elementary workarounds
- Heuristic procedures for the reals

Overview of the formalization

To start with, we needed good supporing libraries:

- a theory of the natural numbers and integers, including properties of primes and divisibility, and the fundamental theorem of arithmetic
- a library for reasoning about finite sets, sums, and products
- a library for the real numbers, including properties of ln

More specific supporting libraries include:

- properties of the μ function, combinatorial identities, and variants of the Möbius inversion formula
- a library for asymptotic "big O" calculations
- a number of basic identities involving sums and ln
- Chebyshev's theorems

Overview of the formalization

Specific components of the Selberg proof are:

- the Selberg symmetry formula
- the inequality involving $R(n)$
- a long calculation to show $R(n)$ approaches 0

This outline is clearly discernible in the list of theory files, online at

http://www.andrew.cmu.edu/user/avigad/isabelle

Overview of the formalization

Here is a formulation of Möbius inversion:

$$\begin{aligned}&\mathit{ALL}\ n.\ (0 < n \longrightarrow\\ &\quad f\ n = (\sum\ d \mid d\ \mathit{dvd}\ n.\ g(n\ \mathit{div}\ d))) \Longrightarrow 0 < (n::\mathit{nat}) \Longrightarrow\\ &\quad g\ n = (\sum\ d \mid d\ \mathit{dvd}\ n.\ \mathit{of\text{-}int}(\mathit{mu}(\mathit{int}(d))) * f\ (n\ \mathit{div}\ d))\end{aligned}$$

Here is one of the identities given above:

$$\begin{aligned}&(\lambda x.\ \sum i{=}1..\mathit{natfloor}(\mathit{abs}\ x).\\ &\quad \mathit{ln}\ (\mathit{real}\ i)\ /\ (\mathit{real}\ i)) =o\\ &\quad (\lambda x.\ \mathit{ln}(\mathit{abs}\ x + 1)\hat{}2\ /\ 2) +o\ O(\lambda x.\ 1)\end{aligned}$$

Overview of the formalization

Here is a version of Selberg's symmetry formula:

$$(\lambda x.\ \sum n = 1..natfloor\ (abs\ x) + 1.$$
$$Lambda\ n * ln\ (real\ n)) + (\lambda x.\ \sum n{=}1..natfloor\ (abs\ x) + 1.$$
$$(\sum\ u \mid u\ dvd\ n.\ Lambda\ u * Lambda\ (n\ div\ u)))$$
$$=o\ (\lambda x.\ 2 * (abs\ x + 1) * ln\ (abs\ x + 1)) +o\ O(\lambda x.\ abs\ x + 1)$$

Finally, here is the error estimate provided above:

$$(\lambda x.\ abs\ (R\ (abs\ x + 1)) * ln\ (abs\ x + 1)\ \hat{}\ 2) <o$$
$$(\lambda x.\ 2 * (\sum\ n = 1..natfloor\ (abs\ x) + 1.$$
$$abs\ (R\ ((abs\ x + 1)\ /\ real\ n)) * ln\ (real\ n))) =o$$
$$O(\lambda x.\ (abs\ x + 1) * (1 + ln\ (abs\ x + 1)))$$

Overview of the formalization

There are at least three reasons not to provide too much detail:

- Our proof followed textbook presentations (due to Shapiro, Nathanson) closely.
- The proof scripts have not been polished, and so are not particularly nice.
- Much of it is not optimal; we know it is possible to do better.

Instead I will focus on:

- Details that diverge from the mathematical presentation.
- Novel features of the formalization.
- Areas where better support should be possible.

Overview of the formalization

Some statistics regarding length, and time, are given in the associated paper.

A lot of time and effort was spent:

- Building basic libraries of easy facts.
- Spelling out "straightforward" inferences.
- Finding the right lemmas and theorems to apply.
- Entering long formulas and expressions formally and correctly.

We suspect that these requirements will continue to diminish.

On a personal note, I am entirely convinced that formal verification of mathematics will eventually become commonplace.

Outline

- Historical background
- Overview of the Selberg proof
- Overview of the formalization
- Interesting aspects of the formalization
  - Asymptotic reasoning
  - Calculations with reals
  - Casts between natural numbers, integers, and reals
  - Combinatorial reasoning with sums
  - Elementary workarounds
- Heuristic procedures for the reals

Asymptotic reasoning

Define $O(g) = \{f \mid \exists C\ \forall x\ (|f(x)| \leq C|g(x)|)\}$.

Then take "equals" to be "element of" in $f = O(g)$.

The expression makes sense for any function type for which the codomain is an ordered ring.

We used Isabelle's axiomatic type classes to develop the theory in full generality.

Asymptotic reasoning

Define

$$f + g \equiv \lambda x.(f(x) + g(x))$$
$$a +_o B \equiv \{c \mid \exists b \in B\ (c = a + b)\}$$
$$a =_o B \equiv a \in B$$

This gives $f =_o g +_o O(h)$ the intended meaning.

Note that $x^2 + 3x = x^2 + O(x)$ really means

$$(\lambda x.\ x\hat{}2 + 3 * x) =o\ (\lambda x.\ x\hat{}2) +o\ O(\lambda x.\ x)$$

Asymptotic reasoning

Rewrite rules for addition of elements and sets:

set-plus-rearrange	$(a +_o C) + (b +_o D) = (a + b) +_o (C + D)$
set-plus-rearrange2	$a +_o (b +_o C) = (a + b) +_o C$
set-plus-rearrange3	$(a +_o C) + D = a +_o (C + D)$
set-plus-rearrange4	$C + (a +_o D) = a + (C + D)$

These put terms in the form $(a + b + \ldots) +_o (C + D + \ldots)$.

Asymptotic reasoning

Some monotonicity and arithmetic rules:

set-plus-intro	$[\|a \in C, b \in D\|] \Rightarrow a + b \in C + D$
set-plus-intro2	$b \in C \Rightarrow a + b \in a + C$
set-plus-mono	$C \subseteq D \Rightarrow a + C \subseteq a + D$
set-plus-mono2	$[\|C \subseteq D, E \subseteq F\|] \Rightarrow C + E \subseteq D + F$
set-plus-mono3	$a \in C \Rightarrow a + D \subseteq C + D$
set-plus-mono4	$a \in C \Rightarrow a + D \subseteq D + C$

Asymptotic reasoning

Some properties of O sets:

bigo-elt-subset	$f \in O(g) \Rightarrow O(f) \subseteq O(g)$
bigo-refl	$f \in O(f)$
bigo-plus-idemp	$O(f) + O(f) = O(f)$
bigo-plus-subset	$O(f + g) \subseteq O(f) + O(g)$
bigo-mult4	$f \in k + oO(h) \Rightarrow g \cdot f \in g \cdot k + oO(g \cdot h)$
bigo-compose1	$f \in O(g) \Rightarrow (\lambda x.\ f(k(x))) \in O(\lambda x.\ g(k(x)))$

Asymptotic reasoning

An annoyance: how do you indicate that $x^3 + 3x^2 + 1 = x^3 + O(x^2)$ for $x \geq 1$?

Options:

1. Define a type of positive reals (or integers).
2. Formalize "$f = O(g)$ on S"
3. Formalize "$f = O(g)$ eventually"
4. Write $(\lambda x.x^3 + 3x^2 + 1) =_o (\lambda x.x^3) +_o O(\lambda x.x^2 + 1)$

We chose the last. This accounts for the endless instances of "+1" and *abs* in our proofs.

The other options have drawbacks, too.

Calculations with real numbers

The very last part of the proof has, by far, the worst length ratio: a difficult 5 page calculation became 89 pages of formal text.

Reason: the need to carry out straightforward calculations by hand, especially involving inequalities.

Isabelle has:

- A term simplifier with ordered rewriting
- Decision procedures of linear and Presburger arithmetic

But lots of easy calculations go just beyond that.

Calculations with real numbers

$$(1 + \frac{\varepsilon}{3(C^* + 3)}) \cdot real(n) < Kx$$

follows from:

$$real(n) \leq (K/2)x$$

$$0 < C^*$$

$$0 < \varepsilon < 1$$

- Need monotonicity rules for arithmetic operations.
- Need to determine signs.
- Need to remember names like "mult-left-mono." "add-pos-nonneg," "order-le-less-trans," "exp-less-cancel-iff," "pos-divide-le-eq."
- Often need to type in long expressions, or cut and paste, or use explicit rules to manipulate terms

Calculations with the real numbers

Sign calculations keep coming back. Consider, for example,

$$1/(1 + st) < 1/(1 + su).$$

These inferences are covered by decision procedures for real closed fields, but

- They are slow.
- Worse: they do not extend to straightforward inferences with monotone functions, trigonometric functions, exponentiation and logarithm, etc.

Consider $x < y \Rightarrow 1/(1 + e^y) < 1/(1 + e^x)$.

Conclusion: we need principled heuristic procedures. (I will come back to this.)

Casting between domains

One can think of θ, ψ, and π as functions from $\mathbb{N}$ to $\mathbb{R}$ or from $\mathbb{R}$ to $\mathbb{R}$.

- Proofs use arithmetic properties of $\mathbb{N}$.
- Ultimately need to cast them to reals.

Recall that μ takes values $\{-1, 0, 1\}$, so we need to deal with integers too.

Casting was an endless source of headaches.

- We had parallel theories of primes and divisibility for ints and nats.
- We had to develop properties of *floor* and *ceiling* functions.
- We had to do annoying manipulations of mixed expressions, e.g. moving $+1$'s in and out of casts, etc.

Casting between domains

When extending a domain (e.g. nats to ints, or ints to reals):

- some operations are extended, like addition and multiplication
- some new operations are mirrored imperfectly in the smaller domain (e.g. $x \dot{-} y$ requires $x \geq y$, $x \mathit{div}\, y$ requires $y|x$).
- some properties depend on the choice of a left inverse, e.g.

$$(n \leq \lfloor x \rfloor) \equiv (\mathit{real}(n) \leq x).$$

The guiding motto should be: anything that is transparent to us should be transparent to a mechanized proof assistant.

Outline

- Historical background
- Overview of the Selberg proof
- Overview of the formalization
- Interesting aspects of the formalization
  - Asymptotic reasoning
  - Calculations with reals
  - Casts between natural numbers, integers, and reals
  - Combinatorial reasoning with sums
  - Elementary workarounds
- Heuristic procedures for the reals

Combinatorial reasoning with sums

Some of our theorems are now in Isabelle's HOL library. For example:

$$\mathit{inj\text{-}on}\ f\ B \Longrightarrow (\sum x{\in}f`B.\ h\ x) = (\sum x{\in}B.\ (h \circ f)(x))$$

"reindexes" a sum.

It is needed, for example, to show

$$\sum_{d|n} h(n) = \sum_{d|n} h(n/d),$$

using $f(d) = n/d$, and

$$\sum_{dd'=c} h(d, d') = \sum_{d|c} h(d, c/d),$$

using $f(d) = \langle d, c/d \rangle$.

Combinatorial reasoning with sums

In the Isabelle formalization, $\sum_{x \in A} f(x)$ is notation for *setsum A f*.

This really only makes sense when A is finite, so finiteness verifications keep popping up in calculations.

(Defining *setsum A f* to be 0 when A is infinite helps.)

According to our motto, there should be better support for finiteness and reindexing.

Elementary workarounds

We relied on the Selberg proof because Isabelle didn't (and still doesn't) have a complex analysis library.

We still don't have a sense of how long it would take to:

- develop a sufficient complex analysis library
- formalize the complex-analytic proof

Of course, the task of finding elementary workarounds is part of the business. It can be oddly enjoyable.

Alas, the need to do this will diminish as formal libraries improve.

Elementary workarounds

Question: how to prove $\ln(1+x) \approx x$ when x is small?

The Isabelle library did not compute the derivative of ln.

It had:

- By definition, $e^x = \sum_{n=0}^{\infty} x^n/n!$
- e^x strictly increasing
- $e^0 = 1$, $e^{x+y} = e^x e^y$
- e^x is surjective on the positive reals
- By definition, $\ln x$ is a left inverse to e^x

Puzzle: show $|\ln(1+x) - x| \leq x^2$ when x is positive and small enough.

Elementary workarounds

Our solution: $x \geq 0$ implies $e^x \geq 1 + x$, so $x \geq \ln(1 + x)$. Replacing x by x^2, we also have $e^{x^2} \geq 1 + x^2$.

On the other hand, the definition of e^x can be used to show

$$e^x \leq 1 + x + x^2$$

when $0 \leq x \leq 1/2$. From these we get

$$e^{x-x^2} = e^x / e^{x^2} \leq (1 + x + x^2)/(1 + x^2) \leq 1 + x.$$

Taking logarithms of both sides, we have

$$x - x^2 \leq \ln(1 + x) \leq x$$

when $0 \leq x \leq 1/2$, as required.

Elementary workarounds

Another puzzle: show

$$\sum_{n \le x} 1/n = \ln x + O(1)$$

without integration. When x is positive, write

$$\begin{aligned}
\ln x &= \sum_{n \le x-1} (\ln(n+1) - \ln n) \\
&= \sum_{n \le x-1} \ln(1 + 1/n) \\
&= \sum_{n \le x-1} 1/n + O(\sum_{n \le x} 1/n^2) \\
&= \sum_{n \le x} 1/n + O(1).
\end{aligned}$$

Outline

- Historical background
- Overview of the Selberg proof
- Overview of the formalization
- Interesting aspects of the formalization
  - Asymptotic reasoning
  - Calculations with reals
  - Casts between natural numbers, integers, and reals
  - Combinatorial reasoning with sums
  - Elementary workarounds
- Heuristic procedures for the reals

Heuristic procedures for the reals

Remember the example: verify

$$(1 + \frac{\varepsilon}{3(C^* + 3)}) \cdot real(n) < Kx$$

using the following hypotheses:

$$real(n) \leq (K/2)x$$

$$0 < C^*$$

$$0 < \varepsilon < 1$$

Idea: work backwards, applying obvious monotonicity rules.

Heuristic procedures for the reals

Problems:

1. Case splits: e.g. $st > 0 \equiv (s > 0 \wedge t > 0) \vee (s < 0 \wedge t < 0)$.
2. Nondeterminism: e.g. many ways to show $s + t < u + v + w$.

Observations:

1. "Straightforward" inferences usually don't need case splits.
2. In practice, Fourier-Motzkin is efficient for linear inequalities.
3. Modulo cases over signs, the same thing works for the multiplicative fragment of the reals.

Heuristic procedures for the reals

Let T_1 be the theory of $\langle \mathbb{R}, 0, +, < \rangle$. T_1 is decidable.

Let T_2 be the theory of $\langle \mathbb{R}, 1, \times, < \rangle$. T_2 is decidable.

Let $T = T_1 \cup T_2$. By Nelson-Oppen methods, the universal fragment of T is decidable.

Problem: T is too weak; it doesn't prove $2 \times 2 = 4$.

Heuristic procedures for the reals

A better version: let $f_a(x) = ax$ for rational constants a.

Let $T_1[\mathbb{Q}]$ be the theory of $\langle \mathbb{R}, 0, 1, +, -, <, \ldots, f_a, \ldots \rangle$.

Let $T_2[\mathbb{Q}]$ be the theory of $\langle \mathbb{R}, 0, 1, \times, \div, \sqrt[n]{\cdot}, <, \ldots, f_a, \ldots \rangle$.

Let $T[\mathbb{Q}] = T_1[\mathbb{Q}] \cup T_2[\mathbb{Q}]$.

Both of these are decidable, but Nelson-Oppen methods fail when there is a nontrivial overlap.

The situation here is much more complex!

Heuristic procedures for the reals

This is joint work with Harvey Friedman.

Here are some things we (think we) know:

- $T[\mathbb{Q}]$ has good normal forms.
- Valid equations are independent of the ordering.
- $T[\mathbb{Q}]$ is undecidable.
- In fact, the $\forall\forall\forall\exists \ldots \exists$ fragment is complete r.e.
- Assuming that the solvability of Diophantine equations in the rationals is undecidable, then so is the existential fragment of $T[\mathbb{Q}]$.
- The universal fragment of $T[\mathbb{Q}]$ *is* decidable.

We have similar results, for example, with the real algebraic numbers A in place of $\mathbb{Q}$.

Heuristic procedures for the reals

Our decidability results are not practical. But the proofs provide ideas and guidelines.

General strategy for amalgamation:

- Maintain a database of facts in the common language.
- Iteratively use each of T_1 and T_2 to add new facts.

Issues:

- Heuristically, how to decide *which* facts to focus on?
- When to split on cases?
- How to look for disjunctions?
- How to incorporate distributivity?
- How to amalgamate other local decision or heuristic procedures?

Conclusions

Formally verified mathematics is becoming increasingly important:

- Proofs are getting very complex.
- Proofs rely on extensive computations.

Fortunately, we are entering "the golden age of metamathematics" (Shankar).

Continued progress will require

- thoughtful reflection
- good theory
- solid engineering

This makes the field an auspicious combination of theory and practice.

Bishop's set theory[1]

Erik Palmgren
Uppsala Universitet
www.math.uu.se/~palmgren

TYPES summer school
Göteborg
August 2005

[1] Errett Bishop (1928-1983) constructivist mathematician.

Introduction - What is a set?

The iterative notion of set (G. Cantor 1890, E. Zermelo 1930)

- sets built up by collecting objects, or other sets, according to some selection criterion $Q(x)$

$$\{x \mid Q(x)\}$$

Frege's "naive" set theory is inconsistent (Russell's paradox). Remedy: introduce size limitations, use explicit set constructions as power sets, products or function sets, start from given sets X

$$\{x \in X \mid Q(x)\}$$

Encoding of mathematical objects as iterative sets

All mathematical objects are built from the empty set (E. Zermelo 1930)

Natural numbers are for example usually encoded as

$$0 = \emptyset \quad 1 = 0 \cup \{0\} = \{\emptyset\} \quad 2 = 1 \cup \{1\} = \{\emptyset, \{\emptyset\}\} \quad \cdots \quad .$$

Pairs of elements can be encoded as $\langle a, b \rangle = \{\{a\}, \{a, b\}\}$. Functions are certain sets of pairs objects ... etc.

Quotient structures are constructed by the method of equivalence classes — only one notion of equality is necessary.

(J.Myhill and P.Aczel (1970s): constructive versions of ZF set theory.)

What is a set? A more basic view

"A set is not an entity which has an ideal existence: a set exists only when it has been defined. To define a set we prescribe, at least implicitly, what we (the constructing intelligence) must do in order to construct an element of the set, and what we must do to show that two elements are equal" (Errett Bishop, Foundations of Constructive Analysis, 1967.)

Martin-Löf type theory conforms to this principle of defining sets.

Abstraction levels

One may disregard the particular representations of set-theoretic constructions, and describe their properties abstractly (in the spirit of Bourbaki).

For instance, the cartesian product of two sets A and B may be described as a set $A \times B$ together with two *projection* functions

$$\pi_1 : A \times B \to A \qquad \pi_2 : A \times B \to B,$$

such that for each $a \in A$ and each $b \in B$ there exists a unique element $c \in A \times B$ with $\pi_1(c) = a$ and $\pi_2(c) = b$. Thus π_k picks out the kth component of the abstract pair.

Reference to the particular encoding of pairs is avoided. This is a good principle in mathematics as well as in program construction.

Some references using Bishop's set theory

E. Bishop and D.S. Bridges (1985). *Constructive Analysis.* Springer-Verlag.

D.S. Bridges and F. Richman (1987). *Varieties of Constructive Mathematics.* London Mathematical Society Lecture Notes, Vol. 97. Cambridge University Press.

R. Mines, F. Richman and W. Ruitenburg (1988). *A Course in Constructive Algebra.* Springer.

Among constructivists, one often says that **constructive mathematics is mathematics based on intuitionistic logic.**

Plan of lectures

(Based on Ch. 3 and 4 of *Type-theoretic foundation of constructive mathematics* by T. Coquand, P. Dybjer, E. Palmgren and A. Setzer, version August 5, 2005.)

1. Introduction

2. Terminology for type theory

3. Intuitionistic logic

4. Sets and equivalence relations

5. Choice sets and axiom of choice

6. Relations and subsets

7. Finite sets and relatives

8. Quotients

9. Universes and restricted power sets

10. Categories

11. Relation to categorical logic

Exercises: see lecture notes.

2. Terminology for type theory

later Martin-Löf	lect. notes	Bishop	early M.-L.	other
type	sort		category	kind
set	type	preset	type	
extensional set	set	set		setoid, E-set
function	operation	operation	function	
extensional function	function	function		setoid map, E-function

(Thanks for the table, Peter!)

The application of an operation $f : A \to B$ to an element $a : A$ is denoted

$$f\ a$$

Recall: A proposition may be regarded as a type according to the following translation scheme

$(\forall x : A)P\,x$	$(\Pi x : A)P\,x$
$(\exists x : A)P\,x$	$(\Sigma x : A)P\,x$
$P \wedge Q$	$P \times Q$
$P \vee Q$	$P + Q$
$P \Rightarrow Q$	$P \to Q$
$\top$	N_1
$\bot$	N_0
$\neg P \quad (= P \Rightarrow \bot)$	$P \to \mathrm{N}_0$

The judgement

$$A \text{ is true}$$

means that there is some p so that $p : A$.

Relations and predicates on types

A *predicate P on a type X* is a family of propositions $P\,x\ (x : X)$.

A *relation R between types X and Y* is a family of propositions $R\,x\,y\ (x : X, y : Y)$. If $X = Y$, we say that R is a *binary relation* on X.

A binary relation R on X is an *equivalence relation* if there are functions ref, sym and tra with

$$ref\ a : R\,a\,a \quad (a : X),$$

$$sym\,a\,b\,p : R\,b\,a \quad (a : X, b : X, p : R\,a\,b),$$

$$tra\,a\,b\,c\,p\,q : R\,a\,c \quad (a, b, c : X, p : R\,a\,b, q : R\,b\,c).$$

We may suppress the proof objects and simply write, for instance in the last line

$$R\ a\ c\ \mathrm{true} \qquad (a, b, c : X, R\ a\ b\ \mathrm{true}, R\ b\ c\ \mathrm{true}),$$

which is equivalent to

$$(\forall a : X)(\forall b : X)(\forall c : X)(R\ a\ b \wedge R\ b\ c \Rightarrow R\ a\ c)\ \mathrm{true}.$$

3. Intuitionistic logic

The logic governing the judgements of the form

$$A \text{ true}$$

is intuitionistic logic. It is best described by considering the derivation rules for natural deduction and then remove the Reductio Ad Absurdum rule (principle of indirect proof):

Derivation rules:

$$\frac{A \qquad B}{A \wedge B}\,(\wedge I) \qquad \frac{A \wedge B}{A}\,(\wedge E1) \qquad \frac{A \wedge B}{B}\,(\wedge E2)$$

$$\frac{\begin{array}{c}\overline{A}^{h}\\ \vdots\\ B\end{array}}{A \to B}(\to I, h) \qquad \frac{A \to B \qquad A}{B}\,(\to E)$$

$$\frac{A}{A \vee B}\,(\vee I1) \qquad \frac{B}{A \vee B}\,(\vee I2) \qquad \frac{A \vee B \qquad \begin{array}{c}\overline{A}^{h_1}\\ \vdots\\ C\end{array} \qquad \begin{array}{c}\overline{B}^{h_2}\\ \vdots\\ C\end{array}}{C}\,(\vee E, h_1, h_2)$$

$$\frac{A}{(\forall x)A}\,(\forall I) \qquad \frac{(\forall x)A}{A[t/x]}\,(\forall E)$$

$$\frac{A[t/x]}{(\exists x)A}\,(\exists I)$$

$$\frac{(\exists x)A \qquad \begin{array}{c}\overline{A}^{h}\\ \vdots \\ C\end{array}}{C}\,(\exists E,h)$$

$$\frac{\bot}{A}\,(\bot E)$$

$$\frac{\begin{array}{c}\overline{\neg A}^{h}\\ \vdots \\ \bot\end{array}}{A}\,(RAA,h)$$

4. Sets and equivalence relations

Definition A *set* X is a type $\underline{X}$ together with an equivalence relation $=_X$ on $\underline{X}$. Write this as

$$X = (\underline{X}, =_X).$$

We shall also write $x \in X$ for $x : \underline{X}$.

Remark

In Bishop (1967) $\underline{X}$ is called a *preset,* rather than a type.

In the type theory community $X = (\underline{X}, =_X)$ is often known as a *setoid*.

Examples Let N be the type of natural numbers. Define equivalence relations

$x =_{\mathbb{N}} y$ iff $\mathrm{Tr}\ (\mathrm{eq}_{\mathrm{N}}\ x\ y)$

(Here $\mathrm{eq}_N : N \to N \to \mathrm{Bool}$ is the equality tester for N and $\mathrm{Tr}\ \mathrm{tt} = \top$ and $\mathrm{Tr}\ \mathrm{ff} = \bot$)

$x =_n y$ iff $x - y$ is divisble by n

Then

- $\mathbb{N} = (\mathrm{N}, =_{\mathbb{N}})$ is the *set of natural numbers*
- $\mathbb{Z}_n = (\mathrm{N}, =_n)$ is the *set of integers modulo* n.

Functions vs operations

What is usually called functions in type theory, we call here *operations*.

Definition. A *function* f from the set X to the set Y is a pair $(\underline{f}, \mathrm{ext}_f)$ where $\underline{f} : \underline{X} \to \underline{Y}$ is an operation so that

$$(\mathrm{ext}_f\ a\ b\ p) : \underline{f}\ a =_Y \underline{f}\ b \quad (a, b : \underline{X}, p : a =_X b).$$

To conform with usual mathematical notation, function application will be written

$$f(a) =_{\mathrm{def}} \underline{f}\ a$$

Two functions $f, g : X \to Y$ are *extensionally equal*, $f =_{[X \to Y]} g$, if there is e with

$$e\ a : f(a) =_Y g(a) \quad (a \in X).$$

Set constructions

The product of sets A and B is a set $P = (\underline{P}, =_P)$ where $\underline{P} = \underline{A} \times \underline{B}$ (cartesian product as types) and the equality is defined by

$(x, y) =_P (u, v)$ iff $x =_A u$ and $y =_B v$.

Standard notation for this P is $A \times B$. Projection function are $\pi_1(x, y) = x$ and $\pi_1(x, y) = y$. This construction can be verified to satisfy the abstract property (page 5). (It can as well be expressed by the categorical universal property for products.)

The disjoint union $A \dot{\cup} B$ (or $A + B$) is definied by considering the corresponding type construction.

The functions from A to B form a set B^A defined to be the type

$$(\Sigma f : \underline{A} \to \underline{B})(\forall x, y : \underline{A})[x =_A y \to f\,x =_B f\,y],$$

together with the equivalence relation

$$(f, p) =_{B^A} (g, q) \iff_{\text{def}} (\forall x : \underline{A})\, f\,x =_B g\,x.$$

The evaluation function $\mathrm{ev}_{A,B} : B^A \times A \to B$ is given by

$$\mathrm{ev}_{A,B}((f, p), a) = f\,a$$

Proposition. Let A, B and X be sets. For every function $h : X \times A \to B$ there is a unique function $\hat{h} : X \to B^A$ with

$$\mathrm{ev}_{A,B}(\hat{h}(x), y) = h(x, y) \qquad (x \in X, y \in A).$$

A set X is called *discrete,* if for all $x, y \in X$

$$(x =_X y) \vee \neg (x =_X y).$$

In classical set theory all sets are discrete. This is not so constructively, but we have

Proposition. The unit set $\mathbf{1}$ and the set of natural numbers $\mathbb{N}$ are both discrete. If X and Y are discrete sets, then $X \times Y$ and $X + Y$ are discrete too.

However, the assumption that $\mathbb{N}^{\mathbb{N}}$ is discrete implies a nonconstructive principle (WLPO):

$$(\forall n \in \mathbb{N}) f(n) = 0 \vee \neg (\forall n \in \mathbb{N}) f(n) = 0$$

Coarser and finer equivalences

An equivalence relation $\sim$ is *finer* than another equivalence relation $\approx$ on a type $\underline{A}$ if for all $x, y : A$

$$x \sim y \Longrightarrow x \approx y.$$

It is easy to prove by induction that $=_{\mathbb{N}}$ is the finest equivalence relation on $\mathbb{N}$.

If there is a finest equivalence relation $=_A$ on a type $\underline{A}$, the set $A = (\underline{A}, =_A)$ has the *substitutivity property*

$$x =_A y \Longrightarrow (Px \Leftrightarrow Py)$$

for any predicate P on the type $\underline{A}$.

Sets are rarely substitutive, and the notion is not preseved by isomorphisms. $\mathbb{Z}_n$ as constructed above is not substitutive; an isomorphic construction yields substitutivity.

Theorem. To any type $\underline{A}$, the identity type construction Id assigns a finest equivalence $\mathrm{Id}\,\underline{A}$. The resulting set, also denoted $\underline{A}$, is substitutive.

Remark. Substitutive sets are however very convenient for direct formalisation in e.g. Agda, as extensionality proofs can be avoided.

5. Choice sets and axiom of choice

A set S is a *choice set,* if for any surjective function $f : X \to S$, there is right inverse $g : S \to X$, i.e.

$$f(g(s)) = s \qquad (s \in S).$$

Theorem. Every substitutive set is a choice set.

(Zermelo's) Axiom of Choice may be phrased thus:

Every set is a choice set.

Theorem. Zermelo's AC implies the law of excluded middle.

Though Zermelo's AC is incompatible with constructivism, there is related axiom (theorem of type theory) freely used in Bishop constructivism.

Theorem. For any set A there is a choice set $\underline{A}$ and surjective function $p : \underline{A} \to A$. (In categorical logic often referred to as "existence of enough projectives".)

As a consequence, Dependent Choice is valid (see notes, p. 76).

Theorem. If A and B are choice sets, then so are $A \times B$ and $A + B$.

6. Relations and subsets

Definition A *(extensional) property* P of the set X is a family of propositions $P\,x$ $(x \in X)$ with

$$x =_X y, P\,x \Longrightarrow P\,y.$$

We also say that P is a *predicate* on X.

A *relation* R between sets X and Y is a family of propositions $R\,x\,y$ $(x \in X, y \in Y)$ such that

$$x =_X x', y =_Y y', R\,x\,y \Longrightarrow R\,x'\,y'.$$

The relation is *univalent* if $y =_Y y'$, whenever $R\,x\,y$ and $R\,x\,y'$.

Write $P(x)$, $R(x,y)$ etc. in the extensional situation.

Restatement of choice principles for relations.

The following is the theorem of *unique choice.*

Thm. Let R be a univalent relation between the sets X and Y. It is total if, and only if, there exists a function $f : X \to Y$, called a *selection function,* such that

$$R(x, f(x)) \qquad (x \in X).$$

(This function is necessarily unique if it exists.)

An alternative characterisation of choice sets is

Thm. A set X is a choice set iff for every set Y, each total relation R between X and Y has a selection function $g : X \to Y$ so that

$$R(x, g(x)) \qquad (x \in X).$$

Dependent choice

Dependent choice. Let A be a set which is the surjective image of a choice set. Let R be a binary relation on A such that

$$(\forall x \in A)(\exists y \in A)R(x,y).$$

Then for each $a \in A$, there exists a function $f : \mathbb{N} \to A$ with $f(0) = a$ and

$$R(f(n), f(n+1)) \qquad (n \in \mathbb{N}).$$

Proof. Let $p : P \to A$ be surjective, where P is a choice set. By surjectivity, we have

$$(\forall u \in P)(\exists v \in P)R(p(u), p(v)).$$

Since P is a choice set we find $h : P \to P$ with $R(p(u), p(h(u)))$ for all $u \in P$.

For $a \in A$, there is $b_0 \in P$ with $a = p(b_0)$.

Define by recursion $g(0) = b_0$ and $g(n+1) = h(g(n))$, and let $f(n) = p(g(n))$. Thus $R(p(g(n)), p(g(n+1)))$, so f is indeed the desired choice function. $\square$

Remark Thus we have proved the general dependent choice theorem in type theory with identity types. We also get another proof of countable choice, without requiring a particular subsititutive construction of natural numbers.

Subsets as injective functions

Let X be a set. A *subset* of X is a pair $S = (\partial S, \iota_S)$ where ∂S is a set and $\iota_S : \partial S \to X$ is an injective function.

An element $a \in X$ is a *member of S* (written $a \in_X S$) if there exists $d \in \partial S$ with $a =_X \iota_S(d)$.

Inclusion $\subseteq_X$ and equality $\equiv_X$ of subsets of X can be defined in the usual logical way.

Prop. For subsets A and B of X, the inclusion $A \subseteq_X B$ holds iff there is a function $f : \partial A \to \partial B$ with $\iota_B \circ f = \iota_A$. (Such f are unique and injective.)

The subsets are equal iff f is a bijection.

Separation of subsets

For a property P on a set X, the subset

$$\{x \in X \mid P(x)\} = \Big(\{x \in X : P(x)\}, \mathfrak{1}\Big)$$

is defined by the data:

$$\underline{\{x \in X : P(x)\}} =_{\text{def}} (\Sigma x \in X) P(x)$$

and

$$\langle x, p\rangle =_{\{x \in X:\, P(x)\}} \langle y, q\rangle \Longleftrightarrow_{\text{def}} x =_X y$$

and $\mathfrak{1}(\langle x, p\rangle) =_{\text{def}} x$.

(Note the pedantic syntactic distinction of ":" and "$\mid$".)

Note that

$$\begin{aligned} a \in_X \{x \in X \mid P(x)\} &\Leftrightarrow (\exists d \in \{x \in X \,:\, P(x)\})\, a = \mathfrak{l}(d) \\ &\Leftrightarrow (\exists x \in X)(\exists p : \underline{P}\, x)\, a = \mathfrak{l}(\langle x, p\rangle) \\ &\Leftrightarrow \underline{P}\, a \\ &\Leftrightarrow P(a) \end{aligned}$$

The usual set-theoretic operations $\cap$, $\cup$, $\overline{(\ )}$ can now be defined "logically" for subsets.

A subset S of X is *decidable*, or *detachable,* if for all $a \in X$

$$a \in_X S \vee \neg(a \in_X S).$$

Union of subsets: logical definition.

Let $A = (\partial A, \iota_A)$ and $B = (\partial B, \iota_B)$ be subsets of X.

Their union is the following subset of X

$$A \cup B = \{z \in X \mid z \in_X A \text{ or } z \in_X B\}.$$

Taking $U = A \cup B$ apart as $U = (\partial U, \iota_U)$ we see that $\underline{\partial U}$ is

$$(\Sigma z : \underline{X})(z \in_X A \text{ or } z \in_X B) \quad = \quad (\Sigma z : \underline{X})((z \in_X A) + (z \in_X B)).$$

whereas $\iota_U(z, p) = z$.

Complement

The complement of the subset A of X is defined as

$$\overline{A} = \{z \in X \mid \neg z \in_X A\}.$$

For $\overline{A} = (\partial C, \mathfrak{u}_C)$ we have

$$\underline{\partial C} = (\Sigma z : \underline{X})((z \in_X A) \to \bot).$$

That A is a decidable subset of X can be expressed as $A \cup \overline{A} = X$.

The decidable subsets form a boolean algebra.

Partial functions

A *partial function* f from A to B consists of a subset $(\mathrm{D}_f, \mathrm{d}_f)$ of A, its *domain of definition* (denoted $\mathrm{dom}\, f$) and a function $\mathrm{m}_f : \mathrm{D}_f \to B$. We write this with a special arrow symbol as $f : A \rightharpoonup B$.

Such $f : A \rightharpoonup B$ is *total* if its domain of definition equals A as a subset, or equivalently, if d_f is an isomorphism.

Another partial function $g : A \rightharpoonup B$ *extends* f, writing $f \subseteq g : A \rightharpoonup B$, if for each $s \in \mathrm{D}_f$ there exists $t \in \mathrm{D}_g$ with $\mathrm{d}_f(s) = \mathrm{d}_g(t)$ and $\mathrm{m}_f(s) = \mathrm{m}_g(t)$. If both $f \subseteq g$ and $g \subseteq f$, we define f and g to be equal as partial functions.

Example. Let $F = (F, \cdot, +, 0, 1)$ be a field, and let

$$U = \{x \in F \mid (\exists y \in F) x \cdot y = 1\}$$

be the subset of invertible elements. Define a function $\mathrm{m}_r : \partial U \to F$ to be $\mathrm{m}_r(x) = y$, where y is unique such that $x \cdot y = 1$. Thus the reciprocal is a partial function $r = (\cdot)^{-1} : F \rightharpoonup F$.

In fact, for any univalent relation R between sets X and Y there is partial function $f_R = (D, d, m)$ given by

$$\partial D = \{u \in X \times Y : R(\pi_1(u), \pi_2(u))\}$$

$d = \pi_1 \circ \mathfrak{l}_D$ and $m = \pi_2 \circ \mathfrak{l}_D$.

Example For any pair of subsets A and B of X that are disjoint $A \cap B = \emptyset$, we may define a partial characteristic function

$$\chi : X \rightharpoonup \{0,1\}$$

satisfying

$\chi(z) = 0$ iff $z \in_X A$,

$\chi(z) = 1$ iff $z \in_X B$,

by considering the univalent relation $R(z,n)$:

$$(z \in_X A \wedge n = 0) \vee (z \in_X B \wedge n = 1).$$

Partial functions are composed in the following manner: if $f : A \rightharpoonup B$ and $g : B \rightharpoonup C$, define the composition $h = g \circ f : A \rightharpoonup C$ by

$$\mathrm{D}_h = \{(s,t) \in \mathrm{D}_f \times \mathrm{D}_g : \mathrm{m}_f(s) = \mathrm{d}_g(t)\}$$

The function $\mathrm{d}_h : \mathrm{D}_h \to A$ given by composing the projection to D_f with d_d is injective. The function $\mathrm{m}_h : \mathrm{D}_h \to C$ is defined by the composition of the projection to D_g and d_g.

7. Finite sets and relatives

The *canonical n-element set* is

$$\mathbb{N}_n = \{k \in \mathbb{N} : k < n\} \hookrightarrow \mathbb{N}.$$

Any set X isomorphic to such a set is called *finite.* It may be written

$$\{x_0, \ldots, x_{n-1}\}$$

where $k \mapsto x_k : \mathbb{N}_n \to X$ is the isomorphism.

Since $x_j = x_k$ iff $j = k$, we can always decide whether two elements of a finite set are equal by comparison of indices.

A related notion is more liberal:

A set X is called *subfinite,* or *finitely enumerable,* if there is, for some $n \in \mathbb{N}$, a surjection $x : \mathbb{N}_n \to X$.

Here we are only required to enumerate the elements, not tell them apart.

We can always tell whether a subfinite set is empty by checking if $n = 0$.

Remark. A subset of a finite set need not be finite, or even subfinite. Consider

$$\{0 \in \mathbb{N}_1 : P\}$$

where P is some undecided proposition.

Some basic properties

Let X and Y be sets. Then:

(i) X finite $\iff$ X subfinite and discrete

(ii) X subfinite, $f : X \to Y$ surjective $\implies$ Y subfinite

(iii) Y discrete, $f : X \to Y$ injective $\implies$ X discrete

(iv) Y discrete, $X \hookrightarrow Y \implies X$ discrete

(v) Y finite, $X \hookrightarrow Y$ decidable $\implies X$ finite.

8. Quotients

Let $X = (\underline{X}, =_X)$ be a set and let $\sim$ be a relation on this set. Then by the extensionality of the relation

$$x =_X y \Longrightarrow x \sim y. \tag{1}$$

Thus if $\sim$ is an equivalence relation on X

$$X/\sim = (\underline{X}, \sim)$$

is a set, and $q : X \to X/\sim$ defined by $q(x) = x$ is a surjective function.

We have the following extension property. If $f : X \to Y$ is a function with

$$x \sim y \Longrightarrow f(x) =_Y f(y), \tag{2}$$

then there is a unique function $\overline{f} : X/\!\sim \to Y$ (up to extensional equality) with

$$\overline{f}(i(x)) =_Y f(x) \qquad (x \in X).$$

We have constructed *the quotient of X with respect to $\sim$*: $q : X \to X/\!\sim$

Remark. Every set is a quotient of a choice set. Namely, X is the quotient of $\underline{X}$ w.r.t. $=_X$.

Proposition. A set is subfinite iff it is the quotient of a finite set.

9. Universes and restricted powersets

A general problem with (or feature of) predicative theories like Martin-Löf type theory is their inability to define a set of *all* subsets of a given set. It is, though, often sufficient to consider certain restricted classes of subsets in a certain situation.

A *set-indexed family* $\mathcal{F} = (F, I)$ *of subsets* of a given set X consists of an *index set* $I = (\underline{I}, =_I)$ and a subset F_i of X for each $i : \underline{I}$, which are such that if $i =_I j$ then F_i and F_j are equal as subsets of X.

A subset S of X *belongs to the family* $\mathcal{F}$, written $S \in \mathcal{F}$, if $S = F_i$ (as subsets of X) for some $i \in I$.

Consider any family of types $\mathcal{U} = (T, U)$, where $T\,i$ is a type for each $i : U$. It represents a collection of sets, the $\mathcal{U}$-sets, as follows.

First, a *$\mathcal{U}$-representation* of a set is a pair $r = (i_0, e)$ where $i_0 : I$ and $e : T\,i_0 \times T\,i_0 \to U$ is an operation so that

$$a =_r b \Leftrightarrow_{\mathrm{def}} T(e\;a\;b)$$

defines an equivalence relation on the type $T\,i_0$. Then this is a set

$$\hat{r} = (T\,i_0, =_r).$$

A set X is *$\mathcal{U}$-representable,* or simply a *$\mathcal{U}$-set*, if it is in bijection with $\hat{r}$ for some $\mathcal{U}$-representation r. The $\mathcal{U}$-sets defines, in fact, a full subcategory of the category of sets, equivalent to a small category.

Example For $U = \mathrm{N}$ and $T\,n = \mathrm{N}_n$, the $(\mathrm{N}, \mathrm{N}_{(-)})$-sets are the finite sets.

Restricted power sets

For any set X and any family of types $\mathcal{U}$, define the family $\mathcal{R}_{\mathcal{U}}(X)$ of subsets of X as follows.

- Its index set I consists of triples (r, m, p) where r is a $\mathcal{U}$-representation, $m : \hat{r} \to X$ is a function and p is a proof that m is injective.
- Two such triples (r, m, p) and (s, n, q) are equivalent, if $(\hat{r}, m)$ and $(\hat{s}, n)$ are equal as subsets.
- For index $(r, m, p) \in I$, the corresponsing subset of X is $F_{(r,m,p)} = (\hat{r}, m)$.

Proposition A subset $S = (\partial S, \iota_S)$ of X belongs to $\mathcal{R}_{\mathcal{U}}(X)$ iff ∂S is a $\mathcal{U}$-set.

Unless $\mathcal{U}$ has some closure properties, $\mathcal{R}_{\mathcal{U}}(X)$ will not be closed under usual set-theoretic operations. We review some common such properties below. Suppose that $\mathcal{U}$ is a type-theoretic universe.

- If $\mathcal{U}$ is closed under Σ, then $\mathcal{R}_{\mathcal{U}}(X)$ is closed under binary $\cap$, and $\bigcup_{i\in I}$ indexed by $\mathcal{U}$-sets I.

- If $\mathcal{U}$ is closed under Π, then $\mathcal{R}_{\mathcal{U}}(X)$ is closed under $\bigcap_{i\in I}$ indexed by $\mathcal{U}$-sets I, and the binary set operation

$$(A \Rightarrow B) = \{x \in X : x \in A \Rightarrow x \in B\}.$$

- If $\mathcal{U}$ is closed under $+$, then $\mathcal{R}_{\mathcal{U}}(X)$ is closed under binary $\cup$.

- If $\mathcal{U}$ contains an empty type, then $\mathcal{R}_{\mathcal{U}}(X)$ contains $\emptyset$.

Standard Martin-Löf type universes $\mathcal{U}$ (see Martin-Löf 1984) satisfies indeed the conditions above. Recall from Dybjer's lecture how such universes are defined:

$$
\begin{array}{rcl@{\qquad}rcl}
\hat{\mathrm{N}} & : & U & T\,\hat{\mathrm{N}} & = & \mathrm{N} \\
\widehat{\mathrm{N}_0} & : & U & T\,\widehat{\mathrm{N}_0} & = & \mathrm{N}_0 \\
\widehat{\mathrm{N}_1} & : & U & T\,\widehat{\mathrm{N}_1} & = & \mathrm{N}_1 \\
(\hat{+}) & : & U \to U \to U & T\,(a \hat{+} b) & = & T\,a + T\,b \\
\hat{\Sigma} & : & (a : U) \to (T\,a \to U) \to U & T\,(\hat{\Sigma}\,a\,b) & = & \Sigma\,(T\,a)\,(\lambda x.T\,(b x)) \\
\hat{\Pi} & : & (a : U) \to (T\,a \to U) \to U & T\,(\hat{\Pi}\,a\,b) & = & \Pi\,(T\,a)\,(\lambda x.T\,(b x)) \\
 & \vdots & & & \vdots &
\end{array}
$$

10. Categories

We use a definition of category where no equality relation between objects is assumed, as introduced in type theory by P. Aczel 1993, P. Dybjer and V. Gaspes 1993. Such categories are adequate for developing large parts of elementary category theory inside type theory (Huet and Saibi 2000).

A *small E-category* $\mathbf{C}$ consists of a type Ob of *objects* (no equivalence relation between objects is assumed) and for all $A, B : \mathrm{Ob}$ there is a set $\mathrm{Hom}(A,B)$ of *morphisms from A to B*. There is a identity morphism $\mathrm{id}_A \in \mathrm{Hom}(A,A)$ for each $A : \mathrm{Ob}$. There is a composition function $\circ : \mathrm{Hom}(B,C) \times \mathrm{Hom}(A,B) \to \mathrm{Hom}(A,C)$. These data satisfy the equations $\mathrm{id} \circ f = f$, $g \circ \mathrm{id} = g$ and $f \circ (g \circ h) = (f \circ g) \circ h$.

For a *locally small E-category* we allow Ob to be a sort.

Example. *The category of sets,* $\mathbf{Sets}$, has as objects sets. The set of functions from A to B is denoted $\mathrm{Hom}(A,B)$. The category $\mathbf{Sets}$ is locally small, but not small.

Example. The *discrete category given by a set* $A = (\underline{A}, =_A)$. The objects of the category are the elements of $\underline{A}$. Define $\mathrm{Hom}(a,b)$ as the type (of proofs of) $a =_A b$. Any two elements of this type are considered equal. (The proofs of reflexivity and transitivity provide id and $\circ$ respectively. Also the proof of symmetry, gives that two objects a and b are isomorphic if, and only if, $a =_A b$.) Denote the discrete category by $A^{\#}$. This is a small category.

Families of sets

Families of sets have more structure than in set theory.

A *family F of sets indexed by a set I* is a functor $F : I^{\#} \to \mathbf{Sets}$.

Explication:

For each element a of I, $F(a)$ is a set.

For any proof object $p : a =_I b$, $F(p)$ is function from $F(a)$ to $F(b)$, a so-called *transporter function*.

Moreover, since any two morphisms p and q from a to b in $I^{\#}$ are identified, we have $F(p) = F(q)$. The functoriality conditions thus degenerate to the following:

(a) $F(p) = \mathrm{id}_{F(a)}$ for any $p : a =_I a$.

(b) $F(q) \circ F(p) = F(r)$ for all $p : a =_I b$, $q : b =_I c$, $r : a =_I c$.

Note that each $F(p)$ is indeed an isomorphism, and that $F(q)$ is the inverse of $F(p)$ as soon as $p : a =_I b$ and $q : b =_I a$.

Remark. If each set in the family F is a subset of a fixed set X, i.e. $i_a : F(a) \hookrightarrow X$ and so that $i_a \circ F(r) = i_b$ for $r : a =_I b$, then $(F(a), i_a) = (F(b), i_b)$ as subsets of X, if $a =_I b$.

Remark. Families of sets are treated in essentially this way in (Bishop and Bridges 1985, Exercise 3.2).

Example. Let $\beta : B \to I$ be any function. Define for each $a \in I$ a set

$$\beta^{-1}(a) \equiv \{u \in B : \beta(u) =_I a\},$$

the *fiber of* β *over* a. Then β^{-1} extends to a functor $I^{\#} \to \mathbf{Sets}$.

This example indicates another way of describing a families of sets indexed by I: as the fibers of a function $\beta : B \to I$. These are in turn precisely the objects of the slice category $\mathbf{Sets}/I$. We have the following equivalence of categories

Thm.

$$\mathbf{Sets}^{I^{\#}} \cong \mathbf{Sets}/I.$$

Constructions:

Given $\beta : B \to I$. Construct functor $\beta^{-1} : I^{\#} \to \mathbf{Sets}$. Define for $r : a =_I b$ a function $\beta^{-1}(r) : \beta^{-1}(a) \to \beta^{-1}(b)$ by

$$\beta^{-1}(r)(u,p) = (u, k_{\beta(u),a,b}(r,p)).$$

Here $k_{a,b,c}$ is the proof object for $b =_I c \to a =_I b \to a =_I c$,

For $F \in \mathbf{Sets}^{I^{\#}}$, define $B = (\Sigma i \in I)F(i)$, where $(a,x) =_B (b,y)$ iff $a =_I b$ and $F(r)(x) =_{F(b)} y$ and $r : a =_I b$. Let $\beta_F : B \to I$ be the first projection.

11. Relation to categorical logic

Category theory provides an abstract way of defining the essential mathematical proporties of sets, in terms of universal constructions.

An *elementary topos* is a category with properties similar to the sets, though neither classical logic (discreteness of sets), or axioms of choice are assumed among these properties.

C. McLarty: *Elementary Categories, Elementary Toposes.* Oxford University Press 1992.

J. Lambek and P.J. Scott: *Introduction to Higher-Order Categorical Logic.* Cambridge University Press 1986.

Also predicative versions of toposes have been developed

I. Moerdijk and E. Palmgren: *Type Theories, Toposes and Constructive Set Theory,* Annals of Pure and Applied Logic 114(2002).

The syntactical category of a type theory

Given is any type theory T including the constructions Σ, Π and $+$ and constants $\mathrm{N}_0, \mathrm{N}_1$. (This can be precised using a logical framework.)

Build a category $\mathcal{S}_T$ of closed terms for sets and functions of T. In this category the standard (Heyting-) algebraic method of interpreting logic can be used.

We associate to any first order formula φ with free variables among $x_1 : X_1, \ldots, x_n : X_n$ a subobject $[\![\varphi]\!]_{x_1,\ldots,x_n}$ of $X_1 \times \cdots \times X_n$ in $\mathcal{S}_T$.

Completeness Theorem. For any first-order formulas φ and ψ whose types are in $\mathcal{S}_T$:

$[\![\varphi]\!]_{x_1,\ldots,x_n} \leq [\![\psi]\!]_{x_1,\ldots,x_n}$ in $\mathcal{S}_T$ iff

$$\vdash_T (\forall x_1 : X_1) \cdots (\forall x_n : X_n)\,(\varphi \to \psi).$$

formalization of mathematics

Freek Wiedijk
Radboud University Nijmegen

TYPES Summer School 2005
Göteborg, Sweden
2005 08 23, 11:10

intro

the best of two worlds

formalization of mathematics is like:

- **computer programming**

  concrete, explicit

  a formalization is much like a computer program

- **doing mathematics**

  abstract, non-trivial

  a formalization is much like a mathematical textbook

you will like it only if you like **both** programming and mathematics

but in that case you will like it very very much!

table of contents: the two parts of this talk

first hour: an overview of

the current state of the art in formalization of mathematics

in the reader: QED manifesto

second hour: an overview of

Mizar, the most 'mathematical' proof assistant

in the reader: Mizar tutorial

first hour:
state of the art in formalization of mathematics

mathematics in the computer

four ways to do mathematics in the computer

- **numerical mathematics**, experimentation, visualisation

  numbers: computer $\longrightarrow$ human

- **computer algebra**

  formulas: computer $\longrightarrow$ human

- **automated theorem provers**

  proofs: computer $\longrightarrow$ human

- **proof assistants**

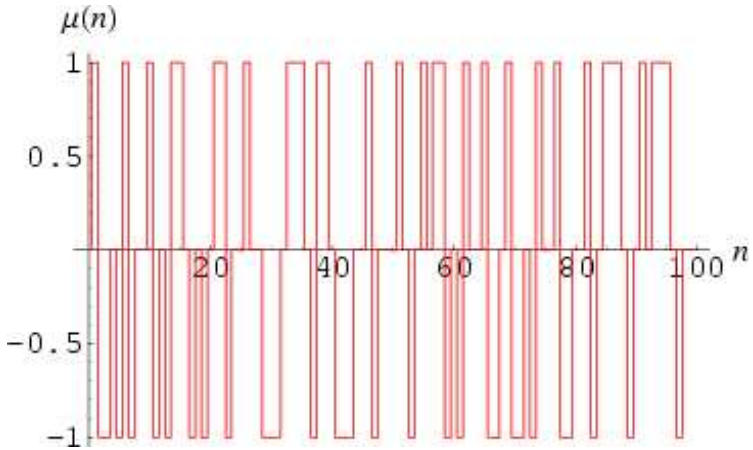
  proofs: human $\longrightarrow$ computer

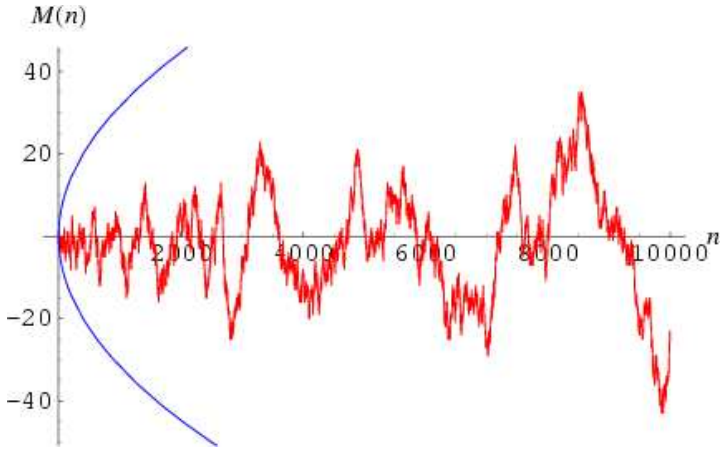
numerical mathematics: Merten's conjecture

Möbius function:

$$\mu(n) = \begin{cases} 0 & \text{when } n \text{ has duplicate prime factors} \\ 1 & \text{when } n \text{ has an even number of different prime factors} \\ -1 & \text{when } n \text{ has an odd number of different prime factors} \end{cases}$$

Mertens, 1897: $$\left|\sum_{k=1}^{n} \mu(n)\right| < \sqrt{n} \quad ?$$





Merten's conjecture (continued)

Odlyzko & te Riele, 1985: Mertens conjecture is false!

50 uur computer time

first n where it fails has tens of digits

indirect proof!

2000 zeroes of the Riemann zeta function to 100 decimals precision

14.13472514173469379045725198356247027078425711569924317568556746014996342980925676494901039317156101 27...
21.02203963877155499262847959389690277733434052490278175462952040358759858606889079971365851418015141 95...
25.01085758014568876321379099256282181865954967255799667249654200674509209844164427784023822455806244 07...
30.42487612585951321031189753058409132018156002371544018096214603699332938933327792029058429390208911 06...
32.93506158773918969066236896407490348881271560351703900928000344078481560863055100593884849613534872 45...
37.58617815882567125721776348070533282140559735083079321833300111362214908961853726473032910494582380 34...
40.91871901214749518739812691463325439572616596277727953616130366725328052872007128299600371988954687 55...
43.32707328091499951949612216540680578264566837183687144687889368552108832230505362645634937106319093 35...
48.00515088116715972794247274942751604168684400114442511777531251981409021641630828133033537230540099 77...
49.77383247767230218191678467856372405772317829967666210078195575043351161151573927873270750740093133 00...
52.97032147771446064414729660888099006382501788882122477990074814031756495030418805413758782709439929 88...
56.44624769706339480436775947670612755278226447171663184545096984395847528027450566690301131427485238 74...
59.34704400260235307965364867499221903109877280646666969812245175474680015269962981183810248707463354 84...
60.83177852460980984425990182452400380291009045121917825710134882480849366729492053843084167039434335 65...
65.11254404808160666087505425318370502934814929516672240596650108667534323266868538441677478443865947 14...
67.07981052949417371447882889652221677010714495174555887419666955169490121895619698353029397508583303 43...
69.54640171117397925292685752655473844301247420960251015732453999966338767227491041953334493317834035 63...
72.06715767448190758252210796982616839048090662145669708668330615148840737239960834836352533041217453 29...

computer algebra: symbolic integration of $\int_0^\infty e^{-\frac{(x-1)^2}{\sqrt{x}}}\,dx$

```
> int(exp(-(x-t)^2)/sqrt(x), x=0..infinity);
```

$$\frac{1}{2}\frac{e^{-t^2}\left(-\frac{3(t^2)^{\frac{1}{4}}\pi^{\frac{1}{2}}2^{\frac{1}{2}}e^{\frac{t^2}{2}}K_{\frac{3}{4}}(\frac{t^2}{2})}{t^2}+(t^2)^{\frac{1}{4}}\pi^{\frac{1}{2}}2^{\frac{1}{2}}e^{\frac{t^2}{2}}K_{\frac{7}{4}}(\frac{t^2}{2})\right)}{\pi^{\frac{1}{2}}}$$

```
> subs(t=1,%);
```

$$\frac{1}{2}\frac{e^{-1}\left(-3\pi^{\frac{1}{2}}2^{\frac{1}{2}}e^{\frac{1}{2}}K_{\frac{3}{4}}(\frac{1}{2})+\pi^{\frac{1}{2}}2^{\frac{1}{2}}e^{\frac{1}{2}}K_{\frac{7}{4}}(\frac{1}{2})\right)}{\pi^{\frac{1}{2}}}$$

```
> evalf(%);
```

$$0.4118623312$$

```
> evalf(int(exp(-(x-1)^2)/sqrt(x), x=0..infinity));
```

$$1.973732150$$

automated theorem proving: Robbins' conjecture

computers

- . . . can in the near future play chess better than a human
- . . . can in the near future do mathematics better than a human?

Robbins, 1933: is every Robbins algebra a Boolean algebra?

EQP, 1996: **yes!**
eight days of computer time

one of the very few proofs that has first been found by a computer
not very conceptual: just searches through very many possibilities

interesting research, but currently not relevant for mathematics

the QED manifesto

let's formalize all of mathematics!

QED manifesto, 1994:

> QED is the very tentative title of a project to build a computer system that effectively represents all important mathematical knowledge and techniques.

pamphlet by anonymous group, led by Bob Boyer
utopian vision

proposed many times
never got very far (yet)

the two kinds of computer proof

- **correctness of computer software and hardware**

  (serious branch of computer science: 'formal methods')

  statements: big
  proofs: shallow
  computer does the main part of the proof

- **correctness of mathematical theorems**

  (slow and thorough style of doing mathematics, still in its infancy)

  statements: small
  proofs: deep
  human does the main part of the proof

a brief overview of proof assistants for mathematics

four prehistorical systems

1968 **Automath**

Netherlands, de Bruijn

1971 **nqthm**

US, Boyer & Moore

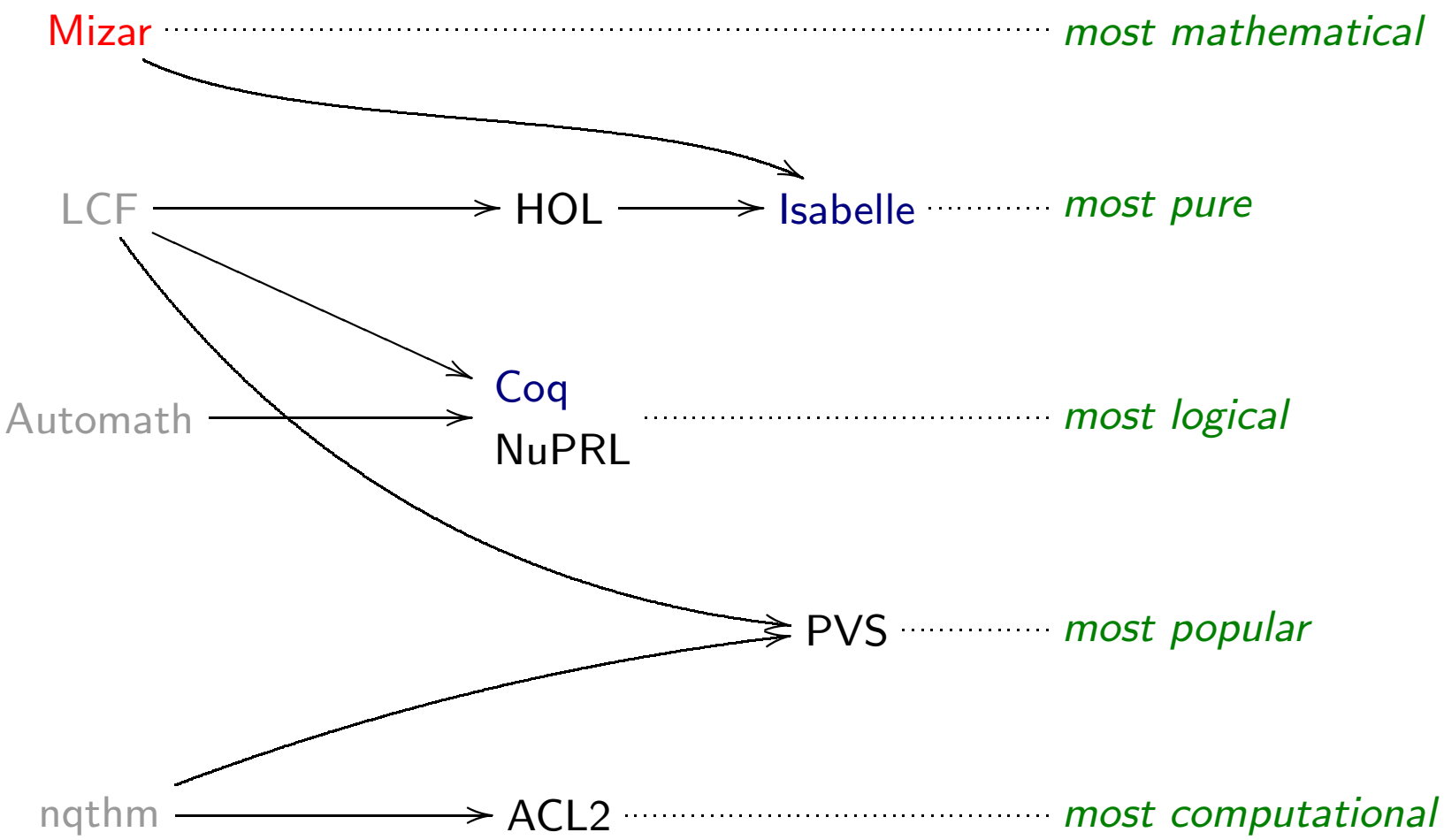
1972 **LCF**

UK, Milner

1973 **Mizar**

Poland, Trybulec

seven current systems for mathematics



a 'top 100' of mathematical theorems

1. The Irrationality of the Square Root of 2 ← *all systems*
2. Fundamental Theorem of Algebra ← Mizar, HOL, Coq
3. The Denumerability of the Rational Numbers ← Mizar, HOL, Isabelle
4. Pythagorean Theorem ← Mizar, HOL, Coq
5. Prime Number Theorem ← Isabelle
6. Gödel's Incompleteness Theorem ← HOL, Coq, nqthm
7. Law of Quadratic Reciprocity ← Isabelle, nqthm
8. The Impossibility of Trisecting the Angle and Doubling the Cube ← HOL
9. *The Area of a Circle*
10. Euler's Generalization of Fermat's Little Theorem ← Mizar, HOL, Isabelle

... ...

63% formalized

`http://www.cs.ru.nl/~freek/100/`

(advertisement) the seventeen provers of the world

LNCS 3600

one theorem

seventeen formalisations + explanations about the systems

HOL, Mizar, PVS, Coq, Otter, Isabelle, Agda, ACL2, PhoX, IMPS,
Metamath, Theorema, Lego, NuPRL, Ωmega, B method, Minlog

`http://www.cs.ru.nl/~freek/comparison/`

state of the art: recent big formalizations

Prime Number Theorem

Bob Solovay's challenge:

> I suspect that fully formalizing the **usual** proof of the prime number theorem [. . .] is beyond the current capacities of the [formalization] community. Say within the next ten years.

Jeremy Avigad e.a.:

```
"pi(x) == real(card(y. y<=x & y:prime))"
"(%x. pi x * ln (real x) / (real x)) ----> 1"
```

1 megabyte = 30,000 lines = 42 files of Isabelle/HOL

the **elementary** proof by Selberg from 1948

Four Color Theorem

Georges Gonthier:

```
(m : map) (simple_map m) -> (map_colorable (4) m)
```

2.5 megabytes = 60,000 lines = 132 files of Coq 7.3.1
streamlined proof by Robertson, Sanders, Seymour & Thomas from 1996

- contains interesting mathematics as well
  'planar hypermaps'
- very interesting 'own' proof language on top of Coq

```
Move=> x' p'; Elim: p' x' => [|y' p' Hrec] x' //=; Rewrite: ~Hrec.
By Congr andb; Congr orb; Rewrite: /eqdf (monic2F_eqd (f_finv (Inode g'))).
```

- heavily relies on **reflection**
  'this formalization really needs Coq'

Jordan Curve Theorem

Tom Hales:

```
‘!C. simple_closed_curve top2 C ==>
   (?A B.  top2 A /\ top2 B /\
     connected top2 A /\ connected top2 B /\
   ~(A = EMPTY) /\ ~(B = EMPTY) /\
    (A INTER B = EMPTY) /\ (A INTER C = EMPTY) /\
        (B INTER C = EMPTY) /\
       (A UNION B UNION C = euclid 2))‘
```

2.1 megabytes = 75,000 lines = 15 files of HOL Light
proof through the Kuratowski characterization of planarity

- ‘warming up exercise’ for the Flyspeck project
- beat the Mizar project at formalizing this first
- also uses an ‘own’ proof style

state of the art: current big projects

the continuous lattices formalization

---

formalize a complete 'advanced' mathematics textbook

A Compendium of Continuous Lattices

by Gierz, Hofmann, Keimel, Lawson, Mislove & Scott

> [...] For if not, then $V \subseteq \bigcup\{L \setminus {\downarrow}v : v \in V\}$; and by quasicompactness and the fact that the $L \setminus {\downarrow}v$ form a directed family, there would be a $v \in V$ with $V \subseteq L \setminus {\downarrow}v$, notably $v \notin V$, which is impossible. [...]

project led by Grzegorz Bancerek

about 70% formalized

4.4 megabytes = 127,000 lines = 58 files of Mizar

the Flyspeck project

Kepler in strena sue de nive sexangula, 1661:

is the way one customarily stacks oranges the most efficient way to stack spheres?

Tom Hales, 1998: **yes !**

proof: depends on computer checking
3 gigabytes programs & data, couple of months of computer time

referees say to be 99% certain that everything is correct

FlysPecK project
'Formal Proof of Kepler'

so why did the qed project not take off?

reason one: differences between systems

---

foundations differ very much

set theory $\longleftrightarrow$ type theory $\longleftrightarrow$ higher order logic $\longleftrightarrow$ PRA

classical $\longleftrightarrow$ constructive

extensional $\longleftrightarrow$ intensional

impredicative $\longleftrightarrow$ predicative

choice $\longleftrightarrow$ only countable choice $\longleftrightarrow$ no choice

two utopias simultaneously?

- **formalization of mathematics**
- **doing mathematics in weak logics**

(advertisement) a questionnaire about intuitionism

`http://www.intuitionism.org/`

ten questions about intuitionism

currently: seventeen sets of answers by various people

3. Do you agree that there are only three infinite cardinalities?
7. Do you agree that for any two statements the first implies the second or the second implies the first?

putting systems together

OMDoc

XML standard for encoding of mathematical documents
developed by Michael Kohlhase

can be used both for natural language documents and for formalizations
modularized language architecture

supports both OpenMath and Content MathML encoding of formulas

does not really address semantical differences between systems

Logosphere

converting between the foundations of various systems

project led by Carsten Schürmann

formalize foundations of each system in the Twelf logical framework

translate all formalizations into Twelf

use Twelf to relate those formalizations

systems that are currently supported:

- **first order resolution provers**
- **HOL**
- **NuPRL**
- **PVS**

reason two: why mathematicians are not interested (yet)

the cost is too high. . .

$$\text{de Bruijn factor} = \frac{\text{size of formalization}}{\text{size of normal text}}$$

question: is this a constant?

experimental: around **4**

$$\text{de Bruijn factor in time} = \frac{\text{time to formalize}}{\text{time to understand}}$$

much larger than **4**

formalizing one textbook page $\approx$ 1 man·week = 40 man·hours

. . . and the gain is too little

l'art pour l'art

Paul Libbrecht in Saarbrücken: 'mental masturbation'

it's not **impossibly** expensive
formalizing all of undergraduate mathematics $\approx$ 140 man·years
the price of about **one** Hollywood movie

but: after formalization we just have a big incomprehensible file
we don't have a good argument yet for spending that money

certainty that it's fully correct?

is that important enough to pay for 140 man·years?

and it does not look like mathematics

most systems: 'proof' = list of tactics = unreadable computer code

even in Mizar and Isar: **still** looks like code

even formulas: too much 'decoding' needed to understand what it says

```
Variable J : interval.      Hypothesis pJ : proper J.
Variable F, G : PartIR.     Hypothesis derG : Derivative J pJ G F.
Let G_inc := Derivative_imp_inc _ _ _ _ derG.

Theorem Barrow : forall a b (H : Continuous_I (Min_leEq_Max a b) F) Ha Hb,
 let Ha' := G_inc a Ha in let Hb' := G_inc b Hb in
 Integral H [=] G b Hb'[-]G a Ha'.
```

$$G' = F \;\Rightarrow\; \int_a^b F(x)\,dx = G(b) - G(a)$$

so what is needed most to promote formalization of mathematics?

- **decision procedures**
  very important, main strength of PVS

- in particular: **computer algebra**
  Macsyma, Maple, Mathematica
  (really: **computer calculus**)

  high school mathematics should be trivial!

$$x = i/n\ ,\quad n = m+1 \quad\vdash\quad n! \cdot x = i \cdot m!$$

$$\frac{k}{n} \geq 0 \quad\vdash\quad \left|\frac{n-k}{n} - 1\right| = \frac{k}{n}$$

$$n \geq 2\ ,\quad x = \frac{1}{n+1} \quad\vdash\quad \frac{x}{1-x} < 1$$

second hour:
a tour of Mizar, a proof assistant for mathematics

why is Mizar interesting?

- **a system for mathematicians**
- **the proof language**

  only other system with similar language: Isabelle/Isar
- **many other interesting ideas**
  - type system

    soft typing

    'attributes'

    multiple inheritance between structure types
  - expression syntax

    type directed overloading

    bracket-like operators

    arbitrary ASCII strings for operators

example formalizations

example: Coq version

```
Definition ge (n m : nat) : Prop :=
  exists x : nat, n = m + x.
Infix ">=" := ge : nat_scope.

Lemma ge_trans :
  forall n m p : nat, n >= m -> m >= p -> n >= p.
Proof.
 unfold ge. intros n m p H H0.
 elim H. clear H. intros x H1.
 elim H0. clear H0. intros x0 H2.
 exists (x0 + x).
 rewrite plus_assoc. rewrite <- H2. auto.
Qed.
```

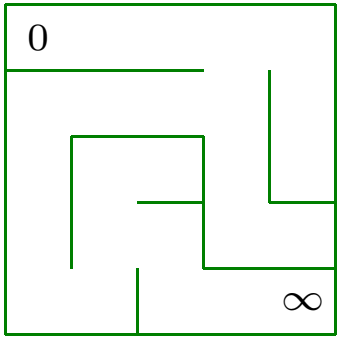
example: Mizar version

```
reserve n,m,p,x,x0 for natural number;

definition let n,m;
 pred n >= m means :ge: ex x st n = m + x;
end;

theorem ge_trans: n >= m & m >= p implies n >= p
proof
 assume that H: n >= m and H0: m >= p;
 consider x such that H1: n = m + x by H,ge;
 consider x0 such that H2: m = p + x0 by H0,ge;
 n = p + (x + x0) by H1,H2;
 hence n >= p by ge;
end;
```

procedural versus declarative



- **procedural**

  E E S E N E S S S W W W S E E E

  HOL, Isabelle, Coq, NuPRL, PVS

- **declarative**

  (0,0) (1,0) (2,0) (3,0) (3,1) (2,1) (1,1) (0,1) (0,2) (0,3) (0,4) (1,4) (1,3) (2,3) (2,4) (3,4) (4,4)

  Mizar, Isabelle

another small example

If every poor person has a rich father,
then there is a rich person with a rich grandfather.

```
 assume that
A1: for x st x is poor holds father(x) is rich and
A2: not ex x st x is rich & father(father(x)) is rich;
 consider p being person;
 now let x;
  x is poor or father(father(x)) is poor by A2;
  hence father(x) is rich by A1;
 end;
 then father(p) is rich & father(father(father(p))) is rich;
 hence contradiction by A2;
```

demo example

Theorem. *There are irrational numbers x and y such that x^y is rational.*

Proof. We have the following calculation

$$\left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\cdot\sqrt{2}} = \sqrt{2}^{2} = 2$$

which is rational. Furthermore Pythagoras showed that $\sqrt{2}$ is irrational. Now there are two cases:

- Either $\sqrt{2}^{\sqrt{2}}$ is rational. Then take $x = y = \sqrt{2}$.
- Or $\sqrt{2}^{\sqrt{2}}$ is irrational. In that case take $x = \sqrt{2}^{\sqrt{2}}$ and $y = \sqrt{2}$. And by the above calculation then $x^y = 2$, which is rational. □

lemmas used in the proof

`AXIOMS:22`	$x \leq y \wedge y \leq z \Rightarrow x \leq z$
`INT_2:44`	2 is prime
`IRRAT_1:1`	p is prime $\Rightarrow \sqrt{p} \notin \mathbb{Q}$
`POWER:38`	$a > 0 \Rightarrow (a^b)^c = a^{bc}$
`SQUARE_1:def 3`	$x^2 = x \cdot x$
`SQUARE_1:def 4`	$0 \leq a \Rightarrow (x = \sqrt{a} \Leftrightarrow 0 \leq x \wedge x^2 = a)$
`SQUARE_1:84`	$1 < \sqrt{2}$
`POWER:53`	`‘a to_power 2 = a^2’`

DEMO

```
reserve x,y for real number;

theorem ex x,y st x is irrational & y is irrational &
  x to_power y is rational
proof
 set r = sqrt 2;
C: r > 0 by SQUARE_1:84,AXIOMS:22;
B1: r is irrational by INT_2:44,IRRAT_1:1;
B2: (r to_power r) to_power r
   = r to_power (r * r) by C,POWER:38
  .= r to_power r^2 by SQUARE_1:def 3
  .= r to_power 2 by SQUARE_1:def 4
  .= r^2 by POWER:53
  .= 2 by SQUARE_1:def 4;
 per cases;
 suppose
A1: r to_power r is rational;
  take x = r, y = r;
  thus thesis by A1,B1;
 end;
 suppose
A2: r to_power r is irrational;
  take x = r to_power r, y = r;
  thus thesis by A2,B1,B2;
 end;
end;
```

example of how Mizar is like English

Hardy & Wright, An Introduction to the Theory of Numbers

Theorem 43 (Pythagoras' theorem). $\sqrt{2}$ is irrational.

The traditional proof ascribed to Pythagoras runs as follows. If $\sqrt{2}$ is rational, then the equation

$$a^2 = 2b^2 \tag{4.3.1}$$

is soluble in integers a, b with $(a, b) = 1$. Hence a^2 is even, and therefore a is even. If $a = 2c$, then $4c^2 = 2b^2$, $2c^2 = b^2$, and b is also even, contrary to the hypothesis that $(a, b) = 1$. □

Mizar language approximation of this text

theorem Th43: sqrt 2 is irrational
proof
  assume sqrt 2 is rational**;**
  consider a, b **such that**
4_3_1: a^2 = 2 * b^2 **and**
    a,b are_relative_prime**;**
  a^2 is even**;**
  a is even**;**
  consider c **such that** a = 2 * c**;**
  4 * c^2 = 2 * b^2**;**
  2 * c^2 = b^2**;**
  b is even**;**
  thus contradiction**;**
end;

full Mizar

```
theorem Th43: sqrt 2 is irrational
proof
  assume sqrt 2 is rational;
  then consider a, b such that
A1: b <> 0 and
A2: sqrt 2 = a/b and
A3: a, b are_relative_prime by Def1;
A4: b^2 <> 0 by A1, SQUARE_1:73;
  2 = (a/b)^2 by A2, SQUARE_1:def 4
    .= a^2/b^2 by SQUARE_1:69;
  then
4_3_1: a^2 = 2 * b^2 by A4, REAL_1:43;
  a^2 is even by 4_3_1, ABIAN:def 1;
  then
A5: a is even by PYTHTRIP:2;
:: continue in next column
  then consider c such that
A6: a = 2 * c by ABIAN:def 1;
A7: 4 * c^2 =(2 * 2) * c^2
    .= 2^2 * c^2 by SQUARE_1:def 3
    .= 2 * b^2 by A6, 4_3_1, SQUARE_1:68;
  2 * (2 * c^2) = (2 * 2) * c^2 by AXIOMS:16
    .= 2 * b^2 by A7;
  then 2 * c^2 = b^2 by REAL_1:9;
  then b^2 is even by ABIAN:def 1;
  then b is even by PYTHTRIP:2;
  then 2 divides a & 2 divides b by A5, Def2;
  then
A8: 2 divides a gcd b by INT_2:33;
  a gcd b = 1 by A3, INT_2:def 4;
  hence contradiction by A8, INT_2:17;
end;
```

some explanations about Mizar

the proof language

forward reasoning

⟨statement⟩ **by** ⟨references⟩

⟨statement⟩ **proof** ⟨steps⟩ **end**

natural deduction

thus ⟨statement⟩	$\longrightarrow$	closes the proof
assume ⟨statement⟩	$\longrightarrow$	$\longrightarrow$-introduction
let ⟨variable⟩	$\longrightarrow$	$\forall$-introduction
thus ⟨statement⟩	$\longrightarrow$	$\wedge$-introduction
consider ⟨variable⟩ **such that** ⟨statement⟩	$\longrightarrow$	$\exists$-elimination
take ⟨term⟩	$\longrightarrow$	$\exists$-introduction
per cases; suppose ⟨statement⟩**;** ...	$\longrightarrow$	$\vee$-elimination

‘semantics’?

Mizar is just first order predicate logic + set theory

Mizar proofs are just Fitch-style natural deduction

but:

- Mizar variables have types...
  ... and these types are quite powerful!
- Mizar has ‘second-order theorems’ called schemes
- Mizar defines function symbols using something like Church’s ι operator (‘unique choice’)

Tarski-Grothendieck set theory

TARSKI:def 3	$X \subseteq Y \Leftrightarrow (\forall x.\, x \in X \Rightarrow x \in Y)$
TARSKI:def 5	$\langle x, y \rangle = \{\{x, y\}, \{x\}\}$
TARSKI:def 6	$X \sim Y \Leftrightarrow \exists Z.\, (\forall x.\, x \in X. \Rightarrow \exists y.\, y \in Y \wedge \langle x, y \rangle \in Z) \wedge$
	$(\forall y.\, y \in Y. \Rightarrow \exists x.\, x \in X \wedge \langle x, y \rangle \in Z) \wedge$
	$(\forall x \forall y \forall z \forall u.\, \langle x, y \rangle \in Z \wedge \langle z, u \rangle \in Z \Rightarrow (x = z \Leftrightarrow y = u))$

TARSKI:def 1	$x \in \{y\} \Leftrightarrow x = y$
TARSKI:def 2	$x \in \{y, z\} \Leftrightarrow x = y \vee x = z$
TARSKI:def 4	$x \in \bigcup X \Leftrightarrow \exists Y.\, x \in Y \wedge Y \in X$
TARSKI:2	$(\forall x.\, x \in X \Leftrightarrow x \in Y) \Rightarrow X = Y$
TARSKI:7	$x \in X \Rightarrow \exists Y.\, Y \in X \wedge \neg \exists x.\, x \in X \wedge x \in Y$
TARSKI:sch 1	$(\forall x\, \forall y\, \forall z.\, P[x, y] \wedge P[x, z] \Rightarrow y = z) \Rightarrow$
	$(\exists X.\, \forall x.\, x \in X \Leftrightarrow \exists y.\, y \in A \wedge P[y, x])$
TARSKI:9	$\exists M.\, N \in M \wedge (\forall X \forall Y.\, X \in M \wedge Y \subseteq X \Rightarrow Y \in M) \wedge$
	$(\forall X.\, X \in M \Rightarrow \exists Z.\, Z \in M \wedge \forall Y.\, Y \subseteq X \Rightarrow Y \in Z) \wedge$
	$(\forall X.\, X \subseteq M \Rightarrow X \sim M \vee X \in M)$

types!

Mizar is based on set theory but it is a typed system

Mizar types are soft types:

$$M : N(t_1, \ldots, t_n)$$

should really be read as a **predicate**

$$N(t_1, \ldots, t_n, M)$$

This means that:

- one Mizar term can have many different types at the same time
- a Mizar typing can be used as a logical formula!

```
let x be Real;     ⟷     assume not x is Nat;
```

types! (continued)

think of Mizar types as predicates that the system keeps track of for you

Mizar types are used for three things:

- **type based overloading**

`x + y`	sum of two numbers
`X + Y`	adding the elements of two sets
`X + y`	mixing these two things
`v + w`	sum of two elements of a vector space
`I + J`	sum of two ideals in a ring
`x + y`	'join' of two elements of a lattice
`p + i`	adding an offset to a pointer

- **inferring implicit arguments**
- **automatic inference of propositions**

types! (continued)

- Mizar has dependent types
  (much like in all the other dependent type systems)
- Mizar has a subtype relation
  every type except the type 'set' has a supertype
- Mizar has 'type modifiers' called attributes
  a type can be prefixed with one or more adjectives
  an adjective is either an attribute or the negation of an attribute
  (behaves like intersection types)

notation

any ASCII string can be used for a Mizar operator

```
  func ].a,b.] -> Subset of REAL means
:: MEASURE5:def 3
   for x being R_eal holds
    x in it iff (a <' x & x <=' b & x in REAL);

  pred a,b are_convergent<=1_wrt R means
:: REWRITE1:def 9
   ex c being set st ([a,c] in R or a = c) & ([b,c] in R or b = c);
```

Mizar in the world

Mizar Mathematical Library

the biggest library of formalized mathematics

49,588	lemmas
1,820,879	lines of 'code'
64	megabytes
165	'authors'
912	'articles'

Mizar, the program

- implemented in Delphi Pascal/Free Pascal
- source not freely available, but

write Mizar 'article'

↓

become member of Association of Mizar Users

↓

get source

- no small proof checking 'kernel'
  correctness of Mizar check depends on correctness of whole program
- users can not automate proofs inside the system

Mizar Mathematical Library

```
theorem :: RUSUB_2:35
for V being RealUnitarySpace, W being Subspace of V,
 L being Linear_Compl of W holds
  V is_the_direct_sum_of L,W & V is_the_direct_sum_of W,L;
```

Formalized Mathematics

(35) Let V be a real unitary space, W be a subspace of V, and L be a linear complement of W. Then V is the direct sum of L and W and the direct sum of W and L.

Mizar versus Isar

some reasons to prefer Mizar over Isar

- the set theory of Mizar is much more powerful and expressive than the HOL logic of Isabelle/HOL
- Mizar is much more able to talk about abstract mathematics, and in particular about algebraic structures, with nice notation
- dependent types are way cool

some reasons to prefer Isar over Mizar

- Isabelle gives you an interactive system
- Isabelle allows you to mix declarative and procedural proof
- Isabelle has much more possibilities of automation
- Isabelle allows you to define binders

is Mizar a difficult system?

no, not difficult at all!

Mizar is about as complex as the Pascal programming language
(proof assistants tend to resemble their implementation language)

reasons that people sometimes think Mizar is a complex language

- lack of proper documentation
- natural language-like syntax

gazing into the crystal ball

Henk's futuristic QED questions

- will proof assistants ever become common among mathematicians?
- if so: when will this happen?
  - the most optimistic answer: it already is here!
  - the experienced user's answer: fifty years

but what do **you** expect?