Pragmatic Higher-Order Theorem Proving via Embedding a Lambda Calculus in First-Order Logic Utilising De Bruijn Indices

Ahmed Bhayat & Giles Reger

The University of Manchester

The University of Manchester

Outline Of Presentation

1. ‘Standard’ translation from higher-order to first-order logic (implemented)
2. Eta-long form translation (ongoing)
3. Deduction Modulo (future work, tying together (1) and (2))

MANCHESTER
1824
The University of Manchester

The Vampire Prover

- Modern, award-winning saturation based, first-order theorem prover
- Implements a resolution and superposition calculus
- Track record of modifiability

$$\underbrace{\boxed{f(x) = x} \qquad g(\boxed{f(x)}, c, b, z)}_{g(x, c, b, z)} \qquad\qquad \underbrace{C \vee \neg P(a) \qquad D \vee P(x)}_{(C \vee D)[x \rightarrow a]}$$

MANCHESTER 1824
The University of Manchester

Vampire Higher-Order

- Project started roughly nine-month ago
- Vampire already being run as back-end to interactive provers
- Why not develop translation module?
  - In control of translation
  - Aware of axioms
  - Can easily modify inference rules

Applicative Translation

More or less ‘standard’:

- Lambda functions translated using combinators
- Application translated using binary *app* function
- Higher-order logical constants and combinators axiomatised

$$mforall = \lambda_{Phi:(i \to o) \to i \to o, W:i}.(\forall_{P:i \to o} : Phi\ P\ W)$$

$$\downarrow$$

$$mforall = app(app(b\_comb, app(b\_comb, \Pi)), c\_comb)$$

Applicative Translation

Drawbacks:

- Structure of original lost
- Head symbol deeply embedded
- Apps and combinators can clog up data structures
- Translation is incomplete. No way to prove:

  $$\exists F : \forall X, Y : F\ X\ Y = g\ Y\ X$$

- Can we do better?

MANCHESTER 1824
The University of Manchester

De Bruijn Indices

- A nameless version of the lambda-calculus

$$\lambda.\lambda.(f\ a\ \lambda.(3\ 2\ b)\ 1)$$

- Lambda is no longer a binder. Can be treated as a unary function
- Indices can be treated as first-order constants
- Partial application:
  - – Use two place *app*
  - – Store all terms in eta-long form ✓

MANCHESTER 1824
The University of Manchester

De Bruijn Translation

- Higher-order variables remain
- Allow them to remain and update provers structures and algorithms to deal with them
- Not obvious how to update superposition
  - Developing simplification orders in the presence of lambdas is a challenge

$$a = (\lambda x.a)\ f \succ a \qquad \text{by sub-term}$$

Pragmatism

- Block superposition from being carried out on terms containing higher-order variables
- Rely on resolution
- To be complete, unification must be modulo beta and eta-reduction
- Higher-order unification
  – Semi-decidable
  – Generates complete sets of unifiers, prolific

Pragmatism (2)

- Unify a sub-class of terms
- Candidate unification algorithms:
  - Pattern unification
  - Prefix unification
- Perhaps implementing these unification algorithms is sufficient to prove a large class of interesting problems?

Prefix Unification

- Unify higher-order variable with prefix term which has same type

$X_{i \to i \to i}\ b\ c$

$f_{int \to i \to i \to i}\ a\ b\ c$ ✓

$g_{int \to int \to int \to int}\ d\ d\ d$ ✗

- Prefix unification is decidable
- Most general unifiers exist

MANCHESTER 1824
The University of Manchester

Prefix Unification

- Vampire uses substitution tree for matching and unification
- All children of a node bind one special variable
- Bound terms stored in order of <u>head symbol</u>

$$
\begin{tikzcd}
& *_0 \arrow[dl] \arrow[dd] \arrow[dr] & \\
*_0 = f_{i\to i\to i\to i}(a, *_1, g(*_2)) \arrow[d] & & *_0 = h_{i\to i\to i}(*_1, g(*_2)) \arrow[d] \\
*_1 = c \arrow[d] & *_0 = s_{i\to o}(d) & *_1 = d \arrow[d] \\
*_2 = e & & *_2 = g(a)
\end{tikzcd}
$$

Solution

- Store terms in ‘buckets’ based on type of head symbol
- Each node stores a list of buckets
- Buckets for node $*0$

$\text{Bucket label} = i \to i \to i \to i$ $\quad$ $\text{Bucket label} = i \to o$

The University of Manchester

Solution

- Query term has variable head:
  - Return all terms with same or larger type in relevant bucket

$$\text{Query term} = X_{i\to i\to i}(Y, g(Z))$$

$\text{Bucket label} = i \to i \to i \to i$ $\quad$ $\text{Bucket label} = i \to o$

Solution

- Query term has rigid head:
  - Return all flexible terms with same or smaller type in relevant bucket

$$\text{Query term} = h_{i \to i \to i}(Y, g(Z))$$

$\text{Bucket label} = i \to i \to i \to i$ $\text{Bucket label} = i \to o$

Future Work

- What is the bigger picture?
- Treat higher-order logic as a first-order theory
- Various axiomatisations possible (Dowek, 2008)
  - With combinators
  - With De Bruijn indices and explicit substitutions
- Axiomatisations can lead to non-goal directed search

Deduction Modulo

- Dowek et al. (2003) introduced deduction modulo
- Treat axioms of theory as rewrite rules
  - Term rewrite rules:

    $$app(app(K, term1), term2) \rightarrow term1$$

  - Propositional rewrite rules:

    $$p \rightarrow \forall X.g(X)$$

Deduction Modulo

- Resolution now becomes resolution modulo

$$\frac{A \vee C_1 \qquad \neg B \vee C_2}{C_1 \vee C_2 \mid A =_E B}$$

- Carry unification constraints
- Unification is modulo set of equations E
- Introduce new inference rule *extended narrowing*

$$\frac{U \mid C}{U' \mid C \wedge U|_p =_E L}$$ where $L \rightarrow R$ is a rewrite rule and $U' = clausified(U[R]_p)$

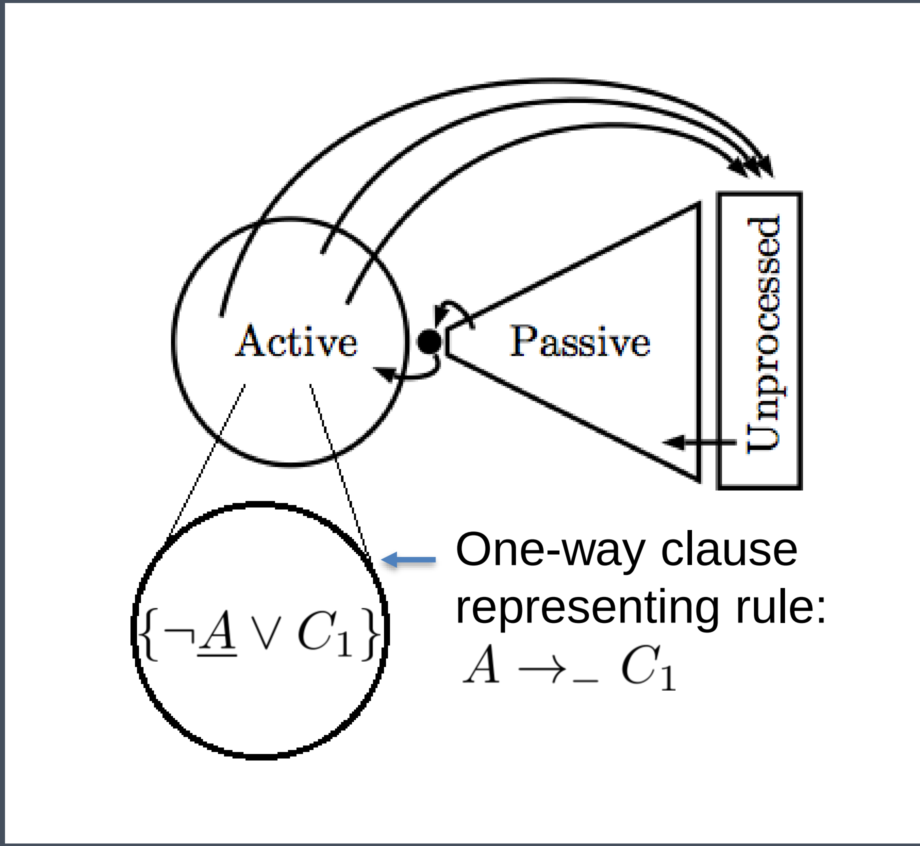
MANCHESTER 1824
The University of Manchester

Deduction Modulo

- Resolution modulo is a complete proof method for any theory that has cut-elimination property
- There has been further work on resolution modulo:
  - Polarised resolution modulo
  - Ordered polarised resolution modulo
- Some strong results for the latter
  - The rewrite rules do not need to be compatible with the ordering relationship $\succ$

Ordered Polarised Resolution Modulo

- Create polarity aware rewrite rules
- No need for clausification
- Add ordering restrictions to deduction modulo
- Still complete



In Practice

- At least two practical attempts at implementation:
  - iProver modulo
  - Zenon modulo
- Both showed some promise
- Many questions, theoretical and practical remain

Open Questions

- Can there be a superposition modulo complete for all theories that enjoy cut-elimination?
- If yes, can the independence between the rewrite rules and $\succ$ be maintained?
- How to recognise unsatisfiable constraints?
- Indexing data structures for unification modulo?

Superposition Modulo?

- Normal completeness proof relies on saturation of clause set with respect to $\succ$
- One-way clauses would have to be saturated as well
- This creates a dependency between the rewrite system and the ordering
- Is this necessary?

Deduction Modulo and Higher-Order Logic

- Both axiomatisation of higher-order logic enjoy cut-elimination
- With combinators unification is modulo:

$$\left.\begin{array}{l} app(app(K, term1), term2) \to term1 \\ app(I, term1) \to term1 \\ \vdots \end{array}\right\} E$$

Deduction Modulo and Higher-Order Logic

- With De Bruijn indices and explicit substitutions unification is modulo the rules of the $\lambda\sigma$-calculus
- Both unification algorithms have been studied
- Both are semi-decidable

An idea:

- Run unification algorithm to some depth
- If small complete set of unifiers returned, apply unifiers
- Otherwise leave as constraint on clause

Further Thoughts

- Is $\lambda\sigma$-calculus the best explicit substitution calculus for the purpose?
- How to update Vampire's highly optimised term structure without harming performance?
- Can substitution trees be updated to handle unification modulo the rewrite rules of either translation?

Questions

?