

# Combining Interactive and Automatic Theorem Proving

Jia Meng

National ICT Australia

August 7, 2006

Introduction

Background

Formalizing Isabelle/HOL in FOL

Improving Integration's Performance – Relevance Filtering

Conclusions and Demo

# Interactive and Automatic Provers

# Interactive and Automatic Provers

- ▶ Interactive Provers (ITPs):
  - ▶ Examples: HOL, Isabelle, PVS, Coq
  - ▶ Rich formalism, used to verify complicated systems
  - ▶ Requires user interaction

# Interactive and Automatic Provers

- ▶ Interactive Provers (ITPs):
  - ▶ Examples: HOL, Isabelle, PVS, Coq
  - ▶ Rich formalism, used to verify complicated systems
  - ▶ Requires user interaction
- ▶ Automatic Provers (ATPs):
  - ▶ Examples: E, SPASS, Vampire, Otter, Bliksem
  - ▶ Completely automatic
  - ▶ Usually based on First Order Logic

# Interactive and Automatic Provers

- ▶ Interactive Provers (ITPs):
  - ▶ Examples: HOL, Isabelle, PVS, Coq
  - ▶ Rich formalism, used to verify complicated systems
  - ▶ Requires user interaction
- ▶ Automatic Provers (ATPs):
  - ▶ Examples: E, SPASS, Vampire, Otter, Bliksem
  - ▶ Completely automatic
  - ▶ Usually based on First Order Logic
- ▶ Can we combine them?

# The Integration of Interactive and Automatic Provers

# The Integration of Interactive and Automatic Provers

- ▶ Motivation: to improve interactive provers' automation

# The Integration of Interactive and Automatic Provers

- ▶ Motivation: to improve interactive provers' automation
- ▶ How: invoke an ATP from and ITP; ITP can then delegate proof tasks to ATP

# The Integration of Interactive and Automatic Provers

- ▶ Motivation: to improve interactive provers' automation
- ▶ How: invoke an ATP from and ITP; ITP can then delegate proof tasks to ATP
- ▶ Problems involved:
  - ▶ Translation of logic formalisms
  - ▶ Translation of types
  - ▶ Coping with large number of lemmas

# Integrating Isabelle with Resolution Provers

# Integrating Isabelle with Resolution Provers

- ▶ Isabelle automation of the L4.verified project, part of the work is in collaboration with Prof. Larry Paulson, University of Cambridge
- ▶ Have implemented linkups between Isabelle and resolution provers
  - ▶ Formalized Isabelle/HOL in FOL
  - ▶ Designed techniques to automatically remove unwanted lemmas

Introduction

Background

Formalizing Isabelle/HOL in FOL

Formalizing Isabelle/HOL Type System in FOL

Translating Typed HOL Formulae

Improving Integration's Performance – Relevance Filtering

The Relevance Problem

Our Relevance Filtering Methods

Effect of Relevance Filtering

Conclusions and Demo

# Isabelle and Resolution Provers

# Isabelle and Resolution Provers

- ▶ Isabelle
  - ▶ Generic, supports multiple logics: HOL, ZF, LCF, etc.
  - ▶ Tactics provide automation: *blast*, *simp*, *auto*
  - ▶ Large number of lemmas available, declared by users

# Isabelle and Resolution Provers

- ▶ Isabelle
  - ▶ Generic, supports multiple logics: HOL, ZF, LCF, etc.
  - ▶ Tactics provide automation: *blast*, *simp*, *auto*
  - ▶ Large number of lemmas available, declared by users
- ▶ Resolution Provers
  - ▶ First-order clauses
  - ▶ Numerous settings
  - ▶ Use Vampire, E and SPASS

# Outline of Integration

# Outline of Integration

- ▶ Invoke ATPs from Isabelle
  - ▶ Via Isabelle method: *apply vampire*

# Outline of Integration

- ▶ Invoke ATPs from Isabelle
  - ▶ Via Isabelle method: *apply vampire*
- ▶ Translate Isabelle's goals and lemmas into FOL clauses

# Outline of Integration

- ▶ Invoke ATPs from Isabelle
  - ▶ Via Isabelle method: *apply vampire*
- ▶ Translate Isabelle's goals and lemmas into FOL clauses
- ▶ Send these clauses to ATPs

# Outline of Integration

- ▶ Invoke ATPs from Isabelle
  - ▶ Via Isabelle method: *apply vampire*
- ▶ Translate Isabelle's goals and lemmas into FOL clauses
- ▶ Send these clauses to ATPs
- ▶ Users can choose to send specific lemmas to ATPs, or send all *automatically filtered* lemmas to ATPs

# Outline of Integration

- ▶ Invoke ATPs from Isabelle
  - ▶ Via Isabelle method: *apply vampire*
- ▶ Translate Isabelle's goals and lemmas into FOL clauses
- ▶ Send these clauses to ATPs
- ▶ Users can choose to send specific lemmas to ATPs, or send all *automatically filtered* lemmas to ATPs
- ▶ Successful proofs from ATP reported to Isabelle
  - ▶ With Isabelle method: finish off the subgoal, as an Oracle

Introduction

Background

**Formalizing Isabelle/HOL in FOL**

Formalizing Isabelle/HOL Type System in FOL

Translating Typed HOL Formulae

Improving Integration's Performance – Relevance Filtering

The Relevance Problem

Our Relevance Filtering Methods

Effect of Relevance Filtering

Conclusions and Demo

# Isabelle/HOL's Type System

# Isabelle/HOL's Type System

- ▶ A *type class*: a collection of types, possibly specified by *axioms* – axiomatic type class
  - ▶ e.g. `real : linorder`
- ▶ A type may belong to multiple type classes
- ▶ An intersection of type classes is a *sort*

# Isabelle/HOL's Type System

- ▶ An arity is about a type constructor, its argument type class and result type class:
  - ▶ e.g. `list :: (linorder) linorder`
- ▶ Overloaded constants with polymorphic types
  - ▶ e.g. `≤: α → α → bool`
  - ▶ `nat → nat → bool, α set → α set → bool`

# Translating Isabelle/HOL Types to FOL

# Translating Isabelle/HOL Types to FOL

- ▶ Types: FOL terms
- ▶ Type classes: FOL unary predicates  
e.g.  $linorder(real)$
- ▶ Arities: FOL formulae

$$\forall \tau [linorder(\tau) \rightarrow linorder(list(\tau))]$$

# Translating Isabelle/HOL Formulae to FOL

# Translating Isabelle/HOL Formulae to FOL

- ▶ FOL formulae admit simpler translation – only need to embed types:
  - ▶ Polymorphic constants: lift to functions and add type variable instantiations as arguments
    - ▶  $UNIV : \alpha \text{ set}$
    - ▶ When instantiated to  $bool \text{ set}$ , it becomes  $UNIV(bool)$
  - ▶ Extra argument of overloaded operator (except equality):
    - ▶  $\leq : \alpha \rightarrow \alpha \rightarrow bool$
    - ▶ When applied to sets, it becomes  $\leq(x, y, set(\alpha))$

# Translating Isabelle/HOL Formulae to FOL

- ▶ FOL formulae admit simpler translation – only need to embed types:
  - ▶ Polymorphic constants: lift to functions and add type variable instantiations as arguments
    - ▶  $UNIV : \alpha \text{ set}$
    - ▶ When instantiated to  $bool \text{ set}$ , it becomes  $UNIV(bool)$
  - ▶ Extra argument of overloaded operator (except equality):
    - ▶  $\leq : \alpha \rightarrow \alpha \rightarrow bool$
    - ▶ When applied to sets, it becomes  $\leq(x, y, set(\alpha))$
- ▶ HOL formulae: need a different treatment

# HOL Constructs to FOL – I

# HOL Constructs to FOL – I

- ▶ Differences between HOL and FOL formulae:
  - ▶ HOL's functions and predicates can be variables
  - ▶ HOL can contain  $\lambda$ -abstractions

# HOL Constructs to FOL – I

- ▶ Differences between HOL and FOL formulae:
  - ▶ HOL's functions and predicates can be variables
  - ▶ HOL can contain  $\lambda$ -abstractions
- ▶ To bridge the differences, we need to:
  - ▶ Regard all HOL predicates, functions are constants
  - ▶ Translate HOL predicate, function variables to first-order variables
  - ▶ A functional equality *fequal*
  - ▶ Use a two-argument function @ for function application:  
 $f(x) \rightarrow @(f, x)$
  - ▶ Use a predicate *B*, meaning “is true” for boolean terms

# HOL Constructs to FOL – I

- ▶ Differences between HOL and FOL formulae:
  - ▶ HOL's functions and predicates can be variables
  - ▶ HOL can contain  $\lambda$ -abstractions
- ▶ To bridge the differences, we need to:
  - ▶ Regard all HOL predicates, functions are constants
  - ▶ Translate HOL predicate, function variables to first-order variables
  - ▶ A functional equality *fequal*
  - ▶ Use a two-argument function @ for function application:  
 $f(x) \rightarrow @(f, x)$
  - ▶ Use a predicate  $B$ , meaning “is true” for boolean terms
- ▶ Example: translate  $\forall F p(F(x))$  to

$$B(@(p, @(F, x)))$$

# HOL Constructs to FOL – II

- ▶ For  $\lambda$ -abstractions, we need:
  - ▶  $\lambda$ -abstractions  $\rightarrow$  combinators **I**, **K**, **C**, **B** and **S**
  - ▶  $\lambda$ -reductions  $\rightarrow$  combinator reduction axioms

# HOL Constructs to FOL – II

- ▶ For  $\lambda$ -abstractions, we need:
  - ▶  $\lambda$ -abstractions  $\rightarrow$  combinators **I**, **K**, **C**, **B** and **S**
  - ▶  $\lambda$ -reductions  $\rightarrow$  combinator reduction axioms

- ▶ Additional axioms

- ▶ Explicit function extensionality:

$$\forall fg [(\forall x f(x) = g(x)) \rightarrow f = g]$$

$$[@(F, @(@(e, F), G)) = (@(G, @(@(e, F), G)))] \leftrightarrow [F = G]$$

- ▶ Lifting function equality to predicate level:

$$B(@(@(fequal, X), Y)) \leftrightarrow X = Y$$

# Embedding Types of HOL Formulae in FOL

- ▶ Need to include types to preserve soundness
- ▶ How much type information to include?
  - ▶ Include all types of all terms: safest, but too large terms, large clauses, bad ATP performance
  - ▶ Omit some types: smaller terms, better ATP performance
    - ▶ But may lead to unsound proofs
    - ▶ Need Isabelle to verify proofs
- ▶ Need a translation that delivers best result overall

# Fully-Typed Translation

# Fully-Typed Translation

- ▶ Every term is typed
  - ▶ Use “typeinfo” to pair each term with its type

# Fully-Typed Translation

- ▶ Every term is typed
  - ▶ Use “typeinfo” to pair each term with its type
- ▶ Example  $P < Q$ :

$$B(T(@ (T(@ (T(<, a \Rightarrow a \Rightarrow tb), T(P, a)), a \Rightarrow tb), T(Q, a)), tb))$$

# Fully-Typed Translation

- ▶ Every term is typed
  - ▶ Use “typeinfo” to pair each term with its type
- ▶ Example  $P < Q$ :

$$B(T(@ (T(@ (T(<, a \Rightarrow a \Rightarrow tb), T(P, a))), a \Rightarrow tb), T(Q, a)), tb))$$

- ▶ Translation is sound
  - ▶ Every term is typed, including variables
  - ▶ Two terms unify iff types are unified
  - ▶ Resolution steps can be reconstructed in Isabelle

# Fully-Typed Translation – Performance

- ▶ Practical performance
  - ▶ Much redundancy: repeated types, large terms
  - ▶ This harms prover performance

# Fully-Typed Translation – Performance

- ▶ Practical performance
  - ▶ Much redundancy: repeated types, large terms
  - ▶ This harms prover performance
- ▶ To improve performance
  - ▶ Omit some types (not all types!)
  - ▶ Retain the most important type information

# Partial-Typed Translation

# Partial-Typed Translation

- ▶ Only include types of functions in function calls

# Partial-Typed Translation

- ▶ Only include types of functions in function calls
- ▶ Insert the type term as a third argument of @

# Partial-Typed Translation

- ▶ Only include types of functions in function calls
- ▶ Insert the type term as a third argument of @
- ▶ Example:

$$B(@(@(<, P, a \Rightarrow a \Rightarrow tb), Q, a \Rightarrow tb))$$

# Partial-Typed Translation

- ▶ Only include types of functions in function calls
- ▶ Insert the type term as a third argument of @
- ▶ Example:

$$B(@(@(<, P, a \Rightarrow a \Rightarrow tb), Q, a \Rightarrow tb))$$

- ▶ Result: much smaller terms

# Partial-Typed Translation

- ▶ Only include types of functions in function calls
- ▶ Insert the type term as a third argument of @
- ▶ Example:

$$B(@(@(<, P, a \Rightarrow a \Rightarrow tb), Q, a \Rightarrow tb))$$

- ▶ Result: much smaller terms
- ▶ Ensures correct overloading of Isabelle most of the time

# Partial-Typed Translation – Soundness Issues

## Partial-Typed Translation – Soundness Issues

- ▶ Admits unsound proofs
- ▶ Example  $\forall x [x = a \vee x = b]$  is translated to

$$X = a \vee X = b$$

- ▶ But we intend to verify proofs

# Constant-Typed Translation

# Constant-Typed Translation

- ▶ Only include type instantiations of polymorphic constants

# Constant-Typed Translation

- ▶ Only include type instantiations of polymorphic constants
- ▶ A polymorphic constant  $\rightarrow$  a FOL function symbol
- ▶ Its instantiated type variables  $\rightarrow$  its argument

## Constant-Typed Translation

- ▶ Only include type instantiations of polymorphic constants
- ▶ A polymorphic constant  $\rightarrow$  a FOL function symbol
- ▶ Its instantiated type variables  $\rightarrow$  its argument
- ▶ Example  $P < Q$

$$B(@(@(< (nat), P), Q))$$

## Constant-Typed Translation

- ▶ Only include type instantiations of polymorphic constants
- ▶ A polymorphic constant  $\rightarrow$  a FOL function symbol
- ▶ Its instantiated type variables  $\rightarrow$  its argument
- ▶ Example  $P < Q$

$$B(@(@(< (nat), P), Q))$$

- ▶ Equality
  - ▶ The built-in equality predicate is untyped
  - ▶ Our boolean equality function is typed

# Constant-Typed Translation

- ▶ Only include type instantiations of polymorphic constants
- ▶ A polymorphic constant  $\rightarrow$  a FOL function symbol
- ▶ Its instantiated type variables  $\rightarrow$  its argument
- ▶ Example  $P < Q$

$$B(@(@(< (nat), P), Q))$$

- ▶ Equality
  - ▶ The built-in equality predicate is untyped
  - ▶ Our boolean equality function is typed
- ▶ This is our most compact form

## Constant-Typed Translation

- ▶ Only include type instantiations of polymorphic constants
- ▶ A polymorphic constant  $\rightarrow$  a FOL function symbol
- ▶ Its instantiated type variables  $\rightarrow$  its argument
- ▶ Example  $P < Q$

$$B(@(@(< (nat), P), Q))$$

- ▶ Equality
  - ▶ The built-in equality predicate is untyped
  - ▶ Our boolean equality function is typed
- ▶ This is our most compact form
- ▶ But admits unsoundness, like partial-typed translation

# Comparisons Between Three Translations

# Comparisons Between Three Translations

- ▶ Fully-typed translation: sound, but excessively large terms

## Comparisons Between Three Translations

- ▶ Fully-typed translation: sound, but excessively large terms
- ▶ Partial- and constant-typed translations: compact terms, but may introduce unsound proofs

# Comparisons Between Three Translations

- ▶ Fully-typed translation: sound, but excessively large terms
- ▶ Partial- and constant-typed translations: compact terms, but may introduce unsound proofs
- ▶ If use the latter two, must verify proofs

# Comparisons Between Three Translations

- ▶ Fully-typed translation: sound, but excessively large terms
- ▶ Partial- and constant-typed translations: compact terms, but may introduce unsound proofs
- ▶ If use the latter two, must verify proofs
- ▶ Experimental results showed compact translations significantly improved ATPs success rates
  - ▶ The significant improvement not due to unsound proofs!
  - ▶ Decide to use compact translations first, and verify proofs

Introduction

Background

Formalizing Isabelle/HOL in FOL

Formalizing Isabelle/HOL Type System in FOL

Translating Typed HOL Formulae

Improving Integration's Performance – Relevance Filtering

The Relevance Problem

Our Relevance Filtering Methods

Effect of Relevance Filtering

Conclusions and Demo

# Relevance Problem at Isabelle-ATP Integration

## Relevance Problem at Isabelle-ATP Integration

- ▶ When invoking ATPs, users can send all available lemmas to ATPs – less human effort
- ▶ Generate and send to ATPs large number of axioms: 1300 ~ 2500 clauses
  - ▶ Many axioms are irrelevant

## Relevance Problem at Isabelle-ATP Integration

- ▶ When invoking ATPs, users can send all available lemmas to ATPs – less human effort
- ▶ Generate and send to ATPs large number of axioms: 1300 ~ 2500 clauses
  - ▶ Many axioms are irrelevant
- ▶ But ATPs performance is affected by *search space*
  - ▶ Difficult for them to detect irrelevant axioms

# Relevance Problem at Isabelle-ATP Integration

- ▶ When invoking ATPs, users can send all available lemmas to ATPs – less human effort
- ▶ Generate and send to ATPs large number of axioms: 1300 ~ 2500 clauses
  - ▶ Many axioms are irrelevant
- ▶ But ATPs performance is affected by *search space*
  - ▶ Difficult for them to detect irrelevant axioms
- ▶ Short runtime

# Relevance Problem at Isabelle-ATP Integration

- ▶ When invoking ATPs, users can send all available lemmas to ATPs – less human effort
- ▶ Generate and send to ATPs large number of axioms: 1300 ~ 2500 clauses
  - ▶ Many axioms are irrelevant
- ▶ But ATPs performance is affected by *search space*
  - ▶ Difficult for them to detect irrelevant axioms
- ▶ Short runtime
- ▶ Need a relevance filter to remove irrelevant axioms before giving them to ATPs

# A Passmark-Based Algorithm – Overview

# A Passmark-Based Algorithm – Overview

- ▶ Include a clause if *sufficiently* close to one existing relevant clause

# A Passmark-Based Algorithm – Overview

- ▶ Include a clause if *sufficiently* close to one existing relevant clause
  - ▶ Distance/closeness – measured by *relevance mark*  $\in [0..1]$

# A Passmark-Based Algorithm – Overview

- ▶ Include a clause if *sufficiently* close to one existing relevant clause
  - ▶ Distance/closeness – measured by *relevance mark*  $\in [0..1]$
  - ▶ Sufficient – must exceed a *pass mark*  $\in [0..1]$

## A Passmark-Based Algorithm – Overview

- ▶ Include a clause if *sufficiently* close to one existing relevant clause
  - ▶ Distance/closeness – measured by *relevance mark*  $\in [0..1]$
  - ▶ Sufficient – must exceed a *pass mark*  $\in [0..1]$
- ▶ Test becomes stricter during each iteration

# A Passmark-Based Algorithm – Implementation

## A Passmark-Based Algorithm – Implementation

- ▶ Two sets: relevant clauses set  $R$ , irrelevant clauses set  $T$ ; a predetermined pass mark  $P$

## A Passmark-Based Algorithm – Implementation

- ▶ Two sets: relevant clauses set  $R$ , irrelevant clauses set  $T$ ; a predetermined pass mark  $P$
- ▶ Each clause is attached with a relevance mark

## A Passmark-Based Algorithm – Implementation

- ▶ Two sets: relevant clauses set  $R$ , irrelevant clauses set  $T$ ; a predetermined pass mark  $P$
- ▶ Each clause is attached with a relevance mark
- ▶ A clause in  $T$  becomes relevant if its relevance mark w.r.t a clause in  $R$  exceeds  $P$

## A Passmark-Based Algorithm – Implementation

- ▶ Two sets: relevant clauses set  $R$ , irrelevant clauses set  $T$ ; a predetermined pass mark  $P$
- ▶ Each clause is attached with a relevance mark
- ▶ A clause in  $T$  becomes relevant if its relevance mark w.r.t a clause in  $R$  exceeds  $P$
- ▶ Add all new relevant clauses from  $T$  to  $R$ , repeat

## A Passmark-Based Algorithm – Implementation

- ▶ Two sets: relevant clauses set  $R$ , irrelevant clauses set  $T$ ; a predetermined pass mark  $P$
- ▶ Each clause is attached with a relevance mark
- ▶ A clause in  $T$  becomes relevant if its relevance mark w.r.t a clause in  $R$  exceeds  $P$
- ▶ Add all new relevant clauses from  $T$  to  $R$ , repeat
- ▶ The relevance mark of a clause  $C$  w.r.t. a relevant clause  $RC$ :

$$M = (m/n) * M_{RC}$$

- ▶  $m$ : number of functions occurring in both  $C$  and  $RC$
- ▶  $n$ : total number of functions occurring in  $C$
- ▶  $M_{RC}$ : relevance mark of  $RC$

# Refinement 1 – Set of Relevant Functions

## Refinement 1 – Set of Relevant Functions

- ▶ Still have relevant clauses set  $R$ , irrelevant clauses set  $T$ , pass mark  $P$

## Refinement 1 – Set of Relevant Functions

- ▶ Still have relevant clauses set  $R$ , irrelevant clauses set  $T$ , pass mark  $P$
- ▶ Accumulate relevant functions set  $F$ : functions occurring in  $R$ 's clauses

## Refinement 1 – Set of Relevant Functions

- ▶ Still have relevant clauses set  $R$ , irrelevant clauses set  $T$ , pass mark  $P$
- ▶ Accumulate relevant functions set  $F$ : functions occurring in  $R$ 's clauses
- ▶ Relevance mark of irrelevant clause  $C$  calculated but w.r.t  $F$

$$M = m/n$$

- ▶  $m$ : number of functions occurring in both  $C$  and  $F$
- ▶  $n$ : total number of functions in  $C$

## Refinement 1 – Set of Relevant Functions

- ▶ Still have relevant clauses set  $R$ , irrelevant clauses set  $T$ , pass mark  $P$
- ▶ Accumulate relevant functions set  $F$ : functions occurring in  $R$ 's clauses
- ▶ Relevance mark of irrelevant clause  $C$  calculated but w.r.t  $F$

$$M = m/n$$

- ▶  $m$ : number of functions occurring in both  $C$  and  $F$
  - ▶  $n$ : total number of functions in  $C$
- ▶ Relevance test becomes stricter after each iteration

$$P = P + (1 - P)/c$$

## Refinement 1 – Set of Relevant Functions

- ▶ Still have relevant clauses set  $R$ , irrelevant clauses set  $T$ , pass mark  $P$
- ▶ Accumulate relevant functions set  $F$ : functions occurring in  $R$ 's clauses
- ▶ Relevance mark of irrelevant clause  $C$  calculated but w.r.t  $F$

$$M = m/n$$

- ▶  $m$ : number of functions occurring in both  $C$  and  $F$
  - ▶  $n$ : total number of functions in  $C$
- ▶ Relevance test becomes stricter after each iteration

$$P = P + (1 - P)/c$$

- ▶ Ran experiments to find best values of  $P$  and  $c$

$$P = 0.6, c = 2.4$$

# Advantages of Set of Relevant Functions

# Advantages of Set of Relevant Functions

- ▶ No need to attach relevance mark with each clause – easier implementation

# Advantages of Set of Relevant Functions

- ▶ No need to attach relevance mark with each clause – easier implementation
- ▶ Can handle clauses related via multiple clauses

## Refinement 2 – Taking Rarity into Account

## Refinement 2 – Taking Rarity into Account

- ▶ Motivation:
  - ▶ Rare functions more preferable
  - ▶ The rarer a relevant function, the more relevant it gets
  - ▶ Relevance marks should be weighted by functions' occurrences

## Refinement 2 – Taking Rarity into Account

- ▶ Motivation:
  - ▶ Rare functions more preferable
  - ▶ The rarer a relevant function, the more relevant it gets
  - ▶ Relevance marks should be weighted by functions' occurrences
- ▶ Relevance mark of a clause:

$$M = wm / (wm + n)$$

## Refinement 2 – Taking Rarity into Account

- ▶ Motivation:
  - ▶ Rare functions more preferable
  - ▶ The rarer a relevant function, the more relevant it gets
  - ▶ Relevance marks should be weighted by functions' occurrences
- ▶ Relevance mark of a clause:

$$M = wm / (wm + n)$$

- ▶  $wm$ : sum of rarity-weighted weights of relevant functions
- ▶  $n$ : sum of non-weighted weights of irrelevant functions

## Refinement 2 – Taking Rarity into Account

- ▶ Motivation:
  - ▶ Rare functions more preferable
  - ▶ The rarer a relevant function, the more relevant it gets
  - ▶ Relevance marks should be weighted by functions' occurrences
- ▶ Relevance mark of a clause:

$$M = wm / (wm + n)$$

- ▶  $wm$ : sum of rarity-weighted weights of relevant functions
  - ▶  $n$ : sum of non-weighted weights of irrelevant functions
- ▶ Need a suitable function to calculate weight of a function based on its rarity
  - ▶ Suitable function found from experiments:

$$weight(n) = 1 + 2 / \log(n + 1)$$

# Effect of Relevance Filtering

# Effect of Relevance Filtering

- ▶ Improved success rates for all ATPs that we use
- ▶ Some more dramatic:
  - ▶ SPASS (default setting) and Vampire success rates went up by almost 20%
  - ▶ E (version 9.0) success rate went up by 8%
- ▶ The filtering techniques are ATP-independent

Introduction

Background

Formalizing Isabelle/HOL in FOL

Formalizing Isabelle/HOL Type System in FOL

Translating Typed HOL Formulae

Improving Integration's Performance – Relevance Filtering

The Relevance Problem

Our Relevance Filtering Methods

Effect of Relevance Filtering

Conclusions and Demo

# Conclusions

# Conclusions

- ▶ Integrated Isabelle with ATPs

# Conclusions

- ▶ Integrated Isabelle with ATPs
- ▶ Formalized Isabelle/HOL in FOL: logic constructs and types

# Conclusions

- ▶ Integrated Isabelle with ATPs
- ▶ Formalized Isabelle/HOL in FOL: logic constructs and types
- ▶ Relevance filtering – improve practical performance

# Conclusions

- ▶ Integrated Isabelle with ATPs
- ▶ Formalized Isabelle/HOL in FOL: logic constructs and types
- ▶ Relevance filtering – improve practical performance
- ▶ Techniques can be applied to other integrations between interactive and automatic theorem provers

# Demo