Interactive Formal Verification
1: Introduction

Tjark Weber
(Slides: Lawrence C Paulson)
Computer Laboratory
University of Cambridge

Motivation

Motivation

- Complex systems almost inevitably contain bugs.

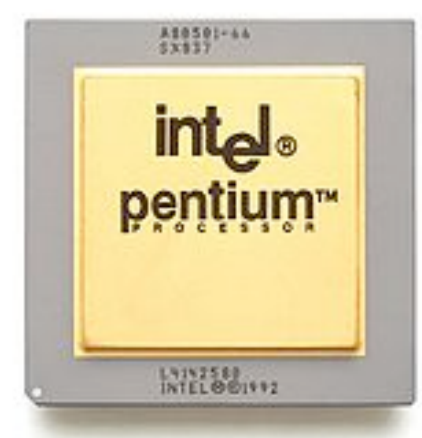
Motivation

- Complex systems almost inevitably contain bugs.

Motivation

- Complex systems almost inevitably contain bugs.
- Program testing can be used to show the presence of bugs, but never to show their absence!

  Edsger W. Dijkstra


intel
pentium™
PROCESSOR

What is Interactive Proof?

What is Interactive Proof?

- Work in a logical formalism
  - precise definitions of concepts
  - formal reasoning system

What is Interactive Proof?

- Work in a logical formalism
  - precise definitions of concepts
  - formal reasoning system
- Construct hierarchies of definitions and proofs
  - libraries of formal mathematics
  - specifications of components and properties

Interactive Theorem Provers

Interactive Theorem Provers

- Based on higher-order logic
  - Isabelle, HOL (many versions), PVS

Interactive Theorem Provers

- Based on higher-order logic
  - Isabelle, HOL (many versions), PVS
- Based on constructive type theory
  - Coq, Twelf, Agda, ...

Interactive Theorem Provers

- Based on higher-order logic
  - Isabelle, HOL (many versions), PVS
- Based on constructive type theory
  - Coq, Twelf, Agda, ...
- Based on first-order logic with recursion
  - ACL2

The LCF Architecture

The LCF Architecture

- A small kernel implements the logic and can generate theorems.

The LCF Architecture

- A small kernel implements the logic and can generate theorems.
- All specification methods and automatic proof procedures expand to full proofs.

The LCF Architecture

- A small kernel implements the logic and can generate theorems.
- All specification methods and automatic proof procedures expand to full proofs.
- Unsoundness is less likely with this architecture

The LCF Architecture

- A small kernel implements the logic and can generate theorems.
- All specification methods and automatic proof procedures expand to full proofs.
- Unsoundness is less likely with this architecture
- ... but the implementation is more complicated, and performance can suffer.

The LCF Architecture

- A small kernel implements the logic and can generate theorems.
- All specification methods and automatic proof procedures expand to full proofs.
- Unsoundness is less likely with this architecture
- ... but the implementation is more complicated, and performance can suffer.
- Used in Isabelle, HOL, Coq but not PVS or ACL2.

Theorem Provers: Characteristic Features

Theorem Provers: Characteristic Features

- Logic (higher-order, type theory etc.)

Theorem Provers: Characteristic Features

- Logic (higher-order, type theory etc.)
- Correctness (LCF vs. non-LCF)

Theorem Provers: Characteristic Features

- Logic (higher-order, type theory etc.)
- Correctness (LCF vs. non-LCF)
- User interface

Theorem Provers: Characteristic Features

- Logic (higher-order, type theory etc.)
- Correctness (LCF vs. non-LCF)
- User interface
- Proof language

Theorem Provers: Characteristic Features

- Logic (higher-order, type theory etc.)
- Correctness (LCF vs. non-LCF)
- User interface
- Proof language
- Automation

Theorem Provers: Characteristic Features

- Logic (higher-order, type theory etc.)
- Correctness (LCF vs. non-LCF)
- User interface
- Proof language
- Automation
- Existing libraries

Theorem Provers: Characteristic Features

- Logic (higher-order, type theory etc.)
- Correctness (LCF vs. non-LCF)
- User interface
- Proof language
- Automation
- Existing libraries
- Tools

Isabelle

Isabelle

- Isabelle is a generic interactive theorem prover, developed by Lawrence Paulson (Cambridge) and Tobias Nipkow (Munich). First release in 1986.

Isabelle

- Isabelle is a generic interactive theorem prover, developed by Lawrence Paulson (Cambridge) and Tobias Nipkow (Munich). First release in 1986.
- Integrated tool support for
    - Automated provers
    - Counterexamples
    - Code generation
    - LaTeX document generation

Higher-Order Logic

"HOL = functional programming + logic"

Higher-Order Logic

- First-order logic extended with functions and sets

"HOL = functional programming + logic"

Higher-Order Logic

- First-order logic extended with functions and sets
- Polymorphic types, including a type of truth values

"HOL = functional programming + logic"

Higher-Order Logic

- First-order logic extended with functions and sets
- Polymorphic types, including a type of truth values
- No distinction between terms and formulas

"HOL = functional programming + logic"

Higher-Order Logic

- First-order logic extended with functions and sets
- Polymorphic types, including a type of truth values
- No distinction between terms and formulas
- ML-style functional programming

"HOL = functional programming + logic"

Basic Syntax of Formulas

formulas *A*, *B*, ... can be written as

`(`A`)`	`t = u`	`~`A
A `&` B	A `|` B	A `-->` B
A `<->` B	`ALL` $x.A$	`EX` $x.A$

(Among many others)

Isabelle also supports symbols such as

$\leq \ \geq \ \neq \ \wedge \ \vee \ \rightarrow \ \leftrightarrow \ \forall \ \exists$

Some Syntactic Conventions

Some Syntactic Conventions

In $\forall x.\ A \wedge B$, the quantifier spans the entire formula

Some Syntactic Conventions

In $\forall x.\ A \land B$, the quantifier spans the entire formula

Parentheses are **required** in $A \land (\forall x\ y.\ B)$

Some Syntactic Conventions

In $\forall x.\ A \wedge B$, the quantifier spans the entire formula

Parentheses are **required** in $A \wedge (\forall x\ y.\ B)$

Binary logical connectives associate to the right: $A \rightarrow B \rightarrow C$ is the same as $A \rightarrow (B \rightarrow C)$

Some Syntactic Conventions

In $\forall x.\ A \wedge B$, the quantifier spans the entire formula

Parentheses are **required** in $A \wedge (\forall x\ y.\ B)$

Binary logical connectives associate to the right: $A \rightarrow B \rightarrow C$ is the same as $A \rightarrow (B \rightarrow C)$

$\neg A \wedge B = C \vee D$ is the same as $((\neg A) \wedge (B = C)) \vee D$

Basic Syntax of Terms

Basic Syntax of Terms

- The typed λ-calculus:
  - constants, c
  - variables, x and *flexible* variables, ?x
  - abstractions λx. t
  - function applications t u

Basic Syntax of Terms

- The typed λ-calculus:
  - constants, c
  - variables, x and *flexible* variables, ?x
  - abstractions λx. t
  - function applications t u
- Numerous infix operators and binding operators for arithmetic, set theory, etc.

Types

Types

- Every term has a type; Isabelle infers the types of terms automatically. We write $t :: \tau$

Types

- Every term has a type; Isabelle infers the types of terms automatically. We write $t :: \tau$
- Types can be *polymorphic*, with a system of type classes (inspired by the Haskell language) that allows sophisticated overloading.

Types

- Every term has a type; Isabelle infers the types of terms automatically. We write $t :: \tau$
- Types can be *polymorphic*, with a system of type classes (inspired by the Haskell language) that allows sophisticated overloading.
- A formula is simply a term of type `bool`.

Types

- Every term has a type; Isabelle infers the types of terms automatically. We write $t :: \tau$
- Types can be *polymorphic*, with a system of type classes (inspired by the Haskell language) that allows sophisticated overloading.
- A formula is simply a term of type `bool`.
- There are types of ordered pairs and functions.

Types

- Every term has a type; Isabelle infers the types of terms automatically. We write $t :: \tau$
- Types can be *polymorphic*, with a system of type classes (inspired by the Haskell language) that allows sophisticated overloading.
- A formula is simply a term of type `bool`.
- There are types of ordered pairs and functions.
- Other important types are those of the natural numbers (`nat`) and integers (`int`).

Product Types for Pairs

Product Types for Pairs

- (x_1, x_2) has type $\tau_1 * \tau_2$ provided $x_i :: \tau_i$

Product Types for Pairs

- (x_1, x_2) has type $\tau_1 * \tau_2$ provided $x_i :: \tau_i$
- $(x_1, \ldots, x_{n\text{-}1}, x_n)$ abbreviates $(x_1, \ldots, (x_{n\text{-}1}, x_n))$

Product Types for Pairs

- (x_1, x_2) has type $\tau_1 * \tau_2$ provided $x_i :: \tau_i$
- $(x_1, \ldots, x_{n\text{-}1}, x_n)$ abbreviates $(x_1, \ldots, (x_{n\text{-}1}, x_n))$
- Extensible record types can also be defined.

Function Types

Function Types

- Infix operators are curried functions
  - `+ :: nat => nat => nat`
  - `& :: bool => bool => bool`
  - Curried function notation: $\lambda x\ y.\ t$

Function Types

- Infix operators are curried functions
  - `+ :: nat => nat => nat`
  - `& :: bool => bool => bool`
  - Curried function notation: $\lambda x\ y.\ t$
- Function arguments can be paired
  - Example: `nat*nat => nat`
  - Paired function notation: $\lambda(x,y).\ t$

Arithmetic Types

Arithmetic Types

- `nat`: the natural numbers (nonnegative integers)
  - inductively defined: `0`, `Suc` ***n***
  - operators include `+ - * div mod`
  - relations include `<` $\leq$ `dvd` (divisibility)

Arithmetic Types

- `nat`: the natural numbers (nonnegative integers)
  - inductively defined: `0`, `Suc` *n*
  - operators include `+ - * div mod`
  - relations include `< ≤ dvd` (divisibility)
- `int`: the integers, with `+ - * div mod` ...

Arithmetic Types

- `nat`: the natural numbers (nonnegative integers)
  - inductively defined: `0`, `Suc` ***n***
  - operators include `+ - * div mod`
  - relations include `<` `≤` `dvd` (divisibility)
- `int`: the integers, with `+ - * div mod` ...
- `rat`, `real`: `+ - * / sin cos ln` ...

Arithmetic Types

- `nat`: the natural numbers (nonnegative integers)
  - inductively defined: `0`, `Suc` *n*
  - operators include `+ - * div mod`
  - relations include `<` $\leq$ `dvd` (divisibility)
- `int`: the integers, with `+ - * div mod` ...
- `rat`, `real`: `+ - * / sin cos ln` ...
- arithmetic constants and laws for these types

HOL as a Functional Language

recursive data type of lists

```
datatype 'a list = Nil | Cons 'a "'a list"

fun app :: "'a list => 'a list => 'a list"  where
   "app Nil ys = ys"
| "app (Cons x xs) ys = Cons x (app xs ys)"

fun rev  where
   "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"
```

recursive functions
(types can be inferred)

Proof by Induction

declaring a lemma

use it to simplify other formulas

```
lemma [simp]: "app xs Nil = xs"
  apply (induct xs)
  apply auto
  done
```

two steps: *induction* followed by *automation*

end of proof

Example of a *Structured Proof*

```
lemma "app xs Nil = xs"
proof (induct xs)
  case Nil
  show "app Nil Nil = Nil"
    by auto
next
  case (Cons a xs)
  show "app (Cons a xs) Nil = Cons a xs"
    by auto
qed
```

Example of a *Structured Proof*

- base case and inductive step can be proved explicitly

```
lemma "app xs Nil = xs"
proof (induct xs)
  case Nil
  show "app Nil Nil = Nil"
    by auto
next
  case (Cons a xs)
  show "app (Cons a xs) Nil = Cons a xs"
    by auto
qed
```

Example of a *Structured Proof*

- base case and inductive step can be proved explicitly

- Invaluable for proofs that need intricate manipulation of facts

```
lemma "app xs Nil = xs"
proof (induct xs)
  case Nil
  show "app Nil Nil = Nil"
    by auto
next
  case (Cons a xs)
  show "app (Cons a xs) Nil = Cons a xs"
    by auto
qed
```