CS 671 Automated Reasoning

Extending Nuprl's Type Theory

1. **Design Decisions for Nuprl's Type Theory**
2. **Product, Union, and List Types**
3. **The Curry-Howard Isomorphism, formally**
4. **Empty and Unit Types**

Design Decisions for Nuprl's Type Theory

- **Syntax:**
  - Expressions will be represented in a uniform term syntax
  - Term display is independent of the internal syntax
- **Semantics:**
  - Semantics models proof, not denotation
  - Semantics is based on judgments and lazy evaluation of noncanonical terms
  - Judgments concern typehood, type equality, membership, and typed equality
- **Proof Theory:**
  - Proofs proceed by applying sequent-style refinement rules
  - A judgment "*t is a member of T*" is represented as T $\lfloor$ext $t\rfloor$
  - Propositions are represented as types

    Basic propositions have `Ax` as only member
  - Typehood is represented by a cumulative hierarchy of universes

See Appendix A of the Nuprl 5 manual for details

Cartesian Products: Building Data Structures

Syntax:

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

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

Evaluation:

$$\frac{e \downarrow \langle e_1, e_2 \rangle \qquad u[e_1, e_2 / x, y] \downarrow val}{\mathsf{let}\ \langle x, y \rangle = e\ \mathsf{in}\ u \downarrow val}$$

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

See Appendix A.3.2 and the library theory `core_2` for further details

Lists: Basic Data Containers

Syntax:

Canonical: T list, [], e_1: : e_2

Noncanonical: list_ind(e; $base$; x, l, f_{xl}. up)

Evaluation:

$$\frac{e \downarrow [\,] \qquad base \downarrow val}{\text{list\_ind}(e;\ base;\ x, l, f_{xl}.\ up) \downarrow val}$$

$$\frac{e \downarrow e_1::e_2 \qquad up[e_1, e_2, \text{list\_ind}(e_2;\ base;\ x, l, f_{xl}.up) \ /\ x, , l, f_{xl}] \downarrow val}{\text{list\_ind}(e;\ base;\ x, l, f_{xl}.\ up) \downarrow val}$$

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^-], ...

See Appendix A.3.10 and the library theory `list_1` for further details

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:

$$\frac{e \downarrow \mathsf{inl}(e') \qquad\qquad u[e' / x] \downarrow val}{\mathsf{case}\ \ e\ \ \mathsf{of}\ \ \mathsf{inl}(x) \mapsto u\ \ |\ \ \mathsf{inr}(y) \mapsto v\ \ \downarrow val}$$

$$\frac{e \downarrow \mathsf{inr}(e') \qquad\qquad v[e' / y] \downarrow val}{\mathsf{case}\ \ e\ \ \mathsf{of}\ \ \mathsf{inl}(x) \mapsto u\ \ |\ \ \mathsf{inr}(y) \mapsto v\ \ \downarrow val}$$

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: ——

See Appendix A.3.3 for further details

The Curry-Howard Isomorphism, formally

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

See the library theory `core_1` for further details

Empty Type void

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: ——

See Appendix A.3.6 and Section 3 of the 1993 CS611 notes for further details

Warning: rules for **void** allows proving semantical nonsense like

x:void ⊢ 0=1 ∈ 2 or ⊢ void⟶2 type

Unit: one element 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