The Nuprl Proof Development System

Christoph Kreitz

Department of Computer Science, Cornell University
Ithaca, NY 14853

http://www.nuprl.org
The Nuprl Project at Cornell University

- Computational formal logics
  - Type Theory

- Proof & program development systems
  - The Nuprl Logical Programming Environment
  - Fast inference engines + proof search techniques
  - Natural language generation from formal mathematics
  - Program extraction + automated complexity analysis

- Application to reliable, high-performance networks
  - Assigning precise semantics to system software
  - Performance Optimizations
  - Assurance for reliability (verification)
  - Verified System Design

Secure software infrastructure
Nuprl’s Type Theory

- **Constructive higher-order logic**
  - Reasoning about types, elements, propositions, proofs, functions . . .

- **Functional programming language**
  - Similar to core ML: polymorphic, with partial recursive functions

- **Expressive data type system**
  - Function, product, disjoint union, $\Pi$- & $\Sigma$-types, atoms, void, top
  - Integers, lists, inductive types, universes
  - Propositions as types, equality type, subsets, subtyping, quotient types
  - (Dependent) intersection, union, records, modules

- **Open-ended**
  - new types can be added if needed

- **User-defined extensions** possible
The Nuprl Proof Development System

- **Beginnings in 1984**
  - Nuprl 1 (Symbolics): proof & program refinement in Type Theory
  - Nuprl 2: Unix Version

  - Constructive machine proofs for unsolved mathematical problems

- **Nuprl 4: System verification and optimization** (1993–2001)
  - Verification of logic synthesis tools & SCI cache coherency protocol
  - Optimization/verification of the Ensemble group communication system

- **Nuprl 5: Open distributed architecture** (2000– . . . )
  - Cooperating proof processes centered around persistent knowledge base
  - Asynchronous, concurrent, and external proof engines
  - Interactive digital libraries of formal algorithmic knowledge
Applications: Mathematics & Programming

- **Formalized mathematical theories**
  - Elementary number theory, real analysis, group theory
  - Discrete mathematics
  - General algebra
  - Finite and general automata
  - Basics of Turing machines
  - Formal mathematical textbook

http://www.nuprl.org/Nuprl4.2/Libraries/Welcome.html

- **Machine proof for unsolved problems**
  - Girard’s paradox
  - Higman’s Lemma

- **Algorithms and programming languages**
  - Synthesis of elementary algorithms: square-root, sorting, ...
  - Simple imperative programming
  - Programming semantics & complexity analysis
  - Type-theoretical semantics of large OCAML fragment
Applications: System Verification and Optimization

Secure software infrastructure

- Verification of a logic synthesis tool (Aagaard & Leeser 1993)
- Verification of the SCI cache coherency protocol (Howe 1996)
- Ensemble group communication toolkit
  - Optimization of application protocol stacks (by factor 3–10) (Kreitz, Hayden, Hickey, Liu, van Renessee 1999)
  - Verification of protocol layers (Bickford 1999)
  - Formal design of new adaptive protocols (Bickford, Kreitz, Liu, van Renessee 2001)
- MediaNet stream computation network
  - Validation of real-time schedules wrt. resource limitations (ongoing)
After more than 15 years . . .

• **Insights**
  - Type theory **expressive enough** to formalize today’s software systems
  - Formal optimization can significantly improve **practical performance**
  - Formal verification **reveals errors** even in well-explored designs
  - Formal design **reveals hidden assumptions** and **limitations** for use of software

• **Ingredients for success in applications. . .**
  - **Precise semantics** for implementation language of a system
  - **Formal models** of: application domain, system model, programming language
  - **Knowledge-based** formal reasoning tools
  - **Collaboration** between systems and formal reasoning groups
Purpose of this course

- Understand Nuprl’s theoretical foundation
- Understand features of the Nuprl proof development system
- Learn how to formalize mathematics and computer science

Additional material can be found at ....

http://www.nuprl.org
http://www.cs.cornell.edu/home/kreitz/Abstracts/02calculemus-nuprl.html
Introduction

1. NUPRL’s Type Theory
   - Distinguishing Features
   - Standard NUPRL Types

2. The NUPRL Proof Development System
   - Architecture and Feature Demonstration

3. Proof Automation in NUPRL
   - Tactics & Rewriting
   - Decision Procedures
   - External Proof Systems

4. Building Formal Theories
   - (Dependent) Records, Algebra, Abstract Data Types

5. Future Directions
Part I:

Nuprl’s Type Theory
The NuPRL Type Theory
An Extension of Martin-Löf Type Theory

● Foundation for computational mathematics
  - Higher-order logic + programming language + data type system
  - Focus on constructive reasoning
  - Reasoning about types, elements, and (extensional) equality . . .

● Open-ended, expressive type system
  - Function, product, disjoint union, Π- & Σ-types, atoms  \(\leadsto\) programming
  - Integers, lists, inductive types  \(\leadsto\) inductive definition
  - Propositions as types, equality type, void, top, universes  \(\leadsto\) logic
  - Subsets, subtyping, quotient types  \(\leadsto\) mathematics
  - (Dependent) intersection, union, records  \(\leadsto\) modules, program composition

  New types can/will be added as needed

● Self-contained
  - Based on “formalized intuition”, not on other theories
Distinguishing Features of Nuprl’s Type Theory

- **Uniform internal notation**
  - Independent display forms support flexible term display
  - Expressions defined independently of their types
  - No restriction on expressions that can be defined
  - Expressions in proofs must be typeable

- **Expressions defined independently of their types**
  - Y combinator
  - “total” functions

- **Semantics based on values of expressions**
  - Judgments state what is true
  - Equality is extensional

- **Refinement calculus**
  - Top-down sequent calculus
  - Proof expressions linked to inference rules
  - Computation rules

- **User-defined extensions possible**
  - User-defined expressions and inference rules

- Expressions in proofs must be typeable
  - “total” functions

- **Semantics based on values of expressions**
  - Judgments state what is true
  - Equality is extensional

- **Refinement calculus**
  - Top-down sequent calculus
  - Proof expressions linked to inference rules
  - Computation rules

- **User-defined extensions possible**
  - User-defined expressions and inference rules
**Syntax Issues**

- **Uniform notation:** \( \text{opid}\{p_i:F_i\}(x_{11},..,x_{m1}.t_1;\ldots;x_{1n},..,x_{mnn}.t_n) \)
  - Operator name \( \text{opid} \) listed in operator tables
  - Parameters \( p_i:F_i \) for base terms (variables, numbers, tokens...)
  - Sub-terms \( t_j \) may contain bound variables \( x_{1j},..,x_{mjj} \)
  - No syntactical distinction between types, members, propositions ...

- **Display forms** describe visual appearance of terms

<table>
<thead>
<tr>
<th>Internal Term Structure</th>
<th>Display Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable({x:v})( )</td>
<td>(x)</td>
</tr>
<tr>
<td>function({})(S ; x . T)</td>
<td>(x:S\rightarrow T)</td>
</tr>
<tr>
<td>function({})(S ; . T)</td>
<td>(S\rightarrow T)</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>lambda({})(x . t)</td>
<td>(\lambda x . t)</td>
</tr>
<tr>
<td>apply({})(f ; t)</td>
<td>(f t)</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
</tr>
</tbody>
</table>

\(\sim\) conventional notation, information hiding, auto-parenthesizing, aliases, ...
(Lazy) **evaluation** of expressions

- Identify **canonical expressions** (values)
- Identify **principal arguments** of non-canonical expressions
- Define reducible non-canonical expressions (**redex**)
- Define reduction steps in redex–contracta table

<table>
<thead>
<tr>
<th>canonical</th>
<th>non-canonical</th>
<th>Redex</th>
<th>Contractum</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow T$</td>
<td></td>
<td>$\lambda x. t$</td>
<td>$\lambda x. u$ $t \xrightarrow{\beta} u[t/x]$</td>
</tr>
</tbody>
</table>

**Judgments**: semantical truths about expressions

- 4 categories: Typehood ($T$ Type), Type Equality ($S=T$), Membership ($t \in T$), Member equality ($s=t$ in $T$)
- Semantics tables define judgments for values of expressions

$S_1 \rightarrow T_1 = S_2 \rightarrow T_2$ iff $S_1 = S_2$ and $T_1 = T_2$

$\lambda x_1. t_1 = \lambda x_2. t_2$ in $S \rightarrow T$ iff $S \rightarrow T$ Type and $t_1[s_1/x_1] = t_2[s_2/x_2]$ in $T$

for all $s_1, s_2$ with $s_1 = s_2 \in S$
Nuprl’s Proof Theory

- **Sequent**  
  \( x_1 : T_1, \ldots, x_n : T_n \vdash C \)  
  “If \( x_i \) are variables of type \( T_i \) then \( C \) has a (yet unknown) member \( t \)”
  - A judgment \( t \in T \) is represented as \( T \) \([\text{ext } t]\)  
  \( \leadsto \) proof term construction
  - Equality is represented as type \( s = t \in T \) \([\text{ext } \text{Ax}]\)  
  \( \leadsto \) propositions as types
  - Typehood represented by (cumulative) universes \( U_i \) \([\text{ext } T]\)

- **Refinement calculus**
  - Top-down decomposition of proof goal  
  \( \leadsto \) interactive proof development
  - Bottom-up construction of proof terms  
  \( \leadsto \) program extraction
  \[
  \Gamma \vdash S \rightarrow T \quad \text{ext} \lambda x. e \quad \text{by lambda-formation } x
  \]
  \[
  \Gamma, x : S \vdash T \quad \text{ext } e
  \]
  \[
  \Gamma \vdash S = S \in U_i \quad \text{ext } \text{Ax}
  \]
  - Computation rules  
  \( \leadsto \) program evaluation

About 8–10 inference rules for each Nuprl type
Executing a Formal Proof Step

**Theorem name**

**Status + position in proof**

**Hypothesis of main goal**

**Conclusion**

**Inference rule**

**First subgoal – status, conclusion**

**Second subgoal – status, new hypotheses**

**conclusion**

<table>
<thead>
<tr>
<th>THM intsqrt</th>
</tr>
</thead>
<tbody>
<tr>
<td># top 1</td>
</tr>
<tr>
<td>1. $x : \mathbb{N}$</td>
</tr>
<tr>
<td>$\vdash \exists y : \mathbb{N}. y^2 \leq x \land x &lt; (y+1)^2$</td>
</tr>
<tr>
<td>BY natE 1</td>
</tr>
<tr>
<td>1# $\vdash \exists y : \mathbb{N}. y^2 \leq 0 \land 0 &lt; (y+1)^2$</td>
</tr>
<tr>
<td>2# 2. $n : \mathbb{N}$</td>
</tr>
<tr>
<td>3. $0 &lt; n$</td>
</tr>
<tr>
<td>4. $v : \exists y : \mathbb{N}. y^2 \leq n - 1 \land n - 1 &lt; (y+1)^2$</td>
</tr>
<tr>
<td>$\vdash \exists y : \mathbb{N}. y^2 \leq n \land n &lt; (y+1)^2$</td>
</tr>
</tbody>
</table>
Methodology for building types

- **Syntax:**
  - Define canonical type
  - Define canonical members of the type
  - Define noncanonical expressions corresponding to the type

- **Semantics**
  - Introduce evaluation rules for non-canonical expressions
  - Define type equality judgment for the type
    - The typehood judgment is a special case of type equality
  - Define member equality judgment for canonical members
    - The membership judgment is a special case of member equality

  *Define judgments only in terms of the new expressions* \(\leadsto\) consistency

- **Proof Theory**
  - Introduce proof rules that are consistent with the semantics
Methodology for defining proof rules

- **Type Formation rules:**
  - When are two types equal? \((\text{type}\text{Equality})\) \(\Gamma \vdash S = T \in U_j\)
  - How to build the type? \((\text{type}\text{Formation})\) \(\Gamma \vdash U_j \ [\text{ext } T]\)

- **Canonical rules:**
  - When are two members equal? \((\text{member}\text{Equality})\) \(\Gamma \vdash s = t \in T\)
  - How to build members? \((\text{member}\text{Formation})\) \(\Gamma \vdash T \ [\text{ext } t]\)

- **Noncanonical rules:**
  - When does a term inhabit a type? \((\text{noncanonical}\text{Equality})\) \(\Gamma \vdash s = t \in T\)
  - How to use a variable of the type \((\text{type}\text{Elimination})\) \(\Gamma, x:S, \Delta \vdash T \ [\text{ext } t]\)

- **Computation rules:**
  - Reduction of redices in an equality \((\text{noncanonical}\text{Reduce}^*)\) \(\Gamma \vdash \text{redex} = t \in T\)

- **Special purpose rules**
Proof Rules for the Function Type

\[ \Gamma \vdash \mathbf{U}_j \text{ ext } x : S \rightarrow T \]
by dependent function Formation \( x \ S \)
\[ \Gamma \vdash S \in \mathbf{U}_j \text{ ext } \text{Ax}_j \]
\[ \Gamma, x : S \vdash \mathbf{U}_j \text{ ext } T \]

\[ \Gamma \vdash \lambda x_1.t_1 = \lambda x_2.t_2 \in x : S \rightarrow T \text{ ext } \text{Ax}_j \]
by lambda equality \( x \ S \)
\[ \Gamma, x' : S \vdash t_1[x'/x_1] = t_2[x'/x_2] \in T[x'/x] \text{ ext } \text{Ax}_j \]
\[ \Gamma \vdash S \in \mathbf{U}_j \text{ ext } \text{Ax}_j \]

\[ \Gamma \vdash f_1.t_1 = f_2.t_2 \in T[t_1/x] \text{ ext } \text{Ax}_j \]
by apply equality \( x : S \rightarrow T \)
\[ \Gamma \vdash f_1 = f_2 \in x : S \rightarrow T \text{ ext } \text{Ax}_j \]
\[ \Gamma \vdash t_1 = t_2 \in S \text{ ext } \text{Ax}_j \]

\[ \Gamma \vdash (\lambda x.t)s = t_2 \in T \text{ ext } \text{Ax}_j \]
by apply reduce
\[ \Gamma \vdash t[s/x] = t_2 \in T \text{ ext } \text{Ax}_j \]

\[ \Gamma \vdash x_1 : S_1 \rightarrow T_1 = x_2 : S_2 \rightarrow T_2 \in \mathbf{U}_j \text{ ext } \text{Ax}_j \]
by function equality \( x \)
\[ \Gamma \vdash S_1 = S_2 \in \mathbf{U}_j \text{ ext } \text{Ax}_j \]
\[ \Gamma, x : S \vdash T_1[x/x_1] = T_2[x/x_2] \in \mathbf{U}_j \text{ ext } \text{Ax}_j \]

\[ \Gamma \vdash x : S \rightarrow T \text{ ext } \lambda x'.t \]
by lambda formation \( x' \)
\[ \Gamma, x' : S \vdash T[x'/x] \text{ ext } t \]
\[ \Gamma \vdash S \in \mathbf{U}_j \text{ ext } \text{Ax}_j \]

\[ \Gamma, f : x : S \rightarrow T, \Delta \vdash C \text{ ext } t[fs, Ax/y, z] \]
by dependent function elimination \( i \ s \ y \ z \)
\[ \Gamma, f : x : S \rightarrow T, \Delta \vdash s \in S \text{ ext } \text{Ax}_j \]
\[ \Gamma, f : x : S \rightarrow T, y : T[s/x], z : y = f s \in T[s/x], \Delta \vdash C \text{ ext } t \]

\[ \Gamma \vdash f_1 = f_2 \in x : S \rightarrow T \text{ ext } t \]
by function extensionality \( j \ x_1 : S_1 \rightarrow T_1 \ x_2 : S_2 \rightarrow T_2 \ x' \)
\[ \Gamma, x' : S \vdash f_1 x' = f_2 x' \in T[x'/x] \text{ ext } t \]
\[ \Gamma \vdash S \in \mathbf{U}_j \text{ ext } \text{Ax}_j \]
\[ \Gamma \vdash f_1 \in x_1 : S_1 \rightarrow T_1 \text{ ext } \text{Ax}_j \]
\[ \Gamma \vdash f_2 \in x_2 : S_2 \rightarrow T_2 \text{ ext } \text{Ax}_j \]

Note: \( e = e \in T \) is usually abbreviated by \( e \in T \)
User-defined Extensions

• Conservative extension of the formal language
  
  = Abstraction:  \textit{new-opid}\{\textit{parms}\}(\textit{sub-terms}) \equiv expr[\textit{parms}, \textit{sub-terms}]

  e.g. \textit{exists}\{}(T; x.A[x]) \equiv x:T \times A[x]

  + Display Form for newly defined term

  e.g. \exists x:T. A[x] \equiv \textit{exists}\{}(T; x.A[x])

Library contains many standard extensions of Type Theory
  
  e.g. Intuitionistic logic, Number Theory, List Theory, Algebra, ...

• Tactics: User-defined inference rules

  – Meta-level programs built using basic inference rules and existing tactics
  
  – May include meta-level analysis of the goal to \textit{find} a proof
  
  – Always result in a valid proof

Library contains many standard tactics and proof search procedures
# Standard Nuprl Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Space</td>
<td>$S \rightarrow T$, $x : S \rightarrow T$</td>
<td>$\lambda x.t$, $f.t$</td>
</tr>
<tr>
<td>Product Space</td>
<td>$S \times T$, $x : S \times T$</td>
<td>$\langle s, t \rangle$, let $\langle x, y \rangle = e$ in $u$</td>
</tr>
<tr>
<td>Disjoint Union</td>
<td>$S + T$</td>
<td>$\text{inl}(s)$, $\text{inr}(t)$, case $e$ of $\text{inl}(x) \mapsto u$</td>
</tr>
<tr>
<td>Universes</td>
<td>$U_j$</td>
<td>$-$ types of level $j$ $-$</td>
</tr>
<tr>
<td>Equality</td>
<td>$s = t \in T$</td>
<td>$\text{Ax}$</td>
</tr>
<tr>
<td>Empty Type</td>
<td>Void</td>
<td>any($x$), $-$ no members $-$</td>
</tr>
<tr>
<td>Atoms</td>
<td>Atom</td>
<td>&quot;token&quot;, if $a=b$ then $s$ else $t$</td>
</tr>
<tr>
<td>Numbers</td>
<td>$\mathbb{Z}$</td>
<td>$0, 1, -1, 2, -2, \ldots$ $s+t$, $s-t$, $s*t$, $s \div t$, $s \text{ rem } t$, $\text{if } a=b \text{ then } s \text{ else } t$, $\text{if } i&lt;j \text{ then } s \text{ else } t$</td>
</tr>
<tr>
<td></td>
<td>$i&lt;j$</td>
<td>$\text{Ax}$</td>
</tr>
<tr>
<td>Lists</td>
<td>$S \text{ list}$</td>
<td>$[]$, $t :: \text{list}$, rec-case $L$ of $[] \mapsto \text{base}$</td>
</tr>
<tr>
<td>Inductive Types</td>
<td>rectype $X = T[X]$</td>
<td>let$^*$ $f(x) = t$ in $f(e)$, $-$ members defined by $T[X]$ $-$</td>
</tr>
<tr>
<td>Subset</td>
<td>${ x : S \mid P[x] }$</td>
<td>$-$ some members of $S$ $-$</td>
</tr>
<tr>
<td>Intersection</td>
<td>$\cap x : S \cdot T[x]$</td>
<td>$-$ members that occur in all $T[x]$ $-$</td>
</tr>
<tr>
<td></td>
<td>$x : S \cap T[x]$</td>
<td>$-$ members $x$ that occur $S$ and $T[x]$ $-$</td>
</tr>
<tr>
<td>Union</td>
<td>$\cup x : S \cdot T[x]$</td>
<td>$-$ members that occur in some $T[x]$, tricky equality$-$</td>
</tr>
<tr>
<td>Quotient</td>
<td>$x, y : S // E[x, y]$</td>
<td>$-$ members of $S$, new equality $-$</td>
</tr>
<tr>
<td>Very Dep. Functions</td>
<td>${ f \mid x : S \rightarrow T[f, x] }$</td>
<td></td>
</tr>
<tr>
<td>Squiggle Equality</td>
<td>$s \sim t$</td>
<td>$-$ a &quot;simpler&quot; equality</td>
</tr>
</tbody>
</table>
**Functions: Basic Programming Concepts**

**Syntax:**
- Canonical: \( S \to T, \lambda x . e \)
- Noncanonical: \( e_1 e_2 \)

**Evaluation:**
\[
\lambda x . u \ x \xrightarrow{\beta} u[t/x]
\]

**Semantics:**
- \( S \to T \) is a type if \( S \) and \( T \) are.
- \( \lambda x_1 . e_1 = \lambda x_2 . e_2 \) in \( S \to T \) if \( S \to T \) type and \( e_1[s_1/x_1] = e_2[s_2/x_2] \) in \( T \) for all \( s_1, s_2 \) with \( s_1 = s_2 \in S \)

**Proof System:** — see above —
Cartesian Products: Building Data Structures

Syntax:
- Canonical: \( S \times T, \langle e_1, e_2 \rangle \)
- Noncanonical: \( \text{let} \ (x, y) = e \text{ in } u \)

Evaluation:
\[
\text{let } (x, y) = \langle e_1, e_2 \rangle \text{ in } u \xrightarrow{\beta} u[e_1, e_2 / x, y]
\]

Semantics:
- \( S \times T \) is a type if \( S \) and \( T \) are
- \( \langle e_1, e_2 \rangle = \langle e_1', e_2' \rangle \) in \( S \times T \) if \( S \times T \) type, \( e_1 = e_1' \) in \( S \), and \( e_2 = e_2' \) in \( T \)

Library Concepts: \( e.1, e.2 \)
**Disjoint Union: Case Distinctions**

**Syntax:**
- **Canonical:** $S+T$, $\text{inl}(e)$, $\text{inr}(e)$
- **Noncanonical:** case $e$ of $\text{inl}(x) \mapsto u$ | $\text{inr}(y) \mapsto v$

**Evaluation:**
- case $\boxed{\text{inl}(e')}$ of $\text{inl}(x) \mapsto u$ | $\text{inr}(y) \mapsto v$ $\xrightarrow{\beta} u[e' / x]$
- case $\boxed{\text{inr}(e')}$ of $\text{inl}(x) \mapsto u$ | $\text{inr}(y) \mapsto v$ $\xrightarrow{\beta} v[e' / y]$

**Semantics:**
- $S+T$ is a type if $S$ and $T$ are
- $\text{inl}(e) = \text{inl}(e')$ in $S+T$ if $S+T$ type, $e = e'$ in $S$
- $\text{inr}(e) = \text{inr}(e')$ in $S+T$ if $S+T$ type, $e = e'$ in $T$

**Library Concepts:**
### The Curry-Howard Isomorphism, Formally

**Propositions are represented as types**

<table>
<thead>
<tr>
<th>Proposition</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \land Q$</td>
<td>$P \times Q$</td>
</tr>
<tr>
<td>$P \lor Q$</td>
<td>$P + Q$</td>
</tr>
<tr>
<td>$P \Rightarrow Q$</td>
<td>$P \rightarrow Q$</td>
</tr>
<tr>
<td>$\neg P$</td>
<td>$P \rightarrow \text{Void}$</td>
</tr>
<tr>
<td>$\exists x : T . P[x]$</td>
<td>$x : T \times P[x]$</td>
</tr>
<tr>
<td>$\forall x : T . P[x]$</td>
<td>$x : T \rightarrow P[x]$</td>
</tr>
</tbody>
</table>

Need an **empty type** to represent “falsehood”

Need **dependent types** to represent quantifiers
Empty Type

Syntax:

Canonical: Void – no canonical elements –

Noncanonical: any(e)

Evaluation: – no reduction rules –

Semantics:

· Void is a type

· e = e’ in Void never holds

Library Concepts: ——

Warning: rules for Void allows proving semantical nonsense like

x : Void ⊢ 0=1 ∈ 2 or ⊢ Void → 2 type
**Singleton Type**

**Syntax:**
- Canonical: Unit, Ax
- Noncanonical: *no noncanonical expressions*

**Evaluation:** *no reduction rules*

**Semantics:**
- Unit is a type
- Ax = Ax in Unit

**Library Concepts:**

Defined type in NUPRL, see the library theory `core_1` for further details
Dependent types

- Allow representing **logical quantifiers** as type constructs

- Allow **typing functions** like \( \lambda x. \text{if } x=0 \text{ then } \lambda x.x \text{ else } \lambda x,y.x \)

- Allow expressing **mathematical concepts** such as finite automata
  - \((Q,\Sigma,q_0,\delta,F)\), where \(q_0 \in Q\), \(\delta:Q \times \Sigma \rightarrow Q\), \(F \subseteq Q\).

- Allow representing **dependent structures in programming languages**
  - Record types \([f_1:T_1; \ldots; f_n:T_n]\)
  - Variant records
    - type date = January of 1..31 | February of 1..28 | ...

- **Nuprl had them from the beginning**
  - ...as did Coq, Alf, ...
  - Other systems have recently adopted them (PVS, SPECWARE, ...)

The Nuprl Proof Development System

I. Type Theory: Standard Nuprl types
Subsumes independent function type
∀ generalizes ⇒

Syntax:
Canonical: \( x:S\rightarrow T, \lambda x.e \)
Noncanonical: \( e_1 e_2 \)

Evaluation:
\[
\lambda x.u \ t \xrightarrow{\beta} u[t/x]
\]

Semantics:
\[\cdot \ x:S\rightarrow T \text{ is a type if } S \text{ is a type and } T[e/x] \text{ is a type for all } e \text{ in } S\]
\[\cdot \ \lambda x_1.e_1 = \lambda x_2.e_2 \text{ in } x:S\rightarrow T \text{ if } x:S\rightarrow T \text{ type and }\]
\[e_1[s_1/x_1]=e_2[s_2/x_2] \text{ in } T[s_1/x] \text{ for all } s_1, s_2 \text{ with } s_1=s_2 \in S\]
Dependent Products (Σ-Types)

Subsumes (independent) cartesian product
\[ \exists \text{ generalizes } \wedge \]

Syntax:
Canonical: \( x : S \times T, \langle e_1, e_2 \rangle \)
Noncanonical: let \( \langle x, y \rangle = e \) in \( u \)

Evaluation:
let \( \langle x, y \rangle = [e_1, e_2] \) in \( u \) \( \xrightarrow{\beta} u[e_1, e_2 / x, y] \)

Semantics:
- \( x : S \times T \) is a type if \( S \) is a type and \( T[e/x] \) is a type for all \( e \) in \( S \)
- \( \langle e_1, e_2 \rangle = \langle e_1', e_2' \rangle \) in \( x : S \times T \) if \( x : S \times T \) type,
  \[ e_1 = e_1' \text{ in } S, \text{ and } e_2 = e_2' \text{ in } T[e_1/x] \]
Well-formedness Issues

- Formation rules for dependent type require checking
  \[ x' : S \vdash T[x'/x] \text{ type} \]
  - \( T \) is a function from \( S \) to types that could involve complex computations,
    e.g. \( T[i] \equiv \text{if } M_i(i) \text{ halts then } \text{IN} \text{ else } \text{Void} \)

  Well-formedness is undecidable
  in (extensional) theories with dependent types

- Programming languages must restrict dependencies
  - Only allow finite dependencies \( \leadsto \text{decidable typechecking} \)

- Typechecking in Nuprl cannot be fully automated
  - Typechecking becomes part of the proof process \( \leadsto \text{heuristic typechecking} \)

- Additional problem
  - What is the type of a function from \( \text{IN} \) to types? \( \leadsto \text{Girard Paradox} \)
Universes

- **Syntactical representation of typehood**
  - \( T \) type expressed as \( T \in \mathbb{U} \) — \( S = T \) expressed as \( S = T \in \mathbb{U} \)

- **Universes are object-level terms**
  - \( \mathbb{U} \) is a type and a universe
  - Girard’s Paradox: a theory with dependent types and \( \mathbb{U} \in \mathbb{U} \) is inconsistent
    \( \mapsto \) No single universe can capture the notion of typehood
  - Typehood \( \triangleq \) cumulative hierarchy of universes \( \mathbb{U} = \mathbb{U}_1 \subseteq \mathbb{U}_2 \subseteq \mathbb{U}_3 \subseteq \ldots \)

**Syntax:**

- Canonical: \( \mathbb{U}_j \)
- Noncanonical: —

**Semantics:**

- \( \mathbb{U}_j \) is a type for every positive integer \( j \)
- \( S = T \) in \( \mathbb{U}_j \) if \( \ldots \) mimic semantics for \( S = T \) as types\ldots
- \( \mathbb{U}_{j_1} = \mathbb{U}_{j_2} \) in \( \mathbb{U}_j \) if \( j_1 = j_2 < j \)
Integers: Basic Arithmetic

Syntax:
Canonical: \( \mathbb{Z} , \, 0, \, 1, \, -1, \, 2, \, -2, \ldots \)  \( i < j \), \( \text{Ax} \)
Noncanonical: \( \text{rec-case } i \text{ of } x < 0 \mapsto [f_x] \cdot s \mid 0 \mapsto b \mid y > 0 \mapsto [f_y] \cdot t \),
\( s + t \), \( s - t \), \( s \cdot t \), \( s \div t \), \( s \text{ rem } t \),
if \( i = j \) then \( s \) else \( t \), if \( i < j \) then \( s \) else \( t \),

Evaluation:
\[
\begin{align*}
\text{rec-case } [0] \text{ of } x < 0 \mapsto [f_x] \cdot s \mid 0 \mapsto b \mid y > 0 \mapsto [f_y] \cdot t & \xrightarrow{\beta} b \\
\text{rec-case } [i] \text{ of } x < 0 \mapsto [f_x] \cdot s \mid 0 \mapsto b \mid y > 0 \mapsto [f_y] \cdot t & \xrightarrow{\beta} t[i, \text{rec-case } i - 1 \text{ of } x < 0 \mapsto [f_x] \cdot s \mid 0 \mapsto b \mid y > 0 \mapsto [f_y] \cdot t / x, f_x] \\
\text{rec-case } [i] \text{ of } x < 0 \mapsto [f_x] \cdot s \mid 0 \mapsto b \mid y > 0 \mapsto [f_y] \cdot t & \xrightarrow{\beta} s[i, \text{rec-case } i + 1 \text{ of } x < 0 \mapsto [f_x] \cdot s \mid 0 \mapsto b \mid y > 0 \mapsto [f_y] \cdot t / x, f_x] \\
\end{align*}
\]
other noncanonical expressions evaluate as usual

Semantics:
\cdot \( \mathbb{Z} \) is a type
\cdot \( i < j \) is a type if \( i \in \mathbb{Z} \) and \( j \in \mathbb{Z} \)
\cdot \( i = i \) in \( \mathbb{Z} \) for all integer constants \( i \)
\cdot \( \text{Ax} = \text{Ax} \) in \( i < j \) if \( i, j \) are integers with \( i < j \)

Library Concepts: see the library theories int_1, int_2, and num_thy ...
**Lists: Basic Data Containers**

**Syntax:**
- Canonical: \( T \text{list}, \ [], \ e_1::e_2 \)
- Noncanonical: \( \text{rec-case } e \text{ of } [] \mapsto \text{base} \mid x::l \mapsto [fxl].up \)

**Evaluation:**
- \( \text{rec-case } [] \text{ of } [] \mapsto \text{base} \mid x::l \mapsto [fxl].up \xrightarrow{\beta} \text{base} \)
- \( \text{rec-case } e_1::e_2 \text{ of } [] \mapsto \text{base} \mid x::l \mapsto [fxl].up \xrightarrow{\beta} \text{base} \)
- \( \xrightarrow{\beta} \text{up}[e_1, e_2 \text{ rec-case } e_2 \text{ of } [] \mapsto \text{base} \mid x::l \mapsto [fxl].up / x, l, fxl} \)

**Semantics:**
- \( T \text{list} \) is a type if \( T \) is
- \( [] = [] \) in \( T \text{list} \) if \( T \text{list} \) is a type
- \( e_1::e_2 = e_1’::e_2’ \) in \( T \text{list} \) if \( T \text{list} \) type, \( e_1 = e_1’ \) in \( T \), and \( e_2 = e_2’ \) in \( T \text{list} \)

**Library Concepts:**
- \( \text{hd}(e), \text{tl}(e), e_1@e_2, \text{length}(e), \text{map}(f;e), \text{rev}(e), e[i], e[i..j^-], \ldots \)
Inductive Types: Recursive Definition

- **Representation of recursively defined data types**
  - Recursive type definition $X = T[X]$  
  - Canonical elements determined by unrolling $T[X]$  
  - Noncanonical form for inductive evaluation of elements

- **Recursion must be well-founded**
  - Least fixed point semantics
  - $T[X]$ must contain a “base” case
  - $X$ must only occur positively in $T[X]$

- **Extensions possible**
  - Parameterized, simultaneous recursion  
    
    rectype $X_1(x_1) = T[X_1] \text{ and } \ldots \text{ } X_n(x_n) = T[X_n]$ select $X_i(a_i)$  
  - Co-inductive type inftype $X = T_X$: greatest fixed point semantics
  - Partial recursive functions $S \not\rightarrow T$: unrestricted recursive induction
**Inductive Types, formally**

**Syntax:**
- Canonical: \( \text{rectype } X = T_X \)
- Noncanonical: \( \text{let } f(x) = t \text{ in } f(e) \)

**Evaluation:**
\[
\text{let* } f(x) = t \text{ in } f(e) \quad \xrightarrow{\beta} \quad t[\lambda y.\text{let* } f(x) = t \text{ in } f(y), e / f, x]
\]

*Termination of \( \text{let* } f(x) = t \text{ in } f(e) \) requires \( e \text{ in rectype } X = T[X] \)*

**Semantics:**
- \( \text{rectype } X_1 = T_{X_1} = \text{rectype } X_2 = T_{X_2} \)
  \( \text{if } T_{X_1}[X/X_1] = T_{X_2}[X/X_2] \text{ for all types } X \)
- \( s = t \text{ in rectype } X = T_X \) \( \text{if } \text{rectype } X = T_X \text{ type } \) \( \text{and } \)
  \( s = t \text{ in } T_X[\text{rectype } X = T_X/X] \)
Subset Types: Hiding Computational Content

- **Representation of mathematical concept of subsets**
  - \( \{ x : S \mid T[x] \} \) formally similar to dependent product \( x : S \times T[x] \)
  - ...but ...
  - Members are elements of \( s \in S \), not pairs \( <s,t> \)
  - Only implicit evidence for \( T[s] \) but no explicit proof component

**Syntax:**

- Canonical: \( \{ x : S \mid T \} \), \( \{ S \mid T \} \)
- Noncanonical: —

**Semantics:**

- \( \{ x_1 : S_1 \mid T_1 \} = \{ x_2 : S_2 \mid T_2 \} \) if \( S_1 = S_2 \) and there are terms \( p_1, p_2 \) and a variable \( x \), which occurs neither in \( T_1 \) nor in \( T_2 \) such that
  
  \[
  p_1 \text{ in } \forall x : S_1. \ T_1[x/x_1] \Rightarrow T_2[x/x_2]
  \]
  
  and \( p_2 \text{ in } \forall x : S_1. \ T_2[x/x_2] \Rightarrow T_1[x/x_1]. \) (violates separation principle)

- \( s = t \) in \( \{ x : S \mid T \} \) if \( \{ x : S \mid T \} \) type,

  \[
  s=t \text{ in } S, \text{ and there is some } p \text{ in } T[s/x].
  \]
Proof rules must manage implicit information

- We “know” $T[s]$ if $s$ in $\{x:S \mid T\}$
- We cannot use the proof term for $T[s]$ computationally
- Proof term for $T[s]$ must be available in non-computational proof parts
- Some refinement rules generate hidden assumptions

\[
\Gamma, z: \{x:S \mid T\}, \Delta \vdash C \quad \text{ext} \ (\lambda y.t) \ z \\
\text{by setElimination} \ i \ y \ v
\]

\[
\Gamma, z: \{x:S \mid T\}, y:S, \llbracket v \rrbracket : T[y/x], \Delta[y/z] \vdash C[y/z] \quad \text{ext} \ t
\]

- Hidden assumptions made visible by refinement rules with extract term $Ax$
**Intersection Types: Polymorphism without parameters**

- **Represent mathematical concept of intersection**
  - $\cap x:S.T[x]$ formally similar to dependent functions $x:S\rightarrow T[x]$
  - ...but ...
  - Members are elements of all $T[s]$ with $s \in S$, not functions
  - “Range parameter” $s \in S$ only implicitly present

### Syntax:
- Canonical: $\cap x:S.T[x]$
- Noncanonical:

### Evaluation:
- —

### Semantics:
- $\cap x:S.T[x]$ is a type if $S$ is a type and $T[e/x]$ is a type for all $e$ in $S$
- $s = t$ in $\cap x:S.T[x]$ if $\cap x:S.T[x]$ type and $s = t$ in $T[e/x]$ for all $e$ in $S$
Quotient Types: User-Defined Equality

- Representation of equivalence classes
  - Members of \( x, y : T // E \) are elements of \( T \) (but \( x, y : T // E \not\in T \))
  - Equality \( s=t \) redefined as \( E[s, t/x, y] \)
  - \( E \) must be type of an equivalence relation

Syntax:
- Canonical: \( x, y : T // E \)
- Noncanonical: —

Semantics:
- \( x_1, y_1 : T_1 // E_1 = x_2, y_2 : T_2 // E_2 \) if \( T_1 = T_2 \) and there are terms \( p_1 p_2 r, s, t \) and variables \( x, y, z \), which occur neither in \( E_1 \) nor in \( E_2 \) such that
  - \( p_1 \) in \( \forall x : T_1. \forall y : T_1. E_1[x, y/x_1, y_1] \Rightarrow E_2[x, y/x_2, y_2] \),
  - \( p_2 \) in \( \forall x : T_1. \forall y : T_1. E_2[x, y/x_2, y_2] \Rightarrow E_1[x, y/x_1, y_1] \),
  - \( r \) in \( \forall x : T_1. E_1[x, x/x_1, y_1] \),
  - \( s \) in \( \forall x : T_1. \forall y : T_1. E_1[x, y/x_1, y_1] \Rightarrow E_1[y, x/x_1, y_1] \),
  - \( t \) in \( \forall x : T_1. \forall y : T_1. \forall z : T_1. E_1[x, y/x_1, y_1] \Rightarrow E_1[y, z/x_1, y_1] \Rightarrow E_1[x, z/x_1, y_1] \)
- \( s = t \) in \( x, y : T // E \) if \( x, y : T // E \) type, \( s \) in \( T \), \( t \) in \( T \), and there is some term \( p \) in \( E[s, t/x, y] \)
Proof rules must manage implicit information

- We “know” $E[s, t/x, y]$ if $s = t$ in $x, y : T // E$
- Proof term for $E[s, t/x, y]$ can only be used non-computationally
- Hidden assumptions generated by decomposing equalities in hypotheses

$$\Gamma, v : s = t \in x, y : T // E, \Delta \vdash C \quad \text{[ext } u\text{]}$$

by quotient\_equalityElimination $i \quad j \quad v'$

$$\Gamma, v : s = t \in x, y : T // E, [v'] : E[s, t/x, y], \Delta \vdash C \quad \text{[ext } u\text{]}$$

$$\Gamma, v : s = t \in x, y : T // E, \Delta \vdash E[s, t/x, y] \in U_j \text{ [Ax]}$$

User-predicates may require type-squashing

- $\downarrow P \equiv \{ x : \text{Top} \mid P \}$: reduce $P$ to it’s truth content
- Necessary if there is too much structure on $x, y : T // E$
**Dependent Intersection**

- **Intersection with self-reference**
  - \( x : S \cap T \) somewhat similar to dependent products \( x : S \times T[x] \)
    
    ...but ...
  - Members are elements \( s \in S \) with \( s \in T[s] \)  
    
    (“very dependent pairs”)

**Syntax:**
- Canonical: \( x : S \cap T \)
- Noncanonical: —

**Evaluation:** —

**Semantics:**

- \( x : S \cap T \) is a type if \( S \) is a type and \( T[e/x] \) is a type for all \( e \) in \( S \)
- \( s = t \) in \( x : S \cap T \) if \( x : S \cap T \) type, \( s = t \) in \( S \), and \( s = t \) in \( T[s] \)

**Useful for representing dependent records, ADT’s, objects, etc.**
Important Defined Types

- Integer ranges: \( \text{IN} \equiv \{i: \mathbb{Z} | 0 \leq i\} \), \( \{j...\} \equiv \{i: \mathbb{Z} | j \leq i\} \), \( \text{IN}^+ \equiv \{i: \mathbb{Z} | 0 < i\} \), \( \{...j\} \equiv \{i: \mathbb{Z} | i \leq j\} \)

- Logic: \( \forall \exists \land \lor \Rightarrow \neg \text{True False} \) (Curry-Howard isomorphism)

- Singleton type: \( \text{Unit} \equiv 0 \in \mathbb{Z} \)

- Boolean: \( \mathbb{B} \equiv \text{Unit} + \text{Unit} \), \( \uparrow b \equiv \text{if } b \text{ then True else False} \)

- Top type: \( \text{Top} \equiv \cap x: \text{Void}.\text{Void} \)

- Subtyping: \( S \sqsubseteq T \equiv \forall x: S. \ x \in T \)

- Type squashing: \( \downarrow P \equiv \{\text{True} \mid P\} \)

- Recursive functions: \( \Upsilon \equiv \lambda f. (\lambda x. f (x \ x)) (\lambda x. f (x \ x)) \)

- (Dependent) records \( \{x_1:T_1; \ x_2:T_2[x_1]; \ldots; \ x_n:T_n[x_1..x_{n-1}]\} \) (\( \rightarrow \) part IV)

See the standard theories of NUPRL 5 for further details
Part II:
The Nuprl System
Interactive proof development

- Supports program extraction and evaluation
- Proof automation through tactics & decision procedures
- Highly customizable: conservative language extensions, term display, ...
- Supports cooperation with other proof systems
System Architecture (Allen et. al, 2000)

- Collection of cooperating processes
  - Asynchronous, distributed & collaborative theorem proving

- Centered around a common knowledge base
  - Library of formal algorithmic knowledge
  - Persistent data base, version control, dependency tracking \( \leadsto \) accountability

- Connected to external systems
  - MetaPRL (fast rewriting, multiple logics) (Hickey & Nogin, 1999)
  - JProver (matrix-based intuitionistic theorem prover) (IJCAR 2001)

- Multiple user interfaces
  - Structure editor, web browser \( \leadsto \) collaborative proving

- Reflective system structure
  - System designed within the system’s library \( \leadsto \) customizability
**Initial Nuprl 5 Screen**

- **Navigator** for browsing and invoking editors
- **ML top loop** for entering meta-level commands
- 3 windows for library, refiner, and editor **Lisp** processes
Features of the Proof Development System

- Interactive proof editor

- Flexible definition mechanism

- Customizable term display

- Structure editor for terms

- Tactics & decision procedures

- Proof objects, program extraction

- Program evaluation

- Library mechanism
  - Large mathematical libraries & tactics collection

- Command interface: navigator + ML top loops

- Formal documentation mechanism
  \(\sim\) readable proofs
  \(\sim\) user-defined terms
  \(\sim\) flexible notation
  \(\sim\) no ambiguities
  \(\sim\) user-defined inferences
  \(\sim\) program synthesis
  \(\sim\) user-theories
  \(\sim\) LaTeX, HTML
Basic Navigator Operations

- Creating, copying, renaming, removing, printing objects, directories, and links
  - Objects will never be destroyed – only references to objects change

- Browsing and searching the library

- Invoking editors on objects

- Checking theories

- Importing and exporting theories

- Invoking operations on collections of objects
Invoke proof editor by opening an object of kind THM

State theorem as top goal, using structured term editor

Prove a goal by entering proof tactics and parameters after the BY

Proof editor refines goal and displays remaining subgoals
- Proof steps are immediately committed to library
- Proof engine may be invoked asynchronously

User can move into subgoal nodes if necessary

Proof editor may generate extract terms from complete proofs

\[
\forall A, B: \mathbb{P}. (((\neg A) \lor (\neg B)) \Rightarrow (\neg (A \land B)))
\]

\[
\begin{align*}
\text{BY D 0} \\
* 1 \\
1. A: \mathbb{P} \\
\vdash \forall B: \mathbb{P}. (((\neg A) \lor (\neg B)) \Rightarrow (\neg (A \land B))) \\
\text{BY Auto} \\
* 1 1 \\
2. B: \mathbb{P} \\
3. (\neg A) \lor (\neg B) \\
\vdash (\neg (A \land B)) \\
\text{BY D 0 THEN D 3 THEN Auto} \\
* 2 \\
\ldots \ldots \text{wf} \ldots \ldots \\
\mathbb{P} \in \mathbb{U}' \\
\text{BY Auto}
\]
The Structured Term Editor

Edit internal structure of terms while showing external display

- Invoke by opening or entering a term slot
- Entering \textit{opid} of term opens template
  - e.g. \texttt{\texttt{\_exists\_\_}} generates the template \texttt{\texttt{\exists[var]:[type].[prop]}}
  - [type] and prop] are new term slots, [var] is a text slot
- Users may navigate through term tree and edit subterms
  - Motion by mouse or \texttt{emacs}-like key combinations (\texttt{m-p}, \texttt{m-b}, \texttt{m-f}, \texttt{m-n})
  - Cutting and pasting of terms possible (\texttt{c-k}, \texttt{m-k}, \texttt{c-y})
  - Text oriented editing possible as well
  - Insert non-ASCII characters with \texttt{c-\#num}
- Internal structure can be made visible
  - Explode (\texttt{c-x ex}) and implode (\texttt{c-x im}) terms
  - Entirely new terms can be inserted by entering \texttt{opid\{parms\}(arity)}
- Terms have hyperlinks to abstractions and display forms
  - Use \texttt{c-x ab} / Mouse-Right and \texttt{c-x df} / Mouse-Middle
Creating Definitions

Define new terms in terms of existing ones

- Click the AddDef* button

Add def : [lhs] == [rhs]

- Insert a new term into the [lhs] slot

\( \exists x : T . P[x] \land (\forall y : T . P[y] \Rightarrow y = x \in T) \)

- Enter its definition into [rhs]

- All free variables of the new term must occur

- Edit the generated display form and wellformedness theorem
Modifying the Term Display

- Open display form object for the term
  - create a new one if necessary

```
<table>
<thead>
<tr>
<th>DISP exists_uni_df</th>
</tr>
</thead>
<tbody>
<tr>
<td>EdAlias exists_uni ::</td>
</tr>
<tr>
<td>exists_uni(&lt;T:T:<em>&gt;;&lt;x:var:</em>&gt;.&lt;P:P:*&gt;)</td>
</tr>
<tr>
<td>== exists_uni(&lt;T&gt;;&lt;x&gt;,&lt;P&gt;)</td>
</tr>
</tbody>
</table>
```

- Edit text on left hand side of ==
  - Special characters may be inserted, e.g. c–# 163 inserts ∃
  - Template slots may be moved or deleted (mark with m–p)
  - Slot description between colons may be modified
  - Precedences for use of parentheses may be described after last colon

```
<table>
<thead>
<tr>
<th>DISP exists_uni_df</th>
</tr>
</thead>
<tbody>
<tr>
<td>EdAlias exists_uni ::</td>
</tr>
<tr>
<td>∃!&lt;x:var:<em>&gt;:&lt;T:type:</em>&gt;. &lt;P:prop:*&gt;</td>
</tr>
<tr>
<td>== exists_uni(&lt;T&gt;;&lt;x&gt;,&lt;P&gt;)</td>
</tr>
</tbody>
</table>
```

- Add additional display forms for iteration and special cases
  - Iteration: instead of ∀x:T. ∀y:T. P display ∀x,y:T. P
  - Special cases: instead of x=y ∈ ℤ display x=y (delete the type slot)
Evaluation of Terms

- Invoke the term evaluator on a Nuprl term by entering `view_showc name term` into the editor ML top loop

<table>
<thead>
<tr>
<th>compute addition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute1* Compute5* Compute10* ComputeAll*</td>
</tr>
<tr>
<td>(3 * 4) - 5 + 6</td>
</tr>
</tbody>
</table>

- Click the buttons to perform one top-level reduction steps
  - Use c− to undo a step
Extracting Programs from Proofs

- Generate extract term of completed proof
  - Close proof editor with `c-z` instead of `c-q`

- Make extract term available for editing
  - Enter `_require_termof (ioid obid)` into the editor ML top loop
  - `obid` is abstract identifier of proof object
  - mark in navigator with left mouse and copy into top loop with `c-y`

- Open term evaluator on extract term
  - Enter `_view_show_co obid` into the editor ML top loop

---

<table>
<thead>
<tr>
<th>compute intsqrt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute1*</td>
</tr>
<tr>
<td>Compute5*</td>
</tr>
<tr>
<td>Compute10*</td>
</tr>
<tr>
<td>ComputeAll*</td>
</tr>
<tr>
<td>TERMOF{intsqrt:o, \v:1}</td>
</tr>
</tbody>
</table>

- Evaluate one step to see the extract
  - Edit term to supply arguments to a NUPRL function, if desired

Should be simplified in the future
Part III:

Proof Automation in Nuprl
Automating the Construction of Proofs

- **Tactics**: Programmed application of inference rules
  - Easy to implement, even by users
  - Flexible, guaranteed to be correct

- **Rewriting**: Replace terms by equivalent ones
  - Computational and definitional equality
  - Derived equivalences in lemmata and hypotheses

- **Decision Procedures**: Solve problems in narrow application domains
  - Translate proof goal into different problem domain
  - Use efficient algorithms for checking translated problems

- **Proof Search Procedures**: Compact representation of proof tree
  - “Unintuitive”, but efficient proof procedure
  - Only for “small” theories
  - Correct integration into interactive proof system?
Tactics: User-defined inference rules

- Meta-level programs built using
  - Basic inference rules
  - Predefined tacticals . . .
  - Meta-level analysis of the proof goal and its context
  - Large collection of standard tactics in the library

- May produce incomplete proofs
  \[\rightarrow\] User has to complete the proof by calling other tactics

- May not terminate
  \[\rightarrow\] User has to interrupt execution

but

Applying a tactic always results in a valid proof
**Basic Tactics**

### Subsume primitive inferences under a common name

**Hypothesis:** Prove \( \vdash C' \) where \( C' \) \( \alpha \)-equal to \( C \)

**Declaration:** Prove \( \vdash x \in T' \) where \( T' \) \( \alpha \)-equal to \( T \)

\[ \text{Variants: NthHyp } i, \text{ NthDecl } i \]

**D \( c \):** Decompose the outermost connective of clause \( c \)

**EqD \( c \):** Decompose immediate subterms of an equality in clause \( c \)

**MemD \( c \):** Decompose subterm of a membership term in clause \( c \)

\[ \text{Variants: EqCD, EqHD } i, \text{ MemCD, MemHD } i \]

**EqTypeD \( c \):** Decompose type subterm of an equality in clause \( c \)

**MemTypeD \( c \):** Decompose type subterm of a membership term in clause \( c \)

\[ \text{Variants: EqTypeCD, EqTypeHD } i, \text{ MemTypeCD, MemTypeHD } i \]

**Assert \( t \):** Assert (or cut) term \( t \) as last hypothesis

**Auto:** Apply trivial reasoning, decomposition, decision procedures ...

**Reduce \( c \):** Reduce all primitive redices in clause \( c \)
PARAMETERS IN TACTICS

- **Position of a hypothesis** to be used
  \( \text{NthHyp } [i] \)

- **Names for newly created variables**
  \( \text{New } [x] \) (D 0)

- **Type of some subterm in the goal**
  \( \text{With } [x:S \rightarrow T] \) (MemD 0)

- **Term to instantiate a variable**
  \( \text{With } [S] \) (D 0)

- **Selection from a number of alternatives**
  \( \text{Sel } [n] \) (D 0)

- **Universe level of a type**
  \( \text{At } [j] \) (D 0)

- **Dependency of a term instance** \( C[z] \)
  on a variable \( z \)
  \( \text{Using } [z,C] \) (D 0)
**Tacticals**

**Compose tactics into new ones**

- $tac_1$ **THEN** $tac_2$: Apply $tac_2$ to all subgoals created by $tac_1$
- $t$ **THENL** $[tac_1; \ldots; tac_n]$: Apply $tac_i$ to the $i$-th subgoal created by $t$
- $tac_1$ **THENA** $tac_2$: Apply $tac_2$ to all auxiliary subgoals created by $tac_1$
- $tac_1$ **THENW** $tac_2$: Apply $tac_2$ to all wf subgoals created by $tac_1$
- $tac_1$ **ORELSE** $tac_2$: Apply $tac_1$. If this fails apply $tac_2$ instead

**Try** $tac$: Apply $tac$. If this fails leave the proof unchanged

**Complete** $tac$: Apply $tac$ only if this completes the proof

**Progress** $tac$: Apply $tac$ only if that causes the goal to change

**Repeat** $tac$: Repeat $tac$ until it fails

**RepeatFor** $i$ $tac$: Repeat $tac$ exactly $i$ times

**AllHyps** $tac$: Try to apply $tac$ to all hypotheses

**OnSomHyp** $tac$: Apply $tac$ to the first possible hypotheses
Advanced Tactics

- **Induction**
  - NatInd $i$: standard natural-number induction on hypothesis $i$
  - IntInd, NSubsetInd, ListInd: induction on $\mathbb{Z}$, $\mathbb{N}$ subranges, lists
  - CompNatInd $i$: complete natural-number induction on hypothesis $i$

- **Case Analysis**
  - BoolCases $i$: case split over boolean variable in hypothesis $i$
  - Cases $[t_1;..;t_n]$: $n$-way case split over terms $t_i$
  - Decide $P$: case split over (decidable) proposition $P$ and its negation

- **Chaining**
  - InstHyp $[t_1;..;t_n]$: instantiate hypothesis $i$ with terms $t_1...t_n$
  - FHyp $i$ $[h_1;..;h_n]$: forward chain through hypothesis $i$
    matching its antecedents against any of the hypotheses $h_1...h_n$
  - BHyp $i$: backward chain through hypothesis $i$
    matching its consequent against the conclusion of the proof
  - Backchain be_names: backchain repeatedly through lemmas and hypotheses

Variants: InstLemma name $[t_1;..;t_n]$, FLemma name $[h_1;..;h_n]$, BLemma name.
Decision Procedures

- **Decide problems in narrow application domains**
  - Translate proof goal into different problem domain
  - Decide translated problem using efficient standard algorithms
  - Implement directly in NUPRL or connect as external proof tool

- **Currently available**
  - *ProveProp*: simple propositional reasoning
  - *Eq*: trivial equality reasoning (limited congruence closure algorithm)
  - *RelRST*: exploit properties of binary relations (find shortest path in relation graph)
  - *Arith*: standard, induction-free arithmetic
  - *SupInf*: solve linear inequalities over \( \mathbb{Z} \)
Arith: INDUCTION-FREE ARITHMETIC

- **Input sequent:** $H \vdash C_1 \lor \ldots \lor C_m$
  - $C_i$ is an arithmetic relation over $\mathbb{Z}$
    built from $<, \leq, >, \geq, =, \neq$, and $\neg$

- **Theory covered:**
  - ring axioms for $+$ and $*$
  - total order axioms of $<$
  - reflexivity, symmetry and transitivity of $=$
  - limited substitutivity

- **Proof procedure:**
  - Translate sequent into a directed graph
    whose edges are labeled with natural numbers
  - Check if the graph contains positive cycles

- **Implemented as NUPRL procedure (Lisp level)**

- **Integrated into the tactic Auto**
SupInf: LINEAR INEQUALITIES OVER $\mathbb{Z}$

- Adaptation of Bledsoe’s Sup-Inf method
  - Complete only for the rationals
  - Sound for integers

- Proof procedure:
  - Convert sequent into conjunction of terms $0 \leq e_i$
    where each $e_i$ is a linear expression over $\mathbb{Q}$ in variables $x_1 \ldots x_n$
  - Check if some assignment of values to the $x_j$ satisfies the conjunction
  - Determine upper and lower bounds for each variable in turn
  - Identify counter-examples if no assignment exists

- Implemented as Nuprl procedure (ML level)

- Integrated into the tactic Auto’
THM intsqrt

* top \( \forall n : \mathbb{N}. \exists r : \mathbb{N}. r^2 \leq n < (r+1)^2 \)

BY allR

* 1 1. \( n : \mathbb{N} \)
   \[ \vdash \exists r : \mathbb{N}. r^2 \leq n < (r+1)^2 \]
   BY NatInd 1

* 1 11 .....basecase.....
   \[ \exists r : \mathbb{N}. r^2 \leq 0 < (r+1)^2 \]
   BY With [0] (D 0) THEN Auto

* 1 12 .....upcase.....
   1. \( i : \mathbb{N} \)
   2. \( 0 < i \)
   3. \( r : \mathbb{N} \)
   4. \( r^2 \leq i-1 < (r+1)^2 \)
   \[ \vdash \exists r : \mathbb{N}. r^2 \leq i < (r+1)^2 \]
   BY Decide \( (r+1)^2 \leq i \) THENW Auto

* 1 12 1 5. \( (r+1)^2 \leq i \)
   \[ \vdash \exists r : \mathbb{N}. r^2 \leq i < (r+1)^2 \]
   BY With \( [r+1] \) (D 0) THEN Auto'

* 1 12 2 5. \( \neg((r+1)^2 \leq i) \)
   \[ \vdash \exists r : \mathbb{N}. r^2 \leq i < (r+1)^2 \]
   BY With \( [r] \) (D 0) THEN Auto
Rewriting: replace terms by equivalent ones

- **Simple rewrite tactics**
  - **Fold** `name c`: fold abstraction `name` in clause `c`
  - **Unfold** `name c`: unfold abstraction `name` in clause `c`
  - **Subst** `t₁=t₂∈T c`: substitute `t₁` by `t₂` in clause `c`
  - **Reduce** `c`: repeatedly evaluate redices in clause `c`

- **Nuprl’s rewrite package**
  - Functions for creating and applying term rewrite rules
  - Supports various equivalence relations
  - Based on tactics for applying conversions to clauses in proofs

- **Conversions**
  - Language for systematically building rewrite rules
  - Transform terms and provide justifications
  - Need to be supported by various kinds of lemmata
  - Organized like tactics: atomic conversions, conversionals, advanced conversions
Atomic Conversions

- **Folding and Unfolding Abstractions**
  - `UnfoldC abs`: Unfold all occurrences of abstraction `abs`
  - `FoldC abs`: Fold all instances of abstraction `abs`
  
  Versions for (un)folding specific instances available as well.

- **Evaluating Redices**
  - `ReduceC`: contract all primitive redices
  - `AbReduceC`: contract primitive and abstract (user-defined) redices

- **Applying Lemmata and Hypotheses**
  - Universally quantified formulas with consequent `a r b`
  - `HypC i`: rewrite instances of `a` into instances of `b`
  - `RevHypC i`: rewrite instances of `b` into instances of `a`
  
  Variants: `LemmaC name`, `RevLemmaC name`
Building Rewrite Tactics

- **Construct advanced Conversions using** Conversionals
  - ANDTHENC, ORTHENC, ORELSEC, RepeatC, ProgressC, TryC

- **Define Macro Conversions**
  - MacroC name c_1 \ t_1 \ c_2 \ t_2: Rewrite instance of t_1 into instance of t_2
    c_1 and c_2 must rewrite t_1 and t_2 into the same term, name is a failure token
  - SimpleMacroC name t_1 \ t_2 abs: Rewrite t_1 into t_2 by unfolding
    abstractions from abs and contracting primitive redices

- **Transform Conversions into Tactics**
  - Rewrite c i: Apply conversion c to clause i
    Variants: RewriteType c i, RWAddr addr c i, RWU, RWD
Writing a tactic-based proof search procedure is easy

Sort rule applications by cost of induced proof search

let simple_prover = Repeat
  (  hypotheses
    ORELSE contradiction
    ORELSE InstantiateAll
    ORELSE InstantiateEx
    ORELSE conjunctionE
    ORELSE existentialE
    ORELSE nondangerousI
    ORELSE disjunctionE
    ORELSE not_chain
    ORELSE iff_chain
    ORELSE imp_chain
  );;

letrec prover = simple_prover
  THEN Try (    Complete (orI1 THEN prover)
    ORELSE (Complete (orI2 THEN prover))
  );;
simple_prover: COMPONENT TACTICS

let contradiction = TryAllHyps falseE is_false_term
and conjunctionE = TryAllHyps andE is_and_term
and existentialE = TryAllHyps exE is_ex_term
and disjunctionE = TryAllHyps orE is_or_term

and nondangerousI pf = let kind = operator_id_of_term (conclusion pf)
                        in
                        if mem mkind ['all'; 'not'; 'implies';
                                      'rev_implies'; 'iff'; 'and']
                        then Run (termkind ^ 'R') pf
                        else failwith 'tactic inappropriate'
                        ;;

let imp_chain pf = Chain impE (select_hyps is_imp_term pf) hypotheses pf
                    ;;
let not_chain = TryAllHyps (\pos. notE pos THEN imp_chain) is_not_term
                   ;;
let iff_chain = TryAllHyps (\pos. (iffE pos THEN (imp_chain
                                         ORELSE not_chain))
                                         ORELSE
                                         (iffE_b pos THEN (imp_chain
                                                          ORELSE not_chain))
                                         ) is_iff_term
                       ;;
## simple_prover: Rule Tactics for First-Order Logic

<table>
<thead>
<tr>
<th>left</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>andE i</strong> $\Gamma, A \land B, \Delta \vdash G$</td>
<td>$\Gamma \vdash A$</td>
</tr>
<tr>
<td></td>
<td>$\Gamma \vdash B$</td>
</tr>
<tr>
<td></td>
<td>$\Gamma \vdash A \land B$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>orE i</strong> $\Gamma, A \lor B, \Delta \vdash G$</td>
<td>$\Gamma \vdash A$</td>
</tr>
<tr>
<td></td>
<td>$\Gamma \vdash B$</td>
</tr>
<tr>
<td></td>
<td>$\Gamma \vdash A \lor B$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>impE i</strong> $\Gamma, A \Rightarrow B, \Delta \vdash G$</td>
<td>$\Gamma, A \vdash \Gamma \vdash B$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>notE i</strong> $\Gamma, \neg A, \Delta \vdash G$</td>
<td>$\Gamma \vdash \neg A$</td>
</tr>
<tr>
<td></td>
<td>$\Gamma \vdash A \neg I$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>exE i</strong> $\Gamma, \exists x : T . B, \Delta \vdash G$</td>
<td>$\Gamma \vdash \exists x : T . B$</td>
</tr>
<tr>
<td></td>
<td>$\Gamma \vdash B[t/x]$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>allE i</strong> $\Gamma, \forall x : T . B, \Delta \vdash G$</td>
<td>$\Gamma \vdash \forall x : T . B$</td>
</tr>
<tr>
<td></td>
<td>$\Gamma, x : T \vdash B$</td>
</tr>
</tbody>
</table>
simple_prover: Matching and instantiation

let InstantiateAll =
  let InstAll_aux pos pf =
    let concl = conclusion pf
    and qterm = type_of_hyp pos pf
    in
    let sigma = match_subAll qterm concl in
    let terms = map snd sigma
    in
    (allEon pos terms THEN (OnLastHyp hypothesis)) pf
    in
    TryAllHyps InstAll_aux is_all_term
  ;;

let InstantiateEx =
  let InstEx_aux pos pf =
    let qterm = conclusion pf
    and hyp = type_of_hyp pos pf
    in
    let sigma = match_subEx qterm hyp
    in
    let terms = map snd sigma
    in
    (exIon terms THEN (hypothesis pos)) pf
    in
    TryAllHyps InstEx_aux (\h.true)
  ;;
• Tactic-based proof search has limitations
  – Many proofs require some “lookahead”
  – Proof search must perform meta-level analysis first

• Complete proof search procedures are “unintuitive”
  – Proof search tree represented in compact form
  – Link similar subformulas that may represent leafs of a sequent proof
  – Proof search checks if all leaves can be covered by connections and if parameters all connected subformulas can be unified

• JProver: inutionistic proof search for NUPRL
  – Find matrix proof of goal sequent and convert it into sequent proof

Formula
\[ \neg A \lor \neg B \Rightarrow \neg B \lor \neg A \]

Annotation
types, polarities, prefixes

Annotated Formula Tree

Matrix Prover
path checking + unification
Substitutions induce ordering \(<\)

Proof Transformation
Search-free traversal of \(<\)
multiple → single-conclusion

Sequent Proof
Communicate formulas in uniform format (MathBus) over INET sockets

Logic module converts between internal term representations

Pre- and postprocessing in NUPRL widens range of applicability
SOLVING THE “AGATHA MURDER PUZZLE”

JProver can run in trusted mode or with all proof details expanded
Part IV:

Building Formal Theories
AN ELEGANT ACCOUNT OF RECORD TYPES

- Express records as (dependent) \textbf{functions from labels to types}
  - $\{x_1:T_1; \ldots ; x_n:T_n\} \equiv l:\text{Labels} \rightarrow \text{if } l=x_i \text{ then } T_i \text{ else } \text{Top}$
  - $\{x_1=t_1; \ldots ; x_n=t_n\} \equiv \lambda l.\text{if } z=x_i \text{ then } t_i \text{ else } ()$
  - $r.l \equiv (r\ l)$

- **Dependent Records** $\{x_1:T_1; \ x_2:T_2[x_1]; \ldots ; \ x_n:T_n[x_1;..x_{n-1}]\}$
  - Type $T_i$ may depend on value of components $x_1;..x_{i-1}$
  - Used for describing algebra, abstract data types, inheritance, …

- Use (dependent) \textbf{intersection to formalize both}

  \[
  \begin{align*}
  \{x:T\} & \equiv z:\text{Labels} \rightarrow \text{if } z=x \text{ then } T \text{ else } \text{Top} \\
  \{R_1; \ R_2\} & \equiv R_1 \cap R_2 \\
  \{x:S; \ y:T[x]\} & \equiv r:\{x:S\} \cap \{y:T[r.x]\} \\
  r.l & \equiv (r\ l) \\
  r.l<-t & \equiv \lambda z. \text{if } z=l \text{ then } t \text{ else } r.z \\
  \{} & \equiv \lambda l.() \\
  \{r; \ l=t\} & \equiv r.l<-t
  \end{align*}
  \]

- $\leadsto \textbf{Subtyping } \{x_1:T\} \sqsubseteq \{x_1:T_1; \ x_2:T_2[x_1]\} \text{ is easy to prove}$

Syntax of iterations can be adjusted using display forms
Formal Algebra: Semigroups

Tuple \((M, \circ)\) where \(M\) is a type and \(\circ: M \times M \to M\) associative

- **Formalization as dependent product** (\(\Sigma\) type)
  \[
  \text{SemiGroup} \equiv M : U \times \circ : M \times M \to M \times \forall x, y, z : M. \ x \circ (y \circ z) = (x \circ y) \circ z \in M
  \]
  \(\leadsto\) semigroups represented as triples \((M, \circ, \text{assoc\_pf})\)

- **Formalization via set types**
  \[
  \text{SemiGroupSig} \equiv M : U \times \circ : M \times M \to M
  \]
  \[
  \text{SemiGroup} \equiv \{\text{sg: SemiGroupSig} \mid \forall x, y, z : M_{\text{sg}}. \ x \circ_{\text{sg}} (y \circ_{\text{sg}} z) = (x \circ_{\text{sg}} y) \circ_{\text{sg}} z \in M_{\text{sg}}\}
  \]
  \(\leadsto\) tedious to access components or use associativity in proofs

- **Formalization via dependent records**
  \[
  \text{SemiGroupSig} \equiv \{M : U; \circ : M \times M \to M\}
  \]
  \[
  \text{SemiGroup} \equiv \{\text{SemiGroupSig}; \text{assoc:} \downarrow (\forall x, y, z : M. \ x \circ (y \circ z) = (x \circ y) \circ z \in M)\}
  \]
  \(\leadsto\) Accessing components and properties straightforward
  \(\leadsto\) Type squashing suppresses explicit proof component
  \(\leadsto\) Subtyping relation \(\text{SemiGroup} \sqsubseteq \text{SemiGroupSig}\) easy to prove
**Formal Algebra: Monoids and Groups**

- **Monoid**: semigroup with identity
  
  MonoidSig \equiv \{ \text{SemiGroupSig}; e: M \}  
  
  Monoid \equiv \{ \text{SemiGroup}; \text{MonoidSig}; \text{id: } \downarrow(\forall x: M. e \circ x = x \in M) \}  
  
  \leadsto \text{natural use of multiple inheritance}

- **Group**: monoid with inverse
  
  GroupSig \equiv \{ \text{MonoidSig}; \, ^{-1}: M \rightarrow M \}  
  
  Group \equiv \{ \text{Monoid}; \, \text{GroupSig}; \text{inv: } \downarrow(\forall x: M. x \circ x^{-1} = e \in M) \}  
  
  \leadsto \text{refinement hierarchy follows directly from definitions}

  \[
  \begin{align*}
  \text{SemiGroup} & \sqsubseteq \text{SemiGroupSig} \\
  \text{Monoid} & \sqsubseteq \text{MonoidSig} \\
  \text{Group} & \sqsubseteq \text{GroupSig}
  \end{align*}
  \]
THE NUPRL PROOF DEVELOPMENT SYSTEM

IV. BUILDING THEORIES

FORMALIZATION: ABSTRACT DATA TYPES

• Abstract Data Type for stacks over a type $T$

  TYPES
  Stack

  OPERATORS
  empty: Stack
  push: Stack $\times$ T $\rightarrow$ Stack
  pop: $\{s: \text{Stack} \mid s \neq \text{empty}\} \rightarrow$ Stack $\times$ T

  AXIOMS
  pushpop: $\forall s: \text{Stack} . \forall t: T . \text{pop(} \text{push(} s , a \text{)} \text{)} = (s , a)$

• Formalization

  – Dependent products unsuited for same reason as above
  – Dependent records lead to “natural formalization”

  $\text{STACKSIG}(T) \equiv \{ \text{Stack: } U$
  $\quad ; \text{empty: } \text{Stack}$
  $\quad ; \text{push: } \text{Stack} \times T \rightarrow \text{Stack}$
  $\quad ; \text{pop: } \{s: \text{Stack} \mid s \neq \text{empty}\} \rightarrow \text{Stack} \times T\}$

  $\text{STACK}(T) \equiv \{\text{STACKSIG}(T); \text{pf: } \downarrow (\forall s: \text{Stack}. \forall t: T . \text{pop(} \text{push(} s , a \text{)} \text{)} = (s , a) \in M)\}$

• Formalizing the implementation of stacks through lists

  $\text{list-as-stack}(T) \equiv \{ \text{Stack = } T \text{ list}$
  $\quad ; \text{empty = } []$
  $\quad ; \text{push} = \lambda s , t . t :: s$
  $\quad ; \text{pop} = \lambda s . <\text{hd}(s) , \text{tl}(s)> \}$

  $\leadsto \text{list-as-stack}(T) \in \text{STACK}(T)$ easy to prove
How to approach large application examples?

Verify and optimize distributed systems (Ensemble)

- Formalize semantics of implementation language
- Build tactics for verification of protocols and system configurations
- Build tactics that optimize performance of configured systems
Embedding system code into Nuprl
Enable formal reasoning on OCaml level

- **Type-theoretical semantics** of OCAML fragment
- **Nuprl** implementation captures syntax & semantics
- Develop **programming logic** for OCaml
- Build **import** and **export** mechanisms

---

**Diagram Description**

- **Programming Environment**
  - OCaml
  - Camlp4 Parser Preprocessor
  - Abstract Syntax Tree
  - Conversion module modified Pretty printer
  - Intermediate Code

- **Deductive System**
  - NuPRL / TYPE THEORY / Meta-Language ML
  - NuPRL Library
  - Term + Object Generators
  - Simulated Ocaml-Code
  - Print Representation
  - Representations of basic Ocaml-constructs
  - Abstractions
  - Display Forms
  - Type Information

---

**Key Components**

- **IMPORT**
- **EXPORT**

---

**Additional Information**

- *Type-theoreticalsemantics* of OCAML fragment
- NuPRL implementation captures syntax & semantics
- Develop programming logic for OCaml
- Build import and export mechanisms

---

**Links**

- The Nuprl Proof Development System
- IV. Building Theories
Formalize system specification and code  
e.g. “Messages are received in the same order in which they were sent”  
  – “Messages may be appended to global event queue and removed from its beginning”  
  – “Messages whose sequence number is too big will be buffered”  
  – ENSEMBLE module Pt2pt.ml: 250 lines of OCAML code  
    All levels represented in type theory

Verification methodology
  – Verify component specifications  
    (benign assumptions — subtle bug detected)  
  – Verify systems by composition  
    (IOA-composition preserves safety properties)  
  – Weave aspects  
  – Verify code
1. Use known optimizations of micro-protocols
2. Compose into optimizations of protocol stacks
3. Integrate message header compression
4. Generate code from optimization theorems and reconfigure system

Fast, error-free, independent of programming language

speedup factor 3-10
DEMO: OPTIMIZING A 24-LAYER PROTOCOL STACK

Performance Test

Original ENSEMBLE System

10000 rounds

After Optimizations

Performance Test

10000 rounds

3–4 times faster

The Nuprl Proof Development System

IV. Building Theories
Part V:

Future Directions
Challenges for Automated Theorem Proving

• **A more expressive theory**
  - **Reflection**: reasoning about syntax and semantics simultaneously
  - Reasoning about objects, inheritance, liveness, distributed processes, …

• **A more widely applicable system**
  - Digital Libraries of Formal Knowledge
  - Cooperation between different proof systems

• **Learn more from large scale applications**
  - Synthesize, verify, and optimize high-assurance software systems
  - Target “unclean” but popular programming languages
  - Aim at pushbutton technology
Directions in Theory: Reflection

- **Embed meta-level of type theory into type theory**
  - Reason about relation between syntactical form and semantical value
    - evaluation, resources, complexity
    - semantical effects of syntactical transformations (reordering, renaming, ...)
    - proofs, tactic applications, dependencies (e.g. proofs $\leftrightarrow$ library contents)
    - relations between different formal theories
      
      ... from within the logic

- **Extremely powerful, but little utilization**

- **Approach: mirror type theory as recursive type**
  - Logically satisfactory, not efficient enough for practical purposes  
    (LICS 1990)

- **New: primitive type of intensional representations**
  - Type Term, closed under quotation
    (Cornell 2001)
  - Theoretically challenging, but much more efficient
**Reflection – basic methodology**

- **Represent object and meta level in type theory**
  - Represent meta-logical concepts as Nuprl terms
  - Express specific object logic in represented meta logic
  - Build hierarchy: level $i$ contains meta level for level $i+1$

- **Reasoning about both levels from the “outside”**

- **Link object logic and meta-logic**
  - Embed object level terms using quotation (operator)
  - Embed object level provability using reflection rule

$$\Gamma \vdash_{i+1} A \quad \text{by reflection } i$$

$$\vdash_i \exists p: \text{Proof}_i. \text{goal}(p) = [\Gamma \vdash_{i+1} A]$$

- **Use same reasoning apparatus for object and meta level**
**Library as platform for cooperating reasoning tools**

- **Connect**
  - Additional proof engines: PVS, HOL, MinLog, ...
  - Multiple browsers (ASCII, web, ...) and editors (structured, Emacs-mode, ...)
  - MathWeb (through OmDoc interface)

**Provide new features**
- Archival capacities (documentation & certification, version control)
- Embedding external library contents (needs data conversion, proof replay, ...)
- A variety of justifications (levels of trust)
- Creation of formal and textual documents
- Asynchronous and distributed mode of operation
- Meta-reasoning (e.g. about relations between theories) and reflection

**Improve cooperation between research groups**

Authoritative reference for reliable software construction
Areas for Study & Research

- **Formal Logics & Type Theory**
  - Classes & inheritance, recursive & partial objects, concurrency, real-time
  - Meta-reasoning, reflection, relating different logics, ...

- **Theorem Proving Environments**
  - Logical accounting, theory modules, interfaces, proof presentation, ...

- **Automated Proof Search Procedures**
  - Matrix methods, inductive theorem proving, rewriting, proof planning
  - Decision procedures, extended type inference, cooperating provers
  - Proof reuse, analogy, distributed proof procedures, ...

- **Applications**
  - Formal CS knowledge: graph theory, automata, trees, arrays, ...
  - Strategies for program synthesis, verification, and optimization
  - Modeling programming languages (OCAML, JAVA, ..)