Mechanizing Bisimulation for a Linear Functional Language: Enforcing Linearity Without Linearity

L Denney

McGill University

Conference of McGill's Epic Programming Language Systems
November 2025

Goal

- **Goal** Adapt the mechanization of Howe's method in Beluga that follows Pitts (1997) to a linear language.
  - Others have done similar proofs about a linear language including Bierman (2000) and Crole (2001) but mechanizations of these would be difficult and forced to struggle with the context splitting
- **Our Idea:** Be more clever and build on an observation by Crary (2010): If a variable occurs once, then the typing assumption will be only used once! In other words, occurrence of variables in the syntax correspond to how often typing assumptions are consumed.
  - This idea has been used in mechanizing other linear systems such as processes [Sano et al. (2023)]

Motivation and Background

Linear Functional Language

- All variables must occur **exactly once** in an expression.
  No duplicating or dropping.
- Encodes resource control into a language: Quantum programming, processes, etc.

The linear functional language we will consider, which is lazily evaluated:

Types	$\tau ::= \top \mid \tau \multimap \tau$
Terms	$M, N ::= x \mid \mathtt{lam}\, x.M \mid M\ N \mid \langle\rangle$
Values	$V ::= \mathtt{lam}\, x.M \mid \langle\rangle$

Typing Function Applications Linearly

$$\frac{\Gamma_1 \vdash M : \sigma \multimap \sigma' \qquad \Gamma_2 \vdash N : \sigma}{\Gamma_1, \Gamma_2 \vdash M\ N : \sigma'}\ \text{app}$$

The Big Idea: Contextual Equivalence

Two sub-programs can be interchanged within a larger program without affecting overall behaviour.

Example:

```
for n in range(1,N): # N incl.   |     sum += N*(N+1)/2
    sum += n                    |
```

Definition

Two programs are contextually equivalent if they are interchangeable within all larger programs without affecting the observable behaviour

Why do we want this? Compiler Optimizations

Bisimilarity

Proving contextual equivalence directly is hard because it requires quantification over every possible context, so to get around this we introduce bisimilarity!

Definition

Two programs (closed terms) are bisimilar if whenever the first evaluates to a value, so does the second one, and all its sub-programs are also bisimilar and vise versa

Howe's Method

Want to prove that applicative bisimilarity is a congruence [Howe (1989)]

Applicative Bisimilarity: a coinductive characterization of observational equivalence for proving program equivalence CBN functional languages

Congruence: a transitive relation respecting the way terms are constructed

We can get bisimilarity from similarity by symmetry, so we only need to show that *applicative similarity* is a *pre-congruence*.

Applicative simulation is a family of typed relations on programs that are related by the result of CBN evaluation. The union of two applicative simulations is still a simulation, and Applicative Similarity is the largest of those.

Pre-Congruence: A compatible transitive relation, where compatible is defined below.

Definition 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 $\Gamma \vdash m_2 \; \mathcal{R}_\tau \; n_2$ entails $\Gamma \vdash (m_1 \; m_2) \; \mathcal{R}_{\tau'} \; (n_1 \; n_2)$;

Howe's Method

- Our language has a variable binding operator `lam` so we need to consider relations over **open terms** to be able to consider all the subprograms.
- We need substitutivity to prove congruence, but proving substitutivity of similarity for open terms directly is hard

Idea [Howe (1996)]: Define some candidate Howe relation that contains open similarity then

1. show it is a pre-congruence
2. prove it coincides with similarity

A good tutorial on this is Pitts (1997).

Aside: Coinduction

Howe's method requires both inductive and coinductive reasoning over open terms, and thankfully Beluga is equipped for both of these.

- Induction - finite data, defined with constructors, analyzed with pattern matching
- Coinduction - infinite data, defined with deconstructors (observations), analyzed with co-pattern matching

"A property holds by induction if there is good reason for it to hold; whereas a property holds by coinduction if there is no good reason for it not to hold." [Kozen and Silva (2017)]

Previous Work on Bisimilarity for Linear Languages

"Observations on a linear PCF" [Bierman (2000)]

- LinPCF (includes both kinds of linear pairs), in terms of evaluation: function application and multiplicative pairs are eager, and additive pairs are lazy.
- Manages the linear context explicitly
- Doesn't mention mechanization at all, it is expected a mechanization of his bisimilarity proof would be inelegant and difficult.

"Completeness of Bisimilarity for Contextual Equivalence in Linear Theories" [Crole (2001)]

- Function space, multiplicative pairs and a divergent term, CBN evaluation
- Also manages the linear context explicitly
- Mentions defining contextual equivalence to be amenable to mechanization, no mention concerning bisimilarity.
- Expected that we can improve upon what a mechanization of his bisimilarity proof would require.

In a Functional Language

Idea:

1. Work in a non linear language
2. Enforce linearity

The language we want to use is a subset of PCFL following Pitts (1997), without the lists and pairs (for now) .

Types	$\tau ::= \top \mid \tau \to \tau$
Terms	$M, N ::= x \mid \mathtt{lam}\, x.M \mid M\ N \mid \langle\rangle$
Values	$V ::= \mathtt{lam}\, x.M \mid \langle\rangle$

It's typing rules and CBN big step operational semantics $m \Downarrow v$ are standard.

Linear Predicate

Enforcing linearity without linearity: The Local Linear Predicate

$\texttt{linear}\ x\ M$ says that x appears linearly in M and is inductively defined as follows:

$$\frac{}{\texttt{linear}\ x\ x}\ \text{lin\_var} \qquad \frac{\Pi x : \text{term}.\texttt{linear}\ y\ (M\ y\ x)}{\texttt{linear}\ y\ (\text{lam}\ \lambda x.M\ x\ y)}\ \text{lin\_lam}$$

$$\frac{\texttt{linear}\ x\ (M\ x)}{\texttt{linear}\ x\ ((M\ x)\ N)}\ \text{lin\_app1} \qquad \frac{\texttt{linear}\ x\ (N\ x)}{\texttt{linear}\ x\ (M\ (N\ x))}\ \text{lin\_app2}$$

A linear predicate has been used in mechanizing other linear systems such as processes [Sano et al. (2023)] Now, we want to leverage it for contextual equivalence - less straightforward than syntactic induction proofs (we have coinduction)

The Meat and Potatoes

Goal and Scope

Goal

First, mechanize a proof of bisimulation for linear terms in PCFL using a linear predicate.
Then, argue this is a nice way of doing it because it:

1. Doesn't require managing the context explicitly
2. Leverages previous PCFL mechanizations

How

Splice the linear predicate in to the Howe's method bisimilarity mechanization in Beluga from Momigliano et al. (2019) and see if I can make it work!

Scope

We limit our terms to functions, applications, and unit.

Beluga

- Beluga supports simultaneous substitutions which we can adapt to preserve linearity. This allows us to encode notoriously painful proofs such as the substitutivity of the Howe relation with ease
- Beluga supports coinductive reasoning
- There is already an elegant mechanization of Howe's method in Beluga from Marabelli and Momigliano (2019) that we can adapt for linear terms without having to write it from scratch.

Example of a co-recursive function in Beluga.

```
rec sim_refl : Sim [|- T] [|- M] [|- M]=
fun .Sim_unit d => d
  | .Sim_lam d => ESim_lam d (mlam R => sim_refl)
;
```

How(e) Howe's method works

Recall It is difficult to show similarity/open similarity is substitutive.

Howe proposes a way to get that similarity is a pre-congruence without proving substitutivity for open similarity directly. [Howe (1996)]

Howe's Idea: Introduce a relation which

1. contains open similarity
2. is a semi-transitive substitutive pre-congruence

and then prove it coincides with similarity

Proof Structure Define the Howe relation in terms of open similarity, then prove 8 lemmata about it. Notably, lemma 6 is substitutivity, and lemma 8 is coincidence with similarity.

Predicate in Relation Definitions

Original version

$$\frac{\Gamma \vdash \langle\rangle \precsim^{\circ}_{\top} n}{\Gamma \vdash \langle\rangle \precsim^{\mathcal{H}}_{\top} n}\ \mathsf{unit} \qquad \frac{\Gamma, x : \tau \vdash x \precsim^{\circ}_{\tau} n}{\Gamma, x : \tau \vdash x \precsim^{\mathcal{H}}_{\tau} n}\ \mathsf{var}$$

$$\frac{\Gamma \vdash m_1 \precsim^{\mathcal{H}}_{\tau \to \tau'} m_1' \qquad \Gamma \vdash m_2 \precsim^{\mathcal{H}}_{\tau} m_2' \qquad \Gamma \vdash m_1'\ m_2' \precsim^{\circ}_{\tau'} n}{\Gamma \vdash m_1\ m_2 \precsim^{\mathcal{H}}_{\tau'} n}\ \mathsf{app}$$

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

New version

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

Changes to the Lemmata

```
LF lin : term A → type =
  | l_unit : lin unit
  | l_app : lin M → lin N
      → lin (app M N)
  | l_lam : linear M
      → ({y:term _ } lin y -> lin (M y))
      → lin (lam \y. M y)
```

Global linearity predicate `lin` m says the the term m is linear
It is defined on the LF level, using `linear` in the lambda case

Howe_lam needs `linear` x M as a premise, but our local linear predicate cannot store information about all the bound variables for us.
We use a global linear predicate `lin` to supply us with the local predicate when we encounter a binder.

```
rec howe_refl : (g:ctx) {M:[g |- term T[]]} [g |- lin M]
                  -> Howe [|- T] [g |- M] [g |- M] =
fun [g |- unit] l => Howe_unit osim_refl
  | [g |- #p] l => Howe_var [g |- #p] osim_refl
  | [g |- lam \x. M] [g |- l_lam l0 (\y. \ly. lm )]
      => Howe_lam (howe_refl [g, x:term _ |- M]  ?)
                  osim_refl [g |- l0]
  | [g |- app M N] [g |- l_app l1 l2]
      => Howe_app (howe_refl [g |- M] [g |- l1])
                  (howe_refl [g |- N] [g |- l2]) osim_refl
```

Additional Lemmas

Two Examples: Renaming preserves local and global linearity

```
rec linear_ren : {g:ctx}{h:ctx} {$S: $[h |-# g]}
                 {M:[g, x:term A[] ⊢ term B[]]} [g |- linear (\x. M)]
                  -> Lin_rename [g] $[h |-# $S]
                  -> [h |- linear (\x. M[$S[..],x])];


rec lin_ren : {g:ctx}{h:ctx} {$S: $[h |-# g]}
              {M:[g ⊢ term B[]]} [g |- lin M]
               -> Lin_rename [g] $[h |-# $S]
               -> [h |- lin M[$S[..]]]
```

Both are used in `howe_ren`, which is used in the proof of `howe_subst_wkn`, which is used in the proof of the 6th lemma, substitutivity.

Changes Important to Mechanization

The Beluga renaming and substitution don't preserve linearity, so we need to construct definitions of them that do.

```
inductive Lin_sub : {g:ctx} (h:ctx) {$S1 : $[h |- g]} ctype =
| LNil_sub : Lin_sub [] $[h |- ^]
| LCons_sub : Lin_sub [g] $[h |- $S1]
->  [h |- lin M]
-> Lin_sub [g, x:term T[]] $[h|- $S1, M] ;

inductive Lin_rename: {g:ctx} (h:ctx) {$S : $[h |-# g]} ctype =
| LNil_ren : Lin_rename [] $[h |-# ^]
| LCons_ren :  Lin_rename [g] $[h |-# $S]
-> Lin_rename [g, x:term T[]] $[h, x:term T[] |-# $S[..], x];
```

Conclusion

Conclusion

- We adapted the Momigliano et al. (2019) mechanization of Howe's method for linear terms.
- We discovered that the linear predicate lets us reuse the mechanization even though its non-linear, and made mechanization easier compared to past attempts by letting us avoid explicit context management.
- It worked even though Howe's method requires both inductive and coinductive reasoning.
- We should add back in the terms we ignored from PCFL, and complete the mechanization for those.
- We are one big step closer to contextual equivalence for linear programs!

Next Step: Reaching Contextual Equivalence

To reach Contextual Equivalence for linear programs:

1. define a linear contextual preorder (this is to contextual equivalence what similarity is to bisimilarity)
2. Show the linear contextual preorder coincides with similarity (the one we got from this mechanization!)

Crole (2001) claims his definition of linear contextual preorder is more suitable for mechanization than Bierman (2000)'s is, although his requires additional lemmas, and Bierman has fewer definitional clauses.

Look at what each of them did and try to simplify with a linear predicate. This could lead us to a cleaner mechanization of Contextual Equivalence than what either of their strategies currently require.

References I

A. Pitts, *Operationally-Based Theories of Program Equivalence*, ser. Publications of the Newton Institute. Cambridge University Press, 1997, p. 241–298.

G. M. Bierman, "Program equivalence in a linear functional language," *Journal of Functional Programming*, vol. 10, no. 2, pp. 167–190, 2000.

R. L. Crole, "Completeness of bisimilarity for contextual equivalence in linear theories," *Logic Journal of the IGPL*, vol. 9, no. 1, pp. 27–51, 01 2001. [Online]. Available: https://doi.org/10.1093/jigpal/9.1.27

K. Crary, "Higher-order representation of substructural logics," *SIGPLAN Not.*, vol. 45, no. 9, p. 131–142, Sep. 2010. [Online]. Available: https://doi.org/10.1145/1932681.1863565

C. Sano, R. Kavanagh, and B. Pientka, "Mechanizing session-types using a structural view: Enforcing linearity without linearity," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, Oct. 2023. [Online]. Available: https://doi.org/10.1145/3622810

References II

D. Howe, "Equality in lazy computation systems," in *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*. Pacific Grove, CA, USA: IEEE Comput. Soc. Press, 1989, pp. 198–203.

D. J. Howe, "Proving congruence of bisimulation in functional programming languages," *Information and Computation*, vol. 124, no. 2, pp. 103–112, 1996. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0890540196900085

D. Kozen and A. Silva, "Practical coinduction," *Mathematical Structures in Computer Science*, vol. 27, no. 7, p. 1132–1152, 2017.

A. Momigliano, B. Pientka, and D. Thibodeau, "A case study in programming coinductive proofs: Howe's method," *Mathematical Structures in Computer Science*, vol. 29, no. 8, p. 1309–1343, 2019.

G. Marabelli and A. Momigliano, "Formalizing program equivalences in dependent type theory," 2019. [Online]. Available: https://ceur-ws.org/Vol-2504/paper23.pdf