AROUND DEDUCING AND PROVING

Dr. Christelle Scharff

Assistant Professor

Pace University
ABOUT ME

- Dr. Christelle Scharff
- PhD from France - 1999
- Research: Automated deduction and theorem proving, Verification of hardware and software, New technologies in education, Languages.
- Teaching in France, in Cambodia, in USA (State University of New York at Stony Brook).
- cscharff@pace.edu
- http://www.csis.pace.edu/~scharff/
• What are deduction and theorem proving?

• **Parallelism** in automatic deduction and theorem proving

• **Simplification** in automatic deduction and theorem proving

• **Constraints** in automatic deduction and theorem proving

• Further work and objectives
GENERAL CONTEXT

MECHANIZATION OF REASONING

Aristotle Syllogism

All men are mortal.
Socrates is a a man.
Therefore, Socrates is mortal.

↓

Model the problem

Hypothesis: \( \text{Man}(x) \Rightarrow \text{Mortal}(x) \)

\( \text{Man}(\text{Socrates}) \)

Goal to prove: \( \text{Mortal}(\text{Socrates}) \)

↓

Prove the goal
DEDUCTION

• Deduce data from existing data.

THEOREM PROVING

• Given a formula $F$.

• Want to know if $F$ is true from existing data.
**HOW TO REPRESENT DATA?**

- A **term** is
  - a **variable** \((x, y, z...)\) or a **constant** \((a, b, c...)\)
  - \(f(t_1, \ldots, t_n)\) where \(f\) is a \(n\)-ary function symbol and each \(t_i\) is a term.

\(u[s]\): The term \(u\) contains the term \(s\).

A ground term \((f(a), g(c, d)...\) has no variables.

A term can be represented by a tree.

\(t = f(a + x, h(f(a, b)))\) is represented by:

![Tree representation of a term](tree.png)
• $\approx$ is a congruence relation i.e.

1. Reflexive: $x \approx x$

2. Symmetric: $x \approx y \Rightarrow y \approx x$

3. Transitive: $x \approx y$ and $y \approx z \Rightarrow x \approx z$

4. Congruence:
   
   $x_1 \approx y_1$ and ... and $x_n \approx y_n \Rightarrow f(x_1, \ldots, x_n) \approx f(y_1, \ldots, y_n)$
• **Equality**

\[ \forall x, y, (x + y \approx y + x) \) (commutativity - C) \]

\[ \forall x, y, z, ((x + y) + z) \approx x + (y + z) \) (associativity - A) \]

\[ f(a) \approx b \]

\[ \forall x, (f(x) \approx a) \]

• **First order logic**

\[ \forall x, y, \ (\text{parent}(x, y) \text{ and parent}(y, z) \Rightarrow \text{grandparent}(x, z)) \]

• **First order logic with equality**

\[ \forall x, \ (x \not\approx 0 \Rightarrow \exists y, \ (y \approx 1/x)) \]
Theorem prover =
Inference rules
+
Proof strategies

- **Inference rules:** How to deduce new data? How to remove data?

- **Strategies:** How to apply inference rules?
  Interactive? automatic?
  Expansion strategies? Simplification strategies? [Bonacina, 99]

**Issue:** Control the deduction of new data.
PROVING BUT...

- **Soundness**
  
  Do not prove that True = False
  
  Proving only true formulas.

- **Completeness**
  
  If a formula is true, it can be proved.

Syntactic/Semantic
CAN WE PROVE EVERYTHING?

- All mathematical truths CANNOT be determined by following a valid logical proof procedure.

- Kurt Goedel (1930) (Czech mathematician) proved this result known as **Incompleteness Theorem of Goedel**.
DIFFERENT PROVING METHODS

- Induction proofs

- Contradiction proofs
  - First order logic
    - Resolution
  - First order logic with equality
    - Paramodulation

- Saturation proofs / Compilation proofs
  - Equational logic
    - Completion
      - Resolution, Paramodulation

...
INDUCTION PROOFS

• Example:

  – The **tower of Hanoi** consists of a fixed number of **disks** stacked on a pole in decreasing size, that is, with the smallest disk at the top.
There are two other poles and the task is to transfer all disks from the first to the third pole, one at a time without ever placing a larger disk on top of a smaller one.

There is an elegant solution to this problem by recursion.
- **Question:** How many moves are needed, at the least, to transfer a tower of $k$ disks?
- Observe that we need to get to the following intermediate configuration, so as to be able to move the largest disk.

That is, we have to transfer the $k - 1$ smaller disks to the middle pole, we can then move the largest disks from the first to the third pole, and finally the $k - 1$ smaller disks from the second pole to the third pole.
Let \( M(k) \) be the minimum number of moves required to transfer \( k \) disks from one pole to another pole.

\[
M(k) = M(k - 1) + 1 + M(k - 1) = 2M(k - 1) + 1 \text{ for all } k > 0.
\]

In addition, we set \( M(0) = 0 \).

Let us evaluate the function for some arguments:

\[
\begin{align*}
M(0) &= 0 \\
M(1) &= 2M(0) + 1 = 1 \\
M(2) &= 2M(1) + 1 = 3 \\
M(3) &= 2M(2) + 1 = 7 \\
M(4) &= 2M(3) + 1 = 15 \\
M(5) &= 2M(4) + 1 = 31 \\
M(6) &= 2M(5) + 1 = 63...
\end{align*}
\]

The values grow fairly fast. In fact one can show that the function \( M \) can be explicitly defined by \( M(k) = 2^k - 1 \) for all \( k \geq 0 \).
— Prove that the 2 definitions of \( M(k) \) are equivalent.

**Case** \( n = 1 \): Since \( M(n) = M(1) = 1 \) and 
\[
2^n - 1 = 2^1 - 1 = 1,
\]
this case is correct.

**Assumption:** Suppose we already know that \( M(k) = 2^k - 1 \), for an arbitrary, but fixed number \( k \geq 1 \).

**Case** \( n = k + 1 \): We have
\[
M(n) = M(k + 1) \\
= 2M(k) + 1 \quad \text{(by recursive identity)} \\
= 2[2^k - 1] + 1 \quad \text{(by assumption)} \\
= 2^{k+1} - 2 + 1 \quad \text{(by basic algebra)} \\
= 2^{k+1} - 1 \quad \text{(by basic algebra)} \\
= 2^n - 1
\]
so that this case is also correct, *under the above assumption*.

— Do these arguments amount to a proof?
• **Principle of Mathematical Induction:**

Let $P(n)$ be a *predicate* defined on the natural numbers and let $a$ be a fixed natural number. Suppose the following two statements are true:

1. $P(a)$ is true.

2. For all integers $k \geq a$, if $P(k)$ true then $P(k + 1)$ is true.

Then $P(n)$ is true for all natural numbers $n$ with $n \geq a$.

• Mathematical induction is a powerful proof technique closely related to recursion.
CONTRADICTION PROOFS

• If one wants to prove $F$,
  
  – one negate $F$ ($F$ is true iff $\neg F$ is false)
  
  – one applies inferences rules and strategy to generate a contradiction.

• **Example:** Given any integer $n$, if $n^2$ is even then $n$ is even.

  \textit{Proof:}

  Suppose $n^2$ is even and $n$ is odd.

  \[ n = 2k + 1 \] so \[ n^2 = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1 \] so $n^2$ is odd.

  Contradiction.
SATURATION PROOFS

• Saturation = Compilation

• Production of a saturated set of data (no data to add or remove) using inference rules and strategy [JA Robinson 1965, Bachmair Ganzinger 1994].

• Completion, Resolution, Paramodulation...
WHY USEFUL?

• Assist mathematicians
  – Moufang identities in alternative rings - **SBREVE** [Anantharaman, Hsiang,90]
  – Robbins algebras - **EQP** [McCune,97]

• Software verification
  – Critical applications - **PVS**
  – ATM conformity - **COQ**
  – Authentication protocols - **Isabelle**, **ELAN**, **DATA**

• Hardware verification
  – Pentium chip - **RRL** [Kapur, Subramaniam,98]

• Artificial intelligence (knowledge representation, expert systems)
• Software engineering (verification)

• Programming language paradigm (logic programming, functional programming, rewrite programming)
**Horror stories**

- “the art of programming is the art of organizing complexity”,

- “we must organize the computations in such a way that our limited powers are sufficient to guarantee that the computation will establish the right effect”, Dijkstra.

- INTEL processor bugs galore.

• The Millenium bug.

• The Ariane 5 satellite launcher malfunction was caused by a faulty software exception floating point to 16-bit integer conversion.

• Eighteen errors were detected during the 10-day flight of Apollo 14. From G. J. Myers, Software Reliability: Principles & Practice, p. 25.

UNDECIDABLE PROBLEMS

- Alan Turing (1930) proved the incompleteness of algorithmic reasoning.

- Turing posed the following problem (since called the Halting Problem):

  It is impossible to write a computer program $H$ that is passed a single parameter $P$, where $P$ is itself a program, such that $H$ always returns true if $P$ halts and false if $P$ does not.
• **Ideas of the proof:**

  – Assume we could write such a program $H$. Then let $H(X)$ be a library routine callable by any other program.

  – Consider the following program:

    ```
    program Contrary;
    #include H;
    begin
      if H(Contrary)
        then
          while true
            \{/*Infinite Loop */}
        else
          halt;
    end;
    ```

  – If $H$ claims that the program $P$ passed as parameter terminates, it deliberately enters an infinite loop.

  – If $H$ claims that the program $P$ passed as parameter will not halt, it immediately halts.
**THE WORD PROBLEM**

- Given a set of equalities $E$ and a goal $s \approx t$, is $s \approx t$ true in all models of $E$?

  **Word problem is undecidable.**

- **Goal:** Find a method which **OFTEN** gives a decision procedure.
  
  - **Solution 1:** Generate all equalities implied by $E$ using inference rules and strategies.
    
    *Disadvantage of solution 1:* Generation of all equalities may not halt, not goal oriented.
  
  - **Solution 2:** Apply equalities to goal until both sides are the same.
    
    *Benefit of solution 2:* Goal oriented.
    
    *Disadvantage of solution 2:* May not halt if goal is false.
• An ordering $\succ$ is a binary relation:
  
  – Reflexive: $x \succ x$
  
  – Transitive: $x \succ y$ and $y \succ z \Rightarrow x \succ z$

• Properties:
  
  – Well-founded: No infinite sequence $x_1 \succ x_2 \succ \ldots x_n \succ \ldots$
  
  – Monotonic: $s \succ t \Rightarrow u[s] \succ u[t]$
  
  – Total: If $s \neq t$, then either $s \succ t$ or $t \succ s$. 
A simple rewrite system

\[
\begin{align*}
\bullet\bullet & \rightarrow \circ \\
\circ\circ & \rightarrow \circ \\
\bullet\circ & \rightarrow \bullet \\
\circ\bullet & \rightarrow \bullet \\
\circ\circ\circ\circ & \rightarrow \circ\circ\circ\circ \rightarrow \bullet
\end{align*}
\]
Analysing the different cases

Disjoint redexes:

\[ \ldots \underline{\times} \underline{\times} \ldots \underline{\times} \underline{\times} \ldots \]
\[ \ldots \underline{\times} \underline{\times} \ldots \underline{\times} \underline{\times} \ldots \]
\[ \ldots \underline{\times} \underline{\times} \ldots \underline{\times} \underline{\times} \ldots \]

is the same as:

\[ \ldots \underline{\times} \underline{\times} \ldots \underline{\times} \underline{\times} \ldots \]
\[ \ldots \underline{\times} \underline{\times} \ldots \underline{\times} \underline{\times} \ldots \]
\[ \ldots \underline{\times} \underline{\times} \ldots \underline{\times} \underline{\times} \ldots \]
No disjoint redexes (central black):

\[
\begin{array}{c|c}
\cdots \bullet \bullet \cdots & \cdots \bullet \bullet \cdots \\
\cdots \bullet \cdots & \cdots \bullet \cdots \\
\cdots \circ \cdots & \cdots \circ \cdots \\
\end{array}
\]

but

\[
\begin{array}{c|c}
\cdots \circ \bullet \cdots & \cdots \bullet \bullet \cdots \\
\cdots \circ \bullet \cdots & \cdots \bullet \bullet \cdots \\
\cdots \circ \circ \cdots & \cdots \circ \circ \cdots \\
\end{array}
\]

or

\[
\begin{array}{c|c}
\cdots \circ \bullet \circ \cdots & \cdots \bullet \bullet \circ \cdots \\
\cdots \circ \bullet \circ \cdots & \cdots \bullet \bullet \circ \cdots \\
\cdots \circ \bullet \cdots & \cdots \circ \bullet \cdots \\
\end{array}
\]

but

\[
\begin{array}{c|c}
\cdots \circ \bullet \circ \cdots & \cdots \bullet \bullet \circ \cdots \\
\cdots \circ \bullet \circ \cdots & \cdots \bullet \bullet \circ \cdots \\
\cdots \circ \bullet \cdots & \cdots \circ \bullet \cdots \\
\end{array}
\]
No disjoint redexes (central white):

but

or

but
Thus in all the cases:

![Diagram](attachment:image.png)

but what about:

![Diagram](attachment:image.png)
REWRITING

• Using equalities with an orientation

If $s \approx t$ and $s \succ t$, then we write $s \rightarrow t$.

$\rightarrow^*$ is the transitive closure of $\rightarrow$.

• **Confluence:** If $s \rightarrow^* t$ and $s \rightarrow^* u$, then there exists $v$ such that $t \rightarrow^* v$ and $u \rightarrow^* v$.

• **Termination:** There exists no infinite sequence $s_1 \rightarrow \ldots \rightarrow s_n \rightarrow \ldots$

  Undecidable property.

• **Convergence:** If $\rightarrow$ is confluent and terminating, then $\rightarrow$ is convergent.
A substitution is a mapping from the set of variables to the set of terms.

Example: \( \{x \mapsto f(b), \ y \mapsto g(z)\} \).

Extended to terms as a morphism.

Solving \( f(x, a) = ? f(b, y) \).

Solution: Substitution \( \sigma = \{x \mapsto b, \ y \mapsto a\} \)

Because \( \sigma(f(x, a)) = \sigma(f(b, y)) \)

Is \( f(x, a) = ? f(b, y) \) satisfiable? Yes.

Solving \( f(x, a) = ? f(b, y) \).

Solving/Satisfiability
SYNTACTIC UNIFICATION

• Expressed using rewriting rules.

• Rewriting is a paradigm.
  Language: ELAN [VKKBМ, 95*]

• Delete:

\[ P \land s =? s \rightarrow P \]

Decompose

\[ P \land f(s_1, ..., s_n) =? f(t_1, ..., t_n) \rightarrow P \land s_1 =? t_1 \land ... \land s_n =? t_n \]

Conflict

\[ P \land f(s_1, ..., s_n) =? g(t_1, ..., t_n) \rightarrow Clash \]
  if \( f \neq g \)

+ Coalesce + Check* + Merge + Check + Eliminate
COMPLETION [Knuth-Bendix 1970]

- Solves the word problem

- Based on **Rewriting** and use an **Ordering**. Use equalities with an orientation: If $s \preceq t$ and $s \succ t$, then we write $s \rightarrow t$.

- Main inference rule: **Critical Pair inference rule**.

- **Compilation** of $E$ using a well-founded, monotonic and total ordering

  $$E \Rightarrow E_\infty$$

  $E_\infty$ is the same equational theory as $E$ but $E_\infty$ is confluent.

- **Rewriting proof**: $s \approx t$ is true if there exist a term $u$ such that $s \rightarrow^* u$ and $t \rightarrow^* u$.

- **Completeness**: $Gr(E_\infty)$ is convergent ($Gr$ stands for ground).
• Critical Pair (CP)

\[
\begin{align*}
\frac{u[s'] \approx v}{\sigma(u[t] \approx v)} & \quad s \approx t
\end{align*}
\]

- \(s'\) is not a variable,
- \(\sigma = \text{mgu}(s = ? s')\) (unification)

• Examples:

- \[g(a) \rightarrow b \quad a \rightarrow c\]
  \[\frac{g(c) \rightarrow b}{g(c) \rightarrow b}\]

- \[g(x) \rightarrow f(x) \quad g(a) \rightarrow b\]
  \[\frac{f(a) \rightarrow b}{f(a) \rightarrow b}\]

Substitution: \(\sigma = \{x \mapsto a\}\).

Ordering: \(g \succ f \succ a \succ b \succ c\)
EXAMPLE

Group

• An additive group $G$ is defined by the set of equalities:

\[
\begin{align*}
x + e & \approx x \\
x + (y + z) & \approx (x + y) + z \\
x + i(x) & \approx e
\end{align*}
\]

• How to check that:

\[
i(x + y) \approx i(y) + i(x)
\]
• An equivalent deterministic term rewrite system $G_\infty$:

\[
\begin{align*}
x + e & \rightarrow x \\
e + x & \rightarrow x \\
x + (y + z) & \rightarrow (x + y) + z \\
x + i(x) & \rightarrow e \\
i(x) + x & \rightarrow e \\
i(e) & \rightarrow e \\
(y + i(x)) + x & \rightarrow y \\
(y + x) + i(x) & \rightarrow y \\
i(i(x)) & \rightarrow x \\
i(x + y) & \rightarrow i(y) + i(x)
\end{align*}
\]

• The proof of $i(x + y) \approx i(y) + i(x)$ is obvious.
CONTROL OF THE EXPANSION OF THE SEARCH SPACE: 3 WAYS

Parallelism

Constraints modulo

Simplificaiton Strategies
PARALLELISM

• Distribution of the search space
  – Processes ⇒ processors
  – Exchange of needed data
  – Synchronization

• Different possibilities of parallelization
  – Fine grain - term
    [Buendgen, Goebel, Kuechlin, 94]
    [CWD, Kirchner, Lynch, Scharff, 96]
  – Medium grain - equality/clause
    [ROO, 92]
  – Coarse grain - search space
    [DARES, 95] [DISCOUNT, 95] [AQUARIUS, 95]
Констриктанты

- Факторизация пространства поиска
- Представление большого количества информации

- $g(x) \approx a[x = ? b]$ представляет $g(b) \approx a$.
- $g(x) \approx a[f(x, a) \Rightarrow_A f(a, x)]$
  представляет бесконечное количество равенств

  $g(a) \approx a$ (Подстановка: $x \mapsto a$)
  $g(f(a, a)) \approx a$ (Подстановка: $x \mapsto f(a, a)$)
  $g(f(a, f(a, a))) \approx a$ (Подстановка: $x \mapsto f(a, f(a, a))$)
  $g(f(a, f(a, f(a, a)))) \approx a$ (Подстановка: $x \mapsto f(a, f(a, f(a, a))))$ ...

- Отдельное следование от вычисления

  [Dowek, Hardin, Kirchner, 98]
SIMPLIFICATION STRATEGIES

- Also called Contraction Strategies

- Remove data from the search space

- Reduce search space

- Crucial in the control of the expansion of the search space
COMBINATION
PARALLELISM - SIMPLIFICATION

- Difficulties due to parallelism

- Very dependent data (conflicts)
COMBINATION CONSTRAINTS - SIMPLIFICATION STRATEGIES

- Keep the major advantage of constraints
- Allowing as many simplifications as possible
- Concrete and understandable simplification strategies
- Complete simplification strategies
A fine-grained concurrent procedure (ground case)

[Kirchner,Lynch,Scharff,RTA96]
SOUR GRAPH BASIC COMPLETION

[Lynch, Strogova, 95]

• Equalities representing by a graph with maximal structure sharing
  - Vertex = Term
  - 
  
  \[ Edges = \begin{cases}
  \text{Subterm} & - S \\
  \text{Orientation} & - O \\
  \text{Unification} & - U \\
  \text{Rewriting} & - R
  \end{cases} \]

• Completion = Saturation of the graph by graph transformations
  - Seek edge patterns
GRAPH TRANSFORMATIONS

- **SUR**

\[
\begin{align*}
(u[s])_\omega & \rightarrow v \\
\quad s & \rightarrow t \\
\quad u[t]_\omega & \rightarrow v
\end{align*}
\]

\[
\begin{array}{cccc}
\text{u} & \text{t} & \text{u} & \text{S} \\
\text{s} & \text{s} & \text{s} & \text{s}
\end{array}
\]

\[
\begin{array}{cccc}
\text{S} & \text{R} & \text{U} & \Rightarrow \\
\text{s} & \text{s} & \text{s} & \text{s}
\end{array}
\]

- **RUR**

\[
\begin{align*}
\quad s & \rightarrow t \\
\quad s & \rightarrow t' \\
\quad t' & \rightarrow t
\end{align*}
\]

\[
\begin{array}{cccc}
\text{t'} & \text{t} & \text{t'} & \text{R} \\
\text{s} & \text{s} & \text{s} & \text{s}
\end{array}
\]

\[
\begin{array}{cccc}
\text{R} & \text{R} & \text{U} & \Rightarrow \\
\text{s} & \text{s} & \text{s} & \text{s}
\end{array}
\]

- **RUR-rhs**

\[
\begin{align*}
\quad s & \rightarrow t \\
\quad t & \rightarrow u \\
\quad s & \rightarrow u
\end{align*}
\]

\[
\begin{array}{cccc}
\text{s} & \text{u} & \text{s} & \text{R} \\
\text{t} & \text{t} & \text{t} & \text{t}
\end{array}
\]

\[
\begin{array}{cccc}
\text{R} & \text{R} & \text{U} & \Rightarrow \\
\text{t} & \text{t} & \text{t} & \text{t}
\end{array}
\]
EXAMPLE (ground case)

\[ E = \begin{cases} 
  f(g(a), g(a)) \approx h(g(a)) \\
  a \approx b 
\end{cases} \]

\[ f > g > h > a > b \]
TERMINATION

- Termination = Processes are idle + No messages are transiting [Dijskra, 83]

- Detection of the termination by a master process

  \[
  \text{State of the master process} = \begin{cases} 
  \text{list\_sent} \\
  \text{list\_receive} 
  \end{cases}
  \]

- Termination = list\_sent and list\_receive are empty.
RESULTS

• Asynchronous implementation

• No broadcast

• No global memory or global control

• No consistency check is needed

• No redundant work is performed

• Use of a simplification based strategy

Implementation in C and PVM
CONTRIBUTION 2

Parallelism

Constraints
   Empty theory

Simplification
   Strategies

BASIC COMPLETION WITH E-CYCLE SIMPLIFICATION

[Lynch, Scharff, AISC98, FI99]
BASIC COMPLETION

• Prohibition of some inferences

• Narrowing of the search space

• Constrained Critical Pair (CCP)

\[
\begin{align*}
    \underline{u[s']} & \approx v[\varphi_1] \quad s \approx t[\varphi_2] \\
    u[t] & \approx v[s \equiv ? s' \land \varphi_1 \land \varphi_2]
\end{align*}
\]

– \( s' \) is not a variable,

– \( (s \equiv ? s') \land \varphi_1 \land \varphi_2 \) is satisfiable.

• The combination Constraints-Simplification implies problems of completeness.
COMBINATION CONSTRAINTS-SIMPLIFICATION

\[
E = \begin{cases} 
    a \approx c \\
    f(g(x)) \approx g(x) \\
    f(g(a)) \approx b 
\end{cases}
\]

- \[
\frac{f(g(x)) \approx g(x)}{g(x) \approx b \[ x = ? a \]}
\]

- \[
\frac{f(g(a)) \approx b}{f(b) \approx b} 
\]

\[E_\infty = \{ a \approx c, \; f(g(x)) \approx g(x), \; g(x) \approx b \[ x = ? a \], \; f(b) \approx b \}\]

Standard Simplification of \( f(g(a)) \approx b \) \( \Rightarrow \) No proof of \( g(c) \approx b \).

INCOMPLETENESS DE CCP+SS
E-CYCLE SIMPLIFICATION

\[ E = \begin{cases} 
  g(x) \approx f(x), \\
  g(a) \approx b, \\
  h(f(a)) \approx b 
\end{cases} \]

- Dependency graph:

- An equality may not simplify its own ancestor (determined by cycle in graph).
## SIMPLIFICATIONS STRATEGIES

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Efficiency</th>
<th>Completeness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Simplification</td>
<td>+++</td>
<td>NO [Nieuwenhuis, Rubio, 92]</td>
</tr>
<tr>
<td>Basic Simplification</td>
<td>+</td>
<td>YES [Nieuwenhuis, Rubio, 92]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Bachmair, Ganzinger, Lynch, Snyder, 95]</td>
</tr>
<tr>
<td>E-cycle Simplification</td>
<td>++</td>
<td>YES</td>
</tr>
</tbody>
</table>
RESULTS

- A procedure: Basic Completion with E-cycle Simplification.

- Implementation in ELAN: ECC

- Abstract setting for simplification strategies
  - Counter-examples for the completeness of some Simplification strategies (Backward Simplification and Forward Simplification).
CONTRIBUTION 3

Parallelism

Constraints modulo an equational theory

Simplification Strategies

BASIC COMPLETION MODULO WITH SIMPLIFICATION

[Lynch, Scharff, LICS2001]
BASIC COMPLETION MODULO $AX$

[Nieuwenhuis, Rubio, Vigneron, Rusinovitch]

Basic Critical Pair Modulo

\[
\frac{u \approx v[\varphi_1]}{u''[t'']} \approx v''[s'' = ?_{AX} s' \land \varphi'_1 \land \varphi'_2]
\]

Advantages

• One equality is deduced

• Constraints modulo $AX$ (satisfiability test)


BASIC COMPLETION MODULO AND SIMPLIFICATION

- Basic Simplification not applicable
  - Constraints solving
    - AC: Doubly exponential [Domenjoud, 92] but satisfiability “only” NP-complete [Kapur, Narendran, 92].
    - A: infinitary but satisfiability is decidable [Makanin, 77].
  - Matching computations
  - No implementation

- E-cycle Simplification not applicable
  - Constraints solving

- Incomplete strategies [McCune]
RESULTS

- Basic Completion modulo

- with simplification

- uses inferences in the constraints

- presented concretely (enough to be implemented)

- does not solve or apply constraints

- involves inferences in the constraints

- completeness

- very original work
LINKED FURTHER WORKS AND OBJECTIVES

- Compatibility of parallelism and simplification strategies
- Validation of the developed techniques in well-known theorem provers
- Implementation of Basic Completion modulo with simplification
- Application of the techniques in verification, deductive databases and data mining
- ELAN Language
  - The definition: Theorem prover = Inferences rules + Strategies fits well in ELAN.
NEW TOPIC 1: CONGRUENCE CLOSURE

- For be useful for program verification a deductive system must be able to reason proficiency about equality.

Example: Array


\[ \Rightarrow \]


- Completion based methods often yield semi-decision procedures. Decision algorithms are too problem specific.

- Completion of ground rewrite systems is always less efficient than congruence closure.
• Congruence closure of a relation on a graph.

• New graph based congruence closure. The idea comes from SOUR Graphs.
NEW TOPIC 2: FUNCTIONAL PROGRAMMING

- **Function evaluation** (not assignment of variables) is the basic concept for a programming paradigm that has been implemented in **functional programming languages**.

- Examples: ML, SML, CAML, LISP, HASKELL

- The basic mode of computation in functional languages is the use of the **definition** and **application** of functions (explicit and recursive).

- The basic cycle of activities has three parts:
  - *read* input from the user,
  - *evaluate* it, and
  - *print* the computed value (or an error message).
fun f 0 = 0
| f(n) = f(n-1)+2*n-1;
val f = fn : int -> int

fun g n:int = n*n;
val g = fn : int -> int

fun length nil = 0
| length L = 1 + length(tl(L));
val length = fn : 'a list -> int

fun comp(x,y) = 6-(x+y);
val f = fn : int*int -> int

fun hanoitower(k,x,y) =
if (k=0 orelse x=y) then []
else if k=1 then [(x,y)]
else hanoitower(k-1,x,comp(x,y))
@ ((x,y)::hanoitower(k-1,comp(x,y),y));
val hanoitower = fn : int * int * int ->
(int * int) list
WHY FUNCTIONAL PROGRAMMING MATTERS?

• Software becomes more and more complex. It is important to structure it well. Structured software is:
  – easy to write
  – easy to debug
  – easy to reuse
• Modular software is generally accepted to be the key to successful software.
  – Divide-and-conquer
  – The ways in which the original problem can be divided up depends directly on the ways in which solutions can be “glued” together.
  – New “glues” are provided in functional programming (Examples: higher-order functions, lazy evaluation, polymorphism, abstract data type).

• High-level language for prototyping.
FEATURES

- The functional ascetics forbid themselves facilities which less pious programmers regard as standard.

- No re-assignment.

- Inference of type.

- No side-effects.
  
  - When a value is assigned it does not change during the execution of the program ⇒ **Property of referential transparency**.
  
  - No global variable or instance of an object.
• No explicit flow of control.

• Higher level than third generation languages.

• Construction of more reliable software $\Rightarrow$ Correctness.

  Proof of the correctness easiest than for imperative programs.
Lists - Reverse example

- fun reverse(L) = if (L=[]) then L
  else if (tl(L)=[]) then L
  else reverse(tl(L))@[hd(L)];

Prove by induction that the reverse function does what we want it to do.

- Basis case: \( L = \text{nil} \) or \( L = [x] \).
  If the list has zero or one element then it is its own reverse (handled by the base cases of the function).

- Induction hypothesis:
  Assume that reverse works for any list of length \( n \) \( (n \geq 0) \).

- We prove that reverse works for any list of length \( n + 1 \).
We label the elements of the list of length \( n + 1 \) as \([a_1, a_2, a_3, \ldots a_n, a_{n+1}]\).

Then \( hd(L) = a_1 \) and
\( tl(L) = [a_2, a_3, \ldots, a_n, a_{n+1}] \).

So \( reverse(L) \)
\( = reverse(tl(L))@[hd(L)] \) (definition of reverse)
\( = reverse([a_2, a_3, \ldots, a_n, a_{n+1}])@[a_1] \) (expanding \( tl(L) \) and \( hd(L) \))
\( = [a_{n+1}, a_n, \ldots, a_3, a_2]@[a_1] \) (by induction hypothesis) \( = [a_{n+1}, a_n, \ldots, a_3, a_2, a_1] \)
RESEARCH TOPICS

• Develop a real application.

• Interfacing ML with another language.

• ELAN and ML.
OTHER REFERENCES

• ELAN: http://www.elan.loria.fr/

• SML:
  http://cm.bell-labs.com/cm/cs/what/smlnj/


• Automated Deduction, Looking ahead, Donald W. Loveland, American association for artificial intelligence, 1999.