A Supposedly Fun Thing I May Have to Do Again *

A HOAS Encoding of Howe's Method

Alberto Momigliano
Dipartimento di Informatica, Università degli Studi di Milano
momigliano@di.unimi.it

Abstract

We formally verify in Abella that similarity in the call-by-name lambda calculus is a pre-congruence, using Howe's method. This turns out to be a very challenging task for HOAS-based systems, as it entails a demanding combination of inductive and coinductive reasoning on open terms, for which no other existing HOAS-based system is equipped for. We also offer a proof using a version of Abella supplemented with predicate quantification; this results in a more structured presentation that is largely independent of the operational semantics as well of the chosen notion of (bi)similarity. While the end result is significantly more succinct and elegant than previous attempts, the exercise highlights some limitations of the two-level approach in general and of Abella in particular.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal definitions and theory—semantics; F.4.1 [*Mathematical Logic*]: Lambda Calculus and Related Systems—Mechanical theorem proving, Proof theory; I.2.3 [*Artificial Intelligence*]: Deduction and Theorem Proving—Deduction, Inference engines, logic programming, meta theory

General Terms Languages, Verification

Keywords Higher order abstract syntax; Co-induction; Similarity in the lambda calculus; Howe's method; Abella; Predicate quantification.

1. Introduction

Howe's method [14, 15] is a remarkably flexible *syntactic* technique to show the congruence properties of (coinductively defined) functional program equalities, in particular in the presence of higher-order functions. In fact, an equality between programs shall better be not only an equivalence relation, but also *compatible*, i.e. it should respect the way programs are constructed. This feature allows equational reasoning to be *compositional*. A very nice and recent introduction to Howe's method for the untyped λ-calculus can be found in [25].

However, the present paper is not concerned about formally verifying the equivalence of concrete code fragments, for which more powerful techniques are now available, and for much more interesting programming languages than call-by name λ-calculus, see for example [16]. I am interested in formally verifying the correctness of Howe's method itself, not because I think it is flawed – it is not, as mechanically first shown in [2] – but because it is a very demanding benchmark for systems wishing to elegantly and effectively formalize the meta-theory of programming languages. More, I argue, than the useful, but by design quite simple examples in [7] and exercising different features than the by now infamous POPLMark challenge [3]. But, if this has already been done, as in the aforementioned [2], what is the fuss, one may ask. The point is in the *elegantly and effectively* qualifiers. The brute force approach using De Bruijn's indexes of [2], as I shall touch upon in Section 6, simply does not scale and in no way can be considered a springboard for more interesting languages and properties. In fact, even the mild generalization to the polymorphic λ-calculus proved to be too painful and lead to the development of the Hybrid tool [8].

These, of course, are not new observations: more than twenty years of research have provided us with alternatives to DB indexes, the prominent one, given my pedigree, being higher-order abstract syntax (HOAS). Or have they? Howe's method turns out to be a very challenging task for HOAS systems as well, as it entails a rare combination of inductive and coinductive reasoning over open terms, for which only one existing HOAS-based system seems to be equipped: *Abella* [10] [1].

This paper offers two (related) formalizations of Howe's method as presented in [24]; the first one basically refines and replays in Abella the proof outlined by Pitts in [20], which had been addressed in Hybrid [8]. The second one uses what we informally refer to as $\forall p$Abella, a version of Abella that permits predicate quantification, available as branch 1.3.6-dev from Abella's current maintainer, Kaustuv Chaudhuri. A word of caution: we do not have yet a consistency proof for this logic, although Alwen Tiu (personal communication), our cut elimination specialist in residence, is positive about that[2]. In any case, predicate quantification allows us to encode Howe's method in its generality [15], making it independent of the operational semantics and of the very notion of (bi)similarity.

While in both cases the end result is significantly more succinct and, especially the one using $\forall p$Abella, more elegant than the previous attempt, the exercise highlights some limitations of the two-level approach in general [19] and of Abella in particular, which can be of general interests for logical frameworks designers.

* To DFW, in memoriam.

LFMTP'12, September 9, 2012, Copenhagen, Denmark.

[1] The competition, in truth, out there being rather scarce: Twelf/Delphin/Beluga do not support coinduction so far and Hybrid only partially supports reasoning over open terms [7].

[2] Note that the lack of a published proof of consistency does not seem to have prevented the use of systems such as Isabelle/HOL and Coq, not to name names.

For the author of this paper, the verification of Howe's method was also a piece of unfinished business. Indeed, some 10 years ago I co-authored a workshop paper [20], which purported to be the first HOAS encoding of Howe's proof. Somewhat to my embarrassment, the "proof" turned out to be unsatisfactory, not just for the reasons we were aware of and mentioned in the paper: the representation not being as adequate as it should be – this was one-level Hybrid [1], after all – and the overhead needed to reason about open terms, which, naively, I thought I could completely eliminate by embracing the two-level architecture. The main culprit lied instead in the proof of the substitutivity property for the Howe relation never being finished[3]. And understandably so, as it turned out to be a major stumbling block for the present development as well. It is with a sense of relief that I offer the present paper as a way to make amends for the previous incomplete attempt.

The contributions of this paper are thus a "full HOAS" proof of Howe's method in two variants and probably the largest development in Abella to date, consisting each of around 50 theorems. At the same time I give some evidence, if any was needed, of the usefulness of $\forall p$Abella. On a secondary note, I describe a couple of folk techniques to get around some of Abella's (hopefully current) limitations and I suggest some improvements, so that the next time I will take on Howe's method, if I choose to, it will be a breeze.

2. Howe's method

Although this is not the main topic of the paper, some background on program equivalence may be in order. Suppose we want to say when two programs (two closed terms) have the same behavior. A well known such relation is Morris-style *contextual equivalence*: m and n are equivalent when, if inserted in *any* larger program fragment (context), both larger programs evaluate to the same value, or equivalently both terminate. While this notion of program equivalence is valuable and intuitive, it is indeed difficult to reason about its meta-theoretical properties, mainly due to the quantification on every possible context. *Bisimilarity* has emerged as a more manageable, yet, in this setting, equivalent idea. Roughly, m and n are *bisimilar* if whenever m evaluates to a value, so does n, and all the subprograms of the resulting values also bisimilar, and vice versa. We will write $m \approx^\circ_\sigma n$ to denote that m and n have the same behavior at type σ and define as: let R_σ be a family of typed binary relations on programs. Define a new binary relation by cases on types by setting, for example at $\sigma \supset \sigma'$, $m\ \Phi(R_{\sigma\supset\sigma'})\ n$ just in case whenever $m \Downarrow \lambda x.\, p$ for any p, there exists a q such that $n \Downarrow \lambda y.\, q$ and for every $r{:}\sigma$, $p[r/x]$ is $R_{\sigma'}$-related to $q[r/y]$; analogously when $n \Downarrow \lambda y.\, q$. If R_σ is a post fixed point, then $m\ R_\sigma\ n$ entails that $m\ \Phi(R_\sigma)\ n$. As we want the relation $\approx^\circ_\sigma$ to characterize "all possible behaviors", we take $\approx^\circ_\sigma$ as the *greatest* relation R_σ which is a post-fixed point of Φ, that is, bisimilarity is the set *coinductively* defined by Φ. This yields a co-induction principle, which can be set-theoretically characterized by the following rule:

$$\frac{\exists S : a \in S \qquad S \subseteq \Phi(S)}{a \in \mathrm{gfp}(\Phi)}\ CI$$

Using this principle, it is a matter of routine to show that bisimilarity is an equivalence relation. On the other hand, establishing the equivalence of specific programs requires providing the correct bisimulation S and this may be arduous. *Equational* reasoning would be helpful and this is why it is crucial to establish bisimulation to be a *congruence*.

Now, to stay closer to our proof, let me drop the symmetry part and limit myself to *similarity* and *pre-congruence*. I will also focus on an inherently trivial language, the simply-typed λ-calculus with units: everything being strongly normalizable, similarity may as well be defined by induction on types, the reader will object. This is indeed true; we need to add at least (lazy) lists to make similarity coinductively interesting[4].

Recapping, we define similarity as the greatest fix point of this coinductive

Definition 1 (Applicative simulation).

- $m \preccurlyeq_{\sigma\supset\sigma'} n$ *iff whenever* $m \Downarrow \lambda x.\, p$ *for any* $p{:}\sigma$, *there exists a* $q{:}\sigma$ *such that* $n \Downarrow \lambda y.\, q$ *and for every* $r{:}\sigma$, $p[r/x] \preccurlyeq_{\sigma'} q[r/y]$;
- $m \preccurlyeq_\top n$ *iff* $m \Downarrow \langle\rangle$ *entails* $n \Downarrow \langle\rangle$.

Let me also recall that Linc-like logics, and so Abella, adopt a proof-theoretical reading of the coinduction rule, which in this instance would read as a right introduction rule for coinductively defined predicates, e.g., in the (interesting) case of simulation:

$$\frac{\Gamma \vdash m\ S\ n \qquad u\ S\ v, u \Downarrow \lambda x.\, p \vdash \exists q, v \Downarrow \lambda y.\, q \wedge \forall r{:}\sigma, p[r/x]\ S\ q[r/y]}{\Gamma \vdash m \preccurlyeq_{\sigma\supset\sigma'} n}\ CIR$$

Some further definitions: a (typed) pre-congruence is a *compatible* transitive relation $\mathcal{R}_\sigma$, i.e., such that it respects the way λ-terms are constructed:

(C1) $\Gamma, x{:}\sigma \vdash x\ \mathcal{R}_\sigma\ x$;

(C2) $\Gamma, x{:}\sigma \vdash m\ \mathcal{R}_{\sigma'}\ n$ entails $\Gamma \vdash (\lambda x.\, m)\ \mathcal{R}_{\sigma\supset\sigma'}\ (\lambda x.\, n)$;

(C3) $\Gamma \vdash m_1\ \mathcal{R}_{\sigma\supset\sigma'}\ n_1$ and $\Gamma \vdash m_2\ \mathcal{R}_\sigma\ n_2$ entails $\Gamma \vdash (m_1\ m_2)\ \mathcal{R}_{\sigma'}\ (n_1\ n_2)$;

Because of the presence of variable-binding operators, we have switched to typed relations over *open* terms, that is families of binary relations indexed by variable typing Γ in addition to depending on types σ. We say that a relation is *substitutive* if

(Sub) $\Gamma, y{:}\sigma \vdash m\ \mathcal{R}_{\sigma'}\ m'$ and $\Gamma \vdash n\ \mathcal{R}_\sigma\ n'$ entails $\Gamma \vdash m[n/y]\ \mathcal{R}_{\sigma'}\ m'[n'/y]$.

Some other properties are admissible: a compatible relation is *reflexive* and typed relations enjoy *weakening* (or monotonicity):

(Mon) $\Gamma \vdash m\ \mathcal{R}_\sigma\ n$ and $\Gamma \subseteq \Gamma'$ entails $\Gamma' \vdash m\ \mathcal{R}_\sigma\ n$.

Moreover, if $\mathcal{R}_\sigma$ is substitutive *and* reflexive, then it is also *closed under substitution*:

(Cus) $\Gamma, y{:}\sigma \vdash m\ \mathcal{R}_{\sigma'}\ m'$ and $\cdot \vdash n : \sigma$ entails $\Gamma \vdash m[n/y]\ \mathcal{R}_{\sigma'}\ m'[n/y]$.

Our aim is to show that similarity is a pre-congruence, but Def. 1 applies only to closed terms. It is therefore customary to *extend* similarity to *open terms* via instantiation. For $\Gamma = x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n$, define

$$\Gamma \vdash m \preccurlyeq^\circ_\sigma m' \text{ iff for all i and closed } p_i{:}\sigma_i,\ m[p_i/x_i] \preccurlyeq_\sigma m'[p_i/x_i]$$

Now, it is easy to show that open similarity is a pre-order and hence (C1) and transitivity hold. Further, (C2) also holds, since simulation satisfies

$$\lambda x.\, m \preccurlyeq_{\sigma\supset\sigma'} \lambda x.\, n \text{ iff for all } p{:}\sigma,\ m[p/x] \preccurlyeq_{\sigma'} n[p/x]$$

However, a direct attempt to prove pre-congruence of open similarity breaks down when dealing with (C3), for which one needs (Sub). In fact, while open similarity is by construction closed under substitution, it is not obvious that it is substitutive. Howe's idea was to introduce a *candidate* relation $\preccurlyeq^{\mathcal{H}}$ (depicted in Fig. 1), which

[3] Not that we were not upfront about it: " Moreover, the proof development is not complete [...] and there are a small number of lemmas which are currently postulated rather than proved." Page 2 of [20]

[4] Alternatively, one could stick to the untyped λ-calculus as in [25], where there is no way to define applicative similarity inductively. Further, the machinery to handle PCF with lazy lists following [24] is exactly what I am going to describe here; however, in the phase of proof exploration, dealing with the plethora of inductive cases generated by a richer language, being Abella strictly a proof-checker, would have made my life miserable. Now that the road is clear, I foresee no problem in treating a non trivial language.

$$\frac{\Gamma \vdash \langle\rangle \preccurlyeq^{\circ}_{\top} n}{\Gamma \vdash \langle\rangle \preccurlyeq^{\mathcal{H}}_{\top} n}\, ep \qquad \frac{\Gamma, x{:}\sigma \vdash x \preccurlyeq^{\circ}_{\sigma} n}{\Gamma, x{:}\sigma \vdash x \preccurlyeq^{\mathcal{H}}_{\sigma} n}\, var \qquad \frac{\Gamma, x{:}\sigma \vdash m \preccurlyeq^{\mathcal{H}}_{\sigma'} m' \qquad \Gamma \vdash \lambda x.\, m' \preccurlyeq^{\circ}_{\sigma\supset\sigma'} n}{\Gamma \vdash \lambda x.\, m \preccurlyeq^{\mathcal{H}}_{\sigma\supset\sigma'} n}\, fun$$

$$\frac{\Gamma \vdash m_1 \preccurlyeq^{\mathcal{H}}_{\sigma\supset\sigma'} m_1' \quad \Gamma \vdash m_2 \preccurlyeq^{\mathcal{H}}_{\sigma} m_2' \qquad \Gamma \vdash m_1'\, m_2' \preccurlyeq^{\circ}_{\sigma'} n}{\Gamma \vdash m_1\, m_2 \preccurlyeq^{\mathcal{H}}_{\sigma'} n}\, app$$

Figure 1. Howe's relation

- contains (open) similarity,
- can be shown to be "almost" a substitutive pre-congruence, i.e., being semi-transitive, see (1),

and then to prove that it does coincide with similarity.

The informal proof consists of several lemmata:

1. Semi-transitivity: the composition of the Howe relation with open similarity is contained in the former. The proof goes by case analysis using transitivity of open similarity.
2. Howe is reflexive. Induction on typing, using reflexivity of open similarity.
3. Compatibility: (C2) and (C3) hold, an easy consequence of (2).
4. Open similarity is contained in Howe, which follows immediately from (1) and (2).
5. The Howe relation is substitutive. By induction on the first premise, using (1) and (Cus) of open similarity.
6. The Howe relation "mimics" the simulation conditions:
   - If $\lambda x.\, m \preccurlyeq^{\mathcal{H}}_{\sigma\supset\sigma'} n$, then $n \Downarrow \lambda x.\, m'$ and for every $q{:}\sigma$ we have $m[q/x] \preccurlyeq^{\mathcal{H}}_{\sigma'} m'[q/x]$. This is proven first by inversion on the Howe relation and similarity. Then apply semi-transitivity and substitutivity of Howe.
   - If $\langle\rangle \preccurlyeq^{\mathcal{H}}_{\top} n$, then $n \Downarrow \langle\rangle$.
7. ("downward closure") If $p \preccurlyeq^{\mathcal{H}}_{\sigma} q$ and $p \Downarrow v$, then $v \preccurlyeq^{\mathcal{H}}_{\sigma} q$. Induction on evaluation, and inversion on Howe and simulation, with an additional case analysis on v.
8. $p \preccurlyeq^{\mathcal{H}}_{\sigma} q$ entails $p \preccurlyeq_{\sigma} q$. By coinduction, using the obvious invariant, point (6) and (7).

Once all of these properties have been proved, we are ready for the main result:

Theorem 2. $\Gamma \vdash p \preccurlyeq^{\mathcal{H}}_{\sigma} q$ *iff* $\Gamma \vdash p \preccurlyeq^{\circ}_{\sigma} q$

Proof Right to left is point (4) above. Conversely, proceed by induction on Γ using (8) for the base case and closure under substitution for the step.

Corollary 3. *Open similarity is a pre-congruence.*

3. Encoding in Abella

I assume familiarity with the HOAS approach to representing object logics (OL), for which many sources are available, e.g. [11]. I also will not motivate the two-level architecture, but I will pay attention at which level judgments are defined.

We start with encoding the static and dynamic semantics of our OL language, which is carried out in the specification logic (SL). This is standard, so we comment no further, except to refer to Section 5 for why we need the `is_ty` judgment.

```
value (abs M).
value ep.

eval C C :- value C.
eval (app F A) C :- eval F (abs M), eval (M A) C.
is_ty (arr S S') :- is_ty S, is_ty S'.
is_ty top.

of (app F A) S :- of F (arr U S), of A U.
of (abs M) (arr S U) :- is_ty S,
                        pi x\ (of x S => of (M x) U).
of ep top.
```

We then establish some classic properties such as determinism of evaluation, value soundness and subject reduction in the usual very neat way that HOAS permits. We exemplify the former just to make a point about Abella's concrete syntax.

```
Theorem eval_det: forall E C1 C2,
                  {eval E C1} -> {eval E C2} -> C1 = C2.
```

Provability in the SL is denoted by curly brackets, so the statement reads: "if there is a (SL) derivation of `eval E C1` and of `eval E C2`, then `C1` and `C2` can be proven to be the same value in the meta-logic" (ML). The proof goes by induction on the height of the first derivation. Note that implication and universal quantification in the ML are denoted by `->` and `forall`, while in the SL by `:-` and `pi`, where `x\ F` is concrete syntax for Abella's lambda abstraction. For the sake of brevity from now on we will suppress the outermost `forall` quantification, unless there are interesting quantifier alternations and use the abbreviation `Thm`.

The OL typing context is translated into a SL context, for which we define a context predicate `ctx` that reifies the informal standard grammar:

$$\Gamma ::= \cdot \mid \Gamma, x{:}\sigma$$

This is adequately and succinctly encoded via inductive definitions in the ML, sporting *nominal abstraction*, more commonly known as "nabla in the head" [12]. We also introduce a predicate that recognizes when a term is a bound variable.

```
Define ctx: olist -> prop by
  ctx nil ;
  nabla x, ctx (of x S :: L) := {is_ty S} /\ ctx L.

Define name : tm -> prop by
  nabla x, name x.
```

Because context predicates are inductively defined, lemma such as the following are ubiquitous, but have very easy stereotyped inductive proofs.

```
Thm ctx_member: ctx L -> member (of X S) L ->
                          name X /\ {is_ty S}.
```

So far, nothing controversial. Looking ahead, the candidate relation in Fig. 1 is any HOAS fan's wildest dream. It begs to be formalized as a third order hypothetical judgments. Further, it is substitutive, which should have a neat proof such as the one in the Abella library for parallel reduction in the Church-Rosser theorem: this points to a SL encoding, where the *fun* case would fold into itself the *var* case like that:

```
howe (abs M) N (arr S S') :-
  (pi x\ (pi Q\ howe x Q S :- sim x Q S) =>
      sigma M'\ howe (M x) (M' x) S',
                sim (abs M') N (arr S S')).
```

There are two problems with this approach, one minor, the other much more serious. Firstly, Abella's SL is currently restricted to second-order logic and does not further allow (contrary to λProlog) existentials in the scope of an implication. We can get around this by rewriting the relevant clauses introducing an undefined judgment, say `hassn`, as a place-holder for the *var* case and an additional one `hbody` to code up the existential:

```
howe (abs M) N (arr S S'):-
     (pi x\ hassn x S => hbody (M x) N (arr S S').
howe  X M S :-
       hassn X S, sim X M S.
hbody (M X) N (arr S S'):-
      howe (M X) (M' X) S',
      sim (abs M') N (arr S S').
```

This is not pretty, but seems viable. The more pressing problem is that the candidate relation *depends* on the notion of simulation and the latter, being coinductively defined, has to live at the ML level, at least in Abella's official architecture. And if a judgment calls anything at the meta-level, then it must be at the meta-level. In other terms, once you go to the "dark side", there is no turning back. The connection between the two levels is "one way" only: the specification-level can never look outside of itself, although the meta-level may look at the specification-level, since the latter is embedded in the former[5]. Thus, we have not much choice in formalizing the candidate relation at the ML.

Back to simulation. Here is a first attempt:

```
CoDefine sim : tm -> tm -> ty -> prop by
 sim M1 M2 (arr S S') :=
     (forall M, {eval M1 (abs M)} ->
        exists M' , {eval M2 (abs M')} /\
          (forall x, {of x S} -> sim (M x) (M' x) S'));
 ...
```

Consider now reflexivity of simulation: $\forall m\,\sigma, m \preccurlyeq_\sigma m$. The informal proof goes by coinduction, followed by a case analysis on σ. However, our meta-logic does not have a notion of recursion on *individuals*, just on predicates, differently from type-theoretic logical frameworks such as Twelf where the two things are identified. We have to simulate this *relationally* via judgments embodying regular types. Hence the judgment `is_ty`. So, in Abella we state the theorem as follows:

```
Thm sim_refl_ty: {is_ty S} -> sim M M S.
```

Now the proof mirrors the informal situation; first apply coinduction and then invert on `is_ty S` to obtain the relevant sub-cases. But here is the rub: every time we use this result, which is often enough, we need to show that `S` is indeed a type. We are faced with the need either to make every relevant theorem using reflexivity of simulation hypothetical on the typing annotation, or have those annotations be *derivable*. However, the first approach is not general enough, since deep in the induction there may be type obligations stemming from existentials – cf. the *app* case of the Howe relation (Fig. 1) – that we will not be able to discharge.

What we have overlooked is that mathematical judgments such as typed similarity yield by construction:

$$m_1 \preccurlyeq_\sigma m_2 \quad \Longrightarrow \quad m_i : \sigma \tag{1}$$

Hence, we revise the definition of simulation so that the above immediately holds:

```
 sim M1 M2 (arr S S') :=
     {of M1 (arr S S')} /\ {of M2 (arr S S')} /\
     (forall M, {eval M1 (abs M)} ->
        exists M', {eval M2 (abs M')} /\
          (forall x, {of x S} -> sim (M x) (M' x) S'));

 sim N N' top :=
     {of N top} /\ {of N' top} /\
     ({eval N ep} -> {eval N' ep}).
```

Note that the typing annotation are in the *empty* context, since we are dealing here with programs. Coming back to reflexivity, we may as well prove the stronger statement:

```
Thm sim_refl: {of M S} -> sim M M S.
```

In fact it is not a complicated matter to show that

```
Thm of_ty: ctx L -> {L |- of M S} -> {is_ty S}.
```

This in turns depends on a sort of strengthening lemma for OL types[6]:

```
Thm ty_strength: ctx L -> {L |- is_ty S} -> {is_ty S}.
```

Now the proof of reflexivity proceeds as in the informal case with the notable exception that in the *fun* case, once we need to use to coinduction to establish `sim (M1 x) (M1 x) S'`, we need to establish `{of (M1 x) S'}`. Here we appeal to the substitution lemma, that is SL instantiation and cut with the assumption `{L, of n S1 |- of (M1 n) S'}` and `{of x S1}`. We mention these details, as this is the last time we will be able to use SL's tactics such as `cut` and `inst`. The statement and proof of transitivity are instead uneventful:

```
Thm sim_trans: sim M1 M2 S -> sim M2 M3 S -> sim M1 M3 S.
```

Now it is the turn of the candidate relationship: we agreed that it has to live at the ML, but in which form? We could try and write it down in its third-order incarnation, similarly to pp. 3:

```
Define howe : tm -> tm -> tp -> prop by
howe (abs M) N (arr S S') :=
  (forall x\ (forall Q\ sim x Q S -> howe x Q S) ->
      exists M'\ howe (M x) (M' x) S' /\
      sim (abs M') N (arr S S'));
...
```

Abella complains that the definition is not stratified. Even if it were, as in the case of our trivial OL language, a further problem is that such a definition does not yield a strong enough induction principle. In the end, we have to bite the bullet and use *explicit* contexts of type information, which mirror the SL typing context. It is important to keep the two in sync, as witnessed by the ubiquitous `ctx L` assumptions in the development. This is going to be significantly more painful – the SL having a much better support for contexts than the ML – but it should get us there. So we type `howe` as `olist -> tm -> tm -> ty -> prop.` But wait a minute: now that Howe is explicitly-contexted, so must be similarity. This brings back the dreaded notion of open simulation, which I hoped I had disposed of:

```
Define osim : olist -> tm -> tm -> ty -> prop by
  osim nil M1 M2 S := sim M1 M2 S;
  nabla x, osim (of x S :: L) (M1 x) (M2 x) S' :=
   nabla x, {L, of x S |- of (M1 x) S'} /\
            {L, of x S |- of (M2 x) S'} /\
            forall N, {of N S} -> osim L (M1 N) (M2 N) S'.
```

We define open simulation by inductively closing every hole, denoted by a name, with a program of the appropriate type. This is similar to the notion of *arbitrary cascading substitutions* used in [11] to generalize the induction hypothesis to open terms in a proof of strong normalization.

A first trivial, yet useful, fact is:

[5] See Section 7 for some speculation on how to overcome this.

[6] Note that the SL handles for free weakening and exchange, but not strengthening.

```
Thm osim_of: ctx L -> osim L M N S ->
               {L |- of M S} /\ {L |- of N S}.
```

Next, a sanity check that closed simulation is contained in the open one:

```
Thm sim_osim : ctx L -> sim M M' S -> osim L M M' S.
```

proven by induction on the first assumption, using one of those *pruning* lemmata that regulate (in this case forbid, since we are dealing with programs) the occurrence of names in terms:

```
Thm prune_sim: nabla (x:tm),
  sim (M x) (N x) S -> exists M', M = y\M' /\
                       exists N', N = y\N'.
```

We extend the preorder properties to open simulation with straightforward inductive proofs on the first judgment:

```
Thm osim_refl: ctx L -> {L |- of P S} -> osim L P P S.
Thm osim_trans: osim L P Q S -> osim L Q R S ->
                 osim L P R S.
```

To recap, we reformulate the `howe` relation as a first-order inductive definition with an *explicit* context. We have a *var* case, distinguished by the `name` predicate, implementing the standard context look-up. The whole encoding should be very familiar to nominal logicians, although this is accomplished via "nabla in the head". The complete listing is in Figure 2.

Let me start with the main development. Although we did not type-annotate the Howe relation, we certainly did for `osim` and this is enough for an inductive proof of:

```
Thm howe_of: ctx L -> howe L M1 M2 S ->
                      {L |- of M1 S} /\ {L |- of M2 S}.
```

Then we can follow the outline at page 3: points (1),(2),(3), and (4) are immediate, exactly mirroring the informal proofs.

```
Thm howe_trans: howe L P Q S -> osim L Q R S ->
                 howe L P R S.
Thm howe_refl: ctx L -> {L |- of M S} -> howe L M M S.
Thm howe_c2: ctx L -> howe L F F' (arr S S') ->
                      howe L A A' S ->
                        howe L (app F A) (app F' A') S'.
Thm howe_c3: nabla x, ctx L ->
                    howe (of x S :: L) (M x)(M' x) S' ->
                      howe L (abs M) (abs M') (arr S S').
Thm osim_howe: ctx L -> {L |- of E S} -> osim L E F S ->
                howe L E F S.
```

The substitution lemma, step (5) at page 2, is where things get hairier. It is easy to formulate, again using a nabla to denote the place where the substitution occurs:

```
Thm howe_subst: nabla x,
     howe (of x S :: L) (A1 x) (A2 x) S' -> {of B2 S} ->
       howe L B1 B2 S -> howe L (A1 B1) (A2 B2) S'.
```

However, we will have to sweat quite a bit to prove this. There are two orders of difficulties, one more general, the other very Abella-specific. First, the informal proof of every substitution lemma[7] requires in the binder case(s) some context manipulations before applying the inductive hypothesis: namely, an application of *exchange* (Xch) to the first derivation, if contexts are seen as *lists*, as in most textbook presentations; then of *weakening* (Mon) to the second one. Were the involved judgments defined at the SL, as in the cited case of parallel reduction, this would be invisibly taken care by the SL infrastructure. Alas, once we have an explicit context in the ML, we need to handle it ourselves. In our setup, the binder case, `(abs y\ A1 y)`, requires weakening `howe L B1 B2 S` to `howe (of y T :: L) B1 B2 S` and exchanging `howe (of y T :: of x S :: L) (A1 x) (A2 x) S'` to `howe (of x S :: of y T :: L) (A1 x) (A2 x) S'`.

As a matter of fact, the proof of (Mon) requires (Xch) for the same reasons outlined above. Further, since `howe` is defined in terms of `osim`, the latter must satisfy (Mon) and (Xch) as well. Finding a good "structural" way to formulate a notion of exchange that works for a tricky encoding such as `osim` turned out to be, to my dismay, the hardest part of the whole endeavor. As this has little to do with the mathematics of Howe's method, we relegate the details of the notion of *context swap* that we have adopted to the Appendix (A). Here, it suffices to say that we prove

```
Thm howe_xch: howe L1 M N S -> swap L1 L2 -> ctx L2 ->
                howe L2 M N S.
```

by induction on the first derivation using the analogous result for `osim`, see the Appendix. Armed with that and with `osim_mono`, we show with a similar induction:

```
Thm howe_mono: howe L M N S -> ctx L -> {is_ty T} ->
                 nabla n, howe (of n T :: L)  M N S.
```

Now for another of Abella's idiosyncrasies. So what, if we have to apply exchange to the context of an hypothesis before applying the IH? Since we do have such a lemma, that should be no problem. Not for Abella, alas. See, Abella's induction is only *structural*, forbidding us to apply a lemma before appealing to the IH, as we comment further in Section 5..

The simplest way out is to induct on a different judgment, one that does not need to be "changed" before calling the IH. In this case, we choose to induct on the shape of `(A1 x)`. We introduce a definition that is essentially a projection of the `howe` relation:

```
Define mtm : tm -> prop by
  mtm  ep;
  mtm (app E1 E2) := mtm E1 /\ mtm E2;
  mtm (abs E)     := nabla x, mtm (E x);
  mtm  X          := name X.
```

and state the substitution lemma as:

```
Thm pre_howe_subst: nabla x, mtm (A1 x) ->
     howe (of x S :: L) (A1 x) (A2 x) S' -> {of B2 S} ->
       howe L B1 B2 S -> howe L (A1 B1) (A2 B2) S'.
```

The case structure of the proof, which proceeds by induction on `mtm (A1 x)` and inversion on `howe (of x S :: L) (A1 x) (A2 x) S'` is interesting: since the "variable" case is not covered by a constructor, we get a number of spurious cases such as `name ep`, which are immediately discharged by inversion. Similarly, inversion on `name (A1 x)`, first leads to the case where the name is equal or not to `x`; in the latter the main sub case is discharged using `howe_trans`, while the absurd one arising from `x` possibly occurring in the rest of `L` is solved by a pruning lemma. The binder case requires some work so that we can apply `howe_mono` as described above. Overall, although reasoning about names is very transparent thanks to the `nabla` quantifier, the proof of a substitution lemma at the ML is significantly more heavy handed than it would be at the SL.

Because it holds that `howe L M N S -> mtm M`, we can, in this case, obtain as a corollary the original `howe_subst` formulation that we were after and use it obtain the (Cus) property.

```
Thm howe_cus: nabla x,
                howe (of x S :: L) (A1 x) (A2 x) S' ->
                  {of B S} -> howe L (A1 B) (A2 B) S'.
```

From this point up, it is reasonably easy sailing: "mimicking" lemmata (6) such as

```
Thm howe_ev_abs: howe nil (abs M) P (arr S S') ->
                  exists N,{eval P (abs N)} /\
                   forall Q, {of Q S} ->
                    howe nil (M Q) (N Q) S'.
```

[7] This observation actually applies to inductive proofs of any other property where we have a name in "head" position in a context, another example being compatibility for abstraction (C2).

```
Define howe : olist -> tm -> tm -> ty -> prop by
  howe L ep N top            := osim L ep N top;
  howe L (app F A) N S'      := exists F' A' S, howe L F F' (arr S S') /\ howe L A A' S /\
                                  osim L (app F' A') N S';
  howe L (abs M) N (arr S S'):= exists M',nabla x, howe (of x S :: L) (M x) (M' x) S' /\
                                    osim L (abs M') N (arr S S');
  howe L X M S               := name X /\ member (of X S) L /\ osim L X M S.
```

Figure 2. Encoding the Howe relation.

follow by Howe reflexivity, transitivity and substitutivity.

Once all this is in place we can state the main lemma (7):

```
Thm down_closed: {eval P V} -> howe nil P Q S ->
                   howe nil V Q S.
```

The proof, by induction on the derivation of the first judgment is very delicate, longish (around 50 instructions for the application case), and lead in a forward chaining fashion. The initial part of the case `{eval (app P1 P2) V}` is not problematic, with the IH being applied immediately after inverting on `howe nil (app P1 P2) Q S`; building on this, and only after appeal to Howe substitutivity and transitivity as well as inversion on simulation, we can then reapply the IH. At this point we proceed by case analysis on `V`, here realized via value soundness. The desired result is then achieved by reasoning on simulation assumptions and composing them by transitivity. The proof script hence contains several intermediate statements being asserted and then discharged. Here automation, in the sense of Isabelle/Coq's `auto`, is not likely to help – the same observation was made by Ambler in his Isabelle development [2], although, of course, at that time there was no *sledgehammer* facility. The latter, however, seems to be used very sparingly (as a matter of fact once) in comparable case studies such as the ones distributed with the *Nominal* package [23].

Home stretch: part (8)

```
Thm howe_sim: howe nil M N S -> sim M N S.
```

is an easy coinduction using (6) and downward closure; we finally extend this to

```
Thm howe_osim: ctx L -> howe L M N S -> osim L M N S.
```

which follows by induction on `ctx L` using `howe_sim` in the base case and closure under substitution in the step. Once we have this, it is an easy feat to move the pre-congruence properties of `howe` to `osim` and we are done.

4. Encoding in $\forall p$Abella

Abella originated as a logical framework with a first-order proof theory, because proving the consistency for Linc-like logics is quite intricate already in this setting – let me refer to [27] for more details. However, a version of Abella with predicate quantification is (informally) available. Here we simply remove the type checking limitations over the occurrences of the **prop** and **o** types. Everything else stays the same: in this sense $\forall p$Abella is conservative w.r.t. first-order Abella.

I do not have to argue about the benefits of predicate quantification: in the present setting it allows us to structure the proof, unsurprisingly, in a much more abstract way. In fact, Howe's definitive account of the method [15] is *generic* in the construction of the candidate relation $\mathcal{R}^H$ from a given putative equality relation $\mathcal{R}$. We have so far instantiated $\mathcal{R}$ with $\preccurlyeq^\circ$ and $\mathcal{R}^H$ with $\preccurlyeq^{\mathcal{H}}$, but this is just one possible choice. In their full generality the candidate relation rules are depicted in Fig. 3, together with the encoding in $\forall p$Abella. Note how the `cand` definition is parametric in an unspecified relation `R` of the appropriate type `olist -> tm -> tm -> ty -> prop`. Note also that $\forall p$Abella is *not* polymorphic (yet) and so we can only talk about typed binary relations over our specific signature.

The next step is "axiomatizing" the properties of $\mathcal{R}$, resolving some issues that were entangled in the concrete definition of `osim`. Beyond being a pre-order, we require our relation to be well-typed, invariant under swapping, monotonic and closed under substitution. We call this a `cus_preorder`:

```
wt R   := forall L M N S, R L M N S ->
            ctx L -> {L |- of M S} /\ {L |- of N S}.
xch R  := forall L1 L2 X Y S,  R L1 X Y S ->
            swap L1 L2 -> ctx L2 -> R L2 X Y S.
mono R := forall L M N S S', nabla n, R L M N S ->
            ctx L -> {is_ty S'} -> R (of n S' :: L) M N S.
cus R  := forall L M1 M2 S S', nabla x,
            R (of x S :: L) (M1 x) (M2 x) S' ->
              forall N, {of N S} -> R L (M1 N) (M2 N) S'.

cus_preorder R := refl R /\ trans R /\ xch R /\
                  mono R /\ cus R /\ wt R.
```

We now lift[8] the relevant "structural" properties from $\mathcal{R}$ to $\mathcal{R}^{\mathcal{H}}$:

```
Thm cand_of:   forall R, wt R -> wt (cand R).
Thm cand_xch:  xch R ->  wt R -> xch (cand R).
Thm cand_mono: cus_preorder R -> mono (cand R).
```

We restate the main properties of the Howe relation that we listed at page 3 and encoded in the previous Section:

```
Thm cand_trans: trans R -> cand R L P Q S ->
                  R L Q R S -> cand R L P R S.
Thm cand_refl: refl R -> refl (cand R).
Thm cand_c2: refl R -> ctx L -> cand R L F F' (arr S S') ->
              cand R L A A' S ->
               cand R L (app F A) (app F' A') S'.
Thm cand_c3: nabla x, refl R -> ctx L ->
                      cand R (of x S :: L) (M x)(M' x) S' ->
                       cand R L (abs M) (abs M') (arr S S').
Thm R_cand: ctx L -> {L |- of E S} ->
              R L E F S -> cand R L E F S.
Thm pre_cand_subst: nabla x, cus_preorder R -> mtm (A1 x) ->
       cand R (of x S :: L) (A1 x) (A2 x) S' -> {of B2 S} ->
         cand R L B1 B2 S -> cand R L (A1 B1) (A2 B2) S'.
Thm cand_cus: nabla x, cus_preorder R ->
       cand R (of x S :: L) (A1 x) (A2 x) S' -> {of B S} ->
         cand R L (A1 B) (A2 B) S'.
```

The proofs are very similar to the previous ones in Section 3, except by referring to the appropriate properties of $\mathcal{R}$ rather than to the relative lemma about `osim`. For example `cand_refl` has,

[8] Thanks to the reviewers for pointing out these more "higher-order" and elegant formulations. However, we will also derive the "unfolded" versions, e.g. in the case of reflexivity, `refl R -> ctx L -> {L |- of M S} -> cand R L M M S`, which are easier to apply in the remainder of the proof.

$$\frac{\Gamma \vdash \langle\rangle\ \mathcal{R}_{\top}\ n}{\Gamma \vdash \langle\rangle\ \mathcal{R}^{\mathcal{H}}_{\top}\ n}\ ep \qquad \frac{\Gamma, x{:}\sigma \vdash x\ \mathcal{R}_{\sigma}\ n}{\Gamma, x{:}\sigma \vdash x\ \mathcal{R}^{\mathcal{H}}_{\sigma}\ n}\ var \qquad \frac{\Gamma, x{:}\sigma \vdash m\ \mathcal{R}^{\mathcal{H}}_{\sigma'}\ m' \qquad \Gamma \vdash \lambda x.\ m'\ \mathcal{R}_{\sigma\supset\sigma'}\ n}{\Gamma \vdash \lambda x.\ m\ \mathcal{R}^{\mathcal{H}}_{\sigma\supset\sigma'}\ n}\ fun$$

$$\frac{\Gamma \vdash m_1\ \mathcal{R}^{\mathcal{H}}_{\sigma\supset\sigma'}\ m_1' \quad \Gamma \vdash m_2\ \mathcal{R}^{\mathcal{H}}_{\sigma}\ m_2' \qquad \Gamma \vdash m_1'\ m_2'\ \mathcal{R}_{\sigma'}\ n}{\Gamma \vdash m_1\ m_2\ \mathcal{R}^{\mathcal{H}}_{\sigma'}\ n}\ app$$

```
Define cand: (olist -> tm -> tm -> ty -> prop) -> olist -> tm -> tm -> ty -> prop by
  cand R L ep N top             := R L ep N top;
  cand R L (app F A) N S'       := exists F' A' S, cand R L F F' (arr S S') /\ cand R L A A' S /\
                                                    R L (app F' A') N S';
  cand R L (abs M) N (arr S S'):= exists M',nabla x, cand R (of x S :: L) (M x) (M' x) S' /\
                                                    R L (abs M') N (arr S S');
  cand R L X M S                := name X /\ member (of X S) L /\ R L X M S.
```

Figure 3. The candidate relationship and its encoding in $\forall p$Abella

after unfolding, the same inductive structure of `howe_refl`, save for inverting on `refl R` rather than using `osim_refl`.

Once we have reached this point, we need to get down to earth and instantiate the general setup. First we show that `osim` is indeed a `cus_preorder` by simply applying the relevant results about `osim`: for example

```
Thm i_osim_refl: refl osim.
```

Then we commit to `osim` as what the candidate relation refines. Note that this is essentially just an abbreviation, but `Define` is the only definitional mechanism in Abella.

```
Define howe : olist -> tm -> tm -> ty -> prop by
howe L M N S:= cand osim L M N S.
```

A property such as `howe_refl` follows now from `cand_refl` by instantiating in the proof `R` with `osim` and using `i_osim_refl`.

Once this preliminary phase is completed, the proof of the main result (steps (6), (7), and (8)) is completely analogous to before and will not be detailed further.

This more abstract treatment of the candidate relation, relying on second-order quantification, is arguably more elegant than our previous attempt, since

- it clearly separates and localizes the assumptions needed in establishing the main properties of the candidate relations – e.g. well-typedness `ty` and closure under substitution `cus`, which were all mixed up in the definition of `osim` play now their different roles;
- it makes the candidate relation *independent* from the operational semantics (be it CBN, CBV, or whatever, provided it satisfies subject reduction, determinacy and value soundness) and from the notion of simulation itself, which we could vary at our leisure.

In fact, let us define *ground similarity* as in [24]:

```
sim M1 M2 (arr S S') := ...
  (forall A, {of A S} -> sim (app M1 A) (app M2 A) S');
```

I have been able to easily and quickly prove it to be a pre-congruence, with only very minor changes in the proofs of the pre-order properties of `sim`, the mimicking lemma and of downward closure.

From the perspective of the two-level style of encoding, our second order treatment has the additional benefit of potentially freeing the candidate relation from being an inductive notion depending on a coinductive one – although the concrete instantiation of $\mathcal{R}$ will likely be coinductive. This begs the question if, after all, we could encode the candidate relation at the SL level and be merrier, not having to deal with ML contexts. I discuss this further in the Conclusion.

5. Evaluation

So, what did we learn from this exercise? Let me start from the paper-and-pencil proof by Andy Pitts [24]. Pitt's account is extremely precise, although it heavily relies on the implicit machinery of type-indexed relations to maintain typing invariants on judgments such as simulation/Howe etc., for which we had to sweat. Hence, I can offer only some very minor corrections: Lemma A.5(i) (semi-transitivity of the candidate relation) is not proved by induction on the first judgment as it says in [24], but simply by inversion. The Howe relation rule for `nil` in Pitts' Figure 7 should read `list S`, not `S` and in the `bool` case there is a spurious assumption. Finally there is a typo in the `app` case of the downward closure lemma, where it should read $M'[A'/x]$, not $M'[A/x]$, but the three latter observations were obvious on paper too – although not present in the *Errata* appendix to the paper[9], suggesting that formal verification entails some serious proof-reading of the informal theorem. But the point of the exercise was not testing Pitt's mathematical skills, which were not in discussion, but the HOAS approach and in particular the two-level architecture.

What about Abella, then? Let me be clear: Abella is a beautifully designed, robust, easy to use, yet very expressive proof assistant, which provides features that no other HOAS system can, at the moment, match. In particular the tactic language is incredibly terse, while being very effective – it can be described in two pages of the already succinct reference manual, which is remarkable compared to other more established proof assistants, here to be left unnamed. This is not to say that all is peachy.

Implicit vs. explicit contexts The most striking lesson is that implicit (SL) contexts are your friends, but explicit (ML) level ones are your foes. This may have been obvious from the get go – after all, it is called a two-level approach for a reason; nevertheless the user is well advised to formulate judgments on open terms via the SL logic, whenever possible. This is not, it is probably useful to stress, just an issue of stratification: certain judgments, such as reducibility [11], can be expressed via implicit context in the ML, informally argued to be stratified, and proven adequate under modest assumptions (e.g. types not being empty). The real issue is that the meta-logic would require yet another level to fully handle in-

[9] `www.cl.cam.ac.uk/~amp12/papers/opebtp/opebtp-errata.pdf`

duction on such judgments; try (and fail) to prove type uniqueness for our OL at the ML. However, once I was compelled to use explicit ML contexts, it became clear there are also Abella-specific issues that, if addressed, could have made my life easier, without changing the design of the meta-logic. The choice of viewing contexts as *lists* is natural, but overly concrete. It probably goes back to λProlog's roots of Linc-like meta logics, where embedded implications are added to the current database in a FIFO discipline (or `asserta`, for the Prolog aficionado)[10]. Lists (and lists predicates) are handy to specify contexts and context predicates, because of the `cons`-notation, but not that handy as reasoning goes. Properties such as weakening, exchange, strengthening may not be deep, but often enough yield surprisingly annoying proof obligations, viz. the case of (Sub). This is an old point, one that HOAS supporters have enjoyed making against other more concrete approaches, and it is somewhat ironical that it crawls back in Abella when working at the meta-level. In fact, the SL is engineered so that contexts are viewed as bags, but this is deep in the OCaml code implementing the prover. However, no such support is offered for meta-level contexts. This also has a bad interaction with Abella's restriction to structural induction, as we elaborate next. So, maybe it is time to have a stronger support for ML contexts[11].

Structural (co)induction Abella employs an annotation-based approach [10] to track when a (co)inductive hypothesis can safely be applied. This means that the user does not have to mess around with numeric values denoting derivations height, as it is the case in Hybrid. However, such a luxury comes with a price: the hassle with non-structural induction. We are not allowed to apply a lemma, even one that does not increase the height of the proof, viz. exchange, before applying the induction hypothesis. Sometimes it is not that difficult to sneak around it, as we did in the proof of `howe_subst`. In other cases, we may have to decorate the judgments with explicit heights and then proceed by complete induction. This is OK in small case studies, but once indexes are introduced, they tend to pollute the rest of the development. It would be worthwhile to relax the annotation-based system to tolerate the application of non-height increasing lemmata. We could begin perhaps, via a `trust me` directive, or just returning a warning, such as the one we receive with (non)stratification.

Weakness of the type system To maintain type invariants such as (1) introduced earlier on page 4, I had to pepper the relevant judgments with enough type annotations so that results as the aforementioned would follow immediately. You have seen the end result in Section 3. This is not only arguably inelegant, but those annotations have an unpleasant backward and forward effect: the judgment `is_ty`, which we introduced to perform case analysis while reasoning about similarity, percolates back to the very definition of the static semantics of the OL language, viz the type annotation in clause:

```
of (abs M) (arr S U) :- is_ty S,
                        pi x\ (of x S => of (M x) U).
```

Conversely, it creates several proof obligations that I had to discharge more often that I care for. In fact, I spent an astounding amount of time trying to minimize those type annotations, to little appreciable result. It goes without saying that in a dependently typed setting, a judgment such as simulation could be given, using Twelf syntax, the intrinsic typing

```
tm  : ty -> type.
sim : {A : ty} tm A -> tm A -> type.
```

This not only provides induction on individuals, which the standard extrinsic encoding would do as well, but keeps type invariants automatically. The issue of type annotations was acknowledged by the Abella designers, see [11], which envisioned an automatic synthesis of those judgments. This has not been followed up and adding recursion on individuals to Linc-like logics seems clearly preferable. As a side remark, in Hybrid we can define OL types (in this case, as they do not contain any binders) as a datatype in the host system, and thus inherit an intrinsic case analysis principle; the informal proof of reflexivity of similarity can then be replicated without the additional judgment. Type invariants in judgments are still an issue though, as Hybrid, even in its Coq incarnation, does not mesh well with dependent types.

Somewhat to my surprise, I did not miss *polymorphism* too much, although together with predicate quantification it would have made my treatment of relations more general.

Brittleness of proof scripts To be fair, this is not just a Abella's problem, but it is more compelling here due to the lack of automation. To begin with, the system has a somewhat naive way to name hypotheses, namely H_{i+1}. Whenever the user has to change the set of assumptions of a given theorem, all the `apply` commands will refer now to the H_i hypotheses and thus the script breaks down. Some relief is offered by the `backchain` tactic, which, by matching the current (atomic) goals with an assumption and keeping the focus, does not need to be told which hypothesis to use. Further, the possibility to use `apply with unknown` again makes the script more independent of hypothesis numbering, although the inductive hypothesis must (unsurprisingly) always be explicitly given [10]. It is natural to suggest the possibility of user-given names to hypotheses[12], but it is a slippery slope: one then complains that the top level interaction with the checker is underdeveloped, especially if that someone is coming from old style Isabelle interaction, and in a flash we have on our hands a full-fledged *tactical* language. I do not have a prescription to resolve the well-known tension between keeping the system simple and moving to a more developed tactical language. Still, I missed simple things as the goal stack, postponing/preferring a goal, or simply renaming teh variables in an assumption, for when Abella's choice of names is not ideal. More in general, the limitations of dealing with a bare-to-the-bones proof checker become painfully clear when the OL system under study gets richer, even in the mildest sense of moving from our λ-calculus to PCF; proofs do not get more difficult, only longer and therefore harder to handle. Just the possibility of creating "macros" of sequences of commands would be a plus.

6. Related work

The first HOAS-like formal verification of the congruence of a notion of bisimilarity concerned the π-calculus [13] and was carried out using the *Theory of Contexts*. Weak HOAS here works reasonably well, if you can live with an axiomatic approach, as this case study does not need hypothetical judgments, which are only partially supported in such a style. However, Abella's take to the same issue [26] is much preferable; that paper details, among so much more, a proof that similarity is pre-congruence for the finite π-calculus, available from the Abella example suite. The encoding is most elegant, where all issues involving bindings, names, and substitutions are handled declaratively without explicit side-conditions, thanks to the ∇-quantifier. However, when the authors move to formally verify the bisimilarity of two specific (very simple) processes, it is fair to say that a proof checker such as Abella is not the right tool for such an endeavor. The web appendix of

[10] This is also the view taken in the Twelf/Beluga school of thought [7], arguably as influenced by logic programming as Abella was.

[11] I am happy to report that, since I started on this case study, the Abella team has committed to provide a primitive notion of bag with which to encode contexts.

[12] Again, I am happy to report that the development version of Abella now provides support for (re)naming hypothesis.

the paper also contains full adequacy proofs relating coinductive notions such as similarity to their formal encoding, to which I refer, in lieu of showing myself the adequacy of my encodings. Those very detailed and non-trivial proofs should be compared to the more sketchy ones done in a Coq setting (e.g. [5]), which seem to appeal to a dubious "structural corecursion", that is to a notion of circular proofs in the mathematical informal world. It should be possible, albeit not without considerable effort, to reformulate this following [26].

In [2] the authors verify in Isabelle/HOL 98[13] the same result of the present paper and a bit more (they also show that similarity coincides with contextual pre-order) for PCFL using DB indexes as an encoding techniques for binders. The development, for the congruence part, consists of around 160 lemmas/theorems, showing not only Ambler's sheer tenacity and capability with a tool that was very different from what we know now – in particular the coinductive package had some limitations, e.g. it would not allow implications in the antecedents, forcing the authors to a more convoluted encoding of applicative simulation – but also, as often remarked by Randy Pollack, that a concrete representation such as DB will get you there, although you may not want to do this again any time soon.

I am not aware of specific related work using the *Nominal* package, but, looking backwards our development of the properties of the candidate relation is not dissimilar to what one would one do in nominal logic, namely dealing somewhat extensively with explicit contexts and fresh names. The lack of HOAS in implementing substitution and SL contexts would be likely leveraged by the automation provided by Isabelle.

Coinduction in Agda is realized by *delay*, i.e. with suspension type constructors akin to the way laziness is implemented in strict functional languages. See [6] for some preliminary applications to operational semantics. In our setting, this would have the benefit of bypassing coinduction completely, and allowing us, at least theoretically, to work entirely in the SL. On the other hand, working with *thunks* is never problem-free [28] and the jury is still out on its feasibility wrt such a complex case study. A similar approach can be taken in Twelf or, more perspicuously in *Celf*, possibly exploiting the CLF monad as a delay monad, see Maxime Beauquier's forthcoming dissertation at ITU.

7. Conclusions and future work

I think we agree that the encoding of Howe's method is an interesting and challenging case study for HOAS-based proof assistants. While I am pleased about the present formalization, especially the one in $\forall p$Abella, there are still other avenues to explore, all leading to avoiding using ML contexts in the first place.

As we left off at the end of Section 4, we could push the candidate relationship at the SL level with the essential use of predicate quantification:

```
type cand (tm -> tm -> ty -> o) -> tm -> tm -> ty -> o.

cand R (abs M) N (arr S S'):-
 (pi x\ (pi Q\ cand R x Q S :- R x Q S) =>
   sigma M'\ cand R (M x) (M' x) S', R (M' x) N (arr S S')).
```

Again, we have to massage it to make it digestible to Abella's SL and this is not completely straightforward. More seriously, any way we decide to axiomatize relations, either at the SL or the ML, we will eventually have to communicate results between the type **o** and the type **prop**, once the instantiation of R with a coinductive notion is done, in other terms we will need some form of *reflection*.

[13] And as such that development, together with [20], is lost, like tears in rain.

A second avenue would be to to try and push coinduction itself to the specification logic, by making the latter coinductive, i.e. admitting infinite proofs. In this way similarity would be encoded and analyzed at that level. While the notion of *cyclic* proofs [4] has been somewhat disappointing, it would be interesting to try and apply Mints' continuous cut elimination to the SL to recover this crucial property w.r.t. infinite SL proofs.

Both ideas point to a general desire in the Abella community to "open up the box" of the SL, so far being wired to second order hereditary Harrop formulæ, to make it "user-programmable", be it higher-order, coinductive, sub-structural, you name it. Of course logical correspondence between levels must be preserved, hence caution is the word.

Eduardo de Filippo used to say "exams never end"; so additional interesting benchmarks for Abella are in store, taking inspirations from a series of papers by Nakata and Uustalu [21, 22], already fully formalized in Coq. The former requires reasoning by mutual coinduction, which is not currently supported by Abella, the latter sports an interesting combinations of induction and coinduction under similar constraints to what we have: being strictly constructive and dealing with a monolithic notion of coinduction, i.e., in Coq the `CoFixpoint` approach, in Abella "structural" coinduction. We are of course better positioned in generalizing Nakata and Uustalu's approach from variants of the *While* language to λ-calculi. One avenue we are actively pursuing is using their trace-based approach to get Leroy's "Coinductive big-step operational semantics" [18] right, fixing its encoding inadequacies, its over-use of classical logic and hopefully providing an operational semantics for which a coinductive big step reading of the progress lemma holds: well typed programs either converge, diverge or raise an exception.

Finally, once we are satisfied with the consistency of $\forall p$Abella, we can embark on HOAS-style relational reasoning in the sense of [17], towards more modular and less labor-intensive proofs – remember the arguably heavy-handed [9] – of results such as the CIU theorem [25].

Proof scripts are available at:
`homepages.inf.ed.ac.uk/amomigl1/Howe`

Acknowledgments

Simon Ambler and Roy Crole introduced me to coinduction and to Howe's method. Thanks to Andrew Gacek for his help with Abella in general and in particular with the inductive structure of the Howe substitution lemma, to David Baelde, who told me to embrace the use of meta-level contexts. This has been made somewhat less painful by discussions with Amy Felty and Brigitte Pientka. Let me also acknowledge a decade-long and still ongoing discussion with Carsten Schürmann about coinduction in LF. Finally, thanks to the reviewers for several insightful remarks.

A. Appendix: on exchange

After many wrong turns, I settled on a notion of *context swap* inspired by the elegant structural definition of permutations given by Paulson, Rasmussen and Voelker in `http://isabelle.in.tum.de/library/HOL/HOL-Algebra/Permutation.html`. As I do not need the full power of permutations, I dropped the transitivity requirement and ended up with a notion of a single swap occurring somewhere in a context.

```
Define swap : olist -> olist -> prop by
       swap nil nil;
       swap (X :: Y :: L) (Y :: X :: L);
       swap (X :: XS) (X :: YS) := swap XS YS.
```

Once I had the "right" definition, all following theorems have reasonably easy proofs.

While swapping is reflexive and symmetric, we only need one direction of the fact that swapping does not affect the set underlying the context:

```
Thm swap_member: swap XS YS -> member X XS -> member X YS.
```

The usual pruning lemmata allow us to remove spurious dependencies from swappable contexts:

```
Thm swap_prune1: nabla (n:tm), swap XS (YS n) ->
                   exists YS', YS = y\YS'.
Thm swap_prune2: nabla (n:tm), swap (YS n) XS ->
                   exists YS', YS = y\YS'.
```

Lastly, I need to establish my swapping credentials w.r.t. OL types: first, that swapping preserves contexts and then, of course, typing judgments.

```
Thm swap_ctx_pres: swap L1 L2 -> ctx L2 -> ctx L1.
Thm swap_of_pres:  swap L1 L2 -> ctx L1 ->
                     {L1 |- of M S} -> {L2 |- of M S}.
```

Now we are ready to establish (Xch) and (Mon) for open simulation:

```
Thm osim_xch: osim L1 M N S -> swap L1 L2 -> ctx L2 ->
                osim L2 M N S .
```

The proof goes by induction on `osim L1 M N S` and case analysis on `swap L1 L2`. The base case follows from `sim_osim`. The step employs `swap_ctx_pres, swap_of_pres` to get the typing invariants right. Finally, (Mon) makes an essential use of `osim_xch`:

```
Thm osim_mono: osim L M N S -> ctx L -> {is_ty T} ->
                 nabla n, osim (of n T :: L) M N S.
```

References

[1] Simon Ambler, Roy Crole, and Alberto Momigliano. Combining higher order abstract syntax with tactical theorem proving and (co)induction. In V. A. Carreño, editor, *TPHOLs*, volume 2342 of *LNCS*. Springer Verlag, 2002.

[2] Simon Ambler and Roy L. Crole. Mechanized operational semantics via (co)induction. In Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, and Laurent Théry, editors, *TPHOLs*, volume 1690 of *Lecture Notes in Computer Science*, pages 221–238. Springer, 1999.

[3] Brian E. Aydemir et al. Mechanized metatheory for the masses: the POPLMARK challenge. In Joe Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference*, Lecture Notes in Computer Science, pages 50–65. Springer, 2005.

[4] James Brotherston and Alex Simpson. Sequent calculi for induction and infinite descent. *J. Log. Comput.*, 21(6):1177–1216, 2011.

[5] Alberto Ciaffaglione. A coinductive semantics of the unlimited register machine. In Fang Yu and Chao Wang, editors, *INFINITY*, volume 73 of *EPTCS*, pages 49–63, 2011.

[6] Nils Anders Danielsson. Operational semantics using the partiality monad. Agda Implementors' Meeting XI, Awaji Yumebutai, 2010.

[7] Amy Felty and Brigitte Pientka. Reasoning with higher-order abstract syntax and contexts: A comparison. In *ITP 2011*, volume 6172 of *LNCS*, pages 227–242. Springer, 2010.

[8] Amy P. Felty and Alberto Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *Journal of Automated Reasoning*, 48(1):43–105, 2012.

[9] Jonathan Ford and Ian A. Mason. Formal foundations of operational semantics. *Higher-Order and Symbolic Computation*, 16(3):161–202, 2003.

[10] Andrew Gacek. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems*. PhD thesis, University of Minnesota, September 2009.

[11] Andrew Gacek, Dale Miller, and Gopalan Nadathur. Reasoning in Abella about structural operational semantics specifications. *Electr. Notes Theor. Comput. Sci.*, 228:85–100, 2009.

[12] Andrew Gacek, Dale Miller, and Gopalan Nadathur. Nominal abstraction. *Inf. Comput.*, 209(1):48–73, 2011.

[13] Furio Honsell, Marino Miculan, and Ivan Scagnetto. Π-calculus in (co)inductive type theories. *Theoretical Computer Science*, 2(253):239–285, 2001.

[14] D. J. Howe. Equality in lazy computation systems. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 198–203, 1989.

[15] Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

[16] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. The marriage of bisimulations and kripke logical relations. In John Field and Michael Hicks, editors, *POPL*, pages 59–72. ACM, 2012.

[17] S. B. Lassen. *Relational Reasoning about Functions and Nondeterminism*. PhD thesis, Dept of Computer Science, Univ of Aarhus, 1998.

[18] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, 2009.

[19] Raymond McDowell and Dale Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic*, 3(1):80–136, January 2002.

[20] Alberto Momigliano, Simon Ambler, and Roy L. Crole. A Hybrid encoding of Howe's method for establishing congruence of bisimilarity. *Electr. Notes Theor. Comput. Sci.*, 70(2), 2002.

[21] Keiko Nakata and Tarmo Uustalu. Trace-based coinductive operational semantics for while. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 375–390. Springer, 2009.

[22] Keiko Nakata and Tarmo Uustalu. Resumptions, weak bisimilarity and big-step semantics for while with interactive I/O: An exercise in mixed induction-coinduction. In Luca Aceto and Pawel Sobocinski, editors, *SOS*, volume 32 of *EPTCS*, pages 57–75, 2010.

[23] Nominal Methods Group. Nominal Isabelle. `isabelle.in.tum.de/nominal`, 2008, Accessed Sun Feb 14 2010.

[24] A. M. Pitts. Operationally Based Theories of Program Equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, 1997.

[25] A. M. Pitts. Howe's method for higher-order languages. In D. Sangiorgi and J. Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, volume 52, chapter 5, pages 197–232. Cambridge University Press, November 2011.

[26] Alwen Tiu and Dale Miller. Proof search specifications of bisimulation and modal logics for the π-calculus. *ACM Trans. Comput. Logic*, 11(2):1–35, 2010.

[27] Alwen Tiu and Alberto Momigliano. Cut elimination for a logic with induction and co-induction. *CoRR*, abs/1009.6171, 2010.

[28] Philip Wadler, Walid Taha, and David B. MacQueen. How to add laziness to a strict language withouth even being odd. In *Proceedings of the 1998 ACM Workshop on ML*, pages 24–30, Baltimore, 1998.