Constructive Type Theory and dependent types in Isabelle

Michael Färber

Institute of Computer Science
University of Innsbruck
Austria

Seminar 3 13 June, 2012

Quiz: Spot the error

```c
int main()
{
    int array[10];
    for (int i = 0; i <= 10; i++)
        array[i] = 0;

    return 0;
}
```

Overview

- Introduction
- CTT
- Dependent types in CTT
- List type in CTT

Motivation/Objective

Motivation

- Isabelle conceived as general logic framework
- however, mostly used for HOL/set theory

Objective

- study CTT and dependent types
- implement small theory with dependent types in CTT
- find limitations of dependent types in CTT

Isabelle

Isabelle

- generic theorem prover
- implemented in Standard ML
- hand-checked kernel
- meta-logic 'Pure'
- logics implemented on top of Pure

Meta-logic operators

- Implication: $\Longrightarrow$
- Equality: $\equiv$
- Universal quantifier: $\bigwedge$
- $[\![p_1; p_2; p_3]\!] \equiv p_1 \Longrightarrow \ldots \Longrightarrow p_n$

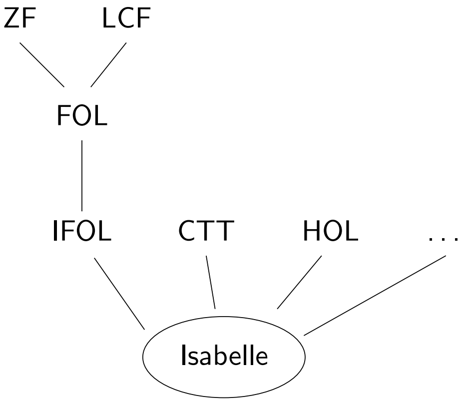


Figure: Overview of existing object logics in Isabelle.

Classical first-order logic

Basis for ...

- ZF (Zermelo-Fraenkel Set Theory)
- LCF (Logic for Computable Functions)
  - logic for Edinburgh LCF, theorem prover from 1972
  - motivated development of ML
  - successor HOL is predecessor of Isabelle

Rules

given via natural deduction

Summary of Natural Deduction (1)

	introduction	elimination
$\wedge$	$\dfrac{\phi \quad \psi}{\phi \wedge \psi}\ \wedge\mathrm{i}$	$\dfrac{\phi \wedge \psi}{\phi}\ \wedge\mathrm{e}_1 \qquad \dfrac{\phi \wedge \psi}{\psi}\ \wedge\mathrm{e}_2$
$\vee$	$\dfrac{\phi}{\phi \vee \psi}\ \vee\mathrm{i}_1 \qquad \dfrac{\psi}{\phi \vee \psi}\ \vee\mathrm{i}_2$	$\dfrac{\phi \vee \psi \quad \boxed{\begin{array}{c}\phi\\ \vdots\\ \chi\end{array}} \quad \boxed{\begin{array}{c}\psi\\ \vdots\\ \chi\end{array}}}{\chi}\ \vee\mathrm{e}$
$\rightarrow$	$\dfrac{\boxed{\begin{array}{c}\phi\\ \vdots\\ \psi\end{array}}}{\phi \rightarrow \psi}\ \rightarrow\mathrm{i}$	$\dfrac{\phi \quad \phi \rightarrow \psi}{\psi}\ \rightarrow\mathrm{e}$

Isabelle code for natural deduction

```
axiomatization
  False :: o and
  conj :: "[o, o] => o"  (infixr "&" 35) and
  disj :: "[o, o] => o"  (infixr "|" 30) and
  imp :: "[o, o] => o"  (infixr "-->" 25)
where
  conjI: "[| P;  Q |] ==> P&Q" and
  conjunct1: "P&Q ==> P" and
  conjunct2: "P&Q ==> Q" and

  disjI1: "P ==> P|Q" and
  disjI2: "Q ==> P|Q" and
  disjE: "[| P|Q;  P ==> R;  Q ==> R |] ==> R" and

  impI: "(P ==> Q) ==> P-->Q" and
  mp: "[| P-->Q;  P |] ==> Q" and
```

The history of CTT

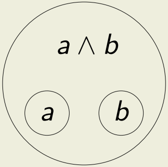
- first version by Per Martin-Löf in 1971
  - impredicative
  - Girard's paradox $\rightarrow$ inconsistent!
- later versions predicative

Principles part 1

Propositions as sets

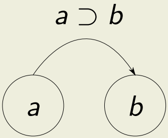
A proposition is identified with the set of its proofs.

Conjunction



A proof of $P \wedge Q$ is a pair $\langle a, b\rangle$ where a is a proof of P and b is a proof of Q.

Implication



A proof of $P \supset Q$ is a function which converts a proof of P into a proof of Q.

Principles part 2

Two kinds of meta-logic types

- type
- instance

Natural numbers

- N type
- $0 \in N$
- $n \in N \Longrightarrow succ(n) \in N$

Principles part 3

Equality

- Type equality: $A = B$
- Instance equality: $a = b \in A$

Functions

- Identity function: $\lambda\!\lambda x.\ x \in A \longrightarrow A$
- Application with β: $(\lambda\!\lambda x.\ x)$ ‘ $a = a \in A$
- Function product: $\Pi\ x \in A.\ B(x)$

Dependent types

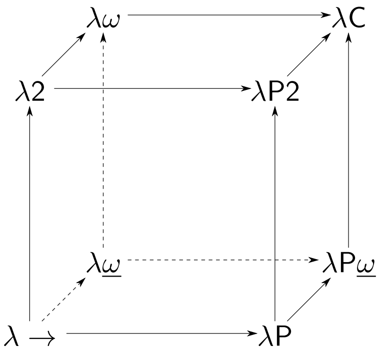


Figure: The λ-cube.

Examples

- $\lambda \to$: terms depending on terms, e.g. $f\ a$
- $\lambda 2$: terms depending on types, e.g. $I_A = \lambda x : A.x$
- $\lambda \underline{\omega}$: types depending on types, e.g. $A \to A$ for a given A
- λP: types depending on terms, e.g. $A^n \to B$ with

$$A^0 \to B = B;$$
$$A^{n+1} \to B = A \to (A^n \to B).$$

Our list type

Idea

- List type depends on a type (the type of elements in the list) and a term (list length)
- such a list type prevents out-of-bounds errors at compile-time due to type checking

Examples

- nil(A) $\in$ List(A,0)
- cons(A) ‘ 0 ‘ a ‘ nil(A) $\in$ List(A,1)

Introduction

axiomatization

```
  List :: "[t,i]⇒t"
```

where

```
  list_type: "⟦A type; n ∈ N⟧ ⟹ List(A,n) type"
```

axiomatization

```
  nil  :: "t ⇒ i" and
  cons :: "t ⇒ i"
```

where

```
  nil_type: "A type ⟹ nil(A) ∈ List(A,0)" and
 cons_type: "A type ⟹ cons(A) ∈ (∏ n ∈ N. A ⟶
List(A,n) ⟶ List(A,succ(n)))"
```

Head and Tail

```
axiomatization
  hd :: "t⇒i" and
  tl :: "t⇒i"
where
  hd_type: "A type ⟹ hd(A) ∈ (Π n ∈ N. List(A,succ(n)) ⟶ A)" and
  hd_appl: "A type ⟹ hd(A) ‘ succ(n) ‘ (cons(A) ‘ n ‘ h ‘ t) = h ∈ A" and

  tl_type: "A type ⟹ tl(A) ∈ (Π n ∈ N. List(A,succ(n)) ⟶ List(A,n))" and

  tl_appl: "A type ⟹ tl(A) ‘ succ(n) ‘ (cons(A) ‘ n ‘ h ‘ t) = t ∈ List(A,n)"
```

List recursion

axiomatization

```
  listrec :: "t⇒(i⇒t)⇒i"
```

where

```
  listrec_type: "⟦A type; (Π x ∈ N. B(x)) type⟧ ⟹
                 listrec(A,B) ∈ Π n ∈ N. List(A,n) ⟶
                 (Π bn ∈ B(0). Π bc ∈
                  (Π nt ∈ N. A ⟶ List(A,nt) ⟶ B(nt) ⟶ B(succ(nt))). B(n))" and

  listrec_nappl: "⟦A type; (Π x ∈ N. B(x)) type⟧ ⟹
                  listrec(A,B) ‘ 0 ‘ nil(A) ‘ bn ‘ bc = bn ∈ B(0)" and
  listrec_cappl: "⟦A type; (Π x ∈ N. B(x)) type⟧ ⟹
                  listrec(A,B) ‘ succ(n) ‘ (cons(A) ‘ n ‘ h ‘ t) ‘ bn ‘ bc =
                  bc ‘ n ‘ h ‘ t ‘ (listrec(A,B) ‘ n ‘ t ‘ bn ‘ bc) ∈ B(succ(nt))" and

  listrec_appl: "⟦A type; (Π x ∈ N. B(x)) type; n ∈ N; l ∈ List(A,n);
                 bn ∈ C(0,nil(A));
                 ⋀nc hc tc rc. ⟦nc ∈ N; hc ∈ A; tc ∈ List(A,nc); rc ∈ C(nc,tc)⟧ ⟹
                  bc ‘ nc ‘ hc ‘ tc ‘ rc ∈ C(succ(nc),cons(A)‘nc‘hc‘tc)⟧ ⟹

                 listrec(A,B) ‘ n ‘ l ‘ bn ‘ bc ∈ C(n,l)"
```

Map

```
definition
  map :: "t⇒t⇒i" where
  "map(A,B) == λλn l f. listrec(A,List(B)) ‘ n ‘ l ‘
nil(B) ‘
    (λλn h t r. cons(B) ‘ n ‘ (f ‘ h) ‘ r)"
```

Properties proved

- appending an element to a nil list gives a list of length 1
- appending an element to any list gives a list of length by one greater than original list
- map function returns list of same length as input list
- recursively defined length function returns length of list

Conclusion

Results

- list type works
- properties cumbersome to prove (especially application elimination!) $\Longrightarrow$ provide tactics?
- "faked" dependency of types on types with Isabelle's meta-logic

Outlook

implement list type in suitable logic (minimally $\lambda P\underline{\omega}$)