CSci 8980, Fall 2012
Specifying and Reasoning About Computational Systems
Reasoning about Specifications

Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

Lectures in Fall 2012

Specifications as Definitions

We want to treat program clauses as *definitions* of relations

Computational interpretation gives only one half of this reading

For example, given

```
append nil L L.
append (X::L1) L2 (X::L3) :- append L1 L2 L3.
```

we can derive goals like

```
?- append (1::nil) (2::nil) (1::2::nil).
```

but not

```
?- append (1::nil) nil nil => false.
```

Deducing negative properties from definitions can be important to meta-theoretic reasoning

Of course, validating negative properties depends on a "closed-world" assumption

Extensional Interpretation of Universals

The specification logic treats universals intensionally, i.e. proof must be generic and independent of domain

If we interpret specifications in an "if and only if" fashion, extensional treatment may also be useful

For example, given

```
nat z.
nat (s N) :- nat N.

plus z L L.
plus (s N) M (s P) :- plus N M P.
```

we want to be able to prove not only

```
pi L\ (plus z L L)
```

but also the following

```
pi L\ (nat L) => (plus L z L)
```


Proving Subject Reduction (Case Study)

Call-by-name evaluation and typing were rendered into *hohh* specifications as follows

```
eval (abs T R) (abs T R).
eval (app T1 T2) V :-
      eval T1 (abs R), eval (R T2) V.

of (app T1 T2) Ty2 :-
   of T1 (Ty1 --> Ty2), of T2 Ty1.
of (abs Ty1 R) (Ty1 --> Ty2) :-
   pi x\ (of x Ty1) => (of (R x) Ty2).
```

In this setting, we want to be able to prove

$$\forall \texttt{e}\, \forall \texttt{v}\, \forall \texttt{ty}\, ((\texttt{eval e v}) \wedge (\texttt{of e ty}) \supset (\texttt{of v ty}))$$

This reasoning task depends on both the closed world reading and the extensional treatment of universals

Informal Proof of Subject Reduction

At the outermost level, we use an induction on the derivation of `eval e v` in the specification logic

Then we consider the cases for the clause used at the end of this derivation

First Case:

The clause used at the end is

```
eval (abs T R) (abs T R).
```

Here *e* and *v* are identical and it follows immediately that the same typing derivations hold for both

Note: The case analysis is leading us to consider instantiations for the eigenvariables *e* and *v*

Informal Proof of Subject Reduction (Second Case)

Here, `e` must have the form `(app e1 e2)` and there must be shorter derivations for

(1) `(eval e1 (abs t r))` and (2) `(eval (r e2) v)`

Since `(of (app e1 e2) ty)` is also derivable

(3) `(of e1 (ty1 --> ty))` is derivable for some `ty1`

(4) `(of e2 ty1)` is derivable for the same `ty1`

By induction on (1) and using (3), it follows that `(of (abs ty1 r) (ty1 --> ty))` is derivable

But then `(pi x\ (of x ty1) => (of (r x) ty))` must be derivable in the specification logic

Using (4) and the instantiation principle for that logic, `(of (r e2) ty)` must be derivable

Using the induction hypothesis now with (2) it follows that `(of v ty)` is derivable

Desiderata for Reasoning Logic

To be able to formalize the style of reasoning illustrated by the subject reduction argument, we need a logic that

- Has the capability to embed definitions in a way that permits case analysis based on a closed world assumption
- Understands the properties of the specification logic and allows these to be used in reasoning
- Allows for the instantiation of eigenvariables to support extensional interpretations of universal quantifiers
- Supports inductive (and perhaps also co-inductive) arguments based on definitions

To realize these requirements, we will introduce a logic of (fixed-point) definitions

Note: We will implicitly assume a simply typed setting with no quantification over predicate types

Universal versus Generic Quantification

The reasoning logic needs also a form of universal quantification whose every instance is proved uniformly

- Such a "generic" reading is what was intended of the universal quantifier in the specification logic, e.g. consider

```
of (abs Ty1 R) (Ty1 --> Ty2) :-
   pi x\ (of x Ty1) => (of (R x) Ty2).
```

- Actually, we will also need a *distinctness of binding* property for our generic quantifiers

  For example, in most logics, the following is derivable

  $$\forall x\, \forall y\, (P\ x\ y) \supset \forall z\, (P\ z\ z)$$

  This can be problematic if universal quantification is used in treating names, e.g. in an encoding of the π-calculus

Comment: the latter issue becomes relevant only when we also have to reason about formulas (premises) in *negative* contexts

The ∇ (Nabla) Quantifier

This is a new quantifier for representing generic quantification

The informal meaning of $\nabla x\ B$:

> *B has a proof that does not depend on the form of x but the proof can use its freshness*

In a minimal formalization of this idea, judgements (i.e. formulas) are decorated with a local context of parameters

The rules for the ∇ quantifier then are the following

$$\frac{\Sigma; \Gamma, ((a, \sigma) \rhd B[a/x]) \vdash \Delta}{\Sigma; \Gamma, (\sigma \rhd \nabla x\ B) \vdash \Delta}\ \nabla\mathcal{L} \qquad \frac{\Sigma; \Gamma \vdash (a, \sigma \rhd B[a/x])}{\Sigma; \Gamma \vdash (\sigma \rhd \nabla x\ B)}\ \nabla\mathcal{R}$$

where a is a parameter that is new to the sequent

Also, local contexts are treated as binders: two (extended) judgements match if they do so under renaming

Note: We will consider only an intuitionistic version of the logic

Quantifier Rules with Local Contexts

The usual quantifier rules must now be adjusted to account for the scopes of ∇ quantifiers as well

For essential existential quantification, parameters from local contexts must be permitted in forming valid terms

$$\frac{\Sigma; \Gamma \vdash (\sigma \rhd B[t/x]) \qquad t \text{ is a } (\Sigma \cup \sigma)\text{-term}}{\Sigma; \Gamma \vdash (\sigma \rhd \exists x\ B)}\ \exists R$$

$$\frac{\Sigma; \Gamma, (\sigma \rhd B[t/x]) \vdash \Delta \qquad t \text{ is a } (\Sigma \cup \sigma)\text{-term}}{\Sigma; \Gamma, (\sigma \rhd \forall x\ B) \vdash \Delta}\ \forall L$$

Considering the local context in the instantiation/generalization term takes into account the ∇ quantifiers with larger scope

Quantifier Rules with Local Contexts (Continued)

For essential universal quantifiers, we must also take into account that eigenvariables may be instantiated later

To properly constrain such instantiations, we use raising

Formally, the rules become

$$\frac{h, \Sigma; \Gamma \vdash (\sigma \rhd B[(h\ \sigma)/x])}{\Sigma; \Gamma \vdash (\sigma \rhd \forall x\ B)}\ \forall R$$

$$\frac{h, \Sigma; \Gamma, (\sigma \rhd B[(h\ \sigma)/x]) \vdash \Delta}{\Sigma; \Gamma, (\sigma \rhd \exists x\ B) \vdash \Delta}\ \exists L$$

Here, h is a new eigenvariable and $(h\ \sigma)$ denotes $(h\ x_1, \ldots, x_n)$ if σ is $x_1, \ldots, x_n$

Raising introduces the permitted dependencies where eigenvariable instantiations will be limited to closed terms

Local Contexts and Propositional Rules

Essentially, the local context must distribute over the connective

$$\frac{}{\Sigma; \Gamma \vdash (\sigma \rhd \top)}\ \top R \qquad \frac{}{\Sigma; (\sigma \rhd \bot), \Gamma \vdash \Delta}\ \bot L$$

$$\frac{\Sigma; \Gamma, (\sigma \rhd B_1) \vdash \Delta \quad \Sigma; \Gamma, (\sigma \rhd B_2) \vdash, \Delta}{\Sigma; \Gamma, (\sigma \rhd B_1 \vee B_2) \vdash \Delta}\ \vee L$$

$$\frac{\Sigma; \Gamma \vdash (\sigma \rhd B_1)}{\Sigma; \Gamma \vdash (\sigma \rhd B_1 \vee B_2)}\ \vee R_1 \qquad \frac{\Sigma; \Gamma \vdash (\sigma \rhd B_2)}{\Sigma; \Gamma \vdash (\sigma \rhd B_1 \vee B_2)}\ \vee R_2$$

$$\frac{\Sigma; \Gamma, (\sigma \rhd B_1) \vdash \Delta}{\Sigma; \Gamma, (\sigma \rhd B_1 \wedge B_2) \vdash \Delta}\ \wedge L_1 \qquad \frac{\Sigma; \Gamma, (\sigma \rhd B_2) \vdash \Delta}{\Sigma; \Gamma, (\sigma \rhd B_1 \wedge B_2) \vdash \Delta}\ \wedge L_2$$

$$\frac{\Sigma; \Gamma \vdash (\sigma \rhd B_1) \quad \Sigma; \Gamma \vdash (\sigma \rhd B_2)}{\Sigma; \Gamma \vdash (\sigma \rhd B_1 \wedge B_2)}\ \wedge R$$

$$\frac{\Sigma; \Gamma \vdash (\sigma \rhd B_1) \quad \Sigma; \Gamma, (\sigma \rhd B_2) \vdash \Delta}{\Sigma; \Gamma, (\sigma \rhd B_1 \supset B_2) \vdash \Delta}\ \supset L$$

$$\frac{\Sigma; \Gamma, (\sigma \rhd B_1) \vdash (\sigma \rhd B_2)}{\Sigma; \Gamma \vdash (\sigma \rhd B_1 \supset B_2)}\ \supset R$$

The Identity Rules

These are the only rules in which we have to "match up" different formulas

Here, we implicitly use renaming by requiring the local context "binders" to be identical

$$\frac{}{\Sigma; \Gamma, (\sigma \rhd B) \vdash (\sigma \rhd B)}\ \mathit{init}$$

$$\frac{\Sigma; \Gamma_1 \vdash (\sigma \rhd B) \qquad \Sigma; \Gamma_2, (\sigma \rhd B) \vdash \Delta}{\Sigma; \Gamma_1, \Gamma_2 \vdash \Delta}\ \mathit{cut}$$

Notice though that

- the binders have to be of equal length, and
- occurrences in formula and location in the binder must be correlated

Adding Definitions to the Logic

Predicate constants will be interpreted via clauses of the form

$$\forall \vec{x}.H \stackrel{\triangle}{=} B$$

where H is a (rigid) atomic formula called the *head* of the clause, and B is a (first-order) formula called the *body*

These clauses seem similar to program clauses but there are some important differences

- Collections of clauses (definitions) will *parameterize* the logic and *will not* be part of a sequent
- The body is not syntactically limited in the way program clauses are
- The logic will give a set $\mathcal{D}$ of such clauses, called a definition, a stronger *fixed-point* interpretation
- To ensure consistency of such a logic, definitions will be required to satisfy some "stratification conditions"

A Stratification Condition for Definitions

Let *lvl* be a function that assigns natural numbers or "levels" to predicate constants

We extend this assignment into one for arbitrary formulas

- $\mathit{lvl}(\top) = \mathit{lvl}(\bot) = 0$
- $\mathit{lvl}(p\ t_1\ \ldots\ t_n) = \mathit{lvl}(p)$
- $\mathit{lvl}(B \wedge C) = \mathit{lvl}(B \vee C) = \max(\mathit{lvl}(B), \mathit{lvl}(C))$
- $\mathit{lvl}(B \supset C) = \max(\mathit{lvl}(B) + 1, \mathit{lvl}(C))$
- $\mathit{lvl}(\forall x\ B) = \mathit{lvl}(\exists x\ B) = \mathit{lvl}(\nabla x\ B) = \mathit{lvl}(B)$

A definition $\mathcal{D}$ is then *good* if it satisfies the following condition

There is some assignment lvl to predicates such that $\forall \vec{x}.H \stackrel{\triangle}{=} B \in \mathcal{D}$ *implies* $\mathit{lvl}(H) \leq \mathit{lvl}(B)$

Essentially, this amounts to a monotonicity condition where the entire definition of a predicate is fixed in one go

Introduction Rules for Atomic Formulas

We must first make precise some notions needed to talk about matching the head of a definition with atomic judgements

Definition

The raising of $\forall x_1 \ldots \forall x_n . H \stackrel{\triangle}{=} B$ over σ is given as follows

$$\forall h_1 \ldots \forall h_n . \sigma \rhd (H\theta) \stackrel{\triangle}{=} \sigma \rhd (B\theta)$$

where $\theta = \{(h_1\ \sigma)/x_1, \ldots, (h_n\ \sigma)/x_n\}$ for new variables $h_1, \ldots, h_n$ of appropriate types

Definition

The application of a substitution θ to $x_1, \ldots, x_n \rhd B$, denoted by $(x_1, \ldots, x_n \rhd B)\theta$, is given as follows

$(x_1, \ldots, x_n \rhd B)\theta = y_1, \ldots, y_n \rhd B'$ if $(\lambda x_1 \ldots \lambda x_n\ B)\theta = \lambda y_1 \ldots \lambda y_n\ B'$

Definition

We write $\Sigma\theta$ for the signature that results from Σ by removing variables in the domain of θ and adding those in its range

Introduction Rules for Atomic Formulas (Continued)

Given a set of clauses $\mathcal{D}$, let $\sigma \rhd \mathcal{D}$ denote the raising of each member of $\mathcal{D}$ over σ

Then the introduction rules for atomic formulas are the following

$$\frac{\Sigma; \Gamma \vdash (\sigma \rhd B)\theta}{\Sigma; \Gamma \vdash \sigma \rhd A}\ def\mathcal{R} \qquad \sigma \rhd A = H\theta, \forall \vec{x}.H \stackrel{\triangle}{=} B \in \sigma \rhd \mathcal{D}$$

$$\frac{\{\Sigma\theta; \Gamma\theta, (\sigma \rhd B)\theta \vdash C\theta \mid (\sigma \rhd A)\theta = H\theta, \forall \vec{x} H \stackrel{\triangle}{=} B \in \sigma \rhd \mathcal{D}\}}{\Sigma; \Gamma, \sigma \rhd A \vdash C}\ def\mathcal{L}$$

Comments

- $def\mathcal{R}$ has the flavour of backchaining, $def\mathcal{L}$ of case analysis
- $def\mathcal{L}$ considers substitutions for eigenvariables
- $def\mathcal{L}$ must consider all clauses and all substitutions
- For cut-free proofs, it suffices to consider only complete sets of unifiers

Using Definitions (Example)

Suppose that the set of definition clauses is

$$p\ a \stackrel{\triangle}{=} \top, \quad p\ b \stackrel{\triangle}{=} \top, \quad q\ a \stackrel{\triangle}{=} \top, \quad q\ b \stackrel{\triangle}{=} \top, q\ c \stackrel{\triangle}{=} \top$$

Then $; \vdash \cdot \rhd \forall x\ p\ x \supset q\ x$ is derivable

In fact, the following is a derivation for it:

$$\dfrac{\dfrac{\dfrac{}{; \cdot \rhd \top \vdash \cdot \rhd \top}}{; \cdot \rhd \top \vdash \cdot \rhd q\ a}\ def\mathcal{R} \quad \dfrac{\dfrac{}{; \cdot \rhd \top \vdash \cdot \rhd \top}}{; \cdot \rhd \top \vdash \cdot \rhd q\ b}\ def\mathcal{R}}{\dfrac{Y; \cdot \rhd p\ Y \vdash \cdot \rhd q\ Y}{; \vdash \cdot \rhd \forall x\ p\ x \supset q\ x}\ \forall R; \supset R}\ def\mathcal{L}$$

Notice that the $def\mathcal{L}$ rule instantiates an eigenvariable

Mixing Finite Success with Finite Failure

A possible style of reasoning in the logic of definitions

- Work as usual with a formula on the right of the sequent
- When an atom becomes available on the left
  - simplify it immediately to the $\top$, computing the corresponding substitution for the eigenvariables
  - prove the formula on the right for each of these substitutions

For this style to work, some requirements have to be satisfied

- Only invertible rules should be applicable on the left
- Every "solution path" for an atom on the left must be finite

Restricting formulas to be proved can yield the first property and limiting specifications yields the second

Mixing Finite Success with Finite Failure (Continued)

The property we want of formulas in the left of a sequent is obtained by a kind of stratification

Specifically, we identify two levels of formulas

Level 0: $G ::= \top \mid \bot \mid A \mid G \wedge G \mid G \vee G \mid \exists x\ G \mid \nabla x\ G$

Level 1: $D ::= \top \mid \bot \mid A \mid D \wedge D \mid D \vee D \mid G \supset D$
$\exists x\ D \mid \nabla x\ D \mid \forall x\ D$

Also there are restrictions on the Level 0 and Level 1 atoms:

- Level 0 atoms must be defined by level 0 formulas
- Level 1 atoms can be defined by level 0 or level 1 formulas

Notice that Level 0 formulas encompass goals in Horn clause logic and that level 1 formulas include level 0 ones

The end goal is always to prove a sequent of the form

$$; \vdash \cdot \rhd D$$

where D is a level 1 formula

Mixing Finite Success with Finite Failure (Continued)

An observation concerning sequents seen by the prover:

> *Only (decorated) G formulas appear on the left and all the rules applicable to them are invertible*

Also, there is a correspondence between uniform provability and the application of left rules to (decorated) *G* formulas

Thus, proof search for $; \vdash \sigma \rhd G \supset D$ can be structured as follows

Step 1 Run a logic programming interpreter with $\sigma \rhd G$, treating eigenvariables as logic variables

Step 2 Collect *all* answer substitutions in Step 1 and attempt to prove *D* under each

If there are *no* answers in Step 1, the prover succeeds immediately

The Bedwyr System

An implementation in OCaml of the described proof search strategy

The main ingredients of the implementation

- uses a logic programming interpreter on both the right and left sides of the sequent
- interprets eigenvariables as instantiatable on the left and produces all answers in a lazy stream-based manner
- treats local contexts via abstractions and uses higher-order pattern matching to match atoms and clause heads

Bedwyr has been used in several prototyping examples, e.g.

- checking (open) bisimilarity in the finite π-calculus
- identifying winning strategies in games
- reasoning about security protocols: the SPEC system `http://users.cecs.anu.edu.au/~tiu/spec/index.html`.

Limitations of Bedwyr

- Eigenvariables and logic variables used for full automation lead to incompleteness if present together

  For example, consider proving the formula

  $$\forall x\, \exists y\, (px \wedge py \wedge x = y \supset \bot)$$

  where p is defined as $\{pa \stackrel{\triangle}{=} \top, pb \stackrel{\triangle}{=} \top, pc \stackrel{\triangle}{=} \top\}$

  Completing this proof requires solving *disunification* problems: For each x, find an y such that $x \neq y$
- Reasoning process relies intrinsically on the finiteness of success and failure for specifications

  The system has been extended to treat some forms of (co-)inductive reasoning by using tabling and cyclic proofs

We will discuss a richer proof system that treats definitions (co-)inductively to extend the second kind of capability

We will also turn our focus to interactive theorem proving

Treating Equality as a Defined Symbol

Our interpretation of definitions is related to a "closed-world" treatment of equality

This treatment can be made explicit via introduction rules for the equality symbol

$$\frac{}{\Sigma; \Gamma \vdash \sigma \rhd (t = t)}\ eqR$$

$$\frac{\{\Sigma\theta; \Gamma\theta \vdash \Delta\theta \mid (\lambda x_1 \ldots \lambda x_n\ u)\theta =_\lambda (\lambda x_1 \ldots \lambda x_n\ v)\theta\}}{\Sigma; \Gamma, x_1, \ldots, x_n \rhd u = v \vdash \Delta}\ eqL$$

The real content is in the *eqL* rule that enforces a syntactic understanding of equality

Once again, for practicality, we may limit ourselves to a complete set of unifiers when considering cut-free proofs

Definitions and Equality

Definitions can now be reduced to at most one clause for each predicate that has the form

$$\forall \vec{x}.p\ \vec{x} \stackrel{\triangle}{=} B\ p\ \vec{x}$$

where p and $\vec{x}$ do not appear in B

Further, the rules for definitions can be simplified to just unfolding:

$$\frac{\Sigma; \Gamma, \sigma \rhd B\ p\ \vec{t} \vdash \Delta}{\Sigma; \Gamma, \sigma \rhd p\ \vec{t} \vdash \Delta}\ def\mathcal{L} \qquad \frac{\Sigma; \Gamma \vdash \sigma \rhd B\ p\ \vec{t}}{\Sigma; \Gamma \vdash \sigma \rhd p\ \vec{t}}\ def\mathcal{R}$$

In the presence of the equality rules, the earlier rules for definitions become derived ones

Definitions and Equality (Example)

For example, consider the clauses

$$nat\ 0 \stackrel{\triangle}{=} \top \qquad \forall N.nat\ (s\ N) \stackrel{\triangle}{=} nat\ N$$

These can be transformed into

$$\forall N.nat\ N \stackrel{\triangle}{=} (N = 0) \vee \exists N'\ (N = s\ N' \wedge nat\ N')$$

The following derivation then realizes the effect of the earlier $def\mathcal{L}$ rule:

$$\dfrac{\dfrac{\{\Sigma\theta; \Gamma\theta \vdash \Delta\theta \mid t\theta = 0\}}{\Sigma; \Gamma, \sigma \rhd t = 0 \vdash \Delta}\ eqL \quad \dfrac{\dfrac{\{(\Sigma, N')\theta; \Gamma\theta, \sigma \rhd (nat\ N')\theta \vdash \Delta\theta \mid t\theta = (s\ N')\theta\}}{\Sigma, N'; \Gamma, \sigma \rhd (t = s\ N') \wedge nat\ N' \vdash \Delta}\ eqL}{\Sigma; \Gamma, \sigma \rhd \exists N'\ (t = s\ N') \wedge nat\ N' \vdash \Delta}\ \exists L}{\Sigma; \Gamma, \sigma \rhd (t = 0) \vee \exists N'\ (t = s\ N' \wedge nat\ N') \vdash \Delta}\ \vee L$$
$$\Sigma; \Gamma, \sigma \rhd nat\ t \vdash \Delta\ def\mathcal{L}$$

A similar observation holds with respect to the earlier $def\mathcal{R}$ rule

Inductive and Co-Inductive Definitions

The unfolding rules amount to treating $\forall \vec{x}.p\ \vec{x} \stackrel{\triangle}{=} B\ p\ \vec{x}$ as a fixed-point of the operator B

The following rule interprets it as the least of the pre-fixed points, leading to an inductive treatment

$$\frac{\vec{x}; B\ S\ \vec{x} \vdash S\ \vec{x} \quad \Sigma; \Gamma, \sigma \rhd S\ \vec{t} \vdash \Delta}{\Sigma; \Gamma, \sigma \rhd p\ \vec{t} \vdash \Delta}\ \mu L$$

Here S is a closed term of the same type as p that serves as an *invariant*

Similarly, the following rule interprets the it as the greatest of the post-fixed points, treating it co-inductively

$$\frac{\Sigma; \Gamma \vdash \sigma \rhd S\ \vec{t} \quad \vec{x}; S\ \vec{x} \vdash B\ S\ \vec{x}}{\Sigma; \Gamma \vdash \sigma \rhd p\ \vec{t}}\ \nu R$$

Here S is a closed term of same type as p called the *co-invariant*

Inductive and Co-Inductive Definitions (Continued)

We have to be careful when mixing inductive and co-inductive definitions

For example, a proof of $; \vdash \bot$ can easily be constructed by interpreting

$$p \stackrel{\triangle}{=} p$$

both inductively and co-inductively

Cut-elimination and, consequently, consistency holds under the following additional conditions

- each clause is treated exclusively inductively, co-inductively or simply as a fixed-point (annotated $\stackrel{\mu}{=}$, $\stackrel{\nu}{=}$ or $\stackrel{\triangle}{=}$)
- the earlier stratification condition holds: $lvl(B\ p\ \vec{x}) \leq lvl(p)$
- "mutual recursion" is prohibited: $lvl(B\ (\lambda \vec{x}\ \top)\ \vec{x}) < lvl(p)$

We will assume these conditions implicitly henceforth

An Example of Inductive Reasoning

Consider the following definition

$$\forall I, J, K.plus\ I\ J\ K \stackrel{\mu}{=} (I = 0 \wedge J = K) \vee \exists M \exists N\ I = (s\ M) \wedge K = (s\ N) \wedge plus\ M\ J\ N$$

Now consider proving that *plus* is functional, i.e.

$$; \vdash \triangleright \forall I \forall J \forall K\ plus\ I\ J\ K \supset \forall L\ plus\ I\ J\ L \supset K = L$$

Eliding the local context that is always empty in case, this proof reduces to one for

$$I, J, K; plus\ I\ J\ K \vdash \forall L\ plus\ I\ J\ L \supset K = L$$

Now use induction on *plus* with the invariant

$$D \equiv \lambda I \lambda J \lambda K\ \forall L\ plus\ I\ J\ L \supset K = L$$

I.e. with the righthand side over which the eigenvariables have been abstracted

An Example of Inductive Reasoning (Continued)

To look at the use of induction, rewrite the definition of *plus* as

$$\forall I, J, K.plus\ I\ J\ K \stackrel{\mu}{=} B\ plus\ I\ J\ K$$

where

$$B = \lambda P \lambda I \lambda J \lambda K\ ((I = 0 \wedge J = K) \vee \exists M \exists N\ I = (s\ M) \wedge K = (s\ N) \wedge plus\ M\ J\ N$$

Then the proof at the end has the following shape

$$\dfrac{L, M, N; B\ D\ L\ M\ N \vdash D\ L\ M\ N \quad \overline{I, J, K; D\ I\ J\ K \vdash D\ I\ J\ K}\ init}{I, J, K; plus\ I\ J\ K \vdash D\ I\ J\ K}\ \mu L$$

Based on the structure of B, we get for the left premise

$$\dfrac{J; \vdash D\ 0\ J\ J \quad L', M, N'; D\ L'\ M\ N' \vdash D\ (s\ L')\ M\ (s\ N')}{L, M, N; B\ D\ L\ M\ N \vdash D\ L\ M\ N}\ \vee L, eqL, \exists L$$

This proof can be completed now by using rules based on the structure of D and case analysis on *plus*

Another Example of Inductive Reasoning

Consider the following definitions

$$\forall N.nat\ N \stackrel{\mu}{=} (N = 0) \vee \exists N'\ N = (s\ N') \wedge nat\ N'$$

$$\forall I, J, K.plus\ I\ J\ K \stackrel{\mu}{=} (I = 0 \wedge J = K) \vee \exists M \exists N\ I = (s\ M) \wedge K = (s\ N) \wedge plus\ M\ J\ N$$

We can then show that *plus* is total on natural numbers

$$; \vdash \cdot \triangleright \forall I\ nat\ I \supset \forall J\ nat\ J \supset \exists K\ plus\ I\ J\ K$$

This can be reduced to proving

$$I; \cdot \triangleright nat\ I \vdash \cdot \triangleright \forall J\ nat\ J \supset \exists K\ plus\ I\ J\ K$$

Now use induction on *nat* with the invariant

$$\lambda I\ \forall J\ nat\ J \supset \exists K\ plus\ I\ J\ K$$

Notice again that this is the righthand side of the sequent with the eigenvariables abstracted

An Easier-to-Use Derived Rule for Induction

Suppose the objective is to prove a formula of the form

$$\forall \vec{x}\ H_1 \supset \cdots \supset (p\ \vec{t}) \supset \cdots \supset H_n \supset A$$

using induction on p which is defined as $\forall \vec{y}.p\ \vec{y} \stackrel{\mu}{=} B\ p\ \vec{y}$

Then, this formula can be changed to

$$\forall \vec{x}\ H_1 \supset \cdots \supset (p^{@}\ \vec{t}) \supset \cdots \supset H_n \supset A$$

and the formula

$$\forall \vec{x}\ H_1 \supset \cdots \supset (p^{*}\ \vec{t}) \supset \cdots \supset H_n \supset A$$

can be added to the assumptions before continuing the proof

The interpretation of $p^{@}$ and p^{*} here are the following

- p^{*} will match with p only if it has the $*$ annotation
- To unfold $p^{@}\ \vec{t}$, replace it with $B\ p^{*}\ \vec{t}$

A proof based on this schema can be transformed into one with the usual induction rule

An Example of the Use of the Derived Rule

Assume the earlier definition of *nat* and consider proving

$$\forall I \forall J\ nat\ I \supset nat\ J \supset \exists K\ plus\ I\ J\ K$$

by induction on the first occurrence of *nat*

Then we would add as a hypothesis

$$IH = \forall I \forall J\ nat^*\ I \supset nat\ J \supset \exists K\ plus\ I\ J\ K$$

After some introduction rules, we would be left with proving

$$I, J; IH, nat^@\ I, nat\ J \vdash \exists K\ plus\ I\ J\ K$$

Unfolding ($nat^@\ I$) now leaves us to prove the two sequents

$$J; IH, nat^*\ 0, nat\ J \vdash \exists K\ plus\ 0\ J\ K \qquad \text{and}$$

$$N, J; IH, nat^*\ N, nat\ J \vdash \exists K\ plus\ (s\ N)\ J\ K$$

These sequents can be proved easily using the induction hypothesis

Some Issues with the Nabla Quantifier

The ∇ quantifier gives us the ability to reason about binding by replacing bound variables with names

In a sense, it allows us to reasoning about "open terms"

When a term is not open, we might expect the reasoning capability to be equivalent to that on closed terms

Thus, we might desire the following formulas to be provable

$$\forall x\ (\nabla n\ p\ x) \supset p\ x$$

$$\forall x\ p\ x \supset (\nabla n\ p\ x)$$

Similarly, what seems important is which "variables" are free, not the order we talk of them

In other words, we might expect the following to hold

$$\nabla x \nabla y\ p\ x\ y \equiv \nabla y \nabla x\ p\ x\ y$$

Neither of these principles hold within the current logic

Open Terms and Induction

Induction over open terms is also not currently possible

Specifically, in the rule that treats inductive definitions

$$\frac{\vec{x}; B\ S\ \vec{x} \vdash S\ \vec{x} \quad \Sigma; \Gamma, \sigma \rhd S\ \vec{t} \vdash \Delta}{\Sigma; \Gamma, \sigma \rhd p\ \vec{t} \vdash \Delta}\ \mu L$$

the local context for establishing the invariant is empty

Thus, the invariant cannot contain any "names" in it

As a specific example, consider the following structural property

$$\nabla n \forall x\ nat\ (x\ n) \supset \exists y\ x = \lambda z\ y$$

i.e., "no name can appear in a natural number"

We could prove this if the following invariant

$$S = \lambda t\ (\forall x\ t = (x\ n) \supset \exists y\ x = \lambda z\ y)$$

where n is a name is acceptable

However, this is not acceptable in the current logic

A Richer Treatment of the Nabla Quantifier

To overcome the first set of issues we extend the logic to include the following equivalences

$$\nabla x \nabla y\ B\ x\ y \equiv \nabla y \nabla x B\ x\ y$$

$$\nabla x\ B \equiv B \quad \text{if } x \text{ does not appear in } B$$

The extension is realized in a rather simple way

- Add to the vocabulary a denumerable set of new symbols called *nominal constants*
- Drop local contexts from formulas but use a nominal constant not appearing in the formula in the rules for ∇
- Use nominal constants arbitrarily for existentials, raise over the ones in the formula when introducing eigenvariables
- When matching formulas (e.g. in the *init* or *cut* rules), we consider permutation mappings for nominal constants

The issue related to induction also is solved by this change: nominal constants can appear in the invariant

A Richer Treatment of the Nabla Quantifier

To overcome these issues we extend the logic to include the following equivalences

$$\nabla x \nabla y\ B\ x\ y \equiv \nabla y \nabla x B\ x\ y$$

$$\nabla x\ B \equiv B \quad \text{if } x \text{ does not appear in } B$$

We refer to these equivalences as the interchange and strengthening principles

The addition of these principles also allows us to simplify the proof rules by making local contexts implicit using *nominal constants*

The Logic LG^{ω}

This logic realizes the interchange and strengthening principles related to ∇ quantification

Formally, we change the existing reasoning logic as follows

- We assume a denumerable number of nominal constants at each type where ∇ quantification is permitted
  $\mathcal{C}$ stands for the set of all nominal constants, $\mathcal{K}$ for normal constants
- We drop local contexts from judgements, i.e., these become simply formulas again
- Formulas can now contain nominal constants arbitrarily, but the scope of such constants will be local
  However, definitions *must not* contain nominal constants

Equality between formulas is given by identity modulo λ-conversion and a permutation mapping on nominal constants
Written as $B' \approx B$ and read "B' is equivariant to B"

The Inference Rules for LG^{ω}

There is not much to say about the rules other than the ones involving identity and the quantifiers

For the former, we should use equivariance to match formulas

$$\frac{}{\Sigma; \Gamma, B' \vdash B}\ \mathit{init} \qquad B \approx B'$$

$$\frac{\Sigma; \Gamma_1 \vdash B \quad \Sigma; \Gamma_2, B' \vdash \Delta}{\Sigma; \Gamma_1, \Gamma_2 \vdash \Delta}\ \mathit{cut} \qquad B \approx B'$$

For the quantifier rules, an important question is what to raise over in the $\forall R$ and $\exists L$ rules

We should raise over enough nominal constants but still leave some for any ∇ quantifiers of smaller scope

It turns out to be adequate to raise only over the nominal constants appearing in the quantified formula

The Quantifier Inference Rules of LG^{ω}

Let $\mathit{supp}(B)$ denote a listing of the nominal constants appearing in B

$$\frac{\Sigma; \Gamma, B[a/x] \vdash \Delta}{\Sigma; \Gamma, \nabla x\ B \vdash \Delta}\ \nabla\mathcal{L} \qquad a \notin \mathit{supp}(B)$$

$$\frac{\Sigma; \Gamma \vdash B[a/x]}{\Sigma; \Gamma \vdash \nabla x\ B}\ \nabla\mathcal{R} \qquad a \notin \mathit{supp}(B)$$

$$\frac{\Sigma, h; \Gamma, B[(h\ \sigma)/x] \vdash \Delta}{\Sigma; \Gamma, \exists x\ B \vdash \Delta}\ \exists L \qquad \sigma = \mathit{supp}(B), h \text{ new}$$

$$\frac{\Sigma; \Gamma \vdash B[t/x]\Delta \quad t \text{ is a } (\Sigma \cup \mathcal{C})\text{-term}}{\Sigma; \Gamma \vdash \exists x\ B}\ \exists R$$

$$\frac{\Sigma; \Gamma, B[t/x] \vdash \Delta \quad t \text{ is a } (\Sigma \cup \mathcal{C})\text{-term}}{\Sigma; \Gamma, \forall x\ B \vdash \Delta}\ \forall L$$

$$\frac{\Sigma, h; \Gamma \vdash B[(h\ \sigma)/x]}{\Sigma; \Gamma \vdash \forall x\ B}\ \exists L \qquad \sigma = \mathit{supp}(B), h \text{ new}$$

Nominal Constants and Substitution

Nominal constants are accompanied by an implicit formula level binding that substitutions must be careful to respect

Sometimes, such as in the quantifier rules, we need nominal constants in the substitution terms to be interpreted locally

This is identical to the earlier notion of substitution and we will continue to use that notation ($B\theta$, $\Gamma\theta$, etc) for it

However, at other times, substitutions determined with one formula have to be applied to another

This happens, for example, with the left introduction rule for equality

In this case, the names of nominal constants must be permuted first before applying the substitution to avoid incorrect capture

This is the "logical" notion of substitution and will be written as $B[\![\theta]\!]$ ($\Gamma[\![\theta]\!]$, etc)

A Missing Feature in LG^{ω}

Nominal constants are used to represent free variables in "open" terms

Sometimes we may want to examine the structure of these terms with respect to such free variables

For example, questions such as the following arise naturally in reasoning based on syntax

- Does a given term correspond to a free variable?
- Is a given term devoid of a particular variable, i.e. is the variable fresh to the term?

The current logic does not support such analyses directly

Some of these checks can be done indirectly, e.g. a term is a nominal constant if it cannot be something else

However, this can be a cumbersome encoding that also makes reasoning difficult

A Concrete Example

Consider the following rendition of typing where tokens with capitals are implicitly universally quantified at the top-level

$$of\ \Gamma\ X\ A \stackrel{\mu}{=} member\ (X : A)\ \Gamma$$
$$of\ \Gamma\ (app\ M\ N)\ B \stackrel{\mu}{=} \exists A\ (of\ \Gamma\ M\ (arr\ A\ B) \wedge (of\ \Gamma\ N\ A))$$
$$of\ \Gamma\ (abs\ A\ M)\ (arr\ A\ B) \stackrel{\mu}{=} \nabla x\ (of\ ((x : A) :: \Gamma)\ (M\ x)\ B)$$

Then determinateness of typing can be expressed as

$$\forall\Gamma\forall M\forall A\forall B\ ctx\ \Gamma \supset (of\ \Gamma\ M\ A) \supset (of\ \Gamma\ M\ B) \supset A = B$$

if $ctx\ \Gamma$ holds iff $\Gamma = (x_1 : A_1) :: \ldots :: (x_n, A_n) :: nil$ where

- each x_i is a nominal constant and
- x_i and x_j are distinct if $i \neq j$

Note that these properties of *ctx* must be articulated for the reasoning to go through

Nominal Abstraction

This is a relation between terms that provides a means for inspecting nominal constant occurrences

This relation, written $s \trianglerighteq t$, holds between terms s and t if

- s is of type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau$ and t is of type τ and
- s is λ-convertible to the result of abstracting over n distinct nominal constants in t

E.g, for nominal constants c_1 and c_2, the following relations hold

$$\lambda x\ x \trianglerighteq c_1 \qquad \lambda x p\ x\ c_2 \trianglerighteq p\ c_1\ c_2 \qquad \lambda x \lambda y\ p\ x\ y \trianglerighteq p\ c_1\ c_2$$

but the following do not hold

$$\lambda x\ x \trianglerighteq p\ c_1\ c_2 \qquad \lambda x\ p\ x\ c_2 \trianglerighteq p\ c_2\ c_1 \qquad \lambda x \lambda y\ p\ x\ y \trianglerighteq p\ c_1\ c_1$$

Also, a substitution θ is said to be a *solution* to the nominal abstraction $s \trianglerighteq t$ if $(s \trianglerighteq t)[\![\theta]\!]$ holds

Nominal Abstraction and Term Structure

Existence of solutions allow us to characterize terms based on nominal constant occurrences

For example, consider the following

$$\lambda x \; \mathit{fresh}\; x \; T \trianglerighteq S$$

This is satisfied only by (ground) terms of the form $\mathit{fresh}\; n \; S'$ where n is a nominal constant that does not appear in S'

Similarly, consider the following

$$\lambda x \; \mathit{name}\; x \trianglerighteq \mathit{name}\; S$$

To satisfy this, S must be a nominal constant

Note that $\trianglerighteq$ extends the equality relation and its "structure discerning" capabilities to terms with nominal constants

Proof Rules for Nominal Abstraction

We will use ambiguously the notation $s \rhd t$ to denote in the logic the semantic notion just described

The rules for introducing this relation are the following

$$\frac{}{\Sigma; \Gamma \vdash s \trianglerighteq t}\ \trianglerighteq\mathcal{R} \qquad s \trianglerighteq t \text{ holds}$$

$$\frac{\{\Sigma\theta; \Gamma[\![\theta]\!] \vdash C[\![\theta]\!] \mid \theta \text{ is a solution to } (s \trianglerighteq t)\}}{\Sigma; \Gamma, s \trianglerighteq t \vdash \Delta}\ \trianglerighteq\mathcal{L}$$

Substitutions in the $\trianglerighteq\mathcal{L}$ rule can be restricted in a cut-free setting to a *complete set of nominal abstraction solutions*

This is a set S such that

- each $\theta \in S$ is a solution to $s \trianglerighteq t$, and
- every solution to $s \trianglerighteq t$ is "covered" over Σ by one in S

Moreover, such sets can be computed using higher-order (pattern) unification

Defining Typing Contexts Using Nominal Abstraction

The property we wanted of typing contexts can now be articulated via the following definition

$$\begin{aligned} \mathit{ctx}\; K \quad \triangleq \quad & (K = \mathit{nil}) \vee \\ & \exists T \exists L\, ((\lambda x\, (x : T) :: L) \trianglerighteq K \wedge \mathit{ctx}\; L) \end{aligned}$$

Given this definition, $\mathit{ctx}\; L$ is provable exactly when L is a type assignment to distinct nominal constants

Conversely, if $\mathit{ctx}\; L$ appears as a premise, the definition allows it to be concluded that L satisfies the desired restrictions

Exercise:

Using this definition of *ctx* try to prove determinateness of typing

$$\forall \Gamma \forall M \forall A \forall B\; \mathit{ctx}\; \Gamma \supset (\mathit{of}\; \Gamma\; M\; A) \supset (\mathit{of}\; \Gamma\; M\; B) \supset A = B$$

A Pattern-Based Form for Definitions

An alternative to allowing nominal abstractions is to let a predicate be defined via multiple clauses of the form

$$\forall \vec{x}.(\nabla \vec{z}.(p\; \vec{t}) \triangleq B\; p\; \vec{x})$$

i.e., clauses that use patterns and nabla quantifiers in the head

The Meaning Of Such a Clause

An instance of $p\; \vec{t}$ is true if the corresponding instance of B is true, provided

- the variables in $\vec{z}$ are instantiated with unique nominal constants $\vec{a}$
- the variables in $\vec{x}$ are instantiated with terms not containing constants in $\vec{a}$

Encoding properties of typing contexts with such definitions:

$$\mathit{ctx}\; \mathit{nil} \triangleq \top$$

$$\forall L, A.\ (\nabla x.\mathit{ctx}\; ((x : A) :: L)) \triangleq \mathit{ctx}\; L$$

Clauses with Patterns versus Nominal Abstractions

The effect of a predicate definition based on n "patterned clauses" of the form

$$\forall \vec{x_i}.(\nabla \vec{z_i}.(p\ \vec{t_i}) \triangleq B_i\ p\ \vec{x_i})$$

can be realized by the following definition in the "one clause" format

$$\forall \vec{y}.(p\ \vec{y}) \triangleq \bigvee_{i \in 1,\ldots,n} \exists \vec{x_i}\ (\lambda \vec{z_i}\ (p'\ \vec{t_i}) \trianglerighteq p'\ \vec{y}) \wedge B_i\ p\ \vec{x_i}$$

where p' is a new predicate name with type identical to that of p and the $\vec{y_i}$ are variables chosen to not appear in $\vec{t_i}$

Left and right introduction rules capturing the intended meaning of patterned clauses can be provided based in this translation

Inductive definitions can be provided a similar treatment through patterned clauses with nabla in the head

Introduction Rules for Pattern Based Definitions

Let $\mathcal{D}$ represent the collection of clauses for p in pattern form

- Right introduction rules for such a definition

$$\frac{\Sigma; \Gamma \vdash (B\ p\ \vec{x})[\theta]}{\Sigma; \Gamma \vdash p\ \vec{s}}\ def\mathcal{R}^p$$

  where $\forall \vec{x}.(\nabla \vec{z}.p\ \vec{t}) \triangleq B\ p\ \vec{x} \in \mathcal{D}$ and θ is such that $range(\theta) \cap \Sigma = \emptyset$ and $(\lambda \vec{z}.p\ \vec{t})[\theta] \trianglerighteq p\ \vec{s}$ holds

- Left introduction rule for such a definition

$$\frac{\left\{ \begin{array}{l|l} \Sigma\theta; \Gamma[\![\theta]\!], (B\ p\ \vec{x})[\![\theta]\!] \vdash C[\![\theta]\!] & \forall \vec{x}.(\nabla \vec{z}.p\ \vec{t}) \triangleq B\ p\ \vec{x} \in \mathcal{D} \\ & \text{and } \theta \text{ is a solution} \\ & \text{to } ((\lambda \vec{z}.p\ \vec{t}) \trianglerighteq p\ \vec{s}) \end{array} \right\}}{\Sigma; \Gamma, p\ \vec{s} \vdash C}\ def\mathcal{L}^p$$

These rules can also be used for inductive and co-inductive definitions, i.e. if $\stackrel{\mu}{=}$ or $\stackrel{\nu}{=}$ is used in the definition instead of $\stackrel{\triangle}{=}$

Left Rule for Inductive Definitions

Suppost that p is defined by n clauses of the form

$$\forall \vec{x_i}.(\nabla \vec{z_i}.(p\ \vec{t_i}) \stackrel{\mu}{=} B_i\ p\ (\vec{x_i})$$

Then the following rule can be used to introduce p on the left of a sequent

$$\frac{\left\{\vec{x_i}; B_i\ S\ \vec{x_i} \vdash \nabla \vec{z_i}.S\ \vec{t_i}\right\}_{i \in 1..n} \qquad \Sigma; \Gamma, S\ \vec{s} \vdash C}{\Sigma; \Gamma, p\ \vec{s} \vdash C}\ \mu L^p$$

The Two-Level Logic Approach to Reasoning

Specifications can be encoded and reasoned about directly in $\mathcal{G}$, the fully featured reasoning logic

However, we propose instead to do the following

- Write specifications in a restriction of *hohh* called hH^2
- Embed the specification logic via a definition based on its operational semantics into the reasoning logic
- Reason about the object languages/systems via the encoding of the specification logic

We refer to this as the two level logic approach to reasoning about specifications

Tradeoffs Involved in the Two-Level Logic Approach

There are some drawbacks related to the use of this approach

- The full flexibility of $\mathcal{G}$ is not available for writing specifications
- There is an indirection involved in reasoning: we have to do this through the encoding of the specification logic

However, there are some advantages too

- Specifications can be used both for animation and for reasoning
- Meta-theoretic properties of the specification logic can be exploited in the reasoning process

We will try to see how the reasoning overhead can be reduced sufficiently to make this approach viable

The Abella system and experiments with it provide evidence for the richness of the framework in practice

The hH^2 Logic

The main restrictions to *hohh* that results in hH^2 are

- Predicate quantification is disallowed
- Implication goals can have only atoms on left

Thus, goals and program clauses have the following structure

$$\begin{array}{lcl} G & ::= & \top \mid A_r \mid G \vee G \mid G \wedge G \mid \exists x\ G \mid \forall x\ G \mid A_r \supset G \\ D & ::= & A_r \mid G \supset D \mid \forall x\ D \end{array}$$

The sequents that arise from applying the *hohh* operational semantics to this language have the form

$$\Sigma; \Delta, \mathcal{L} \vdash G$$

with the following connotations

- Δ consists of the program clauses from the original specification
- $\mathcal{L}$ is a set of atoms introduced via implication goals

We will use this property in the embedding

The Encoding of Formulas and Terms

Both $\mathcal{G}$ and hH^2 are based on the simply typed lambda calculus so the encoding of the syntax can be shallow

- Sorts in hH^2 give rise to corresponding ones in $\mathcal{G}$ with one special case
  The type *o* in hH^2 is represented by a type *fmla* in $\mathcal{G}$
- Constants in hH^2 carve out ones in $\mathcal{G}$ with one proviso
  Logical constants become nonlogical ones yielding terms of type *fmla*
- The sort *atm* is used for atomic formula from hH^2
  $\langle A \rangle$ makes a *fmla* from an *atm* A and $\supset$ is encoded via a constant of type *atm*→*fmla*→*fmla*
- Eigenvariables in hH^2 are encoded by nominal constants in $\mathcal{G}$

Adequacy of the encoding, denoted $\{\!\{\cdot\}\!\}$, follows from showing a suitable bijection and compositionality of substitution

Encoding hH^2 Programs

For each clause $\forall \vec{x}\ G_1 \supset \cdots \supset G_n \supset A$ in the fixed hH^2 specification Δ, we will use a clause of the form

$$prog\ A\ (G_1 \wedge \ldots \wedge G_n) \stackrel{\triangle}{=} \top$$

Here, *prog* has type *atm*→*fmla*→*o*

For example, corresponding to the formalization of type evaluation, we would have the clauses

$$prog\ (of\ (abs\ A\ M)\ (arr\ A\ B))$$
$$(\forall x\ (of\ x\ A) \supset \langle of\ (R\ x)\ B \rangle) \stackrel{\triangle}{=} \top$$
$$prog\ (of\ (app\ M\ N))$$
$$(\langle of\ M\ (arr\ A\ B) \rangle \wedge \langle of\ N\ A \rangle) \stackrel{\triangle}{=} \top$$

Notice that these clauses can be generated automatically from a specification as indeed is done by Abella

Encoding the Operational Semantics

Three special definitions are used in the encoding of hH^2 derivations

- For an special sort nt and $z : nt$, $s : nt \to nt$ and $nat : nt \to o$

  $$nat\ z \stackrel{\mu}{=} \top \qquad nat\ (s\ N) \stackrel{\mu}{=} nat\ N$$

  For encoding induction on the lengths of hH^2 derivations

- For lt of type $nt \to nt \to o$

  $$lt\ z\ (s\ N) \stackrel{\mu}{=} \top \qquad lt\ (s\ M)\ (s\ N) \stackrel{\mu}{=} lt\ M\ N$$

  Helps in realizing strong induction over hH^2 derivations

- For sort $atmlist$, $nil : atmlist$, $::$ an infix operator of type $atm \to atmlist \to atmlist$ and $member : atm \to atmlist \to o$

  $$member\ A\ (A :: L) \stackrel{\mu}{=} \top \qquad member\ A\ (B :: L) \stackrel{\mu}{=} member\ A\ L$$

  Used for encoding "lookup" in a context created in the course of a derivation

Encoding the Operational Semantics (Continued)

Let $seq : nt \to atmlist \to fmla \to o$ be defined by the clauses

$$seq\ (s\ N)\ L\ \top \stackrel{\mu}{=} \top$$

$$seq\ (s\ N)\ L\ (G_1 \vee G_2) \stackrel{\mu}{=} seq\ N\ L\ G_1$$

$$seq\ (s\ N)\ L\ (G_1 \vee G_2) \stackrel{\mu}{=} seq\ N\ L\ G_2$$

$$seq\ (s\ N)\ L\ (G_1 \wedge G_2) \stackrel{\mu}{=} seq\ N\ L\ G_1 \wedge seq\ N\ L\ G_2$$

$$seq\ (s\ N)\ L\ (A \supset G) \stackrel{\mu}{=} seq\ N\ (A :: L)\ G$$

$$seq\ (s\ N)\ L\ (\forall x\ B\ x) \stackrel{\mu}{=} \nabla x\ seq\ N\ L\ (B\ x)$$

$$seq\ (s\ N)\ L\ \langle A \rangle \stackrel{\mu}{=} member\ A\ L$$

$$seq\ (s\ N)\ L\ \langle A \rangle \stackrel{\mu}{=} \exists b\ prog\ A\ b \wedge seq\ N\ L\ b$$

Then hH^2 derivability of $\Sigma; \Delta, \mathcal{L} \vdash G$ amounts (for suitable $prog$ clauses) to the $\mathcal{G}$-provability of $\exists n\ ((nat\ n) \wedge (seq\ n\ \{\!\{\mathcal{L}\}\!\}\ \{\!\{G\}\!\}))$

We will abbreviate the latter by $\mathcal{L} \Vdash G$, assuming a fixed Δ

Proving Properties of Specifications

Properties of specifications can be expressed in $\mathcal{G}$ modulo the encoding of derivability in hH^2

For example, given $prog$ clauses encoding evaluation and typing, subject reduction can be expressed via the statement

$$\forall e \forall v \forall ty\ (\Vdash \langle eval\ e\ v \rangle) \supset (\Vdash \langle of\ e\ ty \rangle) \supset (\Vdash \langle eval\ v\ ty \rangle)$$

Proofs of such properties will typically involve induction on the lengths of hH^2 derivations

For example, the subject reduction formula is equivalent to

$$\forall e \forall v \forall ty \forall n\ (nat\ n \wedge seq\ n\ nil\ \langle eval\ e\ v \rangle \supset (\Vdash \langle of\ e\ ty \rangle) \supset (\Vdash \langle eval\ v\ ty \rangle))$$

If we try to prove this by induction on $nat\ n$, this will amount to induction on the length of the hH^2 derivation of $eval\ e\ v$

Using Strong Induction on Derivation Lengths

To prove the subject reduction formula, we will actually need to unfold the seq definition twice

- we first look up the clause for $(eval\ (app\ e_1\ e_2)\ v)$
- we then use the definition of seq for a conjunction

To deal with this, we prove instead the formula

$$\forall e \forall v \forall ty \forall n \forall n\ (nat\ n \wedge (lt\ m\ n) \wedge seq\ m\ nil\ \langle eval\ e\ v \rangle \supset (\Vdash \langle of\ e\ ty \rangle) \supset (\Vdash \langle eval\ v\ ty \rangle))$$

Induction on $(nat\ n)$ and the resulting case analysis lead to assuming the formula for any m for which $(lt\ m\ n)$ holds

To conclude the proof, we only need to observe that the original formula must hold if this generalization does

This is actually a general schema for justifying strong induction in this setting

Leveraging the Specification Logic Meta-Theory

Meta-theoretic properties of the specification logic such as the following can be stated and proved via its encoding in $\mathcal{G}$

Monotonicity

$\forall\Gamma\forall\Delta\forall G\,(\forall X\ \mathit{member}\ X\ \Gamma \supset \mathit{member}\ X\ \Delta) \supset \Gamma \Vdash G \supset \Delta \Vdash G$

Cut Admissibility

$\forall\Gamma\forall A\forall G\,(A :: \Gamma) \Vdash G \supset \Gamma \Vdash A \supset \Gamma \Vdash G$

Instantiation

$\forall\Gamma\forall G\,(\nabla x\,(\Gamma\ x) \Vdash (G\ x)) \supset \forall t\,(\Gamma\ t) \Vdash (G\ t)$

If the framework allows for lemma use, then these results can simplify proofs

For example, cut admissibility and instantiation immediately yield a substitutivity principle for typing judgements

$(\nabla x\,(\mathit{of}\ x\ a) :: L \Vdash \langle \mathit{of}\ (T\ x)\ B\rangle) \supset (L \Vdash \langle \mathit{of}\ u\ a\rangle) \supset L \Vdash \langle \mathit{of}\ (T\ u)\ B\rangle$

The Abella System

This system realizes the two-level logic approach using hH^2 and $\mathcal{G}$

- It automatically creates *prog* clauses from hH^2 specs
- It treats judgements of the form $L \Vdash G$ in a special way, building in the processing of *seq* and *prog* clauses
- It uses the annotation based treatment of induction on $L \Vdash G$ judgements, thereby building in strong induction
- It embeds meta-theoretic properties of the specification logic into tactics by exploiting a lemma facility

These features considerably reduce the syntactic overhead of the two-level logic approach

The Abella website documents many successful applications

Exercise Suggestion: Try using the system to formalize proofs of meta-theorems of intuitionistic logic from the first problem set

Adequacy of the Two-Level Logic Approach

To allow results to be extracted about the object systems, we have to show the following

- The encoding in hH^2 of the object system is adequate
  Has to be done for each encoding, but there is evidence that λ-tree syntax simplifies this task
- The encoding of hH^2 into $\mathcal{G}$ must be adequate
  In particular, theorems proved in $\mathcal{G}$ via the encoding should be transferrable to hH^2
- The results in the previous two items should "compose" to yield object language theorems from one proved in $\mathcal{G}$
  This again is dependent on particular object but seems to be easily achievable

The second step is complicated by habitation questions

One result: This adequacy holds when eigenvariables and nominal constants are permitted only at already inhabited types