Math. Struct. in Comp. Science (2019), *vol.* 29, *pp.* 1309–1343. © Cambridge University Press 2018
doi:10.1017/S0960129518000415 First published online 31 October 2018

A case study in programming coinductive proofs: Howe's method

ALBERTO MOMIGLIANO[†], BRIGITTE PIENTKA[‡]
and DAVID THIBODEAU[‡]

[†]*Dipartimento di Informatica, Università degli Studi di Milano, Milano, Italy*
Email: alberto.momigliano@unimi.it
[‡]*School of Computer Science, McGill University, Montreal, Canada*
Email: bpientka@cs.mcgill.ca, david.thibodeau@mail.mcgill.ca

Received 1 June 2017; revised 25 September 2018

Bisimulation proofs play a central role in programming languages in establishing rich properties such as contextual equivalence. They are also challenging to mechanize, since they require a combination of inductive and coinductive reasoning on open terms. In this paper, we describe mechanizing the property that similarity in the call-by-name lambda calculus is a pre-congruence using Howe's method in the Beluga formal reasoning system. The development relies on three key ingredients: (1) we give a higher order abstract syntax (HOAS) encoding of lambda terms together with their operational semantics as intrinsically typed terms, thereby avoiding not only the need to deal with binders, renaming and substitutions, but keeping all typing invariants implicit; (2) we take advantage of Beluga's support for representing open terms using built-in contexts and simultaneous substitutions: this allows us to directly state central definitions such as open simulation without resorting to the usual inductive closure operation and to encode very elegantly notoriously painful proofs such as the substitutivity of the Howe relation; (3) we exploit the possibility of reasoning by coinduction in Beluga's reasoning logic. The end result is succinct and elegant, thanks to the high-level abstractions and primitives Beluga provides. We believe that this mechanization is a significant example that illustrates Beluga's strength at mechanizing challenging (co)inductive proofs using HOAS encodings.

1. Introduction

Logical frameworks such as LF (Harper *et al.* 1993) and λProlog (Miller and Nadathur 2012) provide a meta-language (ML) for representing formal systems given via axioms and inference rules, factoring out common and recurring issues such as modelling variable bindings. They exploit the idea, dating back to Church, to use a lambda calculus as the ML to uniformly encode variable binding in our formal system. This technique is now commonly known as higher-order abstract syntax (HOAS) or (in a slightly weaker setting) 'lambda'-tree syntax (Miller and Palamidessi 1999). In particular, we can encode uniformly variable binding operators by mapping them to the lambda binder of the ML. As a consequence, variables in the object language (OL) are represented by variables in the ML and inherit thereby α-renaming and substitution from it. Moreover, this encoding technique scales to representing formal systems that use hypothetical and parametric reasoning by providing generic support for managing hypotheses and the corresponding substitution lemmas. As users do not need to build up all this basic mathematical infrastructure, it is easier to prototype proof environments and mechanize formal systems. It can also have substantial benefits for proof checking and proof search.

While representing formal systems is a first step, the interesting question is how we can *reason* about HOAS representations inductively. ML such as LF or λ-Prolog that are used for representing OL are weak calculi and do not include case analysis, recursion or inductive definitions. This is in fact essential to achieve an adequate representation of the OL where each OL term uniquely corresponds to a given representation in the ML. So how can we still reason about such representations?

One solution to this conundrum is the so-called 'two-level' approach, as advocated by McDowell and Miller (1997), where we distinguish between a specification language and a reasoning logic above it, which supports at least some form of induction. The cited paper presented $FOLD^{\mathcal{N}}$, which is basically a first-order logic with *definitions* (fixed points) and induction over natural numbers. Object logics are encoded in a specification language, which may vary and often is based on (possibly sub-structural) fragments of hereditary Harrop formulas. The method was tested on classical benchmarks such as subject reduction for PCF and its imperative variants.

Notably, one of Dale Miller's motivating examples has been the (meta-)theory of process calculi, in particular, the π-calculus. This brought to the forefront the issue of representing and reasoning about *infinite* behaviour. In fact, McDowell *et al.* (1996) were concerned with the representation of transition systems and their bisimulation: in agreement with Milner's original presentation in *A Calculus of Communicating Systems*, bisimilarity was captured *inductively* by computing the greatest fixed point starting from the universal relation and closing downwards by intersection. This is doable, but notoriously awkward to work with and in fact Milner swiftly adopted the notion of *coinduction* in his subsequent *Communication and Concurrency*.

In the late 1990, coinduction was available in general proof assistants such as Isabelle/HOL and Coq, the first by encoding the standard Tarski's fixed point theorem in higher order logic, the latter by guarded induction; several reasonably large case studies were carried out, not without some difficulties (Ambler and Crole 1999; Hirschkoff 1997; Honsell *et al.* 2001). These case studies further demonstrated the challenges in modelling variable bindings and building up such an infrastructure, as lambda-tree syntax is fundamentally incompatible with the foundations of these proof systems. It turns out instead that it is quite natural to step from $FOLD^{\mathcal{N}}$ to support (co)inductive reasoning; Momigliano and Tiu (2003) adopted the view of definitions as least and greatest fixed points adding rules for fixed point induction. This was later shown to be consistent (Tiu and Momigliano 2012). With the orthogonal ingredient of ∇-quantifier to abstract over variable names (Miller and Tiu 2005), this line of research culminated in the Abella proof assistant (Baelde *et al.* 2014), which until recently was the only proof assistant supporting natively both HOAS and coinduction, as exemplified in some non-insignificant case studies (Momigliano 2012; Tiu and Miller 2010).

The other main player in HOAS logical frameworks is LF (Harper *et al.* 1993): Pfenning advocated using it as a *meta*-logical framework by representing inductive proofs as relations. To ensure that a relation describes a valid inductive proof, external checks guarantee that the implemented relation constitutes a total function, i.e. covers all cases and all appeals to the induction hypothesis are well founded. This lead to the proof environment Twelf (Pfenning and Schürmann 1999), which has been used widely, see for a major case study (Lee *et al.* 2007). However, Twelf did not seem to lend itself to coinductive reasoning.

To address these and other shortcomings, Pientka (2008) designed a reasoning logic on top of LF that allows us to directly analyze and manipulate LF objects. Beluga (Pientka and Dunfield 2010) implements this idea. To model derivation trees that depend on assumptions, LF objects are paired with their surrounding context (Nanevski *et al.* 2008; Pientka 2008; Pientka and Dunfield 2008). Inductive proofs are then implemented as *recursive* functions that directly pattern match on contextual LF objects. Beluga provides a proof language that makes explicit context reasoning via built-in contexts and simultaneous substitutions together with their equational theory. Moreover, it supports inductive and stratified definitions in addition to higher order functions (Cave and Pientka 2012; Jacob-Rao *et al.* 2018; Pientka and Cave 2015), thereby going substantially beyond the expressive power of Twelf.

One might say that the proof and the type-theoretic approaches are converging towards a core reasoning logic that supports least and greatest fixed points and equality within first-order logic. This might be more obvious in the inductive case where we are more familiar with the computational interpretation of proofs: we readily interpret pattern matching in a program as case analysis in a proof and accept that recursive calls on structurally smaller objects correspond to well-founded appeals to the induction hypothesis. Coinductive reasoning in type theory is less well understood. In fact, guarded co-recursion in Coq, for example, does not preserve types (Oury 2008). To overcome these and other difficulties, Pientka and collaborators (Abel and Pientka 2016; Abel *et al.* 2013; Thibodeau *et al.* 2016) proposed in prior work a novel computational interpretation of coinductive proofs. While finite (inductive) data is defined using *constructors* and analyzed via *pattern matching*, we define infinite (coinductive) data by the *observations* that we can make about it. We can reason about such observations using *copattern matching*. A function about finite data represents an inductive proof, if we cover all cases and all recursive calls are on structurally smaller objects. This guarantees that the function is *total*, that is, defined on all inputs and terminating. Dually, a total function about infinite data corresponds to a coinductive proof, if we cover all possible observations on the output and all recursive calls are guarded by an observation. This guarantees that the function is defined on all possible outputs and remains productive, as we only proceed to evaluate the co-recursive function when we apply it to an observation.

As a contribution to a better understanding of the relationship between the logical and computational interpretation of coinductive proofs, the present paper reappraises the proof that similarity in the call-by-name lambda calculus with lists is a pre-congruence using Howe's method (Howe 1996). This is a challenging proof since it requires a combination of inductive and coinductive reasoning on open terms. We mechanize this proof in Beluga, relying on the following three key ingredients:

1. We give an HOAS encoding of lambda terms together with their operational semantics as *intrinsically typed* terms, thereby avoiding not only the need to deal with binders, renaming and substitutions, but keeping all typing invariants implicit.

2. We take advantage of Beluga's support for representing open terms using built-in contexts and simultaneous substitutions: this allows us to directly state a notion such as open simulation without resorting to the usual inductive closure operation and to encode neatly notoriously painful proofs such as the substitutivity of the Howe relation.
3. We exploit the possibility of reasoning by coinduction in Beluga's reasoning logic.

The end result is, in our opinion, succinct and elegant, thanks to the high-level abstractions and primitives Beluga provides.

The paper starts in Section 2 with a summary description of Howe's method and discusses the challenges that it poses to its mechanization. The latter is detailed in Section 3, together with a proof of adequacy of our encoding of similarity (Section 3.4) and an example derivation of two terms being similar (Section 3.6). We review related work in Section 4 and conclude in Section 5. Appendix A contains a brief overview of the relevant part of Beluga's syntax. The entire formal development can be retrieved from `https://github.com/Beluga-lang/Beluga/tree/master/examples/codatatypes/howes-method`.

2. A summary of Howe's method

First, let us fix our programming languages as the simply typed λ-calculus with recursion over (lazy) lists, which we call PCFL following Pitts (1997). Its types consist of the unit type (written as $\top$), function types and lists (written as $\mathsf{list}(\tau)$).

$$\begin{array}{llll}
\text{Types} & \tau & ::= & \top \mid \tau \to \tau \mid \mathsf{list}(\tau) \\
\text{Terms} & m, n, p, q & ::= & x \mid \mathsf{lam}\, x.\, p \mid m_1\ m_2 \mid \mathsf{fix}\ x.\ m \mid \langle\rangle \\
 & & & \mid \mathsf{nil} \mid \mathsf{cons}\ m_h\ m_t \mid \mathsf{lcase}\ m\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow n \mid \mathsf{cons}\ h_d\ t_l \Rightarrow p\} \\
\text{Values} & v & ::= & \langle\rangle \mid \mathsf{lam}\, x.\, p \mid \mathsf{nil} \mid \mathsf{cons}\ m_h\ m_t
\end{array}$$

The typing rules for PCFL and the big step lazy operational semantics denoted by $m \Downarrow v$ are standard and we omit them here. In particular, lists are only evaluated lazily, as the definition of values shows. The interested reader can skip ahead to their encoding in LF in the Section 3.1 or consult Pitts (1997).

2.1. *Proving bisimilarity a congruence using Howe's method*

Suppose we want to say when two programs (two closed terms) have the same behaviour. A well-known characterization is Morris-style *contextual equivalence*: occurrences of the first expression in any program can be replaced by the second without affecting the *observable* results of executing the program.

While this notion of program equivalence is intuitive, it is indeed difficult to reason about it, mainly due to the quantification on every possible context.[†] Many techniques

[†] This notion can and has been simplified, starting from Milner's context lemma (Milner 1977) and going through the CIU theorem (Mason and Talcott 1991). Some mechanizitions are also available (Ford and Mason 2003; McLaughlin *et al.* 2018), as we discuss further in Section 4.

have been proposed through the years, ranging from domain theory (Abramsky 1991), game semantics (Ghica and McCusker 2000) to logical relations (Ahmed 2006). The idea of *bisimilarity* has usefully been adapted from concurrency theory to provide yet another characterization of contextual equivalence. Bisimilarity is, similarly to contextual equivalence, parameterized by the notion of *observable* we select: roughly, m and n are *bisimilar* if whenever m evaluates to an observable so does n, and all the subprograms of those are also bisimilar. In the case of *applicative* bisimilarity, evaluation at function type is pushed until values are reached.

To simplify the presentation, we will concentrate on the notion of *similarity*, from which bisimilarity can be obtained by symmetry, that is taking the conjunction of similarity and its inverse; this is possible thanks to determinism of evaluation.

Definition 2.1 (Applicative simulation). An *applicative simulation* is a family of typed relations R_τ on closed terms satisfying the following conditions:

— If $m \mathcal{R}_\top n$, then $m \Downarrow \langle\rangle$ entails $n \Downarrow \langle\rangle$.
— If $m \mathcal{R}_{\mathsf{list}(\tau)} n$, then $m \Downarrow \mathsf{nil}$ entails $n \Downarrow \mathsf{nil}$.
— If $m \mathcal{R}_{\mathsf{list}(\tau)} n$, then $m \Downarrow \mathsf{cons}\ m_h\ m_t$ entails that there are terms m'_h and m'_t such that $n \Downarrow \mathsf{cons}\ m'_h\ m'_t$ for which $m_h \mathcal{R}_\tau m'_h$ and $m_t \mathcal{R}_{\mathsf{list}(\tau)} m'_t$.
— If $m \mathcal{R}_{\tau\to\tau'} n$, then $m \Downarrow \mathsf{lam}\, x. m'$ entails that there is a term n' such that $n \Downarrow \mathsf{lam}\, y. n'$ for which $m'[r/x] \mathcal{R}_{\tau'} n'[r/y]$ for every term r of type τ.

We can make sense of the non-well-founded nature of the last two conditions by noting that there are *non-empty* simulations, e.g. the identity relation and that the *union* of two applicative simulations is still a simulation. Hence, there exists the *largest* one, which we call applicative *similarity*. This relation can also be characterized using the Knaster–Tarski fixed point theorem, as the greatest fixed point of an appropriate endofunction Φ on families of typed relations. For a detailed explanation, we refer again to Pitts (1997), and we just mention that the definition of the function follows the simulation relation, and has, for example, at $\tau \to \tau'$, $m\ \Phi(R_{\tau\to\tau'})\ n$ just in case whenever $m \Downarrow \mathsf{lam}\, x. m'$ for any m', there exists a term n' such that $n \Downarrow \mathsf{lam}\, y. n'$ and for every r of type τ, $m'[r/x]$ is $R_{\tau'}$-related to $n'[r/y]$; hence, similarity is the set *coinductively* defined by Φ, a relation we write as $m \preccurlyeq_\tau n$. This yields a coinduction principle that we describe first in its generality and below we show it instantiated to applicative similarity.

$$\frac{\exists S \text{ s.t. } a \in S \qquad S \subseteq \Phi(S)}{a \in \mathrm{gfp}(\Phi)}\ CI$$

$$\frac{\exists S_\tau \text{ s.t. } m\ S_\tau\ n \qquad S_\tau \text{ is an applicative simulation}}{m \preccurlyeq_\tau n}\ CI-\preceq$$

It is not difficult to show that similarity is a pre-order and we detail the proof of reflexivity using rule $CI-\preceq$ to highlight the similarities with the type-theoretic definition based instead on the notion of observation, on which our mechanization relies.

Theorem 2.1 (Reflexivity of applicative similarity). $\forall m\ \tau, m \preccurlyeq_\tau m$.

Proof. To show the result, we need to provide an appropriate simulation S and check the simulation conditions. Just choose S_τ to be the family $\{(m,m) \mid \cdot \vdash m : \tau\}$, where we use the judgement $m : \tau$ to say that term m has type σ.

We then consider each case in the applicative simulation definition.

—If $m\ \mathcal{S}_\top\ m$, then $m \Downarrow \langle\rangle$ entails $m \Downarrow \langle\rangle$: immediate.
—If $m\ \mathcal{S}_{\mathsf{list}(\tau)}\ m$, then $m \Downarrow \mathsf{nil}$ entails $m \Downarrow \mathsf{nil}$: immediate.
—Assume $m\ \mathcal{S}_{\mathsf{list}(\tau)}\ m$ and $m \Downarrow \mathsf{cons}\ m_h\ m_t$; pick m'_h, m'_t to be m_h, m_t and by the definition of the simulation, it holds that $m_h\ \mathcal{S}_\tau\ m_h$ and $m_t\ \mathcal{S}_{\mathsf{list}(\tau)}\ m_t$.
—Assume $m\ \mathcal{S}_{\tau\to\tau'}\ m$ and $m \Downarrow \mathsf{lam}\, x.\, m'$; again by picking m' for n' and by the definition of the simulation it is obvious that for every $r{:}\tau$, $[r/x]m'\ \mathcal{S}_{\tau'}\ [r/x]m'$.

□

In many cases, we do not have to look much further beyond the statement of the theorem to come up with an appropriate simulation, i.e. we can read off the definition of simulation from it – and this is indeed the case for all the coinductive proofs in the following development. However, to show the equivalence of specific programs, we may have to come up with a complex bisimulation, possibly defined inductively and/or 'up to.' This phenomenon is well known in inductive theorem proving, where sometimes the induction hypothesis coincides with the statement of the theorem, but in other cases, it needs to be generalized in an appropriate lemma. The fixed point rules conflate those two aspects, generalization and lemma application, in one go. With an abuse of language, we will say that we prove a statement by coinduction and say that we appeal to the use of the 'coinductive hypothesis' when the simulation corresponds to the statement of the theorem.

When dealing with program equivalence, *equational* (in addition to coinductive) reasoning would be helpful and this is why it is crucial to establish bisimilarity to be a *congruence*, i.e. a relation respecting the way terms are constructed. Since in this paper, we restrict ourselves to similarity, we target *pre*-congruence. Given the presence of variable-binding operators, we need to consider relations over *open* terms, that is, families of relations over terms indexed by a typing context Γ in addition to a type τ, which we write as $\Gamma \vdash m\ \mathcal{R}_\tau\ n$.

Definition 2.2 (Compatible relation). A relation $\Gamma \vdash m\ \mathcal{R}_\tau\ n$ is *compatible* when

(C0) $\Gamma \vdash \langle\rangle\ \mathcal{R}_\top\ \langle\rangle$;
(C1) $\Gamma, x{:}\tau \vdash x\ \mathcal{R}_\tau\ x$;
(C2) $\Gamma, x{:}\tau \vdash m\ \mathcal{R}_{\tau'}\ n$ entails $\Gamma \vdash (\mathsf{lam}\, x.\, m)\ \mathcal{R}_{\tau\to\tau'}\ (\mathsf{lam}\, x.\, n)$;
(C3) $\Gamma \vdash m_1\ \mathcal{R}_{\tau\to\tau'}\ n_1$ and $\Gamma \vdash m_2\ \mathcal{R}_\tau\ n_2$ entail $\Gamma \vdash (m_1\ m_2)\ \mathcal{R}_{\tau'}\ (n_1\ n_2)$;
(C4) $\Gamma, x{:}\tau \vdash m\ \mathcal{R}_\tau\ n$ entails $\Gamma \vdash (\mathsf{fix}\ x.\ m)\ \mathcal{R}_\tau\ (\mathsf{fix}\ x.\ n)$;
(C5a) $\Gamma \vdash m_h\ \mathcal{R}_\tau\ n_h$ and $\Gamma \vdash m_t\ \mathcal{R}_{\mathsf{list}(\tau)}\ n_t$ entail $\Gamma \vdash (\mathsf{cons}\ m_h\ m_t)\ \mathcal{R}_{\mathsf{list}(\tau)}\ (\mathsf{cons}\ n_h\ n_t)$;
(C5b) $\Gamma \vdash \mathsf{nil}\ \mathcal{R}_{\mathsf{list}(\tau)}\ \mathsf{nil}$;
(C6) $\Gamma \vdash m_1\ \mathcal{R}_{\mathsf{list}(\tau)}\ m_2$, $\Gamma \vdash n_1\ \mathcal{R}_{\tau'}\ n_2$ and $\Gamma, h_d : \tau, t_l : \mathrm{list}(\tau) \vdash p_1 \mathcal{R}_{t'}\ p_2$ entail
$\Gamma \vdash (\mathsf{lcase}\ m_1\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow n_1 \mid \mathsf{cons}\ h_d\ t_l \Rightarrow p_1\})\ \mathcal{R}_{\tau'}\ (\mathsf{lcase}\ m_2\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow n_2 \mid \mathsf{cons}\ h_d\ t_l \Rightarrow p_2\})$.

Well-Typed Simultaneous Substitutions: $\Psi \vdash \sigma : \Gamma$

$$\frac{}{\Psi \vdash \cdot : \cdot} \qquad \frac{\Psi \vdash \sigma : \Gamma \qquad \Psi \vdash m : \tau}{\Psi \vdash \sigma, m/x : \Gamma, x : \tau}$$

Related Simultaneous Substitutions: $\Psi \vdash \sigma_1\ \mathcal{R}_\Gamma\ \sigma_2$

$$\frac{}{\Psi \vdash \cdot\ \mathcal{R}_\cdot\ \cdot} \qquad \frac{\Psi \vdash \sigma_1\ \mathcal{R}_\Gamma\ \sigma_2 \qquad \Psi \vdash m\ \mathcal{R}_\tau\ n}{\Psi \vdash (\sigma_1, m/x)\ \mathcal{R}_{\Gamma,x:\tau}\ (\sigma_2, n/x)}$$

Fig. 1. (Related) simultaneous substitutions.

Definition 2.3 (Pre-congruence). A *pre-congruence* is a compatible transitive relation.

By the very definition of simulation at arrow type, it is clear that a key property for our development is for a relation to be preserved by pairwise substitution:

$$\Gamma, y{:}\tau \vdash m_1\ \mathcal{R}_{\tau'}\ m_2 \text{ and } \Gamma \vdash n_1\ \mathcal{R}_\tau\ n_2 \text{ entails } \Gamma \vdash [n_1/y]m_1\ \mathcal{R}_{\tau'}\ [n_2/y]m_2.$$

We generalize this property here using *simultaneous* substitutions, streamlining our formal development, see Section 3.7. Figure 1 gives rules for well-typed simultaneous substitutions $\Psi \vdash \sigma : \Gamma$, where σ replaces variables in Γ with terms typable in Ψ; then, we state when two such substitutions are $\mathcal{R}$-related:

Definition 2.4 (Substitutive relation). A relation is *substitutive* **(Sub)** iff $\Gamma \vdash m_1\ \mathcal{R}_\tau\ m_2$ and $\Psi \vdash \sigma_1\ \mathcal{R}_\Gamma\ \sigma_2$ entails $\Psi \vdash [\sigma_1]m_1\ \mathcal{R}_\tau\ [\sigma_2]m_2$.

Some other properties are admissible:

Lemma 2.1 (Elementary admissible properties).

(Ref) If a relation is compatible, then it is *reflexive*; further, if $\Gamma \vdash m\ \mathcal{R}_\tau\ m$, then $\Psi \vdash \sigma\ \mathcal{R}_\Gamma\ \sigma$.

(Cus) If $\mathcal{R}_\tau$ is substitutive and reflexive, then it is also *closed under substitution*: $\Gamma \vdash m_1\ \mathcal{R}_\tau\ m_2$ and $\Psi \vdash \sigma : \Gamma$ entails $\Psi \vdash [\sigma]m_1\ \mathcal{R}_\tau\ [\sigma]m_2$.

Weakening (**Wkn**), that is $\Gamma \vdash m\ \mathcal{R}_\tau\ n$ and $\Gamma \subseteq \Gamma'$ entailing $\Gamma' \vdash m\ \mathcal{R}_\tau\ n$, follows from Cus taking σ to be the identity substitution for Γ.

The definition of similarity applies only to closed terms. It is therefore customary to *extend* similarity to *open terms* via substitution. We do this using *grounding* substitutions:

Definition 2.5 (Open similarity). $\Gamma \vdash m \preccurlyeq^{\circ}_{\tau} m'$ iff $[\sigma]m \preccurlyeq_\tau [\sigma]m'$ for any $\cdot \vdash \sigma : \Gamma$.

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

$$\mathsf{lam}\, x.\, m \preccurlyeq_{\tau\to\tau'} \mathsf{lam}\, x.\, n \text{ iff for all } p{:}\tau,\ [p/x]m \preccurlyeq_{\tau'} [p/x]n$$

However, a direct attempt to prove pre-congruence of open similarity breaks down when dealing with (C3) and (C6). Moreover, while it follows simply by construction that open similarity is closed under substitution, it is not obvious that it is substitutive.

$$\frac{\Gamma \vdash \langle\rangle \preceq^{\circ}_{\top} n}{\Gamma \vdash \langle\rangle \preceq^{\mathcal{H}}_{\top} n}\,\mathit{hu} \qquad \frac{\Gamma \vdash \mathsf{nil} \preceq^{\circ}_{\mathsf{list}(\tau)} n}{\Gamma \vdash \mathsf{nil} \preceq^{\mathcal{H}}_{\mathsf{list}(\tau)} n}\,\mathit{hnil} \qquad \frac{\Gamma, x{:}\tau \vdash x \preceq^{\circ}_{\tau} n}{\Gamma, x{:}\tau \vdash x \preceq^{\mathcal{H}}_{\tau} n}\,\mathit{hvar}$$

$$\frac{\Gamma, x{:}\tau \vdash m \preceq^{\mathcal{H}}_{\tau'} m' \qquad \Gamma \vdash \mathsf{lam}\, x.\, m' \preceq^{\circ}_{\tau\to\tau'} n}{\Gamma \vdash \mathsf{lam}\, x.\, m \preceq^{\mathcal{H}}_{\tau\to\tau'} n}\,\mathit{hlam}$$

$$\frac{\Gamma, x{:}\tau \vdash m \preceq^{\mathcal{H}}_{\tau} m' \qquad \Gamma \vdash \mathsf{fix}\ x.\ m' \preceq^{\circ}_{\tau} n}{\Gamma \vdash \mathsf{fix}\ x.\ m \preceq^{\mathcal{H}}_{\tau} n}\,\mathit{hfix}$$

$$\frac{\Gamma \vdash m_1 \preceq^{\mathcal{H}}_{\tau\to\tau'} m_1' \quad \Gamma \vdash m_2 \preceq^{\mathcal{H}}_{\tau} m_2' \qquad \Gamma \vdash m_1'\ m_2' \preceq^{\circ}_{\tau'} n}{\Gamma \vdash m_1\ m_2 \preceq^{\mathcal{H}}_{\tau'} n}\,\mathit{happ}$$

$$\frac{\Gamma \vdash m_h \preceq^{\mathcal{H}}_{\tau} m_h' \quad \Gamma \vdash m_t \preceq^{\mathcal{H}}_{\mathsf{list}(\tau)} m_t' \qquad \Gamma \vdash \mathsf{cons}\ m_h'\ m_t' \preceq^{\circ}_{\mathsf{list}(\tau)} n}{\Gamma \vdash \mathsf{cons}\ m_h\ m_t \preceq^{\mathcal{H}}_{\mathsf{list}(\tau)} n}\,\mathit{hcons}$$

$$\frac{\begin{array}{c}\Gamma \vdash m \preceq^{\mathcal{H}}_{\mathsf{list}(\tau)} m' \quad \Gamma \vdash m_1 \preceq^{\mathcal{H}}_{\tau'} m_1' \quad \Gamma, h_d{:}\tau, t_l{:}\mathsf{list}(\tau) \vdash m_2 \preceq^{\mathcal{H}}_{\tau'} m_2' \\ \Gamma \vdash \mathsf{lcase}\ m'\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow m_1' \mid \mathsf{cons}\ h_d\ t_l \Rightarrow m_2'\} \preceq^{\circ}_{\tau'} n\end{array}}{\Gamma \vdash \mathsf{lcase}\ m\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow m_1 \mid \mathsf{cons}\ h_d\ t_l \Rightarrow m_2\} \preceq^{\mathcal{H}}_{\tau'} n}\,\mathit{hlcase}$$

Fig. 2. Definition of the Howe relation.

Howe's idea (Howe 1996) was to introduce a *candidate* relation $\preceq^{\mathcal{H}}_{\tau}$ (see Figure 2), which contains (open) similarity and can be shown to be almost a substitutive pre-congruence, 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 on the derivation of the Howe relation using transitivity of open similarity.
2. The Howe relation is reflexive. Induction on typing, using reflexivity of open similarity.
3. Compatibility: (C0)–(C6) 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, see Lemma 2.2.
6. The Howe relation 'mimics' the simulation conditions:

   (6.1) If $\langle\rangle \preceq^{\mathcal{H}}_{\top} n$, then $n \Downarrow \langle\rangle$.

   (6.2) If $\mathsf{nil} \preceq^{\mathcal{H}}_{\mathsf{list}(\tau)} n$, then $n \Downarrow \mathsf{nil}$.

   (6.3) If $\mathsf{lam}\, x. m \preceq^{\mathcal{H}}_{\tau\to\tau'} n$, then $n \Downarrow \mathsf{lam}\, x. m'$ and for every $q{:}\tau$, we have $[q/x]m \preceq^{\mathcal{H}}_{\tau'} [q/x]m'$.

   (6.4) If $\mathsf{cons}\ m_h\ m_t \preceq^{\mathcal{H}}_{\mathsf{list}(\tau)} n$, then $n \Downarrow \mathsf{cons}\ p_h\ p_t$, with $m_h \preceq^{\mathcal{H}}_{\tau} p_h$ and $m_t \preceq^{\mathcal{H}}_{\mathsf{list}(\tau)} p_t$.

   By inversion on the Howe relation and definition of similarity, using semi-transitivity and, in the lambda case, substitutivity of the Howe relation.
7. Downward closure: if $p \preceq^{\mathcal{H}}_{\tau} q$ and $p \Downarrow v$, then $v \preceq^{\mathcal{H}}_{\tau} q$. Induction on evaluation, and inversion on the Howe relation and similarity, with an additional case analysis on v.
8. $p \preceq^{\mathcal{H}}_{\tau} q$ entails $p \preceq_{\tau} q$. By coinduction, using the coinductive hypothesis, points (6) and (7).

Once all of these properties have been proved, we are ready for the main result, stating that the Howe relation coincides with applicative similarity, and hence the pre-congruence of the latter follows as a corollary:

Theorem 2.2. $\Gamma \vdash p \preccurlyeq^{\mathcal{H}}_{\tau} q$ iff $\Gamma \vdash p \preccurlyeq^{\circ}_{\tau} q$.

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 2.1. Open similarity is a pre-congruence.

2.2. *On the role of substitutions in Howe's method*

Substitutions play a central role in the overall proof that similarity is a pre-congruence. In the informal proof, we silently exploit equational laws about substitution; however, they can cause significant trouble during mechanization. We summarize the definition of substitution for our term language together with its equational theory in Figure 3. To illustrate how we rely on these substitution properties in proofs, we show here in more detail the proof of substitutivity and pay particular attention to the properties in Figure 3. Recall that the definition of $\Psi \vdash \sigma_1 \preccurlyeq^{\mathcal{H}}_{\Gamma} \sigma_2$ is just an instance of the definition of related simultaneous substitutions.

Lemma 2.2 (Substitutivity of the Howe relation). Suppose we have $\Gamma \vdash m_1 \preccurlyeq^{\mathcal{H}}_{\tau} m_2$ and $\Psi \vdash \sigma_1 \preccurlyeq^{\mathcal{H}}_{\Gamma} \sigma_2$, then $\Psi \vdash [\sigma_1]m_1 \preccurlyeq^{\mathcal{H}}_{\tau} [\sigma_2]m_2$.

Proof. By induction on the derivations of $\Gamma \vdash m_1 \preccurlyeq^{\mathcal{H}}_{\tau} m_2$.

$$\text{Case}\quad \frac{\Gamma \vdash m \preccurlyeq^{\mathcal{H}}_{\tau\to\tau'} m' \quad \Gamma \vdash n \preccurlyeq^{\mathcal{H}}_{\tau} n' \qquad \Gamma \vdash m'\ n' \preccurlyeq^{\circ}_{\tau'} r}{\Gamma \vdash m\ n \preccurlyeq^{\mathcal{H}}_{\tau'} r}\ \mathit{happ}$$

$\Psi \vdash [\sigma_1]m \preccurlyeq^{\mathcal{H}}_{\tau\to\tau'} [\sigma_2]m'$ by IH on first subderivation

$\Psi \vdash [\sigma_1]n \preccurlyeq^{\mathcal{H}}_{\tau} [\sigma_2]n'$ by IH on second subderivation

$\Psi \vdash [\sigma_2](m'\ n') \preccurlyeq^{\circ}_{\tau'} [\sigma_2]r$ by cus on third subderivation

$[\sigma_2](m'\ n') = [\sigma_2]m'\ [\sigma_2]n'$ by def. of substitution (see Figure 3)

$\Psi \vdash [\sigma_1]m\ [\sigma_1]n \preccurlyeq^{\mathcal{H}}_{\tau'} [\sigma_2]r$ by rule *happ*

$\Psi \vdash [\sigma_1](m\ n) \preccurlyeq^{\mathcal{H}}_{\tau'} [\sigma_2]r$ by above line

$$\text{Case}\quad \frac{\Gamma, x{:}\tau \vdash m \preccurlyeq^{\mathcal{H}}_{\tau'} m' \qquad \Gamma \vdash \mathsf{lam}\,x.\,m' \preccurlyeq^{\circ}_{\tau\to\tau'} r}{\Gamma \vdash \mathsf{lam}\,x.\,m \preccurlyeq^{\mathcal{H}}_{\tau\to\tau'} r}\ \mathit{hlam}$$

$\Psi \vdash \sigma_1 \preccurlyeq^{\mathcal{H}}_{\Gamma} \sigma_2$ by assumption

$\Psi, x : \tau \vdash \sigma_1 \preccurlyeq^{\mathcal{H}}_{\Gamma} \sigma_2$ by weakening (Lemma 4.2)

$[\sigma]x \preccurlyeq_{\tau} [\sigma]x$ for any σ where $\cdot \vdash \sigma : \Psi, x{:}\tau$ by reflexivity of similarity (Theorem 2.1)

$\Psi, x{:}\tau \vdash x \preccurlyeq^{\circ}_{\tau} x$ by def. of open similarity

$\Psi, x{:}\tau \vdash x \preccurlyeq^{\mathcal{H}}_{\tau} x$ by rule *hvar*

Equational Theory of Simultaneous Substitution

For $\Gamma' \vdash \sigma : \Gamma$ and $\Gamma \vdash m : \tau$, we define $[\sigma]m$ as follows

$$
\begin{array}{ll}
[\sigma]x & = \sigma(x) \\
[\sigma](\mathsf{lam}\, x.\, m) & = \mathsf{lam}\, x.\, [\sigma, x/x]m \\
[\sigma](m\ n) & = [\sigma]m\ [\sigma]n \\
[\sigma](\langle\rangle) & = \langle\rangle \\
[\sigma](\mathsf{nil}) & = \mathsf{nil} \\
[\sigma](\mathsf{cons}\ m\ n) & = \mathsf{cons}\ [\sigma]m\ [\sigma]n \\
[\sigma](\mathsf{fix}\ x.\ m) & = \mathsf{fix}\ x.\ [\sigma, x/x]m \\
\multicolumn{2}{l}{[\sigma](\mathsf{lcase}\ m\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow n \mid \mathsf{cons}\ h_d\ t_l \Rightarrow p\})} \\
 & = \mathsf{lcase}\ [\sigma]m\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow [\sigma]n \mid \mathsf{cons}\ h_d\ t_l \Rightarrow [\sigma, h_d/h_d, t_l/t_l]p\} \\
 & \\
[\sigma_2](\cdot) & = \cdot \\
[\sigma_2](\sigma_1, m/x) & = [\sigma_2]\sigma_1,\ [\sigma_2]m/x
\end{array}
$$

Lemma 4 (Substitution lemma and weakening property).

1 If $\Gamma' \vdash \sigma : \Gamma$ and $\Gamma \vdash m : \tau$ then $\Gamma' \vdash [\sigma]m : \tau$.
2 If $\Gamma' \vdash \sigma : \Gamma$ then $\Gamma', y{:}\tau \vdash \sigma : \Gamma$.

Lemma 5 (Substitution properties).

1 $[\sigma, n/x]m = [n/x]([\sigma, x/x]m)$
2 $[\sigma', n/x]\sigma = [n/x]([\sigma', x/x]\sigma)$
3 $[\sigma_2]([\sigma_1]m_1) = [[\sigma_2]\sigma_1]m$
4 $[\sigma_2]([\sigma_1]\sigma) = [[\sigma_2]\sigma_1]\sigma$
5 Let $\mathsf{id} = x_1/x_1, \ldots, x_n/x_n$ be the identity substitutions for $\Gamma = x_1{:}\tau_1, \ldots, x_n{:}\tau_n$, then $[\mathsf{id}]m = m$ and $[\mathsf{id}]\sigma = \sigma$. Moreover, $[\sigma]\mathsf{id} = \sigma$. A special case is when $\Gamma = \cdot$. In this case we have $\mathsf{id} = \cdot$. Moreover, $[\cdot]m = m$, $[\cdot]\sigma = \sigma$, and $[\sigma]\cdot = \cdot$.

Fig. 3. Properties of simultaneous substitutions.

$\Psi, x{:}\tau \vdash \sigma_1, x/x \preccurlyeq^{\mathcal{H}}_{\Gamma,x:\tau} \sigma_2, x/x$	by def. of Howe related substitutions
$\Psi, x{:}\tau \vdash [\sigma_1, x/x]m \preccurlyeq^{\mathcal{H}}_{\tau'} [\sigma_2, x/x]m'$	by IH on first subderivation
$\Psi \vdash [\sigma_2](\mathsf{lam}\, x.\, m') \preccurlyeq^{\circ}_{\tau\to\tau'} [\sigma_2]r$	by cus on second subderivation
$[\sigma_2](\mathsf{lam}\, x.\, m') = \mathsf{lam}\, x.\, [\sigma_2, x/x]m'$	by def. of substitution (see Figure 3)
$\Psi \vdash (\mathsf{lam}\, x.\, [\sigma_1, x/x]m) \preccurlyeq^{\mathcal{H}}_{\tau\to\tau'} [\sigma_2]r$	by rule *hlam*
$[\sigma_1](\mathsf{lam}\, x.\, m) = \mathsf{lam}\, x.\, [\sigma_1, x/x]m$	by def. of substitution (see Figure 3)
$\Psi \vdash [\sigma_1](\mathsf{lam}\, x.\, m) \preccurlyeq^{\mathcal{H}}_{\tau\to\tau'} [\sigma_2]r$	by above line

The other cases are analogous. □

3. Mechanizing Howe's method in Beluga

We discuss in this section the proof that similarity in PCFL is a pre-congruence using Howe's method in Beluga.

```
LF tp : type =
| top : tp
| arr : tp → tp → tp
| list: tp → tp;

LF term : tp → type =
| app  : term (arr S T) → term S → term T
| lam  : (term S → term T) → term (arr S T)
| fix  : (term T → term T) → term T
| unit : term top
| nil  : term (list T)
| cons : term T → term (list T) → term (list T)
| lcase: term (list S) → term T → (term S → term (list S) → term T)
      → term T;
```

Fig. 4. LF definition of intrinsically typed terms.

Beluga is a programming environment that supports both specifying formal systems and reasoning about them. To specify formal systems such as PCFL we use the logical framework LF. This allows us to take advantage of HOAS. A key challenge when reasoning about LF objects is that we must consider potentially open objects. In Beluga, this need is met by viewing all LF objects together with the contexts in which they are meaningful (Nanevski *et al.* 2008) as *contextual* LF objects and by abstracting not only over LF objects but also over contexts. We then view contextual objects and contexts as a particular index domain about which we can reason using a first-order logic with (co)induction principles on our index domain and built-in equality on index objects. Under the Curry–Howard isomorphism, this logic corresponds to a functional language with indexed (co)inductive types that supports (co)pattern matching. Meta-theoretic proofs about formal systems are implemented as (co)recursive functions in Beluga. We summarize in Appendix A the source level syntax of Beluga, where we concentrate on the parts that are relevant for our development. It is by no means a complete description, but it may serve as a useful high-level introduction to understanding Beluga programs. For a more formal introduction to the theoretical foundations, we refer the reader to Cave and Pientka (2012) and Thibodeau *et al.* (2016).

3.1. *Encoding syntax in LF*

We adopt the usual HOAS encoding for binding operators in the OL, and make essential use of LF's dependent types (see Figure 4). In particular, the type family `term` encodes *intrinsically* typed terms. This will make our overall mechanization more compact, as all terms are well typed by construction which is enforced by Beluga's type checker.

Variables such as `T` and `S` that are used in declaring the type of the LF constants are abstracted over at the front of the type of the constructor and we rely on type

```
inductive Value : [term T] → ctype =
  | Val_lam  : Value [lam λx.N]
  | Val_unit : Value [unit]
  | Val_nil  : Value [nil]
  | Val_cons : Value [cons M1 M2];

inductive Eval : [term T] → [term T] → ctype =
  | Ev_app       : Eval [M1] [lam λx.N] → Eval [N[M2]]  [V]
                 → Eval [app M1 M2] [V]
  | Ev_val       : Value [V]
                 → Eval [V] [V]
  | Ev_fix       : Eval [M[fix λx.M]] [V]
                 → Eval [fix λx.M] [V]
  | Ev_case_nil : Eval [M] [nil] → Eval [M1] [V]
                 → Eval [lcase M M1 (λh.λt.M2)] [V]
  | Ev_case_cons: Eval [M]  [cons H L] →  Eval [M2[H, L]] [V]
                 → Eval [lcase M M1 (λh.λt.M2)] [V];
```

Fig. 5. Inductive definition of values and evaluation.

reconstruction to infer their types (Pientka 2013). These variables are treated as implicit and we subsequently omit passing them when forming `term` objects.

3.2. *Encoding the operational semantics with indexed inductive types*

To illustrate how we can use inductive types in Beluga, we encode the *value* and *evaluation* judgement as computation-level type families indexed by closed well-typed terms. This is demonstrably equivalent to encoding the same judgements at the LF level.

How do we enforce that an LF object is closed? This is accomplished by a contextual type `[ ⊢ term T]`, where the context that appears to the left-hand side of the turnstile is empty; to improve readability, we simply write `[term T]`. Note that we can embed contextual types into Beluga types, but not vice versa. There is a strict separation between LF definitions that form our index objects and Beluga types that talk about LF definitions.

The inductive type family `Value` defines a subset of closed well-typed expressions, namely those that are the observables of our OL. Similarly, the inductive type family `Eval` relates two closed expressions of the same type, where the first big-step evaluates to the second (see Figure 5).

Beluga has a sophisticated notion of built-in simultaneous substitution. Consider the rule `Ev_app`, where to build the evaluation derivation for `Eval [app M1 M2] [V]`, we have to supply an evaluation derivation for `Eval [M1] [lam λx.N]` and `Eval [N[M2]] [V]`, where `N` stands for a term of type `S` that may refer to `x:term T`. The substitution that in standard LF would be represented as meta-level application `N M2`, here consists of the singleton simultaneous substitution `N[M2]` that keeps its domain, namely `x`, implicit. In general, all capitalized variables denote LF objects that may depend on LF declarations. Given an LF term `N` that depends on a context γ, we can use `N` in a context ψ by associating `N` with a simultaneous substitution σ with domain ψ and co-domain γ, with type $[\psi \vdash \gamma]$. This *closure* is written in post-fix notation as `N[`σ`]`. If σ is the identity substitution, we may drop the closure. We make use of two kinds of *weakening substitutions*: (1) The weakening substitution, written as `[]`, moves a closed object from the empty context to a context γ. For example, the type `T` is closed in `[term T]`. To use the closed type `T` in a context γ, we need to associate it with the weakening substitution `[]`. Hence, $[\gamma \vdash$ `term T[]]` describes a `term` object of type `T` in a context γ and enforces that the type `T` is closed. (2) The weakening substitution, written as `[..]`, moves an object `M` that is defined in a context γ to an extension of γ, for example γ`, x:term T[]`. Finally, we note that `[lam `λ`x.N]` can be expanded to `[lam `λ`x.N[x]]` where the substitution that maps `x` to itself is simply written as `[x]`.

Inductive types in Beluga correspond to least fixed points over an index domain type-theoretically defined using labelled sums and Σ-type (Cave and Pientka 2012; Thibodeau *et al.* 2016). Such inductive types must in general satisfy the positivity condition. The surface definition of `Value` is translated in the following notation where sums are represented as list of labels together with their computation-level types wrapped with $<\ >$.

$$
\begin{array}{l}
\mu\texttt{Value}.\lambda T, M.\\
<\texttt{Val\_lam}:\ \Sigma S_1, S_2{:}[\texttt{tp}].T = (\texttt{arr}\ S_1\ S_2) \times \Sigma N{:}[x{:}\texttt{term}\ S_1 \vdash \texttt{term}\ S_2[]].M = \texttt{lam}\ \lambda x.N,\\
\quad \texttt{Val\_unit}:\ T = \texttt{top} \times M = \texttt{unit},\\
\quad \texttt{Val\_nil}\ :\ \Sigma S{:}[\texttt{tp}].T = \texttt{list}\ S \times M = \texttt{nil}\\
\quad \texttt{Val\_cons}:\ \Sigma S{:}[\texttt{tp}].T = \texttt{list}\ S \times \Sigma M_1{:}[\texttt{term}\ S].\Sigma M_2{:}[\texttt{term}\ (\texttt{list}\ S)].\ M = \texttt{cons}\ M_1\ M_2 >
\end{array}
$$

We use *equality constraints* to express the refinement of the indices T and M in the above encoding. In general, our index domain is contextual LF (Cave and Pientka 2012, 2013), which has canonical forms. Hence, checking whether two contextual LF objects are equal simply boils down to comparing their canonical form.

We view `Value` as a least fixed point definition, although in this case, there is no recursive reference to `Value`. The latter happens in the fragment of the inductive type for `Eval`:

$$
\begin{array}{l}
\mu\texttt{Eval}.\lambda T, M, W.\\
<\texttt{Ev\_app}:\ \Sigma S{:}[\texttt{tp}],\ M_1{:}[\texttt{term}\ (\texttt{arr}\ S\ T)],\ M_2{:}[\texttt{term}\ S],\ N{:}[x{:}\texttt{term}\ T \vdash \texttt{term}\ S[]].\\
\qquad\qquad M = (\texttt{app}\ M_1\ M_2) \times \texttt{Eval}\ [M_1]\ [\texttt{lam}\ \lambda x.N] \times \texttt{Eval}\ [N[M_2]]\ [W]\\
\quad \texttt{E\_fix}:\ \Sigma N{:}[x{:}\texttt{term}\ T \vdash \texttt{term}\ T[]].\ M = \texttt{fix}\ \lambda x.N \times \texttt{Eval}\ [N[\texttt{fix}\ \lambda x.N]]\ [W]\\
\quad \texttt{E\_val}\ :\ M = W \times \texttt{Value}\ [W] \quad \ldots >
\end{array}
$$

3.3. *Encoding similarity using indexed coinductive types*

In Beluga, we also can state coinductive type families and in particular similarity as a coinductive definition that relates closed well-typed terms.

While inductive types are defined by constructors, we characterize *coinductive* types by the *observations* we can make (Abel *et al.* 2013; Thibodeau *et al.* 2016). To define the

```
coinductive Sim : {T:[tp]} [term T] → [term T] → ctype =
| (Sim_unit : Sim [top] [M] [N])
              :: Eval [M] [unit] → Eval [N] [unit]
| (Sim_nil  : Sim [list T] [M] [N])
              :: Eval [M] [nil] → Eval [N] [nil]
| (Sim_cons : Sim [list T] [M] [N])
              :: Eval [M] [cons H L] → Ex_sim_cons [H] [L] [N]
| (Sim_lam  : Sim [arr S T] [M] [N])
              :: Eval [M] [lam λx.M'] → Ex_sim_lam [x:term S ⊢ M'] [N]

and inductive Ex_sim_cons : [term T] → [term (list T)] → [term (list T)]
                          → ctype =
| ESim_cons: Eval [N] [cons H' L']
              → Sim [T] [H] [H'] → Sim [list T] [L] [L']
              → Ex_sim_cons [H] [L] [N]

and inductive Ex_sim_lam : [x:term S ⊢ term T[]] → [term (arr S T)]
                         → ctype =
| ESim_lam: Eval [N] [lam λx.N']
             → ({R:[term S]} Sim [T] [ M'[R] ] [ N'[R] ])
             → Ex_sim_lam [x:term S ⊢ M'] [N]
```

Fig. 6. Coinductive definition of applicative similarity.

coinductive type `Sim [T] [M] [N]`, we declare observations `Sim_unit`, `Sim_nil`, `Sim_cons`, and `Sim_lam`; each one corresponds to a case in our definition of applicative simulation – compare Definition 2.1 to Figure 6.

When we define an indexed inductive type, the indices impose obligations that must be satisfied in order to construct an object. When we define a coinductive one, indices guard what observations we can make. If the guard is true, then we can make the observation and proceed. We write the observation together with its type on the left side of `::` and on the right side we give the result type of the observation that describes our proof obligation. For example, we can make the observation `Sim_unit:Sim [top] [M] [N]`, if we can show that `Eval [M] [unit] → Eval [N] [unit]`. It corresponds directly to '$m \Downarrow \langle\rangle$ entails $n \Downarrow \langle\rangle$' in Definition 2.1. Note that `M` and `N` are implicitly quantified at the outside, as it becomes apparent in the desugared syntax on page 15. The definition of the observation `Sim_nil` follows a similar schema.

The result of the observation `Sim_cons` on `Sim [list T] [M] [N]` requires that if $m \Downarrow \mathsf{cons}\ h\ t$, then there are h' and t' such that $n \Downarrow \mathsf{cons}\ h'\ t'$ for which $h\ \mathcal{R}_\tau\ h'$ and $t\ \mathcal{R}_{\mathsf{list}(\tau)}\ t'$. We hence need a way to encode an *existential* property. Although existentials (i.e. Σ-types) exist in our theoretical foundation, the implementation of Beluga does not support them at the top level, as they always can be realized using indexed inductive types. We therefore define an indexed inductive type `Ex_sim_cons` that relates h, t and n.

Last, we need to represent the result of observing `Sim_lam` that encodes the corresponding part from the definition:

$m \Downarrow \mathsf{lam}\, x. m'$ for any $x{:}\tau \vdash m'{:}\tau'$ entails that there exists a $y{:}\tau \vdash n'{:}\tau'$ such that $n \Downarrow \mathsf{lam}\, y. n'$ for which $m'[r/x]\ \mathcal{R}_{\tau'}\ n'[r/y]$ for every term r of type τ.

We again resort to defining an inductive type `Ex_sim_lam` that relates the term `M'` with type `[x:term S ⊢ term T[]]`, i.e. `M'` has type `term T[]` under the assumption of the variable `x` having type `term S`. Hence, we can simply write `[x:term S ⊢ M']`, as we interpret `M'` within the context `x:term S`. As `T` denotes a closed type, we associate it with a weakening substitution, since it is used in a non-empty context. The relation `Ex_sim_lam` exists if `Eval [N] [lam λx.N']` and for all `R:[term S]` we know `Sim [T] [M'[R]] [N'[R]]`. Finally, we remark that the coinductive type `Sim` and inductive types `Ex_sim_cons` and `Ex_sim_lam` are defined mutually.

In the type-theoretic foundation that underlies Beluga, the coinductive type family `Sim` is encoded using a greatest fixed point that is defined using records, universals (written using Π), and implications.

$$
\begin{array}{l}
\nu\mathtt{Sim}.\lambda T.\lambda M.\lambda N.\\
\{\ \mathtt{Sim\_unit}\ :T = \mathtt{top} \to \mathtt{Eval}\ [M]\ [\mathtt{unit}] \to \mathtt{Eval}\ [N]\ [\mathtt{unit}]\\
\quad\ \mathtt{Sim\_lam}\ \ :\Pi S_1{:}[\mathtt{tp}].\Pi S_2{:}[\mathtt{tp}].\Pi M'{:}[x{:}\mathtt{term}\ S_1 \vdash \mathtt{term}\ S_2[]].\ T = \mathtt{arr}\ S_1\ S_2\\
\qquad\qquad\quad \to \mathtt{Eval}\ [M]\ [\mathtt{lam}\ \lambda x.M']\\
\qquad\qquad\quad \to \Sigma N'{:}[x{:}\mathtt{term}\ S_1 \vdash \mathtt{term}\ S_2[]].\ \mathtt{Eval}\ [N]\ [\mathtt{lam}\ \lambda x.N']\\
\qquad\qquad\qquad \times\Pi R{:}[\mathtt{term}\ S_1].\mathtt{Sim}\ [S_2]\ [M'[R]]\ [N'[R]]\\
\}
\end{array}
$$

We only show the encoding for lambda expressions and omit the observations we can make on lists to keep it readable. In this internal representation, the guards such as $T = \mathtt{top}$ or $T = \mathtt{arr}\ S_1\ S_2$ are made explicit. We further in-lined the definition of `Ex_sim_lam` to keep the definition compact.

3.4. *On the adequacy of coinductive encodings*

We next sketch the adequacy of the encoding of similarity; a full proof, such as those in the electronic appendix of Tiu and Miller (2010) would fill a dozen pages and require to spell out the static and dynamic semantics of (co)inductive Beluga (Thibodeau *et al.* 2016). Instead, we rely on our intuitive understanding of (co)inductive types; to get started, we assume the adequacy of LF encodings, whereby we denote the mapping of terms m and types τ to their encodings as $\ulcorner m \urcorner$ and $\ulcorner \tau \urcorner$, respectively. Conversely, the decoding of an LF object `M` and `T` into terms and types is written $\llcorner \mathtt{M} \lrcorner$ and $\llcorner \mathtt{T} \lrcorner$, respectively.

Lemma 3.1. For any term m and type τ, we have $\llcorner \ulcorner m \urcorner \lrcorner = m$ and $\llcorner \ulcorner \tau \urcorner \lrcorner = \tau$.

Proof. Standard, following for example Pfenning (1997). □

We further build on the adequacy of the encoding of substitutions. In particular, the translation of $[\sigma]m$ is equivalent to first translating σ and the term m to their corresponding representations in LF and then relying on the built-in simultaneous LF substitution operation of applying $\ulcorner \sigma \urcorner$ to $\ulcorner m \urcorner$. The encoding $\ulcorner \sigma \urcorner$ is defined inductively on the substitution σ as expected: $\ulcorner \cdot \urcorner = \hat{}$ and $\ulcorner \sigma, m/x \urcorner = \ulcorner \sigma \urcorner,\ \ulcorner m \urcorner$. Further, recall that we write the application of a simultaneous LF substitution in prefix form, while we write the closure of an LF object together with an LF substitution as a postfix.

Lemma 3.2 (Compositionality). $\ulcorner [\sigma]m \urcorner = [\ulcorner \sigma \urcorner]\ulcorner m \urcorner$.

Proof. Generalization of the compositionality lemma for LF. □

Lemma 3.3 (Soundness). If $\mathtt{Sim}\ [\ulcorner \tau \urcorner]\ [\ulcorner m \urcorner]\ [\ulcorner n \urcorner]$, then $m \preccurlyeq_\tau n$.

Proof. (Sketch) We apply rule $CI{-}\preceq$ to unfold $m \preccurlyeq_\tau n$ selecting the family S_τ to be

$$\{(m, n) \mid \mathtt{Sim}\ [\ulcorner \tau \urcorner]\ [\ulcorner m \urcorner]\ [\ulcorner n \urcorner]\}.$$

We then show that S_τ satisfies the simulation conditions unfolding the definition of S_τ. □

Before addressing the other direction, we briefly contrast the more familiar inductive reasoning with the coinductive reasoning we will use. To prove a conjecture inductively on an object, we consider all possible ways such an object can be constructed and we reason about some notion of *size* of an object or a derivation. In an inductive proof, to show that the property holds for objects of size m, we may assume that it holds for objects of size k where $k < m$.

For example, to prove that for all terms M and V, if $\mathcal{D} : \mathtt{Eval}\ [M]\ [V]$, then there exists $\mathcal{F} : \mathtt{Value}\ [V]$, we proceed by induction on the height n of $\mathcal{D}$, the derivation of $\mathtt{Eval}\ [M]\ [V]$. We therefore prove the following by considering all possible constructors that we can use to build such a derivation $\mathcal{D}$.

IH	For all $k < n$, for all terms M and V, if $\mathcal{D} : \mathtt{Eval}\ [M]\ [V]$ and $\mathsf{size}(\mathcal{D}) = k$, then there is $\mathcal{F} : \mathtt{Value}\ [V]$
To show	For all term M and V, if $\mathcal{D} : \mathtt{Eval}\ [M]\ [V]$ and $\mathsf{size}(\mathcal{D}) = n$, then there is $\mathcal{F} : \mathtt{Value}\ [V]$

Dually, to prove a conjecture *coinductively*, we consider all possible observations we can make about an object and we *reason inductively on the number of observations*, which we refer to as *depth*. In a coinductive proof, we assume that the conjecture holds when we can make k observations about the object, and we show that the conjecture also holds when we make n observations about it where $k < n$. In essence, to prove a statement by coinduction, we reason by complete induction on the number of observations. For example, if we want to prove reflexivity of simulation, i.e. for all terms M and types T, $\mathcal{D} : \mathtt{Sim}\ [T]\ [M]\ [M]$, then we proceed by induction on the number of observations on $\mathcal{D}$ and consider all possible observations we can make about $\mathcal{D}$.

IH	For all $k < n$, for all terms M and types T, $\mathcal{D} : \mathtt{Sim}\ [T]\ [M]\ [M]$ and $\mathsf{depth}(\mathcal{D}) = k$
To show	for all term M and types T, $\mathcal{D} : \mathtt{Sim}\ [T]\ [M]\ [M]$ and $\mathsf{depth}(\mathcal{D}) = n$

We are now ready to address the other direction of the adequacy statement. For a more formal justification of reasoning about inductive data via sizes and coinductive data via observations, we refer the reader to Abel and Pientka (2013, 2016).

Lemma 3.4 (Completeness). If $m \preccurlyeq_\tau n$, then $\texttt{Sim}\ [\ulcorner\tau\urcorner]\ [\ulcorner m\urcorner]\ [\ulcorner n\urcorner]$.

Proof. We proceed by complete induction on the number of observations we can make on $\texttt{Sim}\ [\ulcorner\tau\urcorner]\ [\ulcorner m\urcorner]\ [\ulcorner n\urcorner]$.

IH	For all $k < j$, for all terms m and types τ, If $\mathcal{S} : m \preccurlyeq_\tau n$, then $\mathcal{D} : \texttt{Sim}\ [\ulcorner\tau\urcorner]\ [\ulcorner m\urcorner]\ [\ulcorner n\urcorner]$ and $\mathsf{depth}(\mathcal{D}) = k$
To show	for all terms m and types τ, If $\mathcal{S} : m \preccurlyeq_\tau n$, then $\mathcal{D} : \texttt{Sim}\ [\ulcorner\tau\urcorner]\ [\ulcorner m\urcorner]\ [\ulcorner n\urcorner]$ and $\mathsf{depth}(\mathcal{D}) = j$

Observation `Sim_unit`.
To show: If $m \preccurlyeq_\tau n$, then $(\ulcorner\tau\urcorner = \texttt{top}) \to \texttt{Eval}\ [\ulcorner m\urcorner]\ [\texttt{unit}] \to \texttt{Eval}\ [\ulcorner n\urcorner]\ [\texttt{unit}]$.

Assume $m \preccurlyeq_\tau n$, $\ulcorner\tau\urcorner = \texttt{top}$, and $\texttt{Eval}\ [\ulcorner m\urcorner]\ [\texttt{unit}]$

$\tau = \top$	by definition of $\ulcorner\tau\urcorner$
$m \Downarrow \langle\rangle$ entails $n \Downarrow \langle\rangle$	by Definition 2.1 using the assumption $m \preccurlyeq_\tau n$
$\llcorner\ \texttt{Eval}\ [\ulcorner m\urcorner]\ [\texttt{unit}]\ \lrcorner = m \Downarrow \langle\rangle$	by decoding of `Eval`
$n \Downarrow \langle\rangle$	by previous lines
$\ulcorner n \Downarrow \langle\rangle\urcorner = \texttt{Eval}\ [\ulcorner n\urcorner]\ [\texttt{unit}]$	by encoding of `Eval`

Observation `Sim_lam`.

IH	For all $k < j$, if $m \preccurlyeq_\tau n$, then $\mathcal{D} : \texttt{Sim}\ [\ulcorner\tau\urcorner]\ [\ulcorner m\urcorner]\ [\ulcorner n\urcorner]$ and $\mathsf{depth}(\mathcal{D}) = k$
To show	If $m \preccurlyeq_\tau n$, then $\mathcal{D}\ .\texttt{S\_lam} : \texttt{Sim}\ [\ulcorner\tau\urcorner]\ [\ulcorner m\urcorner]\ [\ulcorner n\urcorner]$ and $\mathsf{depth}(\mathcal{D}\ .\texttt{Sim\_lam}) = j$

Making an observation corresponds to projecting with the dot notation the field `Sim_lam` of the record. We further note that $\mathsf{depth}(\mathcal{D}\ .\texttt{Sim\_lam}) = \mathsf{depth}(\mathcal{D}) + 1$. Since we are making the observation `Sim_lam`, we can unfold the definitions. Hence, it suffices to show

if $m \preccurlyeq_\tau n$, then
for all $\texttt{S}_1, \texttt{S}_2, \texttt{M}'$. $(\ulcorner\tau\urcorner = \texttt{arr}\ \texttt{S}_1\ \texttt{S}_2) \to \texttt{Eval}\ [\ulcorner m\urcorner]\ [\texttt{lam}\ \lambda x.\texttt{M}']$
$\to$there exists $\texttt{N}'$ s.t. $\texttt{Eval}\ [\texttt{N}]\ [\texttt{lam}\ \lambda x.\texttt{N}']$
and (for all $\texttt{R}$, $\mathcal{D} : \texttt{Sim}\ [\texttt{S}_2]\ [\ \texttt{M}'[\texttt{R}]\ ]\ [\ \texttt{N}'[\texttt{R}]\ ]$)

Moreover, $\mathsf{depth}(\mathcal{D})$ is clearly less than $\mathsf{depth}(\mathcal{D}\ .\texttt{Sim\_lam})$ and we may appeal to the induction hypothesis, which can be specialized to the following statement:

If $[r/x]m' \preccurlyeq_{s_2} [r/y]n'$, then $\texttt{Sim}\ [\texttt{S}_2]\ [\ \texttt{M}'[\texttt{R}]\ ]\ [\ \texttt{N}'[\texttt{R}]\ ]$, where $\ulcorner m'\urcorner = \texttt{M}'$, $\ulcorner r\urcorner = \texttt{R}$, $\ulcorner n'\urcorner = \texttt{N}'$.

Assume $m \preccurlyeq_\tau n$, $\ulcorner\tau\urcorner = (\texttt{arr}\ \texttt{S}_1\ \texttt{S}_2)$, and $\texttt{Eval}\ [\ulcorner m\urcorner]\ [\texttt{lam}\ \lambda x.\texttt{M}']$

$\tau = s_1 \to s_2$	since $\ulcorner s_1 \to s_2\urcorner = \texttt{arr}\ \texttt{S}_1\ \texttt{S}_2$, where $s_1 = \llcorner \texttt{S}_1 \lrcorner$ and $s_2 = \llcorner \texttt{S}_2 \lrcorner$.
For any $x{:}s_1 \vdash m'{:}s_2$. $m \Downarrow \mathsf{lam}\, x.\, m'$ entails that there exists a $y{:}s_1 \vdash n'{:}s_2$ such that $n \Downarrow \mathsf{lam}\, y.\, n'$ and for every $r{:}s_1$, $[r/x]m' \preccurlyeq_{s_2} [r/y]n'$;	by definition of $m \preccurlyeq_\tau n$
$\texttt{Eval}\ [\ulcorner m\urcorner]\ [\texttt{lam}\ \lambda x.\texttt{M}'] = \ulcorner m \Downarrow \llcorner \texttt{lam}\ \lambda x.\texttt{M}' \lrcorner\urcorner$	by encoding of `Eval`

```
rec sim_refl : Sim [T] [M] [M] =
fun .Sim_unit d ⇒ d
  | .Sim_nil  d ⇒ d
  | .Sim_cons d ⇒ ESim_cons d sim_refl sim_refl
  | .Sim_lam  d ⇒ ESim_lam d (mlam R ⇒ sim_refl)
```

Fig. 7. Corecursive program showing that similarity is reflexive.

$\texttt{lam}\ \lambda\texttt{x.M}' = \ulcorner \mathsf{lam}\, x.\, m' \urcorner$ by encoding of terms

there exists a $y{:}s_1 \vdash n'{:}s_2$ such that $n \Downarrow \mathsf{lam}\, y.\, n'$
and for every $r{:}s_1$, $[r/x]m' \preccurlyeq_{s_2} [r/y]n'$ by previous lines

$\ulcorner n \Downarrow \mathsf{lam}\, y.\, n' \urcorner = \texttt{Eval}\ \ulcorner n \urcorner\ (\texttt{lam}\ \lambda y.\ulcorner n' \urcorner)$ by encoding of Eval

Assume $\texttt{R} : \texttt{term S}_1$.
$\ulcorner [r/x]m' \urcorner = \texttt{M}'[\texttt{R}]$ and $\ulcorner [r/y]n' \urcorner = \texttt{N}'[\texttt{R}]$ by Theorem 3.2 (Compositionality)
$\texttt{Sim}\ [\texttt{S}_2]\ [\texttt{M}'[\texttt{R}]]\ [\texttt{N}'[\texttt{R}]]$ by the specialized induction hypothesis
using $[r/x]m' \preccurlyeq_{s_2} [r/y]n'$ from the previous line

Therefore, there exists $\texttt{N}'$, namely $\ulcorner n' \urcorner$, and $\texttt{Eval}\ [\ulcorner n \urcorner]\ [\texttt{lam}\ \lambda y.\ulcorner n' \urcorner]$ and for all $\texttt{R}{:}\texttt{term S}_1$, we have $\texttt{Sim}\ [\texttt{S}_2]\ [\texttt{M}'[\texttt{R}]]\ [\texttt{N}'[\texttt{R}]]$. Hence, $\texttt{Sim}\ [\ulcorner s_1 \to s_2 \urcorner]\ [\ulcorner m \urcorner]\ [\ulcorner n \urcorner]$. This concludes this case. □

Theorem 3.1 (Adequacy of encoding of similarity). $m \preccurlyeq_\tau n$ iff $\texttt{Sim}\ [\ulcorner \tau \urcorner]\ [\ulcorner m \urcorner]\ [\ulcorner n \urcorner]$.

We remark that while soundness follows the same structure of Honsell *et al.* (2001) and Tiu and Miller (2010), the possibility to induct on the number of observations establishes completeness in a novel and easier way w.r.t. an analogous result in Tiu and Miller (2010); there, the authors had to resort to a complex induction on the structure of the arguments of the coinductively defined predicate/type.

Subsequently, we will not make the number of observations explicit in coinductive arguments, but simply permit corecursive calls when they are guarded by an observation.

3.5. *Writing coinductive proofs using copattern matching*

In Beluga, we implement (co)inductive proofs as (co)recursive functions using (co)pattern matching.

Let us reconsider first the proof that similarity is reflexive (Figure 7): For all `T`, `M`, we have `Sim [T] [M] [M]`. The type of the function `sim_refl` encodes this statement directly. We leave `T` and `M` implicit, as these arguments can be reconstructed by Beluga.

To prove our statement, we consider each case by writing the observation on the left-hand side of our corecursive function and provide a witness of the appropriate type on the right-hand side. We write observations as *projections* prefixing them with a dot.

We recall that each observation is implicitly guarded by a constraint (for example, `Sim_unit` is guarded by `T = top`) and we can only make the observation if the constraint is satisfied. Beluga reconstructs proofs for such equality guards and associates them implicitly with the observation. If the guard is not satisfied, i.e. no proof that `T = top` for example exists, then the case is trivially satisfied, and we omit it from our function definition.

In general, the arguments of a function definition in Beluga may consist of index objects, patterns describing inductive objects, and copatterns (i.e. observations) defining coinductive objects. In fact, the function `sim_refl` takes first the two arguments that we left implicit, namely `T` and `M`, before it expects each observation.

Observation `Sim_unit` and `T = top`. In this case, we need to construct a witness for `Eval [M] [unit]` → `Eval [M] [unit]`. This is is simply the identity function that maps `d:Eval [M] [unit]` to itself. Note that instead of returning a function, we write `d` on the left-hand side of our function definition.

Observation `Sim_nil` and `T = list T`λ`;'`. Similar to the previous case.

Observation `Sim_cons` and `T = list T`λ`;'`. In this case, `d:Eval [M] [cons H T]` and we need to supply a witness for `Ex_sim_cons [H] [L] [M]`: we choose `d` for `Eval [M] [cons H L]` and make two corecursive calls on `L` and `H`, respectively.

Hence, a corecursive call is only valid if it is guarded by at least an observation on the left-hand side of a function definition, as we comment on next.

Observation `Sim_lam` and `T = (arr S S')`. We have `d:Eval [M] [lam `λ`x.M']` and we need to provide a witness for `Ex_sim_lam [x:term S ⊢ M'] [M]`: we again choose `d` for `Eval [M] [lam `λ`x.M']` and then build a function of type `{R:[term S]} Sim [S'] [M'[R]] [M'[R]]` that makes a corecursive call to `sim_refl`. We note that in this case we build a function using `mlam`, which declares a Π-type quantification over LF objects, as opposed to **fun** that applied to top-level abstraction. The name `R` is required by `mlam`, although it is only actually used in the reconstructed argument to the recursive call.

As we mentioned above, in order for a term like `sim_refl` to be considered a proof, it needs to be *covering* and *productive*. To be covering, a term of codata type needs to have a branch for each possible observation. In this case, since `sim_refl` does not require a particular shape for the type of `M` and `N`, we need to provide all the possible observations. However, if we were to prove reflexivity only for terms of type `list T`, i.e. `Sim [list T] [M] [M]`, then we are justified to consider the branches for `Sim_cons` and `Sim_nil`.

Similarly to other languages with support for coinduction, we achieve productivity through a guardedness check. However, this check is different from the one implemented in systems such as Coq. There, we define coinductive types through lazy constructors. Thus, recursive calls are restricted, or guarded, as to be performed only under such constructors. With copatterns, coinductive terms are defined via observations that serve to delay computation. As such, guardedness restricts recursive occurrences to be performed under observations; hence, each step of unfolding requires at least one observation to be applied. For example, in the arrow case of the above proof, the corecursive call is justified, as it is guarded by the observation `.Sim_lam`, which increases the number of observations made. This is sufficient, as operationally each step of unfolding requires at least one observation to be applied – otherwise the corecursive function could not progress.

```
rec sim_trans : Sim [T] [M] [N] → Sim [T] [N] [R] → Sim [T] [M] [R] =
fun s1 s2 .Sim_unit  d ⇒ s2.Sim_unit (s1.Sim_unit d)
  | s1 s2 .Sim_nil   d ⇒ s2.Sim_nil (s1.Sim_nil d)
  | s1 s2 .Sim_cons  d ⇒
    let ESim_cons d1 s1' s1'' = s1.Sim_cons d in
    let ESim_cons d2 s2' s2'' = s2.Sim_cons d1 in
    ESim_cons d2 (sim_trans s1' s2') (sim_trans s1'' s2'')
  | s1 s2 .Sim_lam   d ⇒
    let ESim_lam d1 s1' = s1.Sim_lam d in
    let ESim_lam d2 s2' = s2.Sim_lam d1 in
    ESim_lam d2 (mlam P ⇒ sim_trans (s1' [P]) (s2' [P]))
```

Fig. 8. Corecursive program showing that similarity is transitive.

Next, we consider the proof that applicative similarity is transitive: if `Sim [T] [M] [N]` and `Sim [T] [N] [R]`, then `Sim [T] [M] [R]` (Figure 8). We comment on a couple of cases:

Observation `Sim_unit` and `T = top`. We have the following assumptions: `s1: Sim [top] [M] [N]`, `s2: Sim [top] [N] [R]`, and `d:Eval [M] [unit]` and we need to provide a witness for `Eval [R] [unit]`. By the observation `Sim_unit` on `s2` (written as the projection `s2.Sim_unit`), we obtain a function `Eval [N] [unit] → Eval [R] [ unit]`. We pass to it the result of `s1.Sim_unit d`.

Observation `Sim_lam` and `T = (arr S S')`. We have the assumptions: `s1:Sim [arr S T] [M] [N]` and `s2:Sim [arr S T] [N] [R]` and `d:[eval M (lam λx.M')]`, and we need to build a witness for `Ex_sim_lam [x:term S ⊢ M'] [R]`. We first observe `s1.Sim_lam` and pass `d:Eval [M] [lam λx.M']`. This hence gives us `Ex_sim_lam [x:term S ⊢ M'] [N]`, which provides us with `d1:Eval [N] [lam λx.N']` and `s1':{P:[term S]} Sim [T] [M '[P]] [N'[P]]`. Similarly, we observe `s2.Sim_lam` passing `d1:Eval [N] [lam λx.N']`. Hence, we obtain `Ex_sim_lam [x:term S ⊢ N'] [R]`, which provides us with `d2:Eval [R] [lam λx.R']` and `s2':{P:[term S]} Sim [T] [N'[P]] [R'[P]]`. Building a witness for `Ex_sim_lam [x:term S ⊢ M'] [R]` requires us to supply a derivation `d2:Eval [R] [lam λx.R']` and a function of type `{P:[term S]} Sim [T] [M'[P]] [R'[P]]`. We build that function making a corecursive call.

3.6. *A concrete example of similarity*

In this section, we show how we can interactively build an actual simulation between two terms, namely that `two` is simulated by `suc one`, following the example in Pitts (2011). We represent the numbers via Church encodings, where `one` $\equiv \mathsf{lam}\, f.\, \mathsf{lam}\, x.\, f\ \ x$, `two` $\equiv \mathsf{lam}\, f.\, \mathsf{lam}\, x.\, f(f\ \ x)$ and `suc` $\equiv \mathsf{lam}\, n.\, \mathsf{lam}\, x.\, \mathsf{lam}\, y.\, x\ (n\ \ x\ \ y)$. We thus want to prove the following theorem:

```
rec sim_two_succ_one :
        Sim [_] [lam λf.lam λx.app f (app f x)]
                [app (lam λn. lam λx. lam λy. app x (app (app n x) y))
```

```
                    (lam λf.lam λx.app f x)]
```

We will build the proof incrementally, by inserting *holes*, denoted by **?** and refining them, analogously to Agda's methodology. We start with the following program body:

```
fun .Sim_lam (Ev_val Val_lam) ⇒
        ESim_lam (Ev_app (Ev_val Val_lam) (Ev_val Val_lam)) sim_lemma1
```

where `sim_lemma1` is used to abstract over the nested copattern matching:

```
rec sim_lemma1 : {M:[exp (arr T T)]}
   Sim [arr T T]
       [lam (λx. app M[] (app M[] x))]
       [lam (λy. app M[] (app (app (lam (λf. lam (λw. app f w))) M[]) y))] =
fun [M] .Sim_lam (Ev_val Val_lam) ⇒
     ESim_lam (Ev_val Val_lam)
              (mlam N ⇒ ?)
```

Here, the goal has type `Sim [_] [app M (app M N)] [app M (app (app (lam (λf. lam (λw. app f w))) M) N)]`, which we cannot prove directly using our definition of similarity: since our evaluation strategy is call-by-name and the metavariable `M` is not a concrete term, the right-hand side will not reduce. Instead, we use the fact that similarity is a pre-congruence, the main result of this work. We only need property (C3) of Definition 2.2, which translates to the following lemma:

```
rec sim_cong_app : Sim [arr S T] [M1] [M2] → Sim [S] [N1] [N2]
                  → Sim [T] [app M1 N1] [app M2 N2]
```

Using this result and reflexivity of similarity, we can thus refine the body of `sim_lemma1`:

```
fun [M] .Sim_lam (Ev_val Val_lam) ⇒
     ExSimlam (Ev_val Val_lam)
              (mlam N ⇒ sim_cong_app sim_refl ?)
```

where the current hole has type:

```
Sim [T] [app M N] [app (app (lam (λu. lam (λw. app u w))) M) N]
```

Now, we can easily use a derivation of the evaluation of the left-hand side to derive the evaluation of the right-hand side of this similarity as follows:

```
rec ev1 : Eval [app M N] [V] →
               Eval [app (app (lam (λu. lam (λw. app u w))) M) N] [V] =
fun d ⇒ Ev_app (Ev_app (Ev_val Val_lam) (Ev_val Val_lam)) d;
```

Constructing the above simulation requires us to match on the possible values that `app M N` can take through the possible observations – in fact all of them as the type `T` is abstract. We then use `ev1` on the derivations of `Eval [app M N] [V]` for the given `V` and reflexivity when needed.

```
rec sim_lemma2 : Sim [T] [app M N]
                          [app (app (lam (λu. lam (λw. app u w))) M) N] =
fun .Sim_lam  d ⇒ ESim_lam (ev1 d) (fun [V] ⇒ sim_refl)
```

```
  | .Sim_unit d ⇒ ev1 d
  | .Sim_nil  d ⇒ ev1 d
  | .Sim_cons d ⇒ ESim_cons (ev1 d) sim_refl sim_refl
```

Using this lemma, we can complete the body of `sim_lemma1`:

```
fun [M] .Sim_lam (Ev_val Val_lam) ⇒
     ExSimlam (Ev_val Val_lam)
              (mlam N ⇒ sim_cong_app sim_refl sim_lemma2)
```

This concludes the proof.

3.7. *Defining open similarity using first-class contexts and substitutions*

Similarity only relates closed terms. However, in general, we want to be able to reason about similarity of open terms, i.e. terms that depend on a context γ. In Beluga, we can declare schemas of contexts that classify contexts in the same way that types classify terms and kinds classify types, describing the shape of each declaration in a context. Moreover, we can take advantage of built-in substitutions to *relate* two contexts. In particular, we can describe *grounding* substitutions with the type `[ ⊢ γ]`, where the range of the substitution is empty.

We begin by defining the schema of contexts that can occur in our development:

```
schema ctx = term T;
```

Here, we declare the schema `ctx` that states that each declaration of a context γ of schema `ctx` can only contain variable declarations of type `term T` for some type `T`. For example, the context `x:term top, y:term (list top)` is a valid context of schema `ctx`. On the other hand, a context `x:term unit, a:tp` is not.

We can now state open similarity as an inductive type relating well-typed terms in the context γ. In the kind of the inductive type `OSim`, we make the type `T` explicit, but leave γ implicit. This distinction is reflected in Beluga's source syntax. We use curly braces `{ }` to describe explicit index arguments and round ones `( )` to give type annotations implicitly.

We can now define open similarity: two terms `[γ ⊢ M]` and `[γ ⊢ N]` are openly similar if they are similar for all grounding substitutions σ. Here, we pass γ explicitly to `OSimC` so that Beluga knows the type of σ.

```
inductive OSim:(γ:ctx){T:[tp]} [γ ⊢ term T[]] → [γ ⊢ term T[]] → ctype =
| OSimC : {γ:ctx}({σ:[ ⊢ γ]} Sim [T] [M[σ]] [N[σ]])
             → OSim [T] [γ ⊢ M] [γ ⊢ N]
```

We can easily show that open similarity is closed under substitutions by simply composing the input substitution σ with the grounding substitution `σ'`.

```
rec osim_cus : (γ:ctx) (ψ:ctx) {σ:[ψ ⊢ γ]} OSim [T] [γ ⊢ M] [γ ⊢ N]
                 → OSim [T] [ψ ⊢ M[σ]] [ψ ⊢ N[σ]] =
fun [ψ ⊢ σ] (OSimC [γ] f) ⇒ OSimC [ψ] (fun [σ'] ⇒ f [σ[σ']])
```

```
inductive Howe: (γ:ctx){T:[tp]}[γ ⊢ term T[] ] → [γ ⊢ term T[] ] → ctype =
  | Howe_unit: OSim [top] [γ ⊢ unit] [γ ⊢ M]
               → Howe [top] [γ ⊢ unit] [γ ⊢ M]
  | Howe_var : {#p:[γ ⊢ term T[]]} OSim [T] [γ ⊢ #p] [γ ⊢ M]
               → Howe [T] [γ ⊢ #p] [γ ⊢ M]
  | Howe_lam : Howe [T] [γ,x:term S[] ⊢ M] [γ,x:term S[] ⊢ N]
               → OSim [arr S T] [γ ⊢ lam λx.N] [γ ⊢ R]
               → Howe [arr S T] [γ ⊢ lam λx.M] [γ ⊢ R]
  | Howe_app : Howe [arr S T] [γ ⊢ M] [γ ⊢ M']
               → Howe [S] [γ ⊢ N] [γ ⊢ N']
               → OSim [T] [γ ⊢ app M' N'] [γ ⊢ R]
               → Howe [T] [γ ⊢ app M N] [γ ⊢ R]
...
```

Fig. 9. The Howe relation.

3.8. *Defining the Howe relation*

The encoding of the Howe relation (see Figure 9) is, in our view, one of the high points of the formalization: it follows very closely its mathematical formulation, while retaining all the powerful abstractions that Beluga offers. This is apparent in the variable case where Beluga's *parameter* variables, denoted #p, range over elements from the context γ. They permit us to precisely characterize when a variable is Howe related to a term M in the given context, while looking remarkably similar to the informal version. The same applies to the lambda abstractions case, where one notes the correct scoping of M, N and R w.r.t. γ. The cases for fix and lcase follow the same principle and we omit them to save space.

The structure of the Howe relation makes it trivial to prove that it is a precongruence. We show here the proof in the application case, which is indirectly used for the example of simulation we presented above. The proof of precongruence of similarity in fact follows immediately once we prove equivalent similarity and the Howe relation, see Section 3.10.

```
rec howe_cong_app : Howe [arr S T] [M1] [M2] → Howe [S] [N1] [N2]
                      → Howe [T] [app M1 N1] [app M2 N2] =
fun h1 h2 ⇒ Howe_app h1 h2 osim_refl;
```

Using reflexivity and transitivity of open similarity, respectively, we can show reflexivity and semi-transitivity of the candidate relation. We only show the types.

```
rec howe_refl : (γ:ctx) {M:[γ ⊢ term T[] ]} Howe [T] [γ ⊢ M] [γ ⊢ M]

rec howe_osim_trans : (γ:ctx) Howe [T] [γ ⊢ M] [γ ⊢ N]
                        → OSim [T] [γ ⊢ N] [γ ⊢ R]
                        → Howe [T] [γ ⊢ M] [γ ⊢ R]
```

From this, it immediately follows that open similarity is a Howe relation.

```
rec osim_howe:(γ:ctx) OSim [T] [γ ⊢ M] [γ ⊢ N]
```

```
inductive Howe_subst:
             {γ:ctx} (ψ:ctx) {σ₁ : [ψ ⊢ γ]} {σ₂ : [ψ ⊢ γ]} ctype =
  | HNil  : Howe_subst [] [ψ ⊢ ] [ψ ⊢ ]
  | HCons : Howe_subst [γ] [ψ ⊢ σ₁] [ψ ⊢ σ₂]
            → Howe [T] [ψ ⊢ M] [ψ ⊢ N]
            → Howe_subst [γ,x:term T[]] [ψ ⊢ σ₁, M] [ψ ⊢ σ₂, N]

rec howe_subst_wkn : Howe_subst [γ] [ψ ⊢ σ₁] [ψ ⊢ σ₂]
        → Howe_subst [γ] [ψ,x:term S[] ⊢ σ₁[…]] [ψ,x:term S[] ⊢ σ₂[…]]
```

Fig. 10. Howe-related substitutions.

```
                        → Howe [T] [γ ⊢ M] [γ ⊢ N] =
fun s ⇒ howe_osim_trans (howe_refl [_ ⊢ _]) s
```

3.9. *Substitutivity of the Howe relation*

As remarked in Section 2.2, a crucial point of the proof is showing that the Howe relation is substitutive. Traditionally, substitution properties tend to be tedious to prove in proof assistants due to the necessity to reason manually about contexts. Here, Beluga's contextual abstractions significantly reduces the amount of boilerplate work needed for that proof.

We first encode (Figure 10) $\Phi \vdash \sigma_1 \preccurlyeq^{\mathcal{H}}_{\Gamma} \sigma_2$ using an inductive type that relates two simultaneous substitutions. The base case relates empty substitutions, written as `[ψ ⊢ ]`. In the inductive case, the substitution `[ψ ⊢ σ₁, M]` and `[ψ ⊢ σ₂, N]` are related, if so are `[ψ ⊢ σ₁]` and `[ψ ⊢ σ₂]` and `[ψ ⊢ M]` is Howe related to `[ψ ⊢ N]`.

In the subsequent proofs, we rely on the weakening property of simultaneous substitutions, namely that weakening preserves Howe-relatedness, see function `howe_subst_wkn` in Figure 10. In Beluga, weakening a substitution is simply achieved by composing it with the weakening substitution `[…]`, which has here domain ψ and range `ψ, x:term S[]`. This is supported in Beluga's theory of simultaneous substitutions (Cave and Pientka 2013), which internalizes the notions in Figure 3. The proof of `howe_subst_wkn` is done by induction over the predicate `Howe_subst` and by appealing to a special case of substitutivity of Howe on renamings. In the interest of space, we omit those proofs. In the following, namely in the proof of `howe_osim` (Figure 12), we will also need the following reflexivity property of `Howe_subst`, which holds by a simple induction on substitutions:

```
rec howe_subst_refl: (γ:ctx)(ψ:ctx){σ:[ψ ⊢γ]}
                              Howe_subst [γ] [ψ ⊢ σ] [ψ ⊢ σ]
```

A fragment of the proof of substitutivity in Beluga appears in Figure 11. We only show here the same two cases we described in the informal proof, together with the variable case, but the remaining cases follow a similar pattern. We make use of the lemmas described above, together with the following additional lemma for the variable case:

```
rec howe_subst : Howe [T] [γ ⊢ M] [γ ⊢ N]
              → Howe_subst [γ] [ψ ⊢ σ₁] [ψ ⊢ σ₂]
              → Howe [T] [ψ ⊢ M[σ₁]] [ψ ⊢ N[σ₂]] =
fun h (hs:Howe_subst [γ] [ψ ⊢ σ₁] [ψ ⊢ σ₂]) ⇒
case h of
...
| Howe_var [γ ⊢ #p] s ⇒ howe_subst_var h hs
| Howe_lam h' s       ⇒
  Howe_lam (howe_subst h' (HCons (howe_subst_wkn hs)
                              (howe_refl [ψ,x:term _] [ψ,x:term _ ⊢x])))
           (osim_cus [ψ ⊢ σ₂] s)
| Howe_app h1' h2' s   ⇒
  Howe_app (howe_subst h1' hs) (howe_subst h2' hs) (osim_cus [ψ ⊢ σ₂] s);
```

Fig. 11. Substitutivity property of the Howe relation.

```
rec howe_subst_var : (γ:ctx) (ψ:ctx) Howe [T] [γ ⊢ #p] [γ ⊢ M]
                   → Howe_subst [γ] [ψ ⊢ σ₁] [ψ ⊢ σ₂]
                   → Howe [T] [ψ ⊢ #p[σ₁]] [ψ ⊢ M[σ₂]] =
```

This is proven by simple induction on the position of the variable in the context which follows the inductive definition of `Howe_subst`.

We can see that `howe_subst` straightforwardly represents the proof of Lemma 2.2. What is remarkable in this program with respect to the informal proof is that there are no explicit references to the substitution properties outside of the weakening of the Howe relation on substitutions.

3.10. *Main theorem*

The key lemma in our main theorem is the proof that the Howe relation is *downward closed*:

```
rec down_closed : Eval [M] [V] → Howe [T] [M] [N] → Howe [T] [V] [N]
```

The proof of this lemma (point 2.1 at page 8) relies on several previous lemmas such as transitivity of closed and open similarity, semi-transitivity and substitutivity of the Howe relation, together with the unfolding of similarity using the observations. The proof is otherwise straightforward but long, and we leave it to the online documentation.

Moving on, we first establish lemmas that mimic the similarity conditions (previous point 2.1). For example: If $\mathsf{lam}\ x.\ m \preccurlyeq^{\mathcal{H}}_{\tau\to\tau'} n$, then $n \Downarrow \mathsf{lam}\ x.\ m'$ and for every $q{:}\tau$, we have $[q/x]m \preccurlyeq^{\mathcal{H}}_{\tau'} [q/x]m'$. Again, as we do not have existential types, we encode the existence of a term `N'` using the inductive types `Howe_abs`. A fragment of the type signature is as follows:

```
inductive Howe_abs: [x:term S ⊢ term T[]] → [term (arr S T)] → ctype =
  | Howe_absC : Eval [N] [lam λx.N']
```

```
rec howe_sim : Howe [T] [M] [N] → Sim [T] [M] [N] =
fun h .Sim_unit e ⇒ howe_ev_unit (down_closed e h)
  | h .Sim_nil  e ⇒ howe_ev_nil (down_closed e h)
  | h .Sim_cons e ⇒
      let Howe_consC e' h1 h2 = howe_ev_cons (down_closed e h) in
          ESim_cons e' (howe_sim h1) (howe_sim h2)
  | h .Sim_lam  e ⇒
      let Howe_absC e' f = howe_ev_abs (down_closed e h) in
          ESim_lam e' (mlam R ⇒ howe_sim (f [R])

rec howe_osim : {γ:ctx} Howe [T] [γ ⊢ M] [γ ⊢ N]
                → OSim [T] [γ ⊢ M] [γ ⊢ N] =
fun [γ] h ⇒ OSimC (mlam σ ⇒ howe_sim (howe_subst h (howeSubst_refl [σ])));
```

Fig. 12. The Howe relation is included in open similarity.

```
                    → ({R:[term T]} Howe [Tλ;'] [M'[R]] [N'[R]])
                    → HoweAbs [x:term S ⊢ M'] [N];

rec howe_ev_abs : Howe [arr S T] [lam λx.M'] [N]
                    → HoweAbs [x:term S ⊢ M'] [N]
```

We are now ready to prove that the Howe relation coincides with open similarity. We do this by first proving that, in the empty context, the Howe relation is a similarity, then we embed the open version into an open similarity. To do so, we construct out of the input substitution σ for open similarity a derivation $\cdot \vdash \sigma \preccurlyeq^{\mathcal{H}}_{\Gamma} \sigma$ by reflexivity. The proofs appear in Figure 12.

4. Related work

As we mentioned before, Beluga implements indexed copatterns following Thibodeau *et al.* (2016). Beluga is certainly not the only system to feature copatterns, which have been first implemented in MiniAgda (Abel 2012) in an early form and then in Agda as a prototype (Abel *et al.* 2013). Copatterns now show up even in mainstream programming languages such as OCaml (Laforgue and Régis-Gianas 2017) as they allow us to integrate elegantly lazy evaluation within an eager language. However, Beluga is the only system supporting coinductive reasoning about HOAS representation using copatterns.

The observation paradigm differs from a more traditional approach using lazy constructors as adopted by Coq (Giménez 1996), in that coinductive types are defined by their elimination rules as opposed to their introduction rules. Coq's intensional type theory offers a strong decidable equational theory through dependent pattern matching, but loses subject reduction in the presence of coinduction (Oury 2008). In fact, the change of paradigm to observations was proposed to overcome that very issue with subject reduction when mixing lazy constructors with dependent case analysis (Abel *et al.* 2013).

Isabelle recently also overhauled (see, for example, Biendarra *et al.* (2017)) their handling of coinduction.

The first HOAS-like formal verification of the congruence of a notion of bisimilarity concerned the π-calculus (Honsell *et al.* 2001) and was carried out in Coq using the weak HOAS approach and instantiating the *Theory of Contexts* to axiomatizing properties of names. As common in many coinductive developments in Coq, the authors soon ran afoul of the guardedness checker in `Cofix`-style proofs and had to resort to an explicit greatest fixed point encoding for strong late bisimilarity. Abella's take on the same issue (Tiu and Miller 2010) seems preferable; that paper details, among so much more, a proof that similarity is a pre-congruence for the finite π-calculus. The encoding is rather elegant, where all issues involving bindings, names and substitutions are handled declaratively without explicit side conditions, thanks to the ∇-quantifier. This style of encoding has been extended in Chaudhuri *et al.* (2015) to handle bisimilarity 'up-to.' This is achieved via a limited form of quantification on relations that does not have, to our knowledge, a consistency proof yet. The authors do not pursue a proof of congruence in the cited paper.

Recent years have seen much work regarding the formalization of process calculi, in particular, using Nominal Isabelle. We do not have the space to discuss this line of work in detail but simply cite some of the many contributions from Parrow and his associates (Bengtson and Parrow 2009; Bengtson *et al.* 2016; Parrow *et al.* 2014) about various versions of the π/ψ-calculus and their congruence properties, proven without resorting to the Howe's proof strategy.

Encoding bisimilarity in the λ-calculus, in particular, via Howe's method, brings in additional challenges, as we have seen. We are aware of several formalizations through the years:

1. In Ambler and Crole (1999), the authors verify in Isabelle/HOL 98 the same result of the present paper and a bit more (they also show that similarity coincides with contextual pre-order) for PCFL using de Bruijn indices as an encoding techniques for binders. The development, for the congruence part, consists of around 160 lemmas/theorems, and it confirms a common belief about (standard) concrete syntax approaches: doable, but very hard going.
2. A partial improvement was presented in Momigliano *et al.* (2002), which was based on the HOAS approach implemented in an early version of the *Hybrid* tool (Felty and Momigliano 2012), but one crucial lemma was left unproven, tellingly: Howe's substitutivity. This was related to the difficulty of lifting substitution as β-conversion to substitution on judgements in one-level Hybrid term.
3. In Momigliano (2012), the author fixed this problem, giving a complete Abella proof for the simply typed calculus with unit. The proof consists of circa 45 theorems, 1/4 of which devoted to maintaining typing invariants in (open) similarity and in the candidate relation, 1/7 of which instead used to make sure that some ∇-quantified variable cannot occur in certain predicates. The main source of difficulty was again in the proof of substitutivity of the Howe relation, in particular, while handling structural properties of explicit contexts.

4. Using the present paper as a blue print, Chaudhuri (2018) gives a proof in Abella of the substitutivity of the Howe relation that is very close to the one discussed here; it is based on a theory of first class simultaneous substitutions encoded via the *copy* clauses as originally suggested by Miller: $m_1/x_1, \ldots, m_n/x_n$ is represented as the Abella context `copy xn Mn :: ... :: copy x1 M1 :: nil`. The application of a substitution $[\sigma]m = n$ becomes the *derivability* of the judgement $\{\ulcorner\sigma\urcorner \vdash$ `copy` $\ulcorner m\urcorner\ \ulcorner n\urcorner\}$. The theory underlying this formalization consists of 15 theorems, four of which are sensitive to the signature: in this sense, the theory has to be stated and re-proven for each signature. Compared to the Beluga development, where substitutions and their equality theory are built-in, this is more labour intense roughly adding a factor of 1.6. Moreover, Abella's proposed handling of simultaneous substitutions is, of course, relational and must be explicitly applied whenever needed. For example, the proof of `osim_ocus`, which in Beluga is a one liner, requires here the appeal to five lemmas to ensure that substitutions and their compositions are functional, that types and well formedness of contexts are preserved, etc.

In another recent paper, McLaughlin *et al.* (2018) give a formalization of the coincidence of observational and applicative approximation not going through the candidate relation, but triangulating with a notion of *logical* (as in logical relations) approximation. This is then extended to CIU approximation. The encoding uses first-order syntax for terms, but a form of weak HOAS for judgements following Allais *et al.* (2017), and it is therefore compatible with a standard proof assistant such as Agda. Similarly to us, it leverages the use of intrinsically well-typed and well-scoped terms and simultaneous substitutions, although the latter are not supported natively by the framework. Interestingly, it offers an elegant notion of concrete context (and thus of Morris approximation) that seems much easier to reason with than previous efforts (Ford and Mason 2003).

Lenglet and Schmitt (2018) present a formalization of Sangiorgi's higher order π-calculus in Coq, using the locally nameless approach to representing name restriction and well-scoped de Bruijn indices for process variables. The formalization includes a proof that strong context bisimilarity is a congruence, via an adaptation to the concurrent setting of Howe's method. The authors seem unaware of HOAS representations of the π-calculus and the representation technique adopted, albeit state of the art among the concrete ones, still requires a lot of boilerplate infrastructure to handle names and related notions.

5. Conclusions and future work

We have outlined how to use Beluga to encode a significant example of reasoning about program equivalence using Howe's method for PCFL. This has reinforced several observations that we have been making in other case studies, viz. (Cave and Pientka 2015, 2018):

— Using intrinsically typed terms instead of working with explicit typing invariants makes our encoding more compact and easier to deal with, since HOAS maintains our terms well scoped. The advantages of intrinsically typed representations have also been observed in non-HOAS setting (Allais *et al.* 2017; Benton *et al.* 2012).

— The support for built-in simultaneous substitutions and contexts lead us to generalize some statements, for example, substitutivity; this paid off in our mechanization, as many crucial lemmas became simpler to prove.

— Thanks to catering for both indexed inductive and coinductive data types in Beluga, the encoding of similarity and of the candidate relation was concise and as close as the informal presentation as one can reasonably hope for.

An important question, when discussing formalizations, regards *trust* in mechanized proofs. There are two aspects that play a role: the first concerns the theoretical foundations underlying a proof environment and the second the implementation of that foundation in a concrete system. Beluga's theoretical foundation provides the justification for reading its programs as proofs. For reasoning inductively directly over contextual LF objects, we exploit the subterm ordering on LF terms, see Pientka (2005) and Pientka and Abel (2015). Our (co)inductive programs give a high-level surface language to the core calculus described in Jacob-Rao *et al.* (2018). Coverage for our programs using (co)pattern matching is described in Thibodeau *et al.* (2016) and the well foundedness of such programs is justified by reasoning about sizes, where the size describes the depth of observations in the co-recursive case (Abel and Pientka 2013, 2016). Hence, the theoretical foundations for justifying the formalization of this case study are well developed. However, the implementation of these theoretical ideas in Beluga (status: July 2018) – as it is often the case – lags slightly behind; in particular, we are in the process of extending the implementation of our totality checker to support simultaneous copattern matching (Abel and Pientka 2013; Thibodeau *et al.* 2016). This would allow us not only to certify inductive proofs, but also coinductive ones.

While we hope that we have convinced the reader that Beluga is an excellent environment for the meta-theory of program equivalence, this case study has shown that is not well suited (yet) to verifying the equivalence of concrete pieces of code. To approach this, we need not only to improve the interactive proof construction mode similarly to what Agda offers, but to investigate proof search (Baelde *et al.* 2007) and model checking (Cheney and Momigliano 2017).

Last but not the least, let us take a look beyond mechanizing bisimilarity that has been the focus of this paper: the attentive reader may have noticed that we have not proven that bisimilarity coincides with contextual equivalence, as e.g. in Ambler and Crole (1999). The challenge lies in the encoding of the latter notion: we would rather avoid using a notion of concrete non-α-equivalent terms with holes, in favour of a context-less formulation, whereby contextual equivalence can be seen as the largest *adequate and compatible* relation (Lassen 1998; Pitts 2011). This requires extending Beluga to at least second-order quantification. This is work in progress, as it would be useful in many other scenarios, e.g. normalization for system $\mathcal{F}$.

Finally, we believe it would be interesting to explore other approaches to proving program equivalence such as *step-indexed logical relations* (Ahmed 2006) and compare this approach with Howe's. This would give us a deeper understanding of program equivalence proofs and provide insights into how both approaches scale to more complex programming languages such as in Pitts (2005) and Crary and Harper (2007).

Appendix A. Overview of Beluga Source Language

A Beluga signature consists of LF declarations, inductive and coinductive definitions, and programs. For each LF type family `a`, we declare its LF kind together with the constants that allow us to form objects that inhabit the type family. We also support mutual LF definitions that we do not capture in our grammar below not to complicate matters further.

$$
\begin{array}{lll}
\text{Signature Decl. } \mathcal{D} & ::= & \textbf{LF}\ \mathtt{a}\ :\ \mathcal{K}_{\mathsf{LF}}\ =\ \mathtt{c}_1\ :\ \mathcal{A}_1 \mid \ldots \mid \mathtt{c}_k\ :\ \mathcal{A}_k\,; \\
 & & \textbf{inductive}\ \mathtt{a}\ :\ \mathcal{K}\ =\ \mathtt{c}_1 : \mathcal{T}_1 \mid \ldots \mid \mathtt{c}_n : \mathcal{T}_n\,; \\
 & & \textbf{coinductive}\ \mathtt{a}\ :\ \mathcal{K}\ =\ (\mathtt{c}_1 : \mathcal{T}_1)\ ::\ \mathcal{T}'_1 \mid \ldots \mid (\mathtt{c}_n : \mathcal{T}_n)\ ::\ \mathcal{T}'_n\,; \\
 & & \textbf{rec}\ \mathtt{c}\ :\ \mathcal{T}\ =\ \mathcal{E}\,;
\end{array}
$$

(Co)inductive definitions correspond to indexed (co)recursive types and semantically are interpreted as (greatest) least fixed points. We define an indexed recursive type family by defining constructors $\mathtt{c}_i$ that can be used to construct its elements. We define a corecursive type by the observations we can make about it using indexed records, where we write the field $\mathtt{c}_i$ together with the type $\mathcal{T}_i$ from which we can project the result $\mathcal{T}'_i$. Given a Beluga term of $\mathcal{E}$ type $\mathcal{T}_i$, we may use the projection $\mathtt{c}_i$ and the result of $\mathcal{E}.\mathtt{c}_i$ is then of type $\mathcal{T}'_i$.

We describe Beluga's type and terms more precisely in Figure 13. The syntax for LF kinds, types and terms is close to the syntax x in the Twelf system. We write curly braces `{ }` for the dependent LF function space and allow users to write simply →, if there is no dependency. LF kinds, types and LF terms used in a signature declaration must be pure, i.e. they cannot refer to closures highlighted in blue and written as $\mathtt{X}[\sigma]$ and $\mathtt{\#p}[\sigma]$. Here, `X` and `#p` are meta-variables that are bound and introduced in Beluga types and patterns.

Substitutions in closures are either empty, written as ␣ here, or a weakening substitution …, which we use in practice to transition from a context to a possible extension. Substitutions can also be built by extending a substitution σ with an LF term $\mathcal{M}$.

LF contexts may be empty, consists of a context variable, or are built by concatenating to an LF context an LF variable declaration.

Contextual types and terms pair a LF context together with an LF type or LF term, respectively. As LF contexts are first class in Beluga, they form valid contextual objects. LF contexts are in general classified by a schema that allows us to state an invariant the LF context satisfies; here, we only add one schema we have used in this development (Section 3.7), namely `ctx`.

Beluga's type language supports indexed dependent function space, $\{\mathtt{X}{:}\mathcal{U}\}\mathcal{T}$, non-dependent functions, embedding contextual types and (co)inductive definitions, described as $\mathtt{a}\ \mathcal{C}_1 \ldots \mathcal{C}_n$. Note that in $\{\mathtt{X}{:}\mathcal{U}\}\mathcal{T}$, we make `X` explicit and hence any computation-level expression of that type expects first an object of type $\mathcal{U}$. We also permit a limited form of implicit type annotation for context variables; by writing $(\mathtt{X}{:}\mathtt{ctx})\mathcal{T}$, we can abstract over the context variable `X` and declare its context schema, while keeping `X` implicit.

Beluga's computation language allows us to make statements about contextual types and objects and we highlight them in blue as well. It is in many ways similar to standard functional programming languages with two exceptions: first, contextual types and objects

LF Kinds	$\mathcal{K}_{\mathsf{LF}}$	$::=$	$\textbf{type} \mid \{\mathtt{X}{:}\mathcal{A}\}\mathcal{K}_{\mathsf{LF}} \mid \mathcal{A} \to \mathcal{K}_{\mathsf{LF}}$
LF Types	$\mathcal{A}$	$::=$	$\mathtt{a}\ \mathcal{M}_1 \ldots \mathcal{M}_n \mid \{\mathtt{X}{:}\mathcal{A}\}\mathcal{A}' \mid \mathcal{A} \to \mathcal{A}'$
LF Terms	$\mathcal{M}$	$::=$	$\mathtt{x} \mid \lambda\mathtt{x}.\mathcal{M} \mid \mathtt{c}\ \mathcal{M}_1 \ldots \mathcal{M}_n \mid \mathtt{X}[\sigma] \mid \#\mathtt{p}[\sigma]$
LF Subst.	σ	$::=$	$␣ \mid \ldots \mid \sigma, \mathcal{M}$
LF Context	$\Psi,\ \Phi$	$::=$	$␣ \mid \psi \mid \Psi,\mathtt{x}{:}\mathcal{A}$
Contextual Type	$\mathcal{U}$	$::=$	$[\Psi \vdash \mathtt{a}\ \mathcal{M}_1 \ldots \mathcal{M}_n] \mid [\mathtt{ctx}] \mid [\Psi \vdash \Phi] \mid \ldots$
Contextual Object	$\mathcal{C}$	$::=$	$[\Psi \vdash \mathcal{M}] \mid [\Psi] \mid \ldots$

Beluga Kinds	$\mathcal{K}$	$::=$	$\textbf{ctype} \mid \{\mathtt{X}{:}\mathcal{U}\}\mathcal{K} \mid \mathcal{U} \to \mathcal{K}$
Beluga Types	$\mathcal{T}$	$::=$	$\{\mathtt{X}{:}\mathcal{U}\}\mathcal{T} \mid (\mathtt{X}{:}\mathtt{ctx})\mathcal{T} \mid \mathcal{T} \to \mathcal{T} \mid \mathcal{U} \mid \mathtt{a}\ \mathcal{C}_1 \ldots \mathcal{C}_n$
Beluga Terms	$\mathcal{E}$	$::=$	$\mathcal{E}\ \mathcal{E} \mid \mathcal{C} \mid \mathtt{x} \mid \mathtt{c} \mid \mathcal{E}.\mathtt{c} \mid \textbf{fun}\ \mathcal{B}_{\mathcal{R}} \mid \textbf{let}\ \mathcal{P} = \mathcal{E}\ \textbf{in}\ \mathcal{E}$
			$\mid \mathtt{mlam}\ \mathtt{X} \Rightarrow \mathcal{E} \mid \textbf{case}\ \mathcal{E}\ \textbf{of}\ \mathcal{B}_{\mathcal{P}}$
Beluga Copat. Branches	$\mathcal{B}_{\mathcal{R}}$	$::=$	$␣ \mid (\mathcal{B}_{\mathcal{R}} \mid \mathcal{R}_1 \ldots \mathcal{R}_n \Rightarrow \mathcal{E})$
Beluga Pat. Branches	$\mathcal{B}_{\mathcal{P}}$	$::=$	$␣ \mid (\mathcal{B}_{\mathcal{P}} \mid \mathcal{P} \Rightarrow \mathcal{E})$
Beluga Pattern	$\mathcal{P}$	$::=$	$\mathtt{x} \mid \mathcal{C} \mid \mathtt{c}\ \mathcal{P}_1 \ldots \mathcal{P}_n$
Beluga Copatterns	$\mathcal{R}$	$::=$	$\cdot \mid \mathcal{P}\ \mathcal{R} \mid .\mathtt{c}\ \mathcal{R}$

Fig. 13. Grammar of Beluga.

are the special domain and we support not only pattern matching, but also copattern matching, i.e. our patterns may include projections describing the observations that we can make about a given (output) type. In particular, we allow functions to be defined via (co)pattern matching using the **fun** keyword. Beluga also supports case analysis using pattern matching independently for convenience. Further, as in OCaml, we not only allow functions to be defined via (co)pattern matching, but to be formed by abstracting over their arguments. For example, functions defined using `mlam` abstract over contextual variables occurring in the function body. Further, our language features let-expressions as a degenerate case of case analysis, together with applications, computation-level variables, constructors, constants and projections. This is by no means a complete account of the computation language of Beluga– we have only described here the part relevant for our case study.

References

Abel, A. (2012). Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types. In: *Proceedings of the Invited Talk at 8th Workshop on Fixed-points in Computer Science (FICS'12)* 1–11.

Abel, A. and Pientka, B. (2013). Well-founded recursion with copatterns: A unified approach to termination and productivity. In: *Proceedings of the 18th International Conference on Functional Programming (ICFP'13)* 185–196.

Abel, A. and Pientka, B. (2016). Well-founded recursion with copatterns and sized types. *Journal of Functional Programming* **26** e2.

Abel, A., Pientka, B., Thibodeau, D. and Setzer, A. (2013). Copatterns: Programming infinite structures by observations. In: *Proceedings of the 40th Symposium on Principles of Programming Languages (POPL'13)*, ACM Press, 27–38.

Abramsky, S. (1991). A domain equation for bisimulation. *Information and Computation* **92** (2) 161–218.

Ahmed, A. (2006). Step-indexed syntactic logical relations for recursive and quantified types. In: Sestoft, P. (eds.) *Proceedings of the 15th European Symposium on Programming (ESOP'06)*, Springer, 69–83.

Allais, G., Chapman, J., McBride, C. and McKinna, J. (2017). Type-and-scope safe programs and their proofs. In: Bertot, Y. and Vafeiadis, V. (eds.) *Proceedings of the 6th Conference on Certified Programs and Proofs (CPP'17)*, ACM, 195–207.

Ambler, S. and Crole, R.L. (1999). Mechanized operational semantics via (co)induction. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C. and Théry, L. (eds.) *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99)*, Lecture Notes in Computer Science, vol. 1690, Springer, 221–238.

Baelde, D., Chaudhuri, K., Gacek, A., Miller, D., Nadathur, G., Tiu, A. and Wang, Y. (2014). Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning* **7** (2) 1–89.

Baelde, D., Gacek, A., Miller, D., Nadathur, G. and Tiu, A. (2007). The Bedwyr system for model checking over syntactic expressions. In: *Proceedings of the 21st Conference on Automated Deduction (CADE)*, Lecture Notes in Computer Science, vol. 4603, Springer, 391–397.

Bengtson, J. and Parrow, J. (2009). Formalising the pi-calculus using nominal logic. *Logical Methods in Computer Science* **5** (2) 1–36.

Bengtson, J., Parrow, J. and Weber, T. (2016). Psi-calculi in Isabelle. *Journal of Automated Reasoning* **56** (1) 1–47.

Benton, N., Hur, C., Kennedy, A. and McBride, C. (2012). Strongly typed term representations in coq. *Journal of Automated Reasoning* **49** (2) 141–159.

Biendarra, J., Blanchette, J. C., Bouzy, A., Desharnais, M., Fleury, M., Hölzl, J., Kuncar, O., Lochbihler, A., Meier, F., Panny, L., Popescu, A., Sternagel, C., Thiemann, R. and Traytel, D. (2017). Foundational (co)datatypes and (co)recursion for higher-order logic. In: Dixon, C. and Finger, M. (eds.) *Proceedings of the 11th International Symposium on Frontiers of Combining Systems (FroCoS'17)*, Lecture Notes in Computer Science, vol. 10483, Springer, 3–21.

Cave, A. and Pientka, B. (2012). Programming with binders and indexed data-types. In: *Proceedings of the 39th Symposium on Principles of Programming Languages (POPL'12)*, ACM Press, 413–424.

Cave, A. and Pientka, B. (2013). First-class substitutions in contextual type theory. In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'13)*, ACM Press, 15–24.

Cave, A. and Pientka, B. (2015). A case study on logical relations using contextual types. In: Cervesato, I. and Chaudhuri, K. (eds.) *Proceedings of the 10th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'15)*, Electronic Proceedings in Theoretical Computer Science, 18–33.

Cave, A. and Pientka, B. (2018). Mechanizing proofs with logical relations – Kripke-style. *Mathematical Structures in Computer Science*, **28** (Special Issue 9) 1606–1638.

Chaudhuri, K. (2018). A two-level logic perspective on (simultaneous) substitutions. In Andronick, J. and Felty, A.P. (eds.) *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018)*, Los Angeles, CA, USA, January 8–9, 2018, ACM, 280–292.

Chaudhuri, K., Cimini, M. and Miller, D. (2015). A lightweight formalization of the metatheory of bisimulation-up-to. In Leroy, X. and Tiu, A. (eds.) *Proceedings of the 2015 Conference on Certified Programs and Proofs (CPP 2015)*, Mumbai, India, January 15–17, 2015, ACM, 157–166.

Cheney, J. and Momigliano, A. (2017). αcheck: A mechanized metatheory model checker. *TPLP* **17** (3) 311–352.

Crary, K. and Harper, R. (2007). Syntactic logical relations for polymorphic and recursive types. *Electronic Notes in Theoretical Computer Science* **172** 259–299.

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

Ford, J. and Mason, I.A. (2003). Formal foundations of operational semantics. *Higher-Order and Symbolic Computation* **16** (3) 161–202.

Ghica, D.R. and McCusker, G. (2000). Reasoning about idealized ALGOL using regular languages. In: Montanari, U., Rolim, J.D.P. and Welzl, E. (eds.) *Proceedings of the 27th International Colloquium, Automata, Languages and Programming (ICALP 2000)*, Lecture Notes in Computer Science, vol. 1853, Springer, 103–115.

Giménez, E. (1996). *Un Calcul de Constructions Infinies et son application à la vérification de systèmes communicants*. PhD thesis, Ecole Normale Supérieure de Lyon, Thèse d'université.

Harper, R., Honsell, F. and Plotkin, G. (1993). A framework for defining logics. *Journal of the ACM* **40** (1) 143–184.

Hirschkoff, D. (1997). A full formalisation of pi-calculus theory in the calculus of constructions. In: Gunter, E.L. and Felty, A.P. (eds.) *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, Lecture Notes in Computer Science, vol. 1275, Springer, 153–169.

Honsell, F., Miculan, M. and Scagnetto, I. (2001). Π-calculus in (co)inductive type theories. *Theoretical Computer Science* **2** (253) 239–285.

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

Jacob-Rao, R., Pientka, B. and Thibodeau, D. (2018). Index-stratified types. In: Kirchner, H. (ed.) *Proceedings of the 3rd International Conference on Formal Structures for Computation and Deduction (FSCD'18)*, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 19:1–19:17.

Laforgue, P. and Régis-Gianas, Y. (2017). Copattern matching and first-class observations in OCaml, with a macro. In: *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, PPDP'17, 97–108.

Lassen, S. B. (1998). *Relational Reasoning About Functions and Nondeterminism.* PhD thesis, Department of Computer Science, University of Aarhus.

Lee, D. K., Crary, K. and Harper, R. (2007). Towards a mechanized metatheory of Standard ML. In: *Proceedings of the 34th Symposium on Principles of Programming Languages (POPL'07)*, ACM Press, 173–184.

Lenglet, S. and Schmitt, A. (2018). Hoπ in coq. In: Andronick, J. and Felty, A.P. (eds.) *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018)*, Los Angeles, CA, USA, January 8–9, 2018, ACM, 252–265.

Mason, I. and Talcott, C. (1991). Equivalence in functional languages with effects. *Journal of Functional Programming* **1** (03) 287–327.

McDowell, R. and Miller, D. (1997). A logic for reasoning with higher-order abstract syntax. In: Winskel, G. (eds.) *Proceedings of the 12th Symposium on Logic in Computer Science*, IEEE Computer Society Press, 434–445.

McDowell, R., Miller, D. and Palamidessi, C. (1996). Encoding transition systems in sequent calculus: Preliminary report. *Electronic Notes in Theoretical Computer Science*, **3** 138–152.

McLaughlin, C., McKinna, J. and Stark, I. (2018). Triangulating context lemmas. In: Andronick, J. and Felty, A.P. (eds.) *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018)*, Los Angeles, CA, USA, January 8–9, 2018, ACM, 102–114.

Miller, D. and Nadathur, G. (2012). *Programming with Higher-Order Logic*, 1st edition, Cambridge University Press, New York, NY, USA.

Miller, D. and Palamidessi, C. (1999). Foundational aspects of syntax. *ACM Computing Surveys* **31** (3es) 1–7.

Miller, D. and Tiu, A. (2005). A proof theory for generic judgments. *ACM Transactions on Computational Logic* **6** (4) 749–783.

Milner, R. (1977). Fully abstract models of typed -calculi. *Theoretical Computer Science* **4** (1) 1–22.

Momigliano, A. (2012). A supposedly fun thing I may have to do again: A HOAS encoding of Howe's method. In: *Proceedings of the 7th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP'12)*, ACM, 33–42.

Momigliano, A., Ambler, S. and Crole, R. L. (2002). A Hybrid encoding of Howe's method for establishing congruence of bisimilarity. *Electronic Notes in Theoretical Computer Science* **70** (2) 60–75.

Momigliano, A. and Tiu, A. (2003). Induction and co-induction in sequent calculus. In: Coppo, M., Berardi, S. and Damiani, F. (eds.) *Post-Proceedings of TYPES 2003*, Lecture Notes in Computer Science, vol. 3085, 293–308.

Nanevski, A., Pfenning, F. and Pientka, B. (2008). Contextual modal type theory. *ACM Transactions on Computational Logic* **9** (3) 1–49.

Oury, N. (2008). Coinductive types and type preservation. Message on the coq-club mailing list.

Parrow, J., Borgström, J., Raabjerg, P. and Pohjola, J.Å. (2014). Higher-order psi-calculi. *Mathematical Structures in Computer Science* **24** (2) 1–36.

Pfenning, F. (1997). Computation and deduction. Accessed January 31st, 2018.

Pfenning, F. and Schürmann, C. (1999). System description: Twelf — A meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, Lecture Notes in Artificial Intelligence, vol. 1632, Springer, 202–206.

Pientka, B. (2005). Verifying termination and reduction properties about higher-order logic programs. *Journal of Automated Reasoning* **34** (2) 179–207.

Pientka, B. (2008). A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In: *Proceedings of the 35th Symposium on Principles of Programming Languages (POPL'08)*, ACM Press, 371–382.

Pientka, B. (2013). An insider's look at LF type reconstruction: Everything you never wanted to know. *Journal of Functional Programming* **23** (1) 1–37.

Pientka, B. and Abel, A. (2015). Structural recursion over contextual objects. In Altenkirch, T. (ed.) *Proceedings of the 13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*, Leibniz International Proceedings in Informatics (LIPIcs) of Schloss Dagstuhl, 273–287.

Pientka, B. and Cave, A. (2015). Inductive Beluga: Programming proofs (system description). In: Felty, A.P. and Middeldorp, A. (eds.) *Proceedings of the 25th International Conference on Automated Deduction (CADE-25)*, Lecture Notes in Computer Science, vol. 9195, Springer, 272–281.

Pientka, B. and Dunfield, J. (2008). Programming with proofs and explicit contexts. In: Antoy, S. and Albert, E. (eds.) *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, July 15–17, 2008, Valencia, Spain, ACM, 163–173.

Pientka, B. and Dunfield, J. (2010). Beluga: A framework for programming and reasoning with deductive systems (System Description). In: Giesl, J. and Haehnle, R.(eds.) *Proceedings of the 5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence, vol. 6173, Springer, 15–21.

Pitts, A.M. (1997). Operationally based theories of program equivalence. In: Dybjer, P. and Pitts, A.M. (eds.) *Semantics and Logics of Computation*, Cambridge University Press, 241–298.

Pitts, A.M. (2005). Typed operational reasoning. In: Pierce, B.C. (ed.) *Advanced Topics in Types and Programming Languages*, chapter 7, The MIT Press, 245–289.

Pitts, A.M. (2011). Howe's method for higher-order languages. In Sangiorgi, D. and Rutten, J. (eds.) *Advanced Topics in Bisimulation and Coinduction*, Cambridge Tracts in Theoretical Computer Science, vol. 52, chapter 5, Cambridge University Press, 197–232.

Thibodeau, D., Cave, A. and Pientka, B. (2016). Indexed codata. In: Garrigue, J., Keller, G. and Sumii, E. (eds.) *Proceedings of the 21st International Conference on Functional Programming (ICFP'16)*, ACM, 351–363.

Tiu, A. and Miller, D. (2010). Proof search specifications of bisimulation and modal logics for the π-calculus. *ACM Transactions on Computational Logic* **11** (2) 1–35.

Tiu, A. and Momigliano, A. (2012). Cut elimination for a logic with induction and co-induction. *Journal of Applied Logic* **10** (4) 330–367.