The Nuprl Proof Development System

Christoph Kreitz

Department of Computer Science, Cornell University

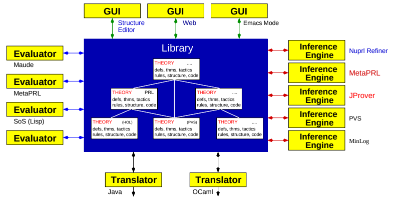
Ithaca, NY 14853

http://www.nuprl.org

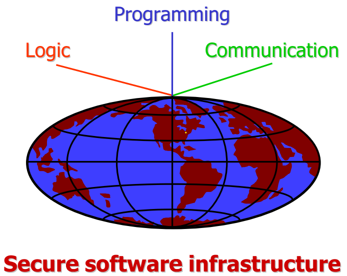
THE NUPRL PROJECT AT CORNELL UNIVERSITY

- **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



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, Π- & Σ-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
  - Book: *Implementing Mathematics . . .* (1986)
  - Nuprl 2: Unix Version
- **Nuprl 3: Mathematical problem solving** (1987–1994)
  - 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
  - $\leadsto$ Interactive digital libraries of formal algorithmic knowledge

Applications: Mathematics & Programming

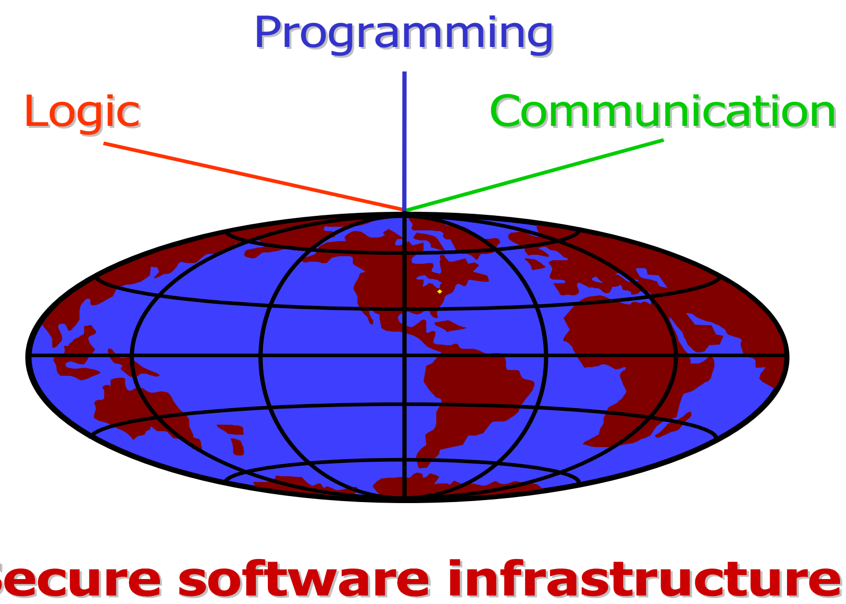
- **Formalized mathematical theories**
  - Elementary number theory, real analysis, group theory
  - Discrete mathematics (Allen, 1994 –. . .)
  - General algebra (Jackson, 1994)
  - Finite and general automata (Constable, Naumov & Uribe 1997, Bickford, 2001)
  - Basics of Turing machines (Naumov, 1998 . . .)
  - Formal mathematical textbook (Constable, Allen 1999)

  http://www.nuprl.org/Nupr14.2/Libraries/Welcome.html

- **Machine proof for unsolved problems**
  - Girard's paradox (Howe 1987)
  - Higman's Lemma (Murthy 1990)

- **Algorithms and programming languages**
  - Synthesis of elementary algorithms: square-root, sorting, . . .
  - Simple imperative programming (Naumov, 1997)
  - Programming semantics & complexity analysis (Benzinger, 2000)
  - Type-theoretical semantics of large OCaml fragment (Kreitz 1997/2002)

Applications: System Verification and Optimization



- **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

Overview

☑ **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 ⇝ programming
  - Integers, lists, inductive types ⇝ inductive definition
  - Propositions as types, equality type, void, top, universes ⇝ logic
  - Subsets, subtyping, quotient types ⇝ mathematics
  - (Dependent) intersection, union, records ⇝ 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 ⇝ free syntax
- **Expressions defined independently of their types**
  - No restriction on expressions that can be defined ⇝ Y combinator
  - Expressions *in proofs* must be typeable ⇝ "total" functions
- **Semantics based on values of expressions**
  - Judgments state what is true ⇝ computational semantics
  - Equality is extensional
- **Refinement calculus**
  - Top-down sequent calculus ⇝ interactive proof development
  - Proof expressions linked to inference rules ⇝ program extraction
  - Computation rules ⇝ program evaluation
- **User-defined extensions possible**
  - User-defined expressions and inference rules ⇝ abstractions & tactics

Syntax Issues

- **Uniform notation:** $opid\{p_i\!:\!F_i\}(x_{11},..,x_{m_1 1}.t_1;\ldots;x_{1n},..,x_{m_n n}.t_n)$
  - Operator name $opid$ listed in operator tables
  - Parameters $p_i\!:\!F_i$ for base terms (variables, numbers, tokens...)
  - Sub-terms t_j may contain bound variables $x_{1j},..,x_{m_j j}$
  - No syntactical distinction between types, members, propositions ...

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

Internal Term Structure	*Display Form*
variable$\{x\!:\!\texttt{v}\}()$	x
function$\{\}(S;\,x.T)$	$x\!:\!S\!\to\!T$
function$\{\}(S;\,.T)$	$S\!\to\!T$
⋮	⋮
lambda$\{\}(x.t)$	$\lambda x.t$
apply$\{\}(f;t)$	$f\ t$
⋮	⋮

$\leadsto$ conventional notation, information hiding, auto-parenthesizing, aliases, ...

Semantics models proof, not denotation

- **(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

canonical	*non-canonical*	*Redex*		*Contractum*
$S \rightarrow T$				
$\lambda x\,.\,t$	$\boxed{f}\ t$	$\boxed{\lambda x\,.\,u}\ t$	$\xrightarrow{\beta}$	$u[t/x]$

- **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 $s_1{=}s_2 \in S$

$\vdots$ $\vdots$

NUPRL'S PROOF THEORY

- **Sequent** $x_1{:}T_1, \ldots, x_n{:}T_n \vdash C \quad \lfloor\textsf{ext}\ t\rfloor$

  "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 $T \ \lfloor\textsf{ext}\ t\rfloor$ $\leadsto$ proof term construction

  – Equality is represented as type $s{=}t \in T \ \lfloor\textsf{ext Ax}\rfloor$ $\leadsto$ propositions as types

  – Typehood represented by (cumulative) universes $\mathbb{U}_i \ \lfloor\textsf{ext}\ T\rfloor$

- **Refinement calculus**

  – Top-down decomposition of proof goal $\leadsto$ interactive proof development

  – Bottom-up construction of proof terms $\leadsto$ program extraction

  $$\Gamma \vdash S{\rightarrow}T \quad \lfloor\textsf{ext}\ \lambda x.e\rfloor \quad \textbf{by}\ \texttt{lambda-formation}\ x$$
  $$\Gamma,\ x{:}S \vdash T \quad \lfloor\textsf{ext}\ e\rfloor$$
  $$\Gamma \vdash S{=}S \in \mathbb{U}_i \quad \lfloor\textsf{ext Ax}\rfloor$$

  – Computation rules $\leadsto$ program evaluation

About 8–10 inference rules for each NUPRL type

Executing a Formal Proof Step

	THM intsqrt
Theorem name	
Status + position in proof	`# top 1`
Hypothesis of main goal	`1. x:ℕ`
Conclusion	$\vdash \exists y{:}\mathbb{N}.\ y^2 \le x \ \wedge\ x < (y+1)^2$
Inference rule	`BY natE 1`
First subgoal – status, conclusion	`1#` $\vdash \exists y{:}\mathbb{N}.\ y^2 \le 0 \ \wedge\ 0 < (y+1)^2$
Second subgoal – status,	`2# 2. n:ℕ`
new hypotheses	`3. 0<n`
	`4. v:`$\exists y{:}\mathbb{N}.\ y^2 \le n-1 \ \wedge\ n-1 < (y+1)^2$
conclusion	$\vdash \exists y{:}\mathbb{N}.\ y^2 \le n \ \wedge\ n < (y+1)^2$

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 $\rightsquigarrow$ 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? ($typeEquality$) $\Gamma \vdash S = T \in \mathbf{U}_j$
  - How to build the type? ($typeFormation$) $\Gamma \vdash \mathbf{U}_j \ \lfloor \text{ext } T \rfloor$
- **Canonical rules:**
  - When are two members equal? ($memberEquality$) $\Gamma \vdash s = t \in T$
  - How to build members? ($memberFormation$) $\Gamma \vdash T \ \lfloor \text{ext } t \rfloor$
- **Noncanonical rules:**
  - When does a term inhabit a type? ($noncanonicalEquality$) $\Gamma \vdash s = t \in T$
  - How to use a variable of the type ($typeElimination$) $\Gamma, x{:}S, \Delta \vdash T \ \lfloor \text{ext } t \rfloor$
- **Computation rules:**
  - Reduction of redices in an equality ($noncanonicalReduce*$) $\Gamma \vdash redex = t \in T$
- **Special purpose rules**

Proof Rules for the Function Type

$\Gamma \vdash \mathbb{U}_j \quad \lfloor \text{ext } x{:}S{\rightarrow}T \rfloor$
by `dependent_functionFormation` x S
$\Gamma \vdash S \in \mathbb{U}_j \quad \lfloor \text{ext Ax} \rfloor$
$\Gamma, x{:}S \vdash \mathbb{U}_j \quad \lfloor \text{ext } T \rfloor$

$\Gamma \vdash \lambda x_1.t_1 = \lambda x_2.t_2 \in x{:}S{\rightarrow}T \quad \lfloor \text{ext Ax} \rfloor$
by `lambdaEquality` j x'
$\Gamma, x'{:}S \vdash t_1[x'/x_1] = t_2[x'/x_2] \in T[x'/x] \quad \lfloor \text{ext Ax} \rfloor$
$\Gamma \vdash S \in \mathbb{U}_j \quad \lfloor \text{ext Ax} \rfloor$

$\Gamma \vdash f_1 t_1 = f_2 t_2 \in T[t_1/x] \quad \lfloor \text{ext Ax} \rfloor$
by `applyEquality` $x{:}S{\rightarrow}T$
$\Gamma \vdash f_1 = f_2 \in x{:}S{\rightarrow}T \quad \lfloor \text{ext Ax} \rfloor$
$\Gamma \vdash t_1 = t_2 \in S \quad \lfloor \text{ext Ax} \rfloor$

$\Gamma \vdash (\lambda x.t)\, s = t_2 \in T \quad \lfloor \text{ext Ax} \rfloor$
by `applyReduce`
$\Gamma \vdash t[s/x] = t_2 \in T \quad \lfloor \text{ext Ax} \rfloor$

$\Gamma \vdash x_1{:}S_1{\rightarrow}T_1 = x_2{:}S_2{\rightarrow}T_2 \in \mathbb{U}_j \quad \lfloor \text{ext Ax} \rfloor$
by `functionEquality` x
$\Gamma \vdash S_1 = S_2 \in \mathbb{U}_j \quad \lfloor \text{ext Ax} \rfloor$
$\Gamma, x{:}S_1 \vdash T_1[x/x_1] = T_2[x/x_2] \in \mathbb{U}_j \quad \lfloor \text{ext Ax} \rfloor$

$\Gamma \vdash x{:}S{\rightarrow}T \quad \lfloor \text{ext } \lambda x'.t \rfloor$
by `lambdaFormation` j x'
$\Gamma, x'{:}S \vdash T[x'/x] \quad \lfloor \text{ext } t \rfloor$
$\Gamma \vdash S \in \mathbb{U}_j \quad \lfloor \text{ext Ax} \rfloor$

$\Gamma, f{:}\,x{:}S{\rightarrow}T, \Delta \vdash C \quad \lfloor \text{ext } t[f\,s, \text{Ax}/y,z] \rfloor$
by `dependent_functionElimination` i s y z
$\Gamma, f{:}\,x{:}S{\rightarrow}T, \Delta \vdash s \in S \quad \lfloor \text{ext Ax} \rfloor$
$\Gamma, f{:}\,x{:}S{\rightarrow}T, y{:}T[s/x], z{:}\,y{=}f\,s \in T[s/x], \Delta \vdash C \quad \lfloor \text{ext } t \rfloor$

$\Gamma \vdash f_1 = f_2 \in x{:}S{\rightarrow}T \quad \lfloor \text{ext } t \rfloor$
by `functionExtensionality` j $x_1{:}S_1{\rightarrow}T_1$ $x_2{:}S_2{\rightarrow}T_2$ x'
$\Gamma, x'{:}S \vdash f_1 x' = f_2 x' \in T[x'/x] \quad \lfloor \text{ext } t \rfloor$
$\Gamma \vdash S \in \mathbb{U}_j \quad \lfloor \text{ext Ax} \rfloor$
$\Gamma \vdash f_1 \in x_1{:}S_1{\rightarrow}T_1 \quad \lfloor \text{ext Ax} \rfloor$
$\Gamma \vdash f_2 \in x_2{:}S_2{\rightarrow}T_2 \quad \lfloor \text{ext Ax} \rfloor$

Note: $e{=}e \in T$ is usually abbreviated by $e \in T$

User-defined Extensions

- **Conservative extension of the formal language**

  = **Abstraction**: $\mathit{new\text{-}opid}\{\mathit{parms}\}(\mathit{sub\text{-}terms}) \equiv \mathit{expr}[\mathit{parms},\ \mathit{sub\text{-}terms}]$

  e.g. $\textbf{exists}\{\}(T\,;\,x\,.\,A[x]) \;\equiv\; x\!:\!T\times A[x]$

  \+ **Display Form** for newly defined term

  e.g. $\exists x\!:\!T\,.\,A[x] \;\equiv\; \textbf{exists}\{\}(T\,;\,x\,.\,A[x])$

  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 *find* a proof

  – Always result in a valid proof

  Library contains many standard tactics and proof search procedures

Standard Nuprl Types

Function Space	$S \rightarrow T$, $x:S \rightarrow T$	$\lambda x.t$, $f\,t$
Product Space	$S \times T$, $x:S \times T$	$\langle s,t \rangle$, let $\langle x,y \rangle = e$ in u
Disjoint Union	$S+T$	inl(s), inr(t), case e of inl$(x) \mapsto u$ | inr$(y) \mapsto v$
Universes	$\mathbf{U}_j$	— *types of level j* —
Equality	$s = t \in T$	Ax
Empty Type	Void	any(x), — *no members* —
Atoms	Atom	"*token*", if a=b then s else t
Numbers	$\mathbb{Z}$	0,1,-1,2,-2,... s+t, s-t, s*t, $s \div t$, s rem t, if a=b then s else t, if i<j then s else t ind$(u;\, x,f_x.s;\; base;\; y,f_y.t)$
	i<j	Ax
Lists	S list	[], t::*list*, rec-case L of [] $\mapsto$ *base* | x::$l \mapsto$ [f_l].t
Inductive Types	rectype $X = T[X]$	let* $f(x) = t$ in $f(e)$, — *members defined by* $T[X]$ —
Subset	$\{x:S \mid P[x]\}$,	— *some members of S* —
Intersection	$\cap x:S.T[x]$,	— *members that occur in all* $T[x]$ —
	$x:S \cap T[x]$	— *members x that occur S and* $T[x]$ —
Union	$\cup x:S.T[x]$	— *members that occur in some* $T[x]$, *tricky equality* —
Quotient	$x,y : S//E[x,y]$	— *members of S, new equality* —
Very Dep. Functions	$\{f \mid x:S \rightarrow T[f,x]\}$	
Squiggle Equality	s~t	— *a "simpler" equality*

Functions: Basic Programming Concepts

Syntax:

Canonical: $S \rightarrow T$, $\lambda x \,.\, e$

Noncanonical: $e_1\, e_2$

Evaluation:

$$\boxed{\lambda x \,.\, u}\; t \xrightarrow{\beta} u[t/x]$$

Semantics:

- $S \rightarrow T$ is a type if S and T are
- $\lambda x_1 . e_1 = \lambda x_2 . e_2$ in $S \rightarrow T$ if $S \rightarrow T$ 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: let $\langle x, y\rangle = e$ in u

Evaluation:

let $\langle x, y\rangle = \boxed{\langle e_1, e_2\rangle}$ 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: $S{+}T$, $\mathsf{inl}(e)$, $\mathsf{inr}(e)$

Noncanonical: $\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u\ |\ \mathsf{inr}(y) \mapsto v$

Evaluation:

$$\mathsf{case}\ \boxed{\mathsf{inl}(e')}\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u\ |\ \mathsf{inr}(y) \mapsto v \quad \xrightarrow{\beta} \quad u[e' / x]$$

$$\mathsf{case}\ \boxed{\mathsf{inr}(e')}\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u\ |\ \mathsf{inr}(y) \mapsto v \quad \xrightarrow{\beta} \quad v[e' / y]$$

Semantics:

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

Library Concepts: ——

The Curry-Howard Isomorphism, formally

Propositions are represented as types

Proposition		**Type**
$P \wedge Q$	$\equiv$	$P \times Q$
$P \vee Q$	$\equiv$	$P + Q$
$P \Rightarrow Q$	$\equiv$	$P \rightarrow Q$
$\neg P$	$\equiv$	$P \rightarrow \mathsf{Void}$
$\exists x{:}T\,.\,P[x]$	$\equiv$	$x{:}T \times P[x]$
$\forall x{:}T\,.\,P[x]$	$\equiv$	$x{:}T \rightarrow P[x]$

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** `λx. if x=0 then λx.x else λx,y.x`

- **Allow expressing mathematical concepts such as finite automata**
  - (Q,Σ,q_0,δ,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`; ...; `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, ...)

Dependent Functions (Π-Types)

Subsumes independent function type

$\forall$ generalizes $\Rightarrow$

Syntax:

Canonical: $x{:}S{\rightarrow}T$, $\lambda x\,.\,e$

Noncanonical: $e_1\; e_2$

Evaluation:

$$\boxed{\lambda x\,.\,u}\; t \xrightarrow{\beta} u[t/x]$$

Semantics:

· $x{:}S{\rightarrow}T$ is a type if S is a type and $T[e/x]$ is a type for all e in S

· $\lambda x_1.\, e_1 = \lambda x_2.\, e_2$ in $x{:}S{\rightarrow}T$ if $x{:}S{\rightarrow}T$ type and

$e_1[s_1/x_1]{=}e_2[s_2/x_2]$ in $T[s_1/x]$ for all s_1, s_2 with $s_1{=}s_2 \in S$

Dependent Products (Σ-Types)

Subsumes (independent) cartesian product

$\exists$ generalizes $\wedge$

Syntax:

Canonical: $x{:}S{\times}T$, $\langle e_1, e_2\rangle$

Noncanonical: $\mathsf{let}\ \langle x, y\rangle = e\ \mathsf{in}\ u$

Evaluation:

$$\mathsf{let}\ \langle x, y\rangle = \boxed{\langle e_1, e_2\rangle}\ \mathsf{in}\ u \xrightarrow{\beta} u[e_1, e_2 / x, y]$$

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,

$e_1{=}e_1'$ in S, and $e_2{=}e_2'$ in $T[e_1/x]$

Well-formedness Issues

- **Formation rules for dependent type require checking**

  $$x'{:}S \vdash T[x'/x] \ \texttt{type}$$

  – T is a function from S to types that could involve complex computations, e.g. $T[i] \equiv$ `if` $\texttt{M}_i$`(i) halts then` $\mathbb{N}$ `else Void`

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

- **Programming languages must restrict dependencies**
  – Only allow finite dependencies $\leadsto$ decidable typechecking

- **Typechecking in Nuprl cannot be fully automated**
  – Typechecking becomes part of the proof process $\leadsto$ heuristic typechecking

- **Additional problem**
  – What is the type of a function from $\mathbb{N}$ to types? $\leadsto$ 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 $\hat{=}$ cumulative hierarchy of universes $\mathbb{U}{=}\mathbb{U}_1 \overset{\in}{\subset} \mathbb{U}_2 \overset{\in}{\subset} \mathbb{U}_3 \overset{\in}{\subset} \ldots$

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 …mimic semantics for $S = T$ as types…

· $\mathbb{U}_{j_1} = \mathbb{U}_{j_2}$ in $\mathbb{U}_j$ if $j_1{=}j_2{<}j$

INTEGERS: BASIC ARITHMETIC

Syntax:

Canonical: $\mathbb{Z}$, 0, 1, -1, 2, -2, ... i<j, Ax

Noncanonical: rec-case i of x<0 ↦ [f_x] . s | 0 ↦ b | y>0 ↦ [f_y] . t,
s+t, s-t, s*t, s÷t, s rem t,
if i=j then s else t, if i<j then s else t,

Evaluation:

rec-case 0 of x<0 ↦ [f_x] . s | 0 ↦ b | y>0 ↦ [f_y] . t $\xrightarrow{\beta}$ b

rec-case i of x<0 ↦ [f_x] . s | 0 ↦ b | y>0 ↦ [f_y] . t where $i > 0$
$\xrightarrow{\beta}$ $t[i,$ rec-case $i-1$ of x<0 ↦ [f_x] . s | 0 ↦ b | y>0 ↦ [f_y] . t $/ x, f_x]$

rec-case i of x<0 ↦ [f_x] . s | 0 ↦ b | y>0 ↦ [f_y] . t where $i < 0$
$\xrightarrow{\beta}$ $s[i,$ rec-case $i+1$ of x<0 ↦ [f_x] . s | 0 ↦ b | y>0 ↦ [f_y] . t $/ x, f_x]$

other noncanonical expressions evaluate as usual

Semantics:

- $\mathbb{Z}$ is a type
- $i < j$ is a type if $i \in \mathbb{Z}$ and $j \in \mathbb{Z}$
- $i = i$ in $\mathbb{Z}$ for all integer constants i
- Ax = 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 list , [], e_1::e_2

Noncanonical: rec-case e of [] $\mapsto$ $base$ | x::l $\mapsto$ [f_{xl}].up

Evaluation:

rec-case [] of [] $\mapsto$ $base$ | x::l $\mapsto$ [f_{xl}].up $\xrightarrow{\beta}$ $base$

rec-case e_1::e_2 of [] $\mapsto$ $base$ | x::l $\mapsto$ [f_{xl}].up

$\xrightarrow{\beta}$ $up[e_1, e_2,$ rec-case e_2 of [] $\mapsto$ $base$ | x::l $\mapsto$ [f_{xl}].up $/ x, l, f_{xl}]$

Semantics:

· T list is a type if T is

· [] = [] in T list if T list is a type

· e_1::e_2 = e_1'::e_2' in T list if T list type, e_1=e_1' in T, and e_2=e_2' in T list

Library Concepts:

hd(e), tl(e), e_1@e_2, length(e), map(f;e), rev(e), e[i], e[i..j^-], ...

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 $X_1(x_1)$ = $T[X_1]$ and ... $X_n(x_n)$ = $T[X_n]$ select $X_i(a_i)$
  - Co-inductive type inftype $X = T_X$: greatest fixed point semantics
  - Partial recursive functions $S \not\to T$: unrestricted recursive induction

Inductive Types, formally

Syntax:

Canonical: $\mathsf{rectype}\ X = T_X$

Noncanonical: $\mathsf{let}^*\ f(x) = t\ \ \mathsf{in}\ \ f(e)$

Evaluation:

$$\mathsf{let}^*\ f(x) = t\ \ \mathsf{in}\ \ f(e) \xrightarrow{\beta} t[\lambda y.\mathsf{let}^*\ f(x) = t\ \ \mathsf{in}\ \ f(y), e\ /\ f, x]$$

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

Semantics:

- $\mathsf{rectype}\ X_1 = T_{X1} = \mathsf{rectype}\ X_2 = T_{X2}$
  if $T_{X1}[X/X_1] = T_{X2}[X/X_2]$ for all types X
- $s = t$ in $\mathsf{rectype}\ X = T_X$ if $\mathsf{rectype}\ X = T_X$ type and
  $s = t$ in $T_X[\mathsf{rectype}\ X = T_X / X]$

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 in $\forall x:S_1.\ T_1[x/x_1] \Rightarrow T_2[x/x_2]$

and p_2 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,

$s{=}t$ in S, and there is some p 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

$$\Gamma,\, z{:}\,\{x{:}S \mid T\},\, \Delta \vdash C \quad \lfloor \mathsf{ext}\ (\lambda y.t)\ z \rfloor$$

by `setElimination` $i\ y\ v$

$$\Gamma,\, z{:}\,\{x{:}S \mid T\},\, y{:}S,\, [\![v]\!]{:}T[y/x],\, \Delta[y/z] \vdash C[y/z] \quad \lfloor \mathsf{ext}\ t \rfloor$$

– 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 $x,y:T//E \not\sqsubseteq T$)
  - 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 $\forall x{:}T_1.\forall y{:}T_1.\ E_1[x,y/x_1,y_1] \Rightarrow E_1[y,x/x_1,y_1]$,
and 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 $E[s,t/x,y]$

Quotient Types: Proof theory

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

$$\Gamma,\, v{:}\, s = t \in x,y : T//E,\, \Delta \vdash C \quad \lfloor \mathsf{ext}\ u \rfloor$$

by `quotient_equalityElimination` i j v'

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

$$\Gamma,\, v{:}\, s = t \in x,y : T//E,\, \Delta \vdash E[s,t/x,y] \in \mathbb{U}_j \quad \lfloor \mathsf{Ax} \rfloor$$

User-predicates may require type-squashing

– $\downarrow P \equiv \{$`x: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

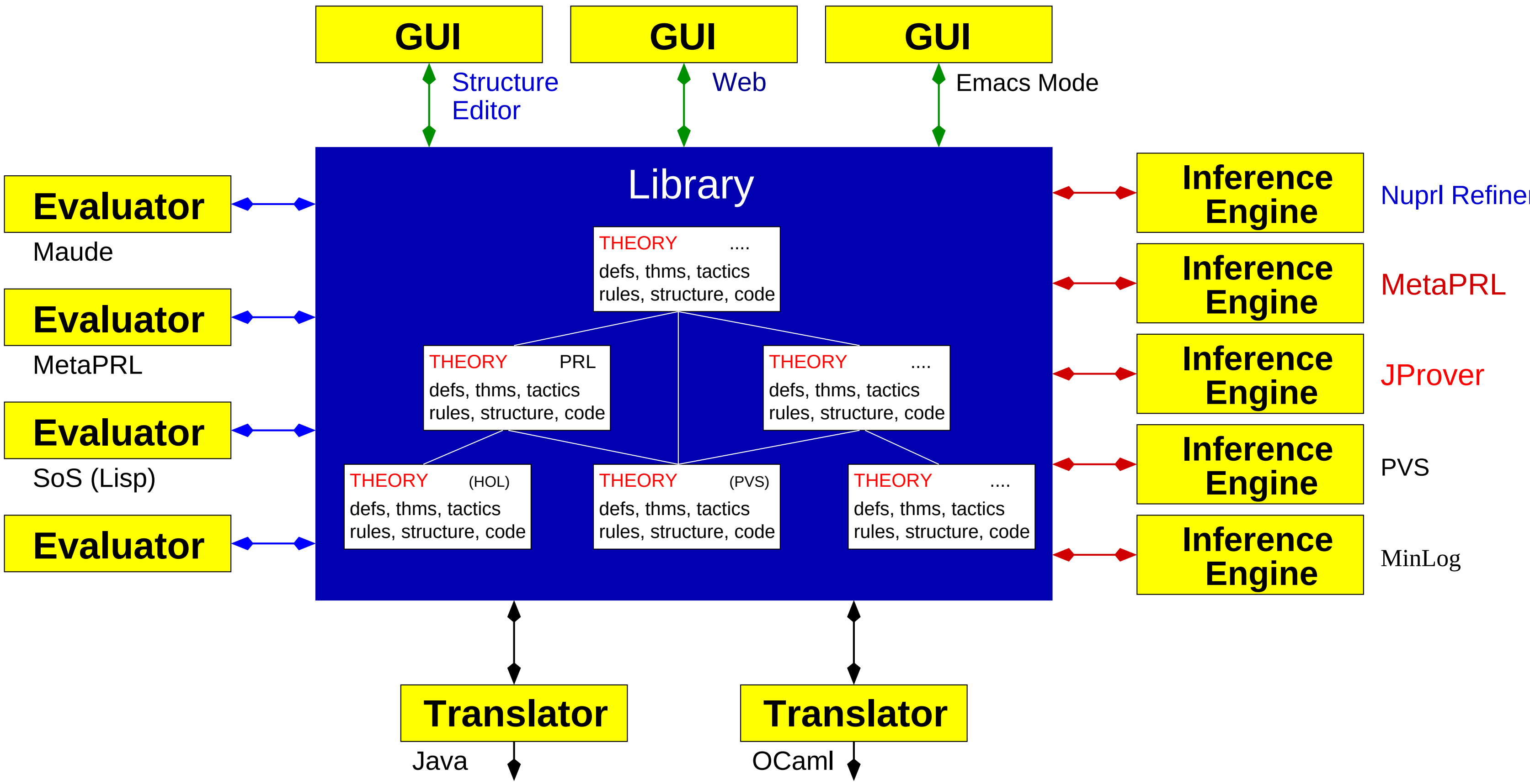
- Integer ranges: $\mathbb{N} \equiv$ `{i:`$\mathbb{Z}$`|0≤i}`, `{j...}` $\equiv$ `{i:`$\mathbb{Z}$`|j≤i}`,
  $\mathbb{N}^+ \equiv$ `{i:`$\mathbb{Z}$`|0<i}`, `{...j}` $\equiv$ `{i:`$\mathbb{Z}$`|i≤j}`
- Logic: $\forall$ $\exists$ $\wedge$ $\vee$ $\Rightarrow$ $\neg$ `True` `False` (Curry-Howard isomorphism)
- Singleton type: `Unit` $\equiv$ `0` $\in \mathbb{Z}$
- Boolean: $\mathbb{B}$ $\equiv$ `Unit + Unit`, $\uparrow b$ $\equiv$ `if` b `then True else False`
- Top type: `Top` $\equiv$ $\cap$`x:Void.Void`
- Subtyping: $S \sqsubseteq T$ $\equiv$ $\forall$`x:`S`.` `x` $\in T$
- Type squashing: $\downarrow P$ $\equiv$ `{True |` P `}`
- Recursive functions: **Y** $\equiv$ `λf. (λx.f (x x)) (λx.f (x x))`
- (Dependent) records $\{x_1{:}T_1;\ x_2{:}T_2[x_1];\ ..;\ x_n{:}T_n[x_1..x_{n-1}]\}$ (→ part IV)

See the standard theories of Nuprl 5 for further details

Part II:

The Nuprl System

Nuprl's Automated Reasoning Environment



Interactive proof development

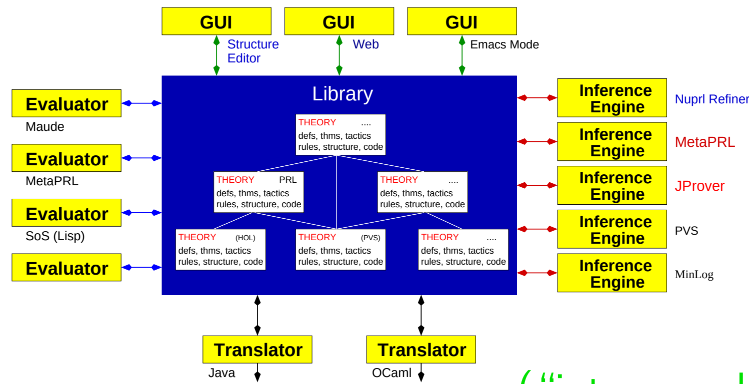
– Supports program extraction and evaluation

– Proof automation through tactics & decision procedures

– Highly customizable: conservative language extensions, term display, . . .

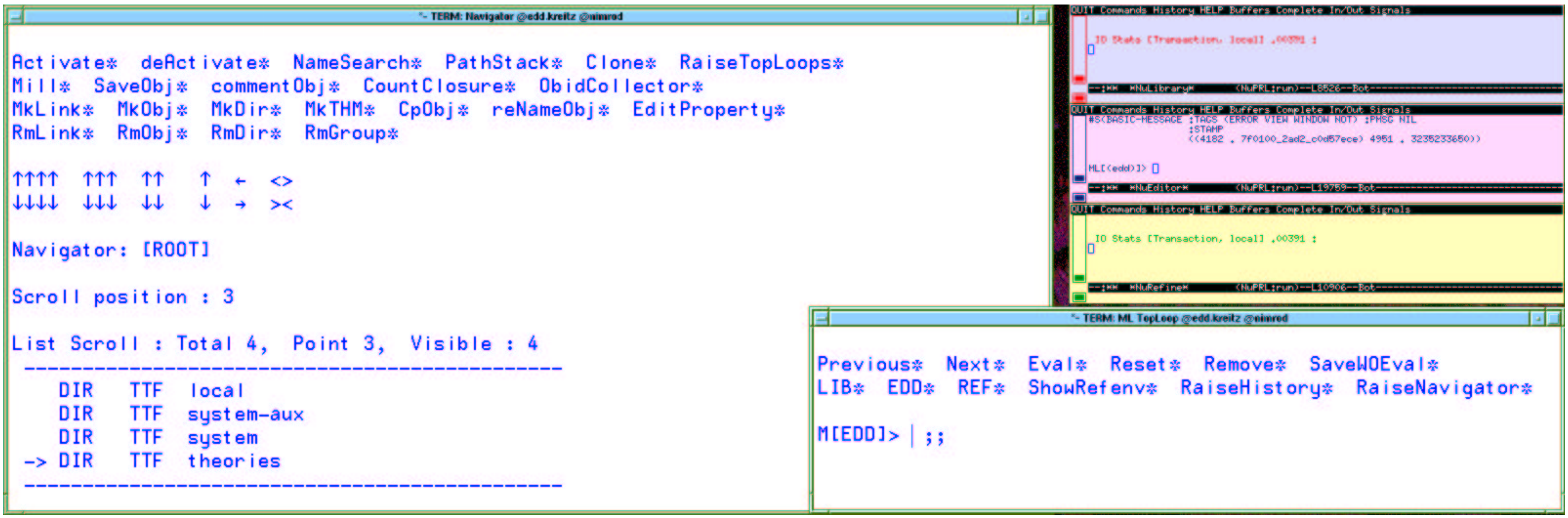
– Supports cooperation with other proof systems

System Architecture (Allen et. al, 2000)



- **Collection of cooperating processes**
  ⇝ Asynchronous, distributed & collaborative theorem proving ("interoperability")
- **Centered around a common knowledge base**
  – Library of formal algorithmic knowledge
  – Persistent data base, version control, dependency tracking ⇝ 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 ⇝ collaborative proving
- **Reflective system structure**
  – System designed within the system's library ⇝ 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** ⇝ readable proofs
- **Flexible definition mechanism** ⇝ user-defined terms
- **Customizable term display** ⇝ flexible notation
- **Structure editor for terms** ⇝ no ambiguities
- **Tactics & decision procedures** ⇝ user-defined inferences
- **Proof objects, program extraction** ⇝ program synthesis
- **Program evaluation**
- **Library mechanism** ⇝ user-theories
  – Large mathematical libraries & tactics collection
- **Command interface: navigator + ML top loops**
- **Formal documentation mechanism** ⇝ 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**

⋮

The Proof Editor

- 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

THM not_over_and

```
* top
∀A,B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))

BY D 0

* 1

1. A:ℙ
⊢ ∀B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))

BY Auto

* 1 1

2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)

BY D 0 THEN D 3 THEN Auto

* 2

.....wf.....
ℙ ∈ 𝕌'

BY Auto
```

- Proof editor may generate **extract terms** from complete proofs

The Structured Term Editor

Edit internal structure of terms while showing external display

- **Invoke by opening or entering a term slot**
- **Entering *opid* of term opens template**
  e.g. ⌞exists↵⌟ generates the template ⌜∃[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 emacs-like key combinations (m-p, m-b, m-f, m-n)
  – Cutting and pasting of terms possible (c-k, m-k, c-y)
  – Text oriented editing possible as well
  – Insert non-ASCII characters with c-#*num*
- **Internal structure can be made visible**
  – Explode (c-x ex) and implode (c-x im) terms
  – Entirely new terms can be inserted by entering *opid*{*parms*}(arity)
- **Terms have hyperlinks to abstractions and display forms**
  – Use c-x ab / Mouse-Right and c-x df / Mouse-Middle

Creating Definitions

Define new terms in terms of existing ones

- **Click the AddDef* button**

```
OK*  Cancel*

add def : [lhs] ==
          [rhs]

----------------------------------------

MkTHY*  OpenThy*  CloseThy*  ExportThy*  ChkThy*  ChkAllThys*  ChkOpenThy*
CheckMinTHY*  MinTHY*  EphTHY*  ExTHY*

Mill*  ObidCollector*  NameSearch*  PathStack*  RaiseTopLoops*
PrintObjTerm*  PrintObj*  MkThyDocObj*  ProofHelp*  ProofStats*  showRefEnvs*  FixRefEnvs*
CpObj*  reNameObj*  EditProperty*  SaveObj*  RmLink*  MkLink*  RmGroup*

ShowRefenv*  SetRefenvSibling*  SetRefenvUsing*  SetRefenv*  ProveRR*  SetInOBJ*
MkTHM*  MkML*  AddDef*  AddRecDef*  AddRecMod*  AddDefDisp*  AbReduce*  NavAtAp*
Act*  DeAct*  MkThyDir*  RmThyObj*  MvThyObj*

↑↑↑↑  ↑↑↑ ↓↓↓↓  ↓↓↓  <>   ><

Navigator: [kreitz; user; theories]

Scroll position : 0

List Scroll : Total 1,  Point 0,  Visible : 1
 ----------------------------------------------
 -> STM   FFF  not_over_and
 ----------------------------------------------
```

- **Insert a new term into the [lhs] slot** ⌜exists_uni(T; x.P[x])⌝
- **Enter its definition into [rhs]** ⌜∃x:T. P[x] ∧ (∀y:T. P[y] ⇒ y=x ∈ T)⌝
  – 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

DISP exists_uni_df

```
EdAlias exists_uni ::
  exists_uni(<T:T:*>;<x:var:*>.<P:P:*>)
  == exists_uni(<T>;<x>.<P>)
```

- **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

DISP exists_uni_df

```
EdAlias exists_uni ::
  ∃!<x:var:*>:<T:type:*>. <P:prop:*>)
  == exists_uni(<T>;<x>.<P>)
```

- **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**

compute addition

```
Compute1*  Compute5*  Compute10*  ComputeAll*

((3 * 4) - 5) + 6
```

- **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

compute intsqrt

```
Compute1*  Compute5*  Compute10*  ComputeAll*

TERMOF{intsqrt:o, \\v:l}
```

- **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**

  $\mapsto$ User has to complete the proof by calling other tactics
- **May not terminate**

  $\mapsto$ 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* $\ulcorner \ldots C \ldots \vdash C' \urcorner$ *where* C' α-*equal to* C

`Declaration`: *Prove* $\ulcorner \ldots x{:}T \ldots \vdash x \in T' \urcorner$ *where* T' α-*equal to* T

Variants: `NthHyp` i, `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*

Variants: `EqCD` , `EqHD` i, `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*

Variants: `EqTypeCD` , `EqTypeHD` i, `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** — `NthHyp` $\boxed{i}$
- **Names for newly created variables** — `New` `[`x`]` `(D 0)`
- **Type of some subterm in the goal** — `With` $\boxed{x\!:\!S\!\rightarrow\!T}$ `(MemD 0)`
- **Term to instantiate a variable** — `With` $\boxed{s}$ `(D 0)`
- **Selection from a number of alternatives** — `Sel` $\boxed{n}$ `(D 0)`
- **Universe level of a type** — `At` $\boxed{j}$ `(D 0)`
- **Dependency of a term instance $C[z]$ on a variable z** — `Using` $\boxed{[z,C]}$ `(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; . . . ; 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] *i*: *instantiate hypothesis i with terms* $t_1 \ldots t_n$
  - FHyp *i* [h_1;..;h_n]: *forward chain through hypothesis i*
    matching its antecedents against any of the hypotheses $h_1 \ldots h_n$
  - BHyp *i*: *backward chain through hypothesis i*
    matching its consequent against the conclusion of the proof
  - Backchain *bc_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 egdes 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'`**

Proving the Existence of an Integer Square Root

THM intsqrt

```
* top       ∀n:ℕ. ∃r:ℕ. r² ≤ n < (r+1)²

            BY allR

* 1         1. n : ℕ
            ⊢ ∃r:ℕ. r² ≤ n < (r+1)²

            BY NatInd 1

* 1 1       .....basecase.....
            ∃r:ℕ. r² ≤ 0 < (r+1)²

            BY With ⌜0⌝ (D 0) THEN Auto

* 1 2       .....upcase.....
            1. i : ℕ
            2. 0 < i
            3. r : ℕ
            4. r² ≤ i-1 < (r+1)²
            ⊢ ∃r:ℕ. r² ≤ i < (r+1)²

            BY Decide ⌜(r+1)² ≤ i⌝ THENW Auto

* 1 2 1     5. (r+1)² ≤ i
            ⊢ ∃r:ℕ. r² ≤ i < (r+1)²

            BY With ⌜r+1⌝ (D 0) THEN Auto'

* 1 2 2     5. ¬((r+1)² ≤ i)
            ⊢ ∃r:ℕ. r² ≤ 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` $t_1{=}t_2{\in}T$ *c*: *substitute* t_1 *by* t_2 *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`
  - `SubC`, `NthSubC`, `AddrC`, `SweepUpC`, `SweepDnC`, `DepthC`, `AllC`, `SomeC`, `FirstC`

- **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

	left	right	
`andE` i	$\Gamma,\ \underline{A \wedge B},\ \Delta \vdash G$ $\quad\Gamma,\ \underline{A},\ \underline{B},\ \Delta \vdash G$	$\Gamma \vdash \underline{A \wedge B}$ $\quad\Gamma \vdash \underline{A}$ $\quad\Gamma \vdash \underline{B}$	`andI`

`orE` i	$\Gamma,\ \underline{A \vee B},\ \Delta \vdash G$
	$\quad\Gamma,\ \underline{A},\ \Delta \vdash G$
	$\quad\Gamma,\ \underline{B},\ \Delta \vdash G$
		$\Gamma \vdash \underline{A \vee B}$
		$\quad\Gamma \vdash \underline{A}$
		$\Gamma \vdash \underline{A \vee B}$
		$\quad\Gamma \vdash \underline{B}$
			`orI1`
			
			`orI2`

`impE` i	$\Gamma,\ \underline{A \Rightarrow B},\ \Delta \vdash G$ $\quad\Gamma,\ \underline{A \Rightarrow B},\ \Delta \vdash \underline{A}$ $\quad\Gamma,\ \Delta,\ \underline{B} \vdash G$	$\Gamma \vdash \underline{A \Rightarrow B}$ $\quad\Gamma,\ \underline{A} \vdash \underline{B}$	`impI`
`notE` i	$\Gamma,\ \underline{\neg A},\ \Delta \vdash G$ $\quad\Gamma,\ \underline{\neg A},\ \Delta \vdash \underline{A}$	$\Gamma \vdash \underline{\neg A}$ $\quad\Gamma,\ \underline{A} \vdash \underline{\texttt{false}}$	`notI`
`exE` i	$\Gamma,\ \underline{\exists x{:}T.B},\ \Delta \vdash G$ $\quad\Gamma,\ \underline{x{:}T},\ \underline{B},\ \Delta \vdash G$	$\Gamma \vdash \underline{\exists x{:}T.B}$ $\quad\Gamma \vdash \underline{B[t/x]}$	`exI` t
`allE` i t	$\Gamma,\ \underline{\forall x{:}T.B},\ \Delta \vdash G$ $\quad\Gamma,\ \underline{\forall x{:}T.B},\ \underline{B[t/x]},\ \Delta \vdash G$	$\Gamma \vdash \underline{\forall x{:}T.B}$ $\quad\Gamma,\ \underline{x{:}T} \vdash \underline{B}$	`allI`

simple_prover: Matching and instantiation

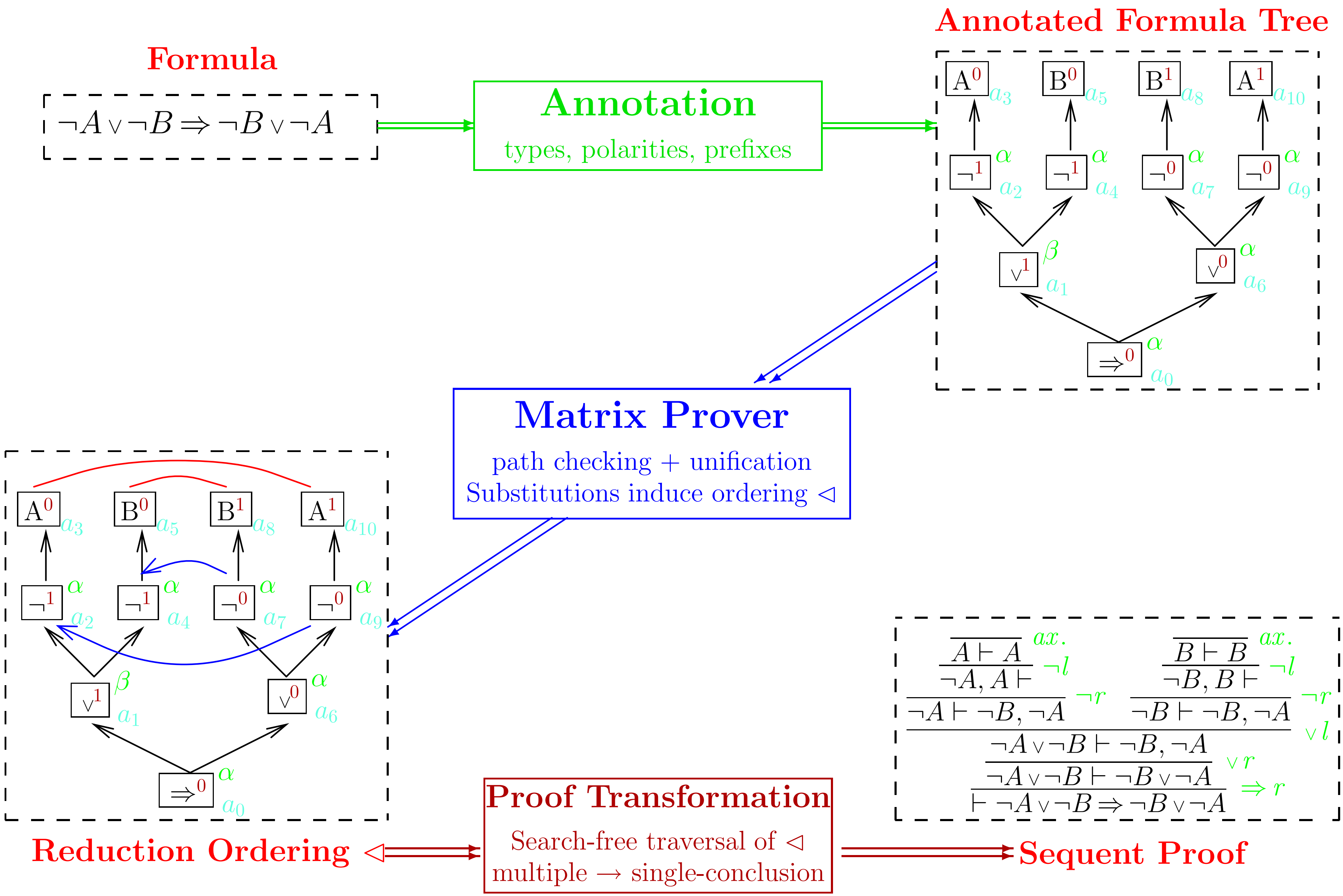
```
 let InstantiateAll =
    let InstAll_aux pos pf =
         let concl = conclusion pf
         and qterm = type_of_hyp pos pf          in
            let sigma = match_subAll qterm concl in
               let terms = map snd sigma         in
                  (allEon pos terms THEN (OnLastHyp hypothesis))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

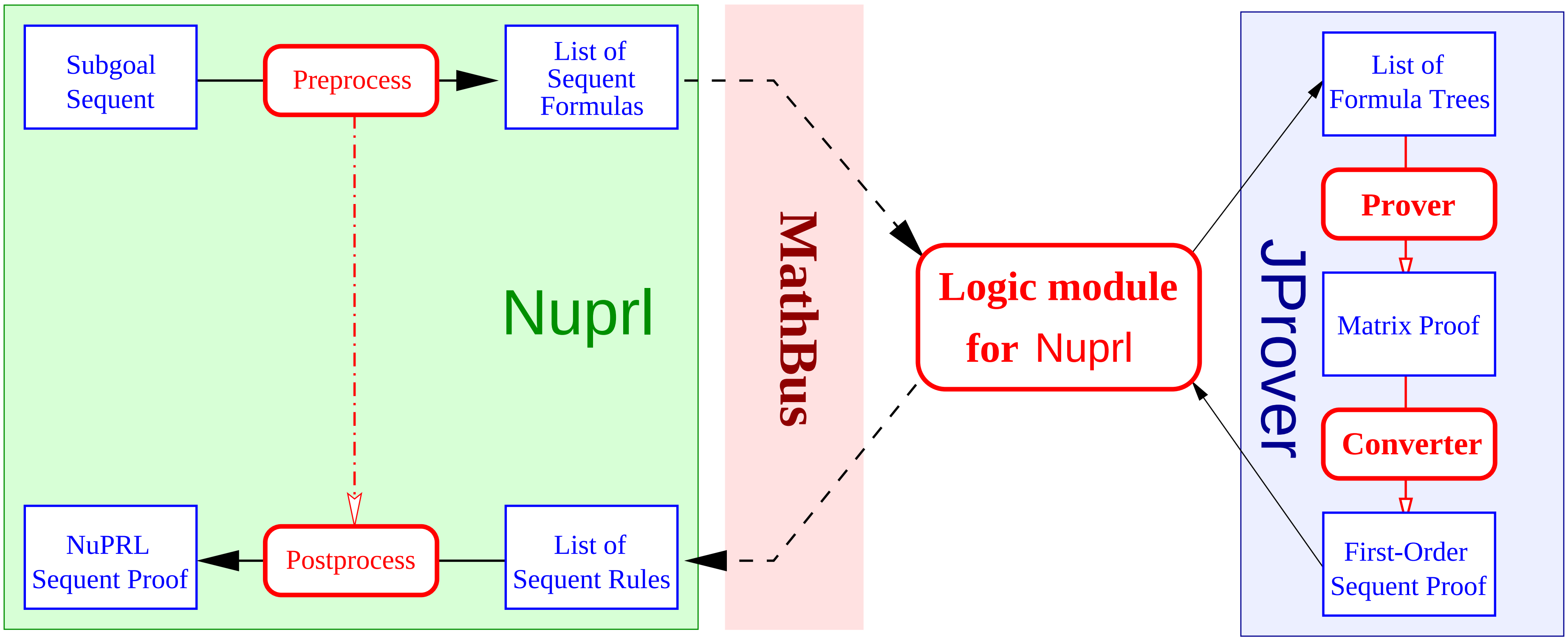
Integrating Complete Proof Search Procedures

- **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

JProver: Proof Methodology (Kreitz, Otten, Schmitt 1995–2000)

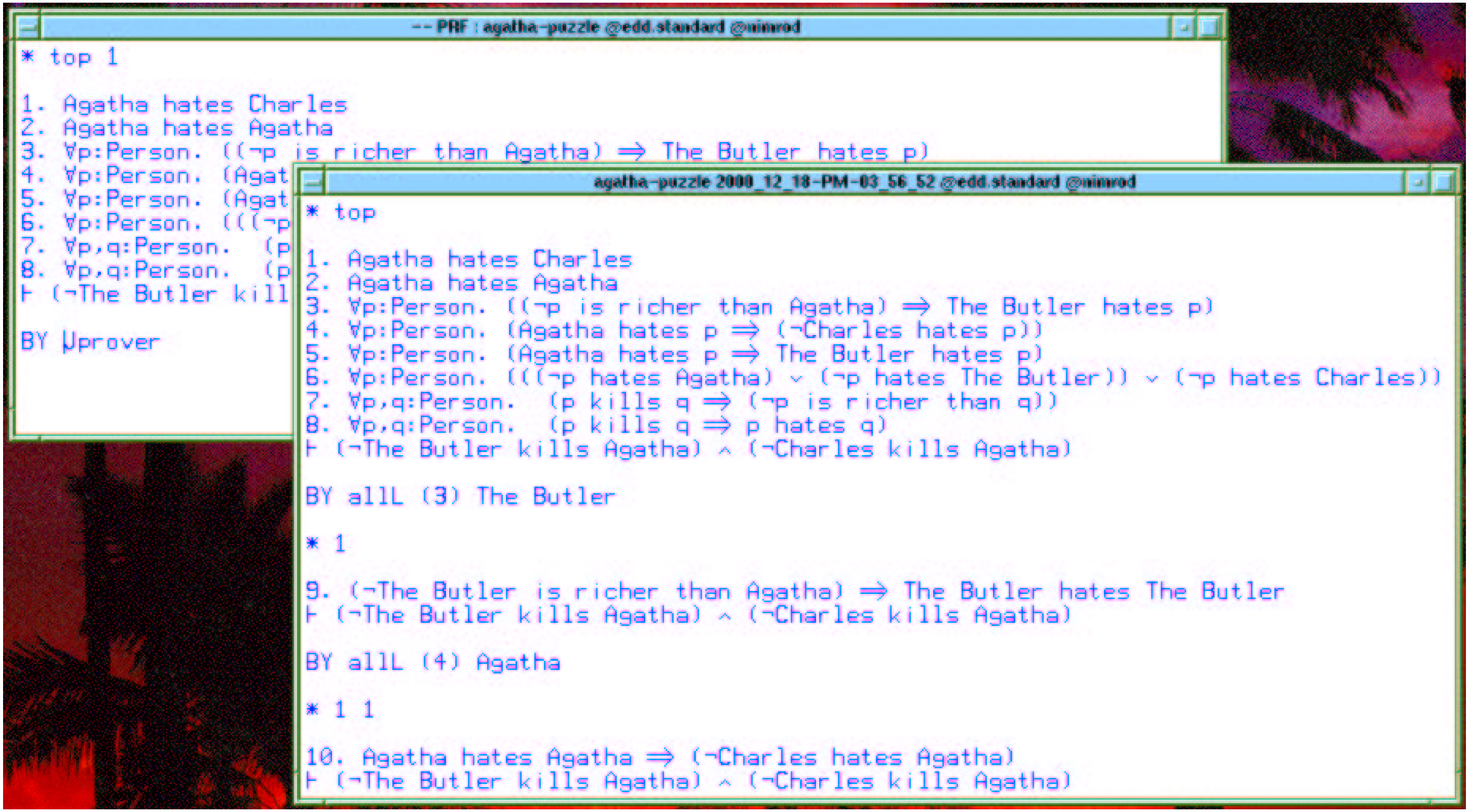


JProver: Integration Architecture (Schmitt, et. al 2001)



- 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) functions from labels to types**
  - $\{x_1{:}T_1;\ ..\ ;\ x_n{:}T_n\} \equiv$ `l:Labels` $\rightarrow$ `if l=`x_i `then` T_i `else Top`
  - $\{x_1{=}t_1;\ ..\ ;\ x_n{=}t_n\} \equiv$ λ`l.if z=`x_i `then` t_i `else ()`
  - $r.l \equiv (r\ l)$
- **Dependent Records** $\{x_1{:}T_1;\ x_2{:}T_2[x_1];\ ..\ ;\ x_n{:}T_n[x_1;..x_{n-1}]\}$
  - Type T_i may depend on value of components $x_1;..x_{i-1}$
  - Used for describing algebra, abstract data types, inheritance, ...
- **Use (dependent) intersection to formalize both**

$\{x{:}T\}$	$\equiv$	`z:Labels` $\rightarrow$ `if z=`x `then` T `else 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	$\equiv$	λ`z. if z=`l `then` t `else` r`.z`
$\{\}$	$\equiv$	λ`l.()`
$\{r;\ l{=}t\}$	$\equiv$	$r.l$`<-`t

$\leadsto$ **Subtyping $\{x_1{:}T\} \sqsubseteq \{x_1{:}T_1;\ x_2{:}T_2[x_1]\}$ 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 $\circ$:$M\times M\rightarrow M$ associative

- **Formalization as dependent product (Σ type)**

  `SemiGroup ≡ M:U × ∘:M×M→M × ∀x,y,z:M. x∘(y∘z) = (x∘y)∘z ∈ M`

  $\rightsquigarrow$ **semigroups represented as triples $(M,\circ,assoc\_pf)$**

- **Formalization via set types**

  `SemiGroupSig ≡ M:U × ∘:M×M→M`

  `SemiGroup ≡ {sg:SemiGroupSig | ∀x,y,z:M`$_{sg}$`. x∘`$_{sg}$`(y∘`$_{sg}$`z) = (x∘`$_{sg}$`y)∘`$_{sg}$`z ∈ M`$_{sg}$`}`

  $\rightsquigarrow$ **tedious to access components or use associativity in proofs**

- **Formalization via dependent records**

  `SemiGroupSig ≡ {M:U; ∘:M×M→M}`

  `SemiGroup ≡ {SemiGroupSig; assoc: ↓(∀x,y,z:M. x∘(y∘z) = (x∘y)∘z ∈ M)}`

  $\rightsquigarrow$ **Accessing components and properties straightforward**

  $\rightsquigarrow$ **Type squashing suppresses explicit proof component**

  $\rightsquigarrow$ **Subtyping relation `SemiGroup ⊑ SemiGroupSig` easy to prove**

Formal Algebra: Monoids and Groups

- **Monoid: semigroup with identity**

$$\texttt{MonoidSig} \equiv \{\ \texttt{SemiGroupSig; e:M}\}$$
$$\texttt{Monoid} \equiv \{\ \texttt{SemiGroup; MonoidSig; id: } \downarrow(\forall \texttt{x:M. e}\circ\texttt{x=x} \in \texttt{M})\}$$

$\leadsto$ **natural use of multiple inheritance**

- **Group: monoid with inverse**

$$\texttt{GroupSig} \equiv \{\ \texttt{MonoidSig; }^{-1}\texttt{:M}\rightarrow\texttt{M}\}$$
$$\texttt{Group} \equiv \{\ \texttt{Monoid; GroupSig; inv: } \downarrow(\forall \texttt{x:M.x}\circ\texttt{x}^{-1}\texttt{=e} \in \texttt{M})\}$$

$\leadsto$ **refinement hierarchy follows directly from definitions**

$$\begin{array}{ccc} \texttt{SemiGroup} & \sqsubseteq & \texttt{SemiGroupSig} \\ \sqsubseteq & & \sqsubseteq \\ \texttt{Monoid} & \sqsubseteq & \texttt{MonoidSig} \\ \sqsubseteq & & \sqsubseteq \\ \texttt{Group} & \sqsubseteq & \texttt{GroupSig} \end{array}$$

Formalization: Abstract Data Types

- **Abstract Data Type for stacks over a type T**

```
TYPES         Stack
OPERATORS     empty:   Stack
              push:    Stack×T → Stack
              pop:     {s:Stack|s≠empty} → Stack×T
AXIOMS        pushpop: ∀s:Stack.∀t:T. pop(push(s,a)) = (s,a)
```

- **Formalization**
  - Dependent products unsuited for same reason as above
  - Dependent records lead to "natural formalization"

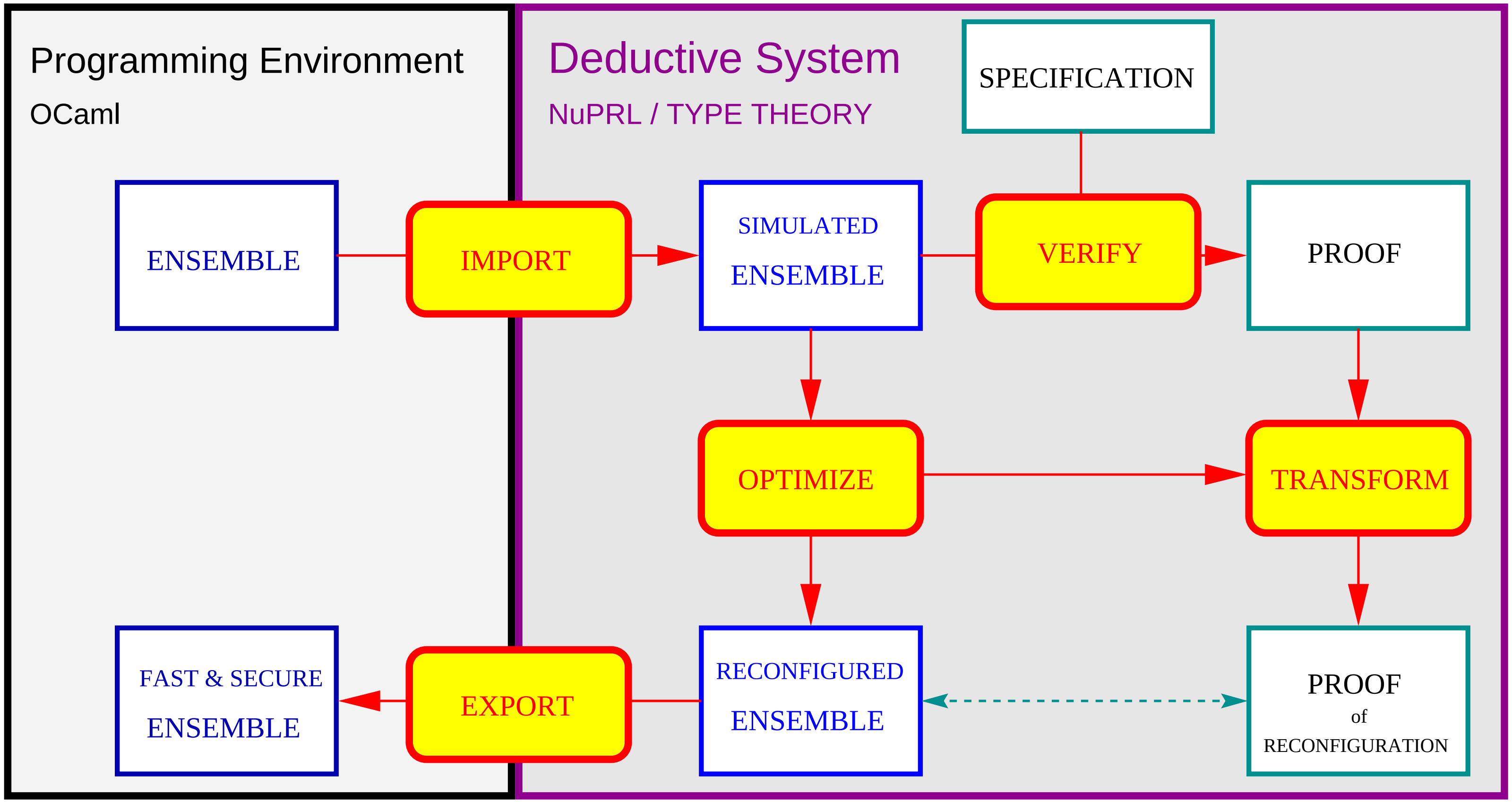
```
STACKSIG(T)  ≡ { Stack:U
               ; empty:   Stack
               ; push:    Stack×T → Stack
               ; pop:     {s:Stack|s≠empty} → Stack×T}
STACK(T)     ≡ {STACKSIG(T); pf: ↓(∀s:Stack.∀t:T.pop(push(s,a))=(s,a) ∈M)}
```

- **Formalizing the implementation of stacks through lists**

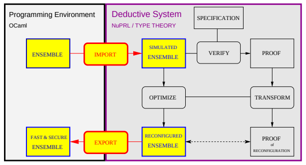
```
list-as-stack(T) ≡ { Stack = T list
                   ; empty = []
                   ; push  = λs,t. t::s
                   ; pop   = λs. <hd(s),tl(s)> }
```

$\leadsto$ list-as-stack(T) $\in$ STACK(T) **easy to prove**

How to approach large application examples?
Verify and optimize distributed systems (Ensemble)



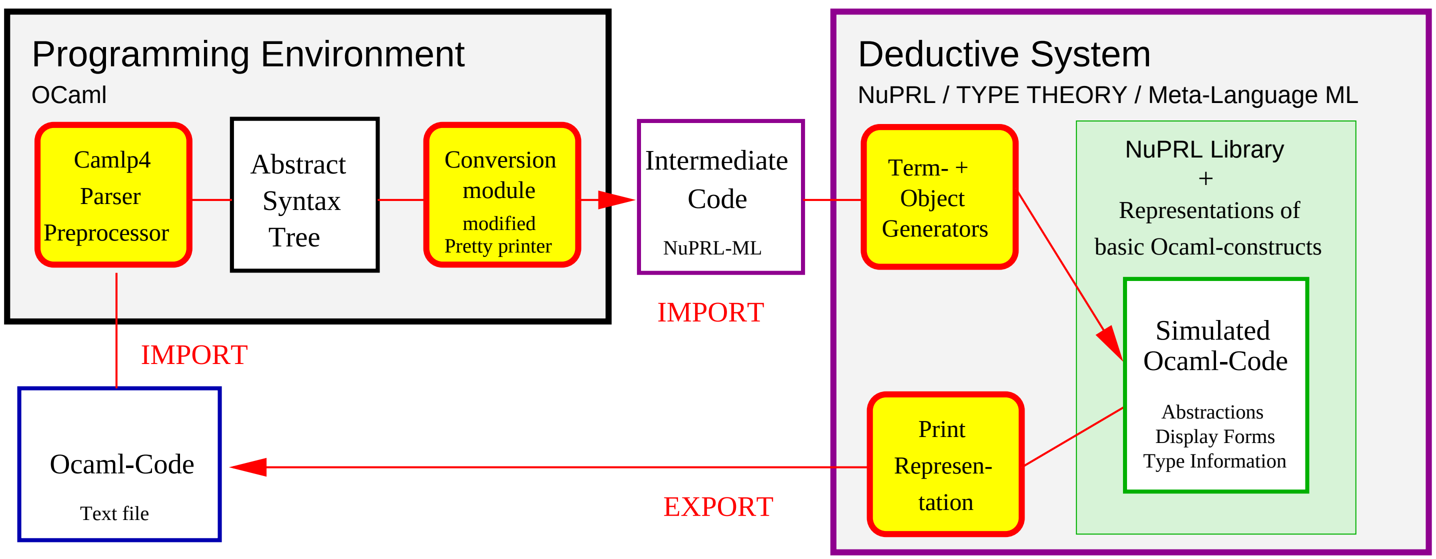
– Formalize semantics of implementation language
– Build tactics for verification of protocols and system configurations
– Build tactics that optimize performance of configured systems

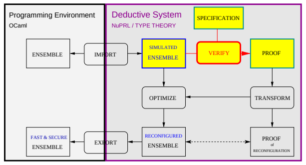


Embedding system code into Nuprl

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**



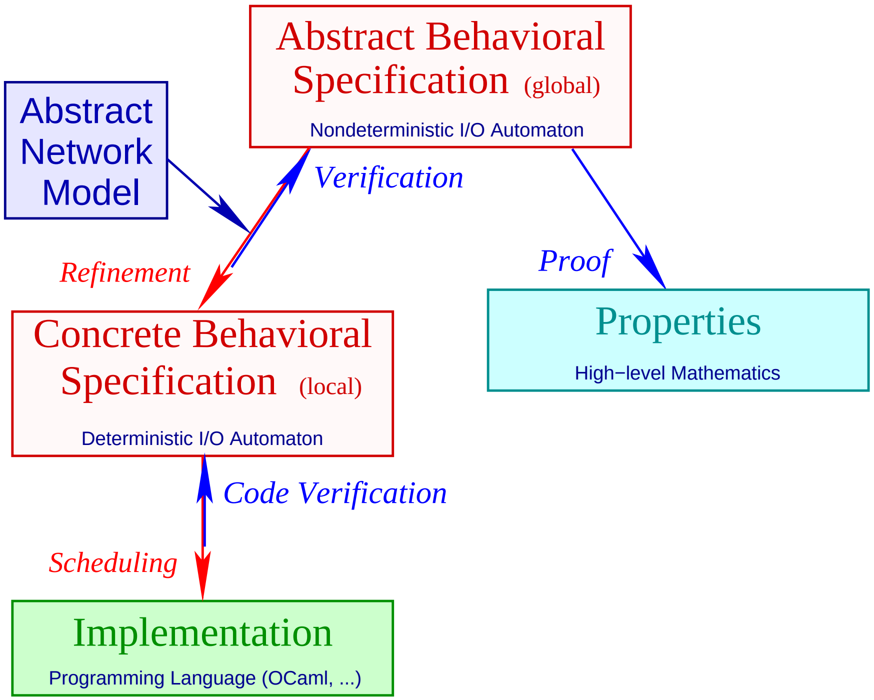


Formalize system specification and code

e.g. *"Messages are received in the same order in which they were sent"*

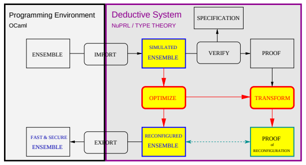
- *"Messages may be appended to global event queue and removed from its beginning"*
- *"Messages whose sequence number is too big will be buffered"*
- ENSEMBLE *module* `Pt2pt.ml`*: 250 lines of* OCAML *code*

All levels represented in type theory

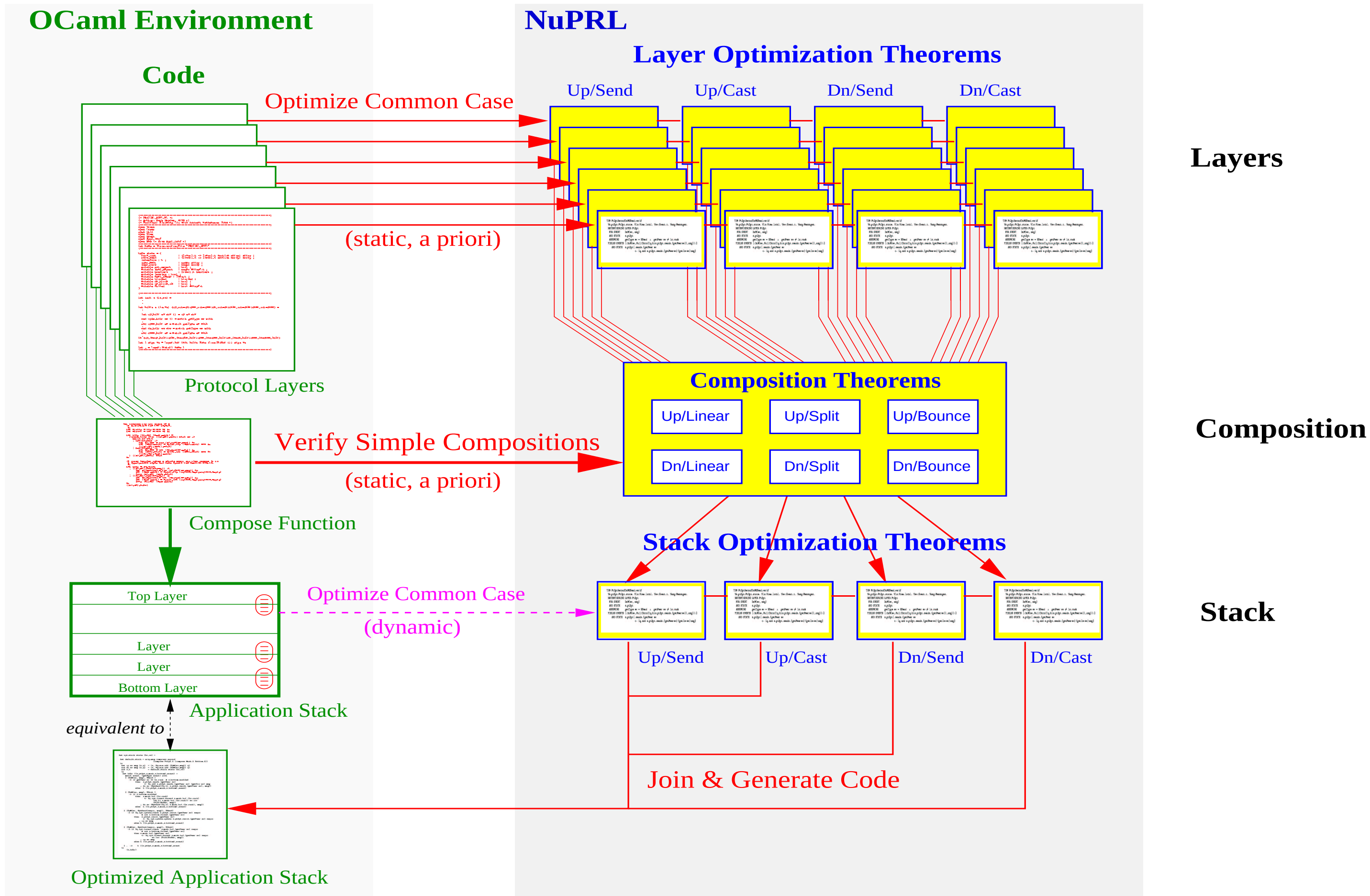


Verification methodology

- Verify component specifications
  (benign assumptions — subtle bug detected)
- Verify systems by composition
  (IOA-composition preserves safety properties)
- Weave aspects
- Verify code



Optimization of Protocol Stacks
Prove and Compose Optimization Theorems

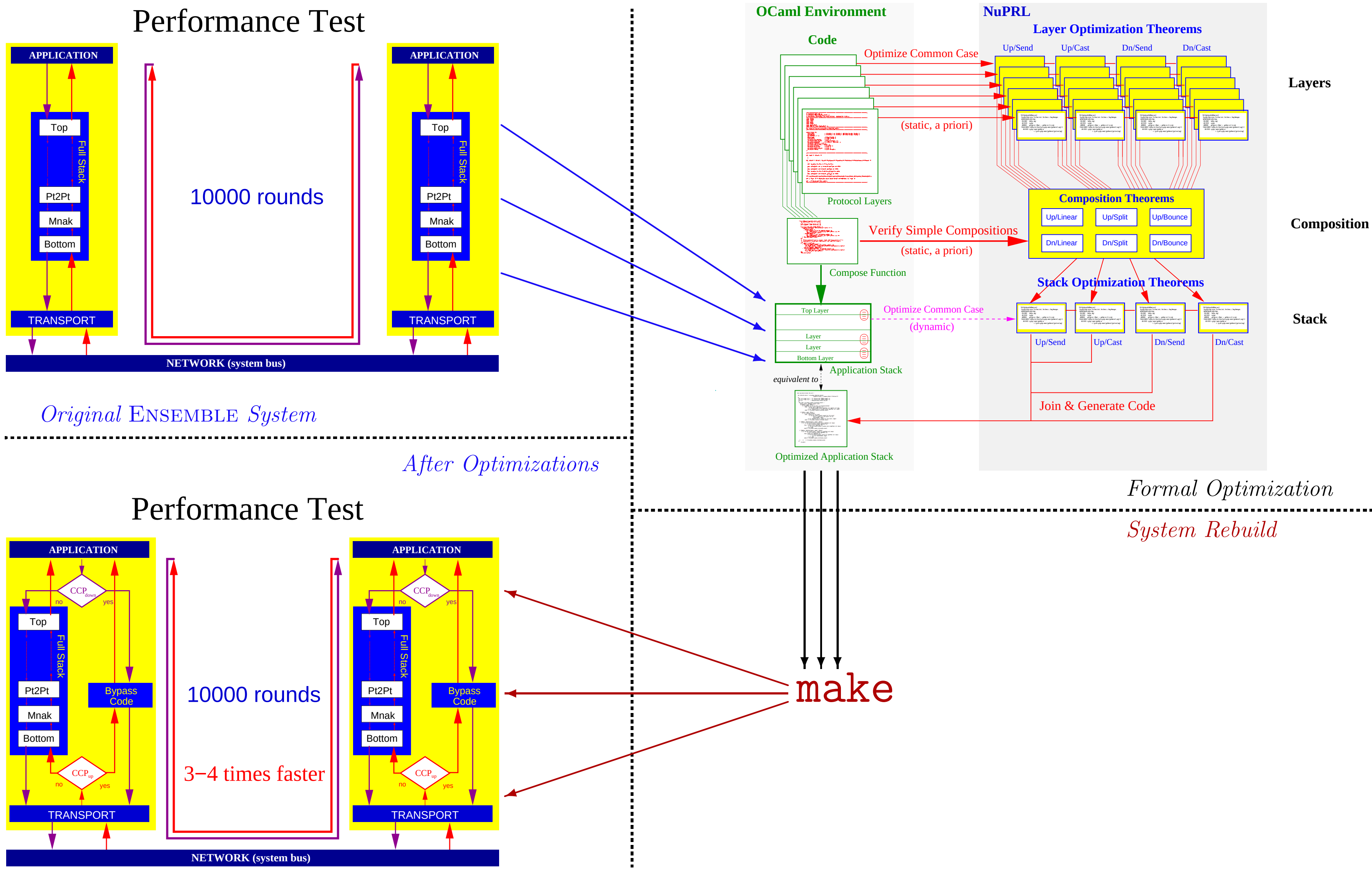


1. Use known optimizations of micro-protocols — A priori: Ensemble + Nuprl experts
2. Compose into optimizations of protocol stacks — automatic: application designer
3. Integrate message header compression — automatic: ⋮
4. Generate code from optimization theorems and reconfigure system — automatic: ⋮

Fast, error-free, independent of programming language **speedup factor 3-10**

DEMO: Optimizing a 24-layer Protocol Stack

Top :: Heal :: Switch :: Migrate :: Leave :: Inter :: Intra :: Elect :: Merge :: Slander :: Sync :: Suspect :: Stable :: Vsync :: Partial_appl :: Total :: Collect :: Local :: Frag :: Pt2ptw :: Mflow :: Pt2pt :: Mnak :: Bottom



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 ↔ library contents)
    - · relations between different formal theories

  ...from within the logic

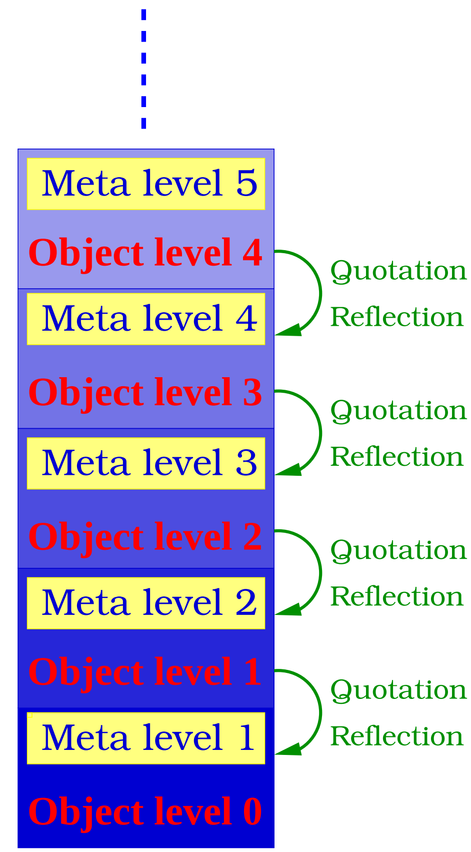
- **Extremely powerful, but little utilization**

- **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"



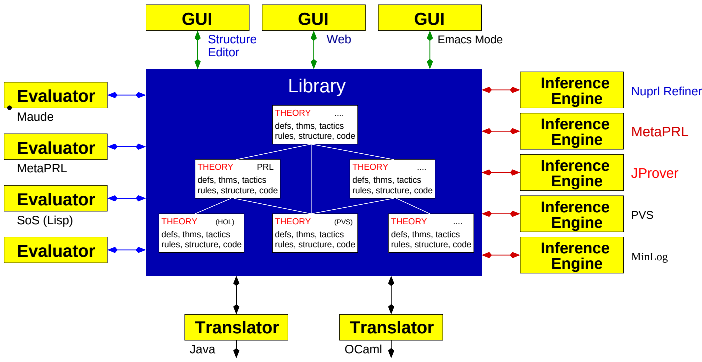
- **Link object logic and meta-logic**
  – Embed object level terms using quotation (operator)
  – Embed object level provability using reflection rule

$$\Gamma \vdash_{i+1} A \qquad \textbf{by}\ \texttt{reflection}\ i$$
$$\vdash_i \exists \texttt{p:Proof}_i.\ \texttt{goal(p) = } \lceil \Gamma \vdash_{i+1} A \rceil$$

- **Use same reasoning apparatus for object and meta level**

Digital Libraries of Formal Algorithmic Knowledge

- **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

$\Downarrow$

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, ..)