Isabelle/HOL

Integrated Theorem Proving

Tjark Weber

webertj@in.tum.de

TECHNISCHE UNIVERSITÄT MÜNCHEN

Cooperation of Deduction Tools Day

April 10th, 2006

Isabelle

Isabelle is a generic proof assistant:

- Highly flexible
- Interactive
- Automatic proof procedures
- Advanced user interface
- Readable proofs
- Large theories of formal mathematics

Isabelle is Highly Flexible

Isabelle provides a meta-logic, Isabelle/Pure, based on the simply-typed λ-calculus.

This meta-logic serves as the basis for several different object logics:

- ZF set theory (Isabelle/ZF)
- First-order logic (Isabelle/FOL)
- Higher-order logic (Isabelle/HOL)
- Modal logics
- ...

Isabelle's Automatic Proof Procedures

Isabelle provides several automatic tactics and decision procedures.

- term rewriting (auto, simp)
- tableau-based (blast)
- Fourier-Motzkin (arith)
- Cooper's quantifier elimination (presburger)
- ...

These can easily be instantiated for different object logics.

Isabelle's User Interface

Isabelle uses Proof General as its default interface.

- Proof script management
- Mathematical symbols
- Document generation
- Theory dependency graph
- ...

A batch mode is available as well.

Isabelle/Isar: Readable Proofs

Isabelle/Isar is a structured proof language, where proofs resemble those found in mathematical textbooks.

```
proof (induct n)
   case 0 { ... }
   case (Suc n) { ... }
qed
```

A tactic-style proof language, where a proof is a sequence of tactic applications, is available as well.

Existing Formalizations in Isabelle

Substantial amounts of mathematics and computer science have been formalized with Isabelle.

- Algebra
- Calculus
- Graph theory
- ...

- Automata theory
- Programming language semantics, program logics
- Java bytecode verification
- ...

Isabelle/HOL

Isabelle/HOL: higher-order logic, based on Church's simple theory of types (1940)

Simply-typed λ-calculus:

- Types: $\sigma ::= \mathbb{B} \mid \alpha \mid \sigma \to \sigma$
- Terms: $t_\sigma ::= x_\sigma \mid (t_{\sigma' \to \sigma}\, t_{\sigma'})_\sigma \mid (\lambda x_{\sigma_1}.\, t_{\sigma_2})_{\sigma_1 \to \sigma_2}$

Two logical constants:

- $\Longrightarrow_{\mathbb{B}\to\mathbb{B}\to\mathbb{B}}, =_{\sigma\to\sigma\to\mathbb{B}}$

Other constants, e.g.

$$\texttt{True} \mid \texttt{False} \mid \neg \mid \wedge \mid \vee \mid \forall \mid \exists \mid \exists!$$

are definable.

Isabelle/HOL

Isabelle/HOL: higher-order logic, based on Church's simple theory of types (1940)

Simply-typed λ-calculus:

- Types: $\sigma ::= \mathbb{B} \mid \alpha \mid \sigma \to \sigma \mid (\alpha_1, \ldots, \alpha_n)c$
- Terms: $t_\sigma ::= x_\sigma \mid (t_{\sigma' \to \sigma}\, t_{\sigma'})_\sigma \mid (\lambda x_{\sigma_1}.\, t_{\sigma_2})_{\sigma_1 \to \sigma_2} \mid c_\sigma$

Two logical constants:

- $\Longrightarrow_{\mathbb{B} \to \mathbb{B} \to \mathbb{B}}, =_{\sigma \to \sigma \to \mathbb{B}}$

Other constants, e.g.

$$\texttt{True} \mid \texttt{False} \mid \neg \mid \wedge \mid \vee \mid \forall \mid \exists \mid \exists!$$

are definable.

Features of Isabelle/HOL

Isabelle/HOL also includes . . .

- axiomatic type classes
- type and constant definitions
- recursive datatypes
- recursive functions
- inductively defined sets
- locales (parameterized theory modules)

Isabelle/HOL is both a specification logic and a programming language.

The Semantics of HOL

Set-theoretic semantics:

- Types denote certain sets.
- Terms denote elements of these sets.

The Semantics of HOL

Set-theoretic semantics:

- Types denote certain sets.
- Terms denote elements of these sets.

A type environment E assigns to each type variable α a non-empty set E_α.

Semantics of types:

- $[\![\mathbb{B}]\!] = \{\top, \bot\}$
- $[\![\alpha]\!] = E_\alpha$
- $[\![\sigma_1 \to \sigma_2]\!] = [\![\sigma_2]\!]^{[\![\sigma_1]\!]}$

LCF-Style Systems

Theorems are implemented as an abstract datatype. They can be constructed only in a very controlled manner, through calling functions from the system's kernel.

There is one such kernel function for each inference rule of the system's logic.

Advanced tactics and decision procedures are built on top of the kernel. They must use the kernel functions (or other existing tactics) to create theorems.

Integrating External Provers

The oracle approach: the system is told to simply accept the external prover's result, bypassing any checking by the kernel.

The LCF-style approach: every statement claimed to be valid by the external prover is proved again in the system, using the kernel's inference rules.

We are aiming for the LCF-style approach. This requires the external system to return not just a theorem, but also its proof (which must then be replayed).

Recent Prover Integrations

- SAT: zChaff, MiniSAT (Tjark Weber, Alwen Tiu et al.)
- SMT: haRVey (Stephan Merz et al.)
- FOL: Spass, Vampire (Jia Meng, Larry Paulson)
- HOL: HOL 4, HOL-Light (Sebastian Skalberg, Steven Obua)

SAT: Motivation

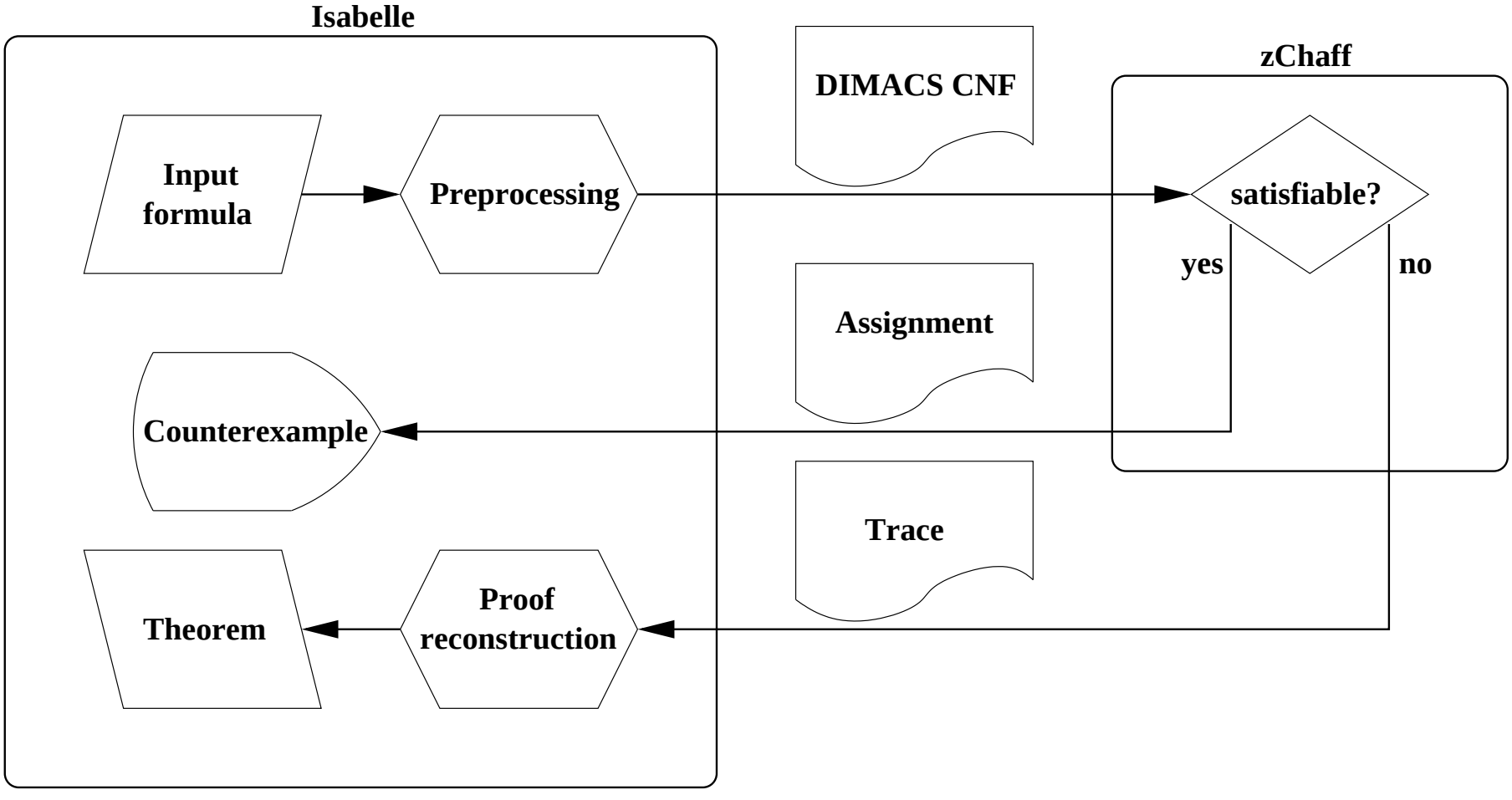
- Verification problems can often be reduced to Boolean satisfiability.
- Recent SAT solver advances have made this approach feasible in practice.

Can an LCF-style theorem prover benefit from these advances?

zChaff

- A leading SAT solver (winner of the SAT 2002 and SAT 2004 competitions in several categories)
- Developed by Sharad Malik and Zhaohui Fu, Princeton University
- Returns a satisfying assignment, or . . .
- . . . a proof of unsatisfiability (since 2003)

System Overview



Preprocessing

Input: propositional formula ϕ

- CNF conversion
- Normalization
- Removal of duplicate literals
- Removal of tautological clauses

Output: a theorem of the form $\phi = \phi^*$

The SAT Solver's Trace

```
CL: 184 <= 173 28 35 142 154
CL: 185 <= 43 4 11 59 55
[...]
VAR: 16 L: 35 V: 0 A: 55 Lits: 29 33
VAR: 26 L: 28 V: 1 A: 202 Lits: 52 98 57
[...]
CONF: 206 == 80 82 64 70 37
```

The SAT Solver's Trace

clause id resolvents

```
CL: 184 <= 173 28 35 142 154
CL: 185 <= 43 4 11 59 55
[...]
VAR: 16 L: 35 V: 0 A: 55 Lits: 29 33
VAR: 26 L: 28 V: 1 A: 202 Lits: 52 98 57
[...]
CONF: 206 == 80 82 64 70 37
```

variable id antecedent

conflict clause id

The SAT Solver's Trace

clause id resolvents

```
CL: 184 <= 173 28 35 142 154
CL: 185 <= 43 4 11 59 55
[...]
VAR: 16 L: 35 V: 0 A: 55 Lits: 29 33
VAR: 26 L: 28 V: 1 A: 202 Lits: 52 98 57
[...]
CONF: 206 == 80 82 64 70 37
```

variable id antecedent

conflict clause id

```
type proof = (int list) Inttab.table * int
```

The Intermediate Proof Format

```
type proof = (int list) Inttab.table * int
```

- Each integer is a clause identifier.
- Clauses of the original problem are numbered consecutively, starting from 0.
- Each list of integers gives the resolvents for the associated key.
- No circular dependencies between clauses.
- (At least) one clause must be the empty clause.
- Its ID is given by the second component of the proof.

Proof Reconstruction

- `resolution : Thm.thm list -> Thm.thm`

- `prove_clause : int -> Thm.thm`

- `replay_proof : (Thm.thm option) array`
  `-> SatSolver.proof -> Thm.thm`

Proof Reconstruction

- `resolution : Thm.thm list -> Thm.thm`

  Input: $[\![x_1; \ldots; a; \ldots; x_n]\!] \Longrightarrow$ `False`

  $[\![y_1; \ldots; \neg a; \ldots; y_m]\!] \Longrightarrow$ `False`

  Result: $[\![x_1; \ldots; x_n; y_1; \ldots; y_m]\!] \Longrightarrow$ `False`

- `prove_clause : int -> Thm.thm`

- `replay_proof : (Thm.thm option) array`
  `-> SatSolver.proof -> Thm.thm`

Proof Reconstruction

- `resolution : Thm.thm list -> Thm.thm`

  Input: $[\![x_1; \ldots; a; \ldots; x_n]\!] \Longrightarrow$ `False`

  $[\![y_1; \ldots; \neg a; \ldots; y_m]\!] \Longrightarrow$ `False`

  Result: $[\![x_1; \ldots; x_n; y_1; \ldots; y_m]\!] \Longrightarrow$ `False`

- `prove_clause : int -> Thm.thm`

```
prove_clause clause_id =
  resolution (map prove_clause
    (resolvents_of clause_id))
```

- `replay_proof : (Thm.thm option) array -> SatSolver.proof -> Thm.thm`

Proof Reconstruction

- `resolution : Thm.thm list -> Thm.thm`

  Input: $[\![x_1; \ldots; a; \ldots; x_n]\!] \Longrightarrow$ `False`

  $[\![y_1; \ldots; \neg a; \ldots; y_m]\!] \Longrightarrow$ `False`

  Result: $[\![x_1; \ldots; x_n; y_1; \ldots; y_m]\!] \Longrightarrow$ `False`

- `prove_clause : int -> Thm.thm`

```
prove_clause clause_id =
  resolution (map prove_clause
    (resolvents_of clause_id))
```

- `replay_proof : (Thm.thm option) array -> SatSolver.proof -> Thm.thm`

```
prove_clause empty_clause_id
```

Evaluation

- Isabelle is several orders of magnitude slower than zverify_df.
- However, zChaff vs. auto/blast/fast . . .
  - 42 propositional problems in TPTP, v2.6.0
    - 19 “easy” problems, solved in less than a second each by auto, blast, fast, and zChaff
    - 23 harder problems

Performance (1)

Problem	Status	auto	blast	fast	sat
MSC007-1.008	unsat.	x	x	x	726.5
NUM285-1	sat.	x	x	x	0.2
PUZ013-1	unsat.	0.5	x	5.0	0.1
PUZ014-1	unsat.	1.4	x	6.1	0.1
PUZ015-2.006	unsat.	x	x	x	10.5
PUZ016-2.004	sat.	x	x	x	0.3
PUZ016-2.005	unsat.	x	x	x	1.6
PUZ030-2	unsat.	x	x	x	0.7
PUZ033-1	unsat.	0.2	6.4	0.1	0.1
SYN001-1.005	unsat.	x	x	x	0.4
SYN003-1.006	unsat.	0.9	x	1.6	0.1
SYN004-1.007	unsat.	0.3	822.2	2.8	0.1
SYN010-1.005.005	unsat.	x	x	x	0.4
SYN086-1.003	sat.	x	x	x	0.1
SYN087-1.003	sat.	x	x	x	0.1
SYN090-1.008	unsat.	13.8	x	x	0.5
SYN091-1.003	sat.	x	x	x	0.1
SYN092-1.003	sat.	x	x	x	0.1
SYN093-1.002	unsat.	1290.8	16.2	1126.6	0.1
SYN094-1.005	unsat.	x	x	x	0.8
SYN097-1.002	unsat.	x	19.2	x	0.2
SYN098-1.002	unsat.	x	x	x	0.4
SYN302-1.003	sat.	x	x	x	0.4

Improvements

- Representation of clauses as implications:
  $l_1 \vee \ldots \vee l_n$ is equivalent to $[\![\neg l_1; \ldots; \neg l_n]\!] \Longrightarrow \texttt{False}$
- Representation of clauses as sequents:
  the above is equivalent to $\{\neg l_1; \ldots; \neg l_n\} \vdash \texttt{False}$
- Proof reconstruction for needed clauses only:
  starting with the conflict clause, resolvents are proved recursively and stored in an array
- Minor implementation optimizations:
  bookkeeping instead of “trial and error” during resolution; reordering resolution steps to minimize the size of the assumption sets

Performance (2)

Problem	Status	sat (naive)	sat (implications)	sat (optimized implications)	sat (sequents)
MSC007-1.008	unsat.	726.5	11.5	7.9	1.2
PUZ015-2.006	unsat.	10.5	2.4	0.7	0.2
PUZ016-2.005	unsat.	1.6	1.2	0.6	0.1
PUZ030-2	unsat.	0.7	0.5	0.4	0.1
PUZ033-1	unsat.	0.1	0.1	0.1	0.1
SYN090-1.008	unsat.	0.5	0.5	0.3	0.1
SYN093-1.002	unsat.	0.1	0.1	0.1	0.1
SYN094-1.005	unsat.	0.8	0.7	0.5	0.1

Performance (2)

Problem	Status	sat (naive)	sat (implications)	sat (optimized implications)	sat (sequents)
MSC007-1.008	unsat.	726.5	11.5	7.9	1.2
PUZ015-2.006	unsat.	10.5	2.4	0.7	0.2
PUZ016-2.005	unsat.	1.6	1.2	0.6	0.1
PUZ030-2	unsat.	0.7	0.5	0.4	0.1
PUZ033-1	unsat.	0.1	0.1	0.1	0.1
SYN090-1.008	unsat.	0.5	0.5	0.3	0.1
SYN093-1.002	unsat.	0.1	0.1	0.1	0.1
SYN094-1.005	unsat.	0.8	0.7	0.5	0.1

- A fast decision procedure for propositional logic
- Proof reconstruction (and preprocessing) takes most of the time.
- A good understanding of the prover's internals is important for an efficient implementation.
- Integration of an incremental SAT solver?

Proof Objects for Isabelle/HOL

Proof objects constitute certificates that are easily verifiable by an external proof checker.

Proof objects in Isabelle (implemented by Stefan Berghofer) use λ-terms, based on the Curry-Howard isomorphism. Information that can be inferred is omitted to reduce the size of the proof object.

Applications: proof-carrying code, proof export, program extraction from proofs

For programs extracted from proofs, a realizability statement is automatically proven to establish the program's correctness.

Code Generation for Isabelle/HOL

Motivation: executing formal specifications, rapid prototyping, program extraction, reflection

ML code can be generated from executable HOL terms (implemented by Stefan Berghofer). Such terms are built from executable constants, datatypes, inductive relations and recursive functions.

Translating inductive relations requires a mode analysis.

Normalization by evaluation: simplifies executable terms (possibly containing free variables) by evaluating them symbolically. Orders of magnitude faster than rewriting.

Reflection

Instead of implementing a decision procedure in ML, we write it in the logic (Isabelle/HOL) – as a recursive function on an appropriately defined datatype representing terms –, prove its correctness, and generate code from the definition.

Motivation: Assuming that the code generator is part of the trusted kernel, the generated code can simply be executed, with no need for proof checking. This can lead to a significant speed-up.

Amine Chaieb has implemented a reflected version of Cooper's quantifier elimination procedure for Presburger arithmetic. A generic reflection interface for user-defined decision procedures is soon to come.

Disproving Non-Theorems

Traditionally, the focus in theorem proving has been on, well, proving theorems.

However, disproving a faulty conjecture can be just as important. In formal verification, initial conjectures are more often false than not, and a counterexample often exhibits a fault in the implementation.

Isabelle provides two essentially different tools for disproving conjectures: quickcheck (by Stefan Berghofer) and refute (by Tjark Weber).

quickcheck

- Inspired by the Haskell `quickcheck` library (for testing Haskell programs)
- Free variables are instantiated with random values.
- The code generator is used to simplify the resulting term.
- Parameters: size of instantiations, number of iterations
- Fast, but not suited for existential or non-executable statements
- Implications with strong premises are problematic.

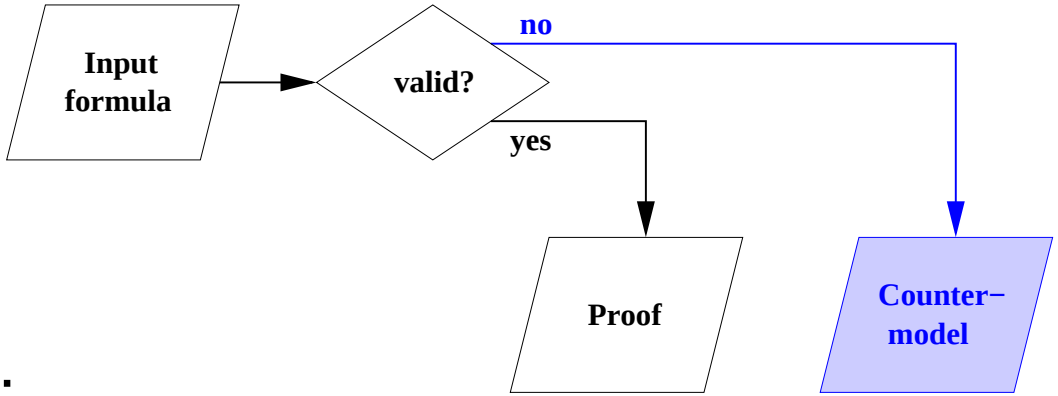
refute

- A SAT-based finite model generator
- A HOL formula is translated into a propositional formula that is satisfiable iff the HOL formula has a model of a given size.
- Parameters: (minimal, maximal) size of the model
- Any statement can be handled, but non-elementary complexity.
- Still useful in practice (“small model property”)

Finite Model Generation

Theorem proving: from formulae to proofs

Finite model generation: from formulae to models



Applications:

- Finding counterexamples to false conjectures
- Showing the consistency of a specification
- Solving open mathematical problems
- Guiding resolution-based provers

Overview

Input: HOL formula ϕ

Output: either a model for ϕ, or “no model found”

Overview

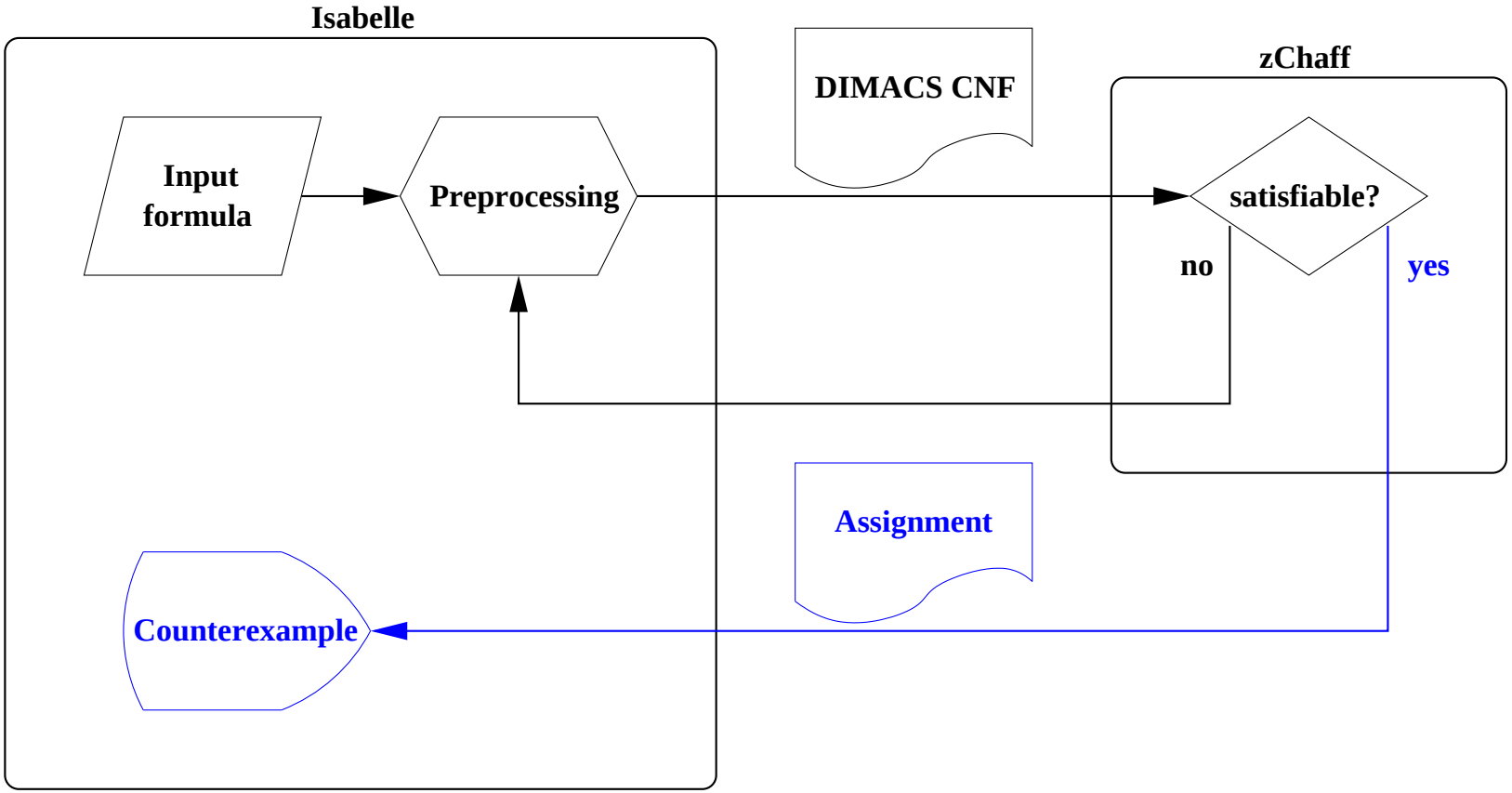
Input: HOL formula ϕ

1. Fix a finite type environment E.
2. Translate ϕ into a Boolean formula that is satisfiable iff $[\![\phi]\!]_E^A = \top$ for some variable assignment A.
3. Use a SAT solver to search for a satisfying assignment.
4. If a satisfying assignment was found, compute from it the variable assignment A. Otherwise repeat for a larger environment.

Output: either a model for ϕ, or “no model found”

Overview

Input: HOL formula ϕ



Output: either a model for ϕ, or "no model found"

Fixing a Finite Environment

Fix a positive integer for every type variable that occurs in the typing of ϕ.

Every type then has a finite size:

- $|\mathbb{B}| = 2$
- $|\alpha|$ is given by the environment
- $|\sigma_1 \to \sigma_2| = |\sigma_2|^{|\sigma_1|}$

Finite model generation is a generalization of satisfiability checking, where the search tree is not necessarily binary.

The SAT Solver

Several external SAT solvers (zChaff, BerkMin, Jerusat, . . .) are supported.

- Efficiency
- Advances in SAT solver technology are “for free”

The SAT Solver

Several external SAT solvers (zChaff, BerkMin, Jerusat, . . .) are supported.

- Efficiency
- Advances in SAT solver technology are “for free”

Simple internal solvers are available as well.

- Easy installation
- Compatibility
- Fast enough for small examples

Some Extensions

Sets are interpreted as characteristic functions.

- $\sigma\ \texttt{set} \cong \sigma \to \mathbb{B}$
- $x \in P \cong P\,x$
- $\{x.\,P\,x\} \cong P$

Non-recursive datatypes can be interpreted in a finite model.

- $(\alpha_1, \ldots, \alpha_n)\sigma ::= C_1\,\sigma_1^1 \ldots \sigma_{m_1}^1 | \ldots | C_k\,\sigma_1^k \ldots \sigma_{m_k}^k$
- $|(\alpha_1, \ldots, \alpha_n)\sigma| = \sum_{i=1}^{k} \prod_{j=1}^{m_i} |\sigma_j^i|$
- Examples: *option*, *sum*, *product* types

Some Extensions

Recursive datatypes are restricted to initial fragments.

- Examples: $\texttt{nat}$, $\sigma\ \texttt{list}$, $\texttt{lambdaterm}$
- $\texttt{nat}^1 = \{0\}$, $\texttt{nat}^2 = \{0, 1\}$, $\texttt{nat}^3 = \{0, 1, 2\}, \ldots$
- This works for datatypes that occur only positively.

Datatype constructors and recursive functions can be interpreted as partial functions.

- Examples: $\mathrm{Suc}_{\mathrm{nat}\to\mathrm{nat}}$, $+_{\mathrm{nat}\to\mathrm{nat}\to\mathrm{nat}}$, $@_{\sigma\mathrm{list}\to\sigma\mathrm{list}\to\sigma\mathrm{list}}$
- 3-valued logic: true, false, unknown

Axiomatic type classes introduce additional axioms that must be satisfied by the model.

Records and inductively defined sets can be treated as well.

Soundness and Completeness

If the SAT solver is sound/complete, we have ...

- Soundness: The algorithm returns “model found” only if the given formula has a finite model.
- Completeness: If the given formula has a finite model, the algorithm will find it (given enough time).
- Minimality: The model found is a “smallest” model for the given formula.

Conclusions

- Isabelle is a powerful interactive theorem prover.
- A reasonably high degree of automation is available through both internal and external tools.
- Definitional packages and existing libraries facilitate the development of new specifications.
- A human-readable proof language, existing lemmas and a good user interface facilitate the development of proofs.
- Several side applications: proof import/export, code extraction, counterexamples for unprovable formulae

Future Work / Wishlist

- The integration of external provers for different logics is ongoing work.
- External provers often lack the ability to produce proofs.
- Even if they can produce proofs, usually no standard proof format exists.
- Performance is an issue – more with Isabelle than with the external tool.