Formalizing Program Equivalences in Dependent Type Theory

Giorgio Marabelli[1] and Alberto Momigliano[2]*

[1] Dipartimento di
Matematica, Università degli Studi di Milano, `giorgio.marabellli@studenti.unimi.it`
[2] Dipartimento di Informatica, Università degli Studi di Milano
`momigliano@di.unimi.it`

Abstract. This brief note summarizes our formalization in a dependently typed setting of the meta-theory of several notions of program equivalences in higher-order programming languages.

1 Introduction

The equivalence of given programs is an important problem, especially in higher-order programming languages. If we manage to prove that two phrases behave "in the same manner" under every context — known as Morris-style *contextual/observational* equivalence [21] — then we can exchange them preserving the behavior of the whole programs. Examples abound in compilers optimization [11], interchangeability of ADT [15] and extend to more intensional properties than behavioral equivalence such as security [20].

Contextual equivalence is an appealing notion: it is easily shown to be a congruence, and this supports equational reasoning via rewriting and Leibniz equality. However, the quantification over *all* contexts makes a direct proof of the equivalence of two code fragments quite hard. Therefore, other more manageable notions of equivalence have emerged: a successful proposal coming from concurrency theory is (applicative) *bisimilarity* [21]: two programs m and n are bisimilar if whenever m evaluates to a value, so does n, and (roughly) all the subprograms of the resulting values are also bisimilar, and vice versa. While it is not immediate that bisimilarity is a congruence [10], it may offer modular and incremental ways to address programs equivalence.

This topic has become a sort of benchmark for researchers interested in *mechanizing* with proof assistants not only the equivalence of concrete programs, but more in general the *meta-theory* of program equivalences, see the formalizations in [18, 16, 14, 3, 17] to name just a few. This is challenging as the logical framework (and associated proof assistant) with which we wish to carry out the formalization needs to support several features that are not easily found together:

– a way to represent syntax with *binders*, which are ubiquitous in higher order languages such as ones based on the λ-calculus: in fact, the representation ought

* Partially funded by the INdAM-GNCS project 2018 "Metodi di prova orientati al ragionamento automatico per logiche non-classiche".

to be abstract enough so that mundane tasks such as α-equivalence or lookup in a context of assumptions do not overwhelm the development, while, at the same time it must not preclude dealing correctly with exotic syntactical entities such as non-capture avoiding contexts;
- tools for general inductive and more crucially coinductive reasoning;
- a logic powerful enough to formulate a general theory of (dependent) *relations* making the development modular and economic.

It is fair to say that none of the research so far has managed to exploit all of the above. In this note, we revisit and significantly extend the approach in [18], namely the proof that (applicative) similarity is a precongruence following Howe's method [22]. Rather than using a specialized proof assistant such as *Beluga* (`http://complogic.cs.mcgill.ca/beluga/`), which supports *higher-order abstract syntax* as a representation technique, but is restricted (so far) to essentially a first order fixed-point logic, we choose to carry out our development in the fully dependently-typed higher-order logic of *Coq* (`https://coq.inria.fr`). Thanks to Coq's impredicativity, we can approach our topic abstractly via Lassen's theory of *relational* reasoning [13]. This has two immediate benefits: for one, we can easily change our choice of *observables*, that is of what we consider the result of evaluating a program, and therefore of observational equivalence, and retain essentially the same proof of congruence for the correspond notion of bisimilarity. Secondly, we go much further than [18] by characterizing contextual equivalence both in terms of concrete well-typed contexts as in [16] and in the "context-less" way advocated by Gordon [9]. We close the circle by showing that the three equivalences (contextual, context-less and similarity) coincide, giving us the leisure to use one or the other at our convenience.

2 Formalizing program equivalences

Syntax First, we fix our model of a higher-order programming language in the guise of a simply-typed λ-calculus with recursion and lazy lists over the unit type (for simplicity), known as PCFL [21]:

$$\begin{array}{lll} \text{Type}\ \tau & ::= & \top \mid \tau \to \tau \mid \tau\ \mathsf{list} \\ \text{Exp}\ \ m & ::= & x \mid \mathsf{lam}\, x.p \mid m_1\ m_2 \mid \mathsf{fix}\ x.\ m \mid \langle\rangle \mid \mathsf{nil} \mid \mathsf{cons}\ m_1\ m_2 \mid \mathsf{lcase}\ m\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow n \mid \mathsf{cons}\ h\ t \Rightarrow p\} \end{array}$$

Selecting a faithful and effective representation for the syntax of an object language is crucial for a successful development. While there are many choices compatible with Coq (to wit [5,6] and see [7] for a critical discussion), we use *well-scoped* and *well-typed* DeBruijn terms [2] to encode expressions as the following inductive definition.

```
Inductive Exp (E:Env) :  Ty → Type :=
| VAR :  ∀ t,  Var E t → Exp E t
| LAM :  ∀ t1 t2,  Exp (t1 ::  E) t2 → Exp E (ARR t1 t2)
| APP :  ∀ t1 t2,  Exp E (ARR t1 t2) → Exp E t1 → Exp E t2 ...
```

In the above snippet of Coq code, for which we assume a passing familiarity, the type family `Exp` represents *intrinsically-typed* PCFL expressions of type `t` over a type environment `E`. In particular, a variable `VAR` is an expression for which there is proof `Var E t` that the nameless variable represented by the position of `t` belongs to `E`. We define lazy *evaluation* as a type-preserving inductive relation between closed expressions (programs). We show the clause for application, where `STmExp {| e2 |} e` denotes the substitution in `e` of `e2` for the nameless variable bound in `Lam e`, that is $e[e2/x]$:

```
Inductive Ev: ∀ t, Exp [] t →Exp [] t →Prop :=
| EvAPP : ∀ t1 t2 e v (e1 : Exp [] (ARR t1 t2)) e2,
    Ev e1 (LAM e) →Ev (STmExp {| e2 |} e) v →Ev (APP e1 e2) v
```

Relations The idea of type-preserving relations is so pervasive that we are lead to the development of a general theory of *dependent* relations: while the Coq library defines `relation (A : Type) := A →A →Prop`, we require `A` to depend on arbitrary indexes:

```
Definition Relations {E1 E2 : Type} (A:E1 →E2 →Type) :=
 ∀ (e1 : E1) (e2 : E2), A e1 e2 →A e1 e2 →Prop.
```

Consider relations over *expressions* `ExpRelations := Relations Exp`: the `E1`,`E2` indexes are fixed to `Env`,`Ty`: in `GrndRelations`, we restrict to programs and we will also talk about *substitutive* relations `Relations Sub`, which are dependent on *substitutions*, namely, functions from `Var E t` to `Exp E' t`. In fact, substitutive relations mediate between properties defined over programs, but which must be extended to open expressions, given the presence of variable binding constructors such as functions etc.

We develop a point-free algebra of dependent intrinsically-typed relations under composition (`CompRel` below), identity and inverse, from which we obtain the usual formulation of an equivalence relation.

```
Definition CompRel {E1 E2} {A:E1 →E2 →Type} (R R':Relations A) : Relations A :=
  fun e1 e2 e f ⇒∃g, R e1 e2 e g ∧R' e1 e2 g f.
```

Any reasonable program equivalence needs to be a congruence. A *congruence* is any `ExpRelation` that is symmetric, transitive and *compatible*, i.e., it respects the constructors of PCFL. Following Gordon [9], we define *contextual equivalence* as the greatest compatible and *adequate* relation, where a relation is adequate if the related terms either both diverge or converge to some value. Depending on the type at which we observe convergence, at any type or at certain *ground* types, we vary the coarseness of the equivalence, which in the latter case is called *observational*.

Similarity and Howe's method As we have mentioned, the above equivalences are hard to use in practice and we turn to *applicative* (bi)similarity. From now on, we discuss only similarity (and pre-orders), since equivalences can be achieved by symmetrization. In the presence of non-termination, the standard way to make sense of the circular definition of similarity (here shown for the function case) is *coinductively*:

```
CoInductive aSim : GrndRelations :=
| aSimARR : ∀ s t (e f:Exp [] (ARR s t)), (∀ e', Ev e (LAM e') →
∃ f', Ev f (LAM f') ∧∀ g, aSim (STmExp {|g|} e') (STmExp {|g|} f')) →aSim e f
```

Next, we extend it to *open* similarity (`OaSim`) via *grounding* substitutions. We do that abstractly as an operation $\mathcal{R}^\circ$ to lift ground relations, which, to increase readability, we describe in mathematical (rather than Coq's) notation: $E \vdash m \mathcal{R}^\circ_\tau m'$ iff $[\gamma] m \mathcal{R}_\tau [\gamma] m'$, for any grounding substitution $\cdot \vdash \gamma : E$, that is substitutions that take open expressions over `E` to programs.

While it is easy to show that (open) similarity is a pre-order, *substitutivity*, the stepping stone towards pre-congruence, does not hold. Howe's idea [10] was to introduce a *candidate* relation that contains open similarity and can be shown with relative ease to be a substitutive pre-congruence. Then one proves that it entails open similarity. Our proof follows quite closely the structure laid out in [18], with the obvious difference that the former is carried out in Beluga writing proof-terms manually, while this one relies on Coq's tactics. The biggest improvement, however, is that we can follow Howe's original account, by which the candidate relation does not refine only open similarity, but *any* pre-order which is also closed under substitution (CuS). Thanks to Coq's *type classes* [23], we can encode Howe's relation as a type preserving inductive family parameterized over any *CuS pre-order* — again, we show only a couple of cases:

```
Inductive Howe R `{CuS_Preorder R} : ∀ E t, Exp E t → Exp E t → Prop :=
| HVAR : ∀ E t (v: Var E t) e, R E t (VAR v) e → Howe (VAR v) e
| HLAM : ∀ E t1 t2 (e f: Exp (t1:: E) t2) g,
            Howe e f → R E (ARR t1 t2) (LAM f) g → Howe (LAM e) g
```

Thus, Howe's properties leading to pre-congruence are proven once and for all, e.g.,

```
Lemma Howe_comp_subs: ∀(R : ExpRelations) (H : CuS_Preorder R),
              Compatible Howe ∧ Substitutive Howe.
```

and can be recycled once we instantiate `R` to any flavor of similarity that happens to be a CuS pre-order. In our case-study, beside applicative, we have also studied the coarser notion of *ground similarity* [21], which differs in the function case:

```
CoInductive gSim : GrndRelations :=
 | gSimARR : ∀ (s t : Ty)(e f : Exp [] (ARR s t)),
             (∀ a : Exp [] s, gSim (APP e a) (APP f a)) → gSim e f
```

We follow the applicative path for exposition sake, the other being very similar modulo the function case. Once we have shown that the Howe's relation mimics the simulation conditions and it is *downward* closed w.r.t. evaluation, both rather labor-intensive lemmas, we can show the relational equality (notation `<:>`) of open similarity and Howe's:

```
Theorem aHowe_is_OaSim : aHowe <:> OaSim.
```

and have our main objective:

```
Corollary OaSim_precongruence : Precongruence OaSim.
```

Implementing the contextual pre-order If we want to use similarity in lieu of the contextual pre-order when proving programs equivalent, we need to show that they coincide. Recall that a PCFL's context $\mathcal{C}$ is an expression with a *hole* $\circ$, which can be thought as a logic variable inside the expression tree; *filling* a hole with a program m, denoted $\mathcal{C}\{m\}$, substitutes m for the hole, possibly capturing some of the free variables of m. We say that m and n are *contextually equivalent* iff for every closing $\mathcal{C}$, $\mathcal{C}\{m\}$ converges iff $\mathcal{C}\{n\}$ does. Again, we distinguish between applicative and observational contextual equivalence depending on the type at which we observe convergence (reflected in the definition of adequacy). The above definition is reputed to be "unpleasantly concrete" ([22]) and the literature [9] has suggested to use a "context-less" representation, namely as the largest compatible and adequate open relation. We have implemented

$$\begin{array}{ccc}
\text{CA} & \subseteq & \text{Observational CA} \\
\| & & \| \\
\text{Applicative Contextual Pre-order} & \subseteq & \text{Observational Contextual Pre-order} \\
\| & & \| \\
\text{Applicative Similarity} & \subseteq & \text{Ground Similarity}
\end{array}$$

Fig. 1. Equivalences and their relations.

both (context-less and context-full) and showed their equivalence, but here we report only the former, which is novel. Gordon's definition translates immediately in:

Definition `CA` : `ExpRelations` := **fun** `E t e f` ⇒
    ∃ `R`, `Compatible R` ∧ `OPreAdequate R` ∧ `R E t e f`.

This definition makes sense since we can show that `CA` is not empty. Then, while it is immediate that similarity is in `CA` (via Howe's result), the converse is more complicated. We have to show that the (ground restriction of) `CA` is a simulation, that is we have to reason by coinduction. On paper, this is straightforward thanks to previous lemmas, but a formal proof runs afoul of Coq's infamous *guardedness* check. Coq in fact implements a limited form of coinduction via guarded induction, realized by a very restrictive syntactic check that essentially prevents to apply any lemma inside a coinductive argument — curiously, Howe's proof did not violate it, suggesting that it is not a good benchmark for coinduction. A way out is offered by *parameterized* coinduction [12] (PACO), which replaces guarded induction with support for reasoning directly with greatest fixed points. To avoid starting from scratch, we show the coincidence of the coinductive and the PACO version of similarity and transfer all relevant lemmas from the former to the latter, a trick suggested in [8]. With this in place we (eventually) have:

Corollary `CA_is_OaSim` : `CA` <:> `OaSim`.

We summarize the relationships among equivalences in Fig. 1.

3 Conclusions

Our full development consists of around 110 theorems and 70 definitions for circa 2300 lines of code including sparse comments; the sources can be downloaded at `http://momigliano.di.unimi.it/FPEDT`. Although originated from a M.S. thesis, with both of the authors being Coq novices, we managed to go a little farther than the state of the art. We do not have the space for an exhaustive comparison, but the closest competitor is [16], which does not use Howe's method but logical relations.

As future work, we plan to extend our object language to a version of PCFL with arbitrary algebraic data types, by viewing object types *coinductively*, as suggested in [1]. We conjecture that our general algebraic setting should make this extension easy to accomplish. Another short-term goal is writing some tactics to make more automatic the verification of equivalence of concrete programs such as the ones listed in [21]. Finally, it would be interesting to see whether the complimentary phase of *validation* of specifications via property-based testing [19, 4] is applicable in this context.

References

1. A. Abel and A. Vezzosi. A formalized proof of strong normalization for guarded recursive types. In *APLAS*, volume 8858 of *LNCS*, pages 140–158. Springer, 2014.
2. N. Benton, C. Hur, A. Kennedy, and C. McBride. Strongly typed term representations in Coq. *J. Autom. Reasoning*, 49(2):141–159, 2012.
3. K. Chaudhuri, M. Cimini, and D. Miller. A lightweight formalization of the metatheory of bisimulation-up-to. In *CPP*, pages 157–166. ACM, 2015.
4. J. Cheney, A. Momigliano, and M. Pessina. Advances in property-based testing for αprolog. In *TAP*, volume 9762 of *LNCS*, pages 37–56. Springer, 2016.
5. A. Ciaffaglione and I. Scagnetto. Mechanizing type environments in weak HOAS. *Theor. Comput. Sci.*, 606:57–78, 2015.
6. A. P. Felty and A. Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *Journal of Automated Reasoning*, 48(1):43–105, 2012.
7. A. P. Felty, A. Momigliano, and B. Pientka. The next 700 challenge problems for reasoning with higher-order abstract syntax representations - part 2 - A survey. *J. Autom. Reasoning*, 55(4):307–372, 2015.
8. E. Giménez. Codifying guarded definitions with recursive schemes. In *TYPES*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 1994.
9. A. D. Gordon. Bisimilarity as a theory of functional programming. *Theor. Comput. Sci.*, 228(1-2):5–47, 1999.
10. D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.
11. C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. In *POPL '11*, pages 133–146, NY, USA, 2011. ACM.
12. C.-K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. The power of parameterization in coinductive proof. In *POPL '13*, pages 193–206, NY, USA, 2013. ACM.
13. S. B. Lassen. *Relational Reasoning about Functions and Nondeterminism.* PhD thesis, Dept of Computer Science, Univ of Aarhus, 1998.
14. S. Lenglet and A. Schmitt. Hoπ in Coq. In J. Andronick and A. P. Felty, editors, *CPP 2018*, pages 252–265. ACM, 2018.
15. J. Matthews and A. Ahmed. Parametric polymorphism through run-time sealing or, theorems for low, low prices! In *ESOP*, volume 4960 of *LNCS*, pages 16–31. Springer, 2008.
16. C. McLaughlin, J. McKinna, and I. Stark. Triangulating context lemmas. In J. Andronick and A. P. Felty, editors, *CPP 2018*, pages 102–114. ACM, 2018.
17. A. Momigliano. A supposedly fun thing I may have to do again: A HOAS encoding of Howe's method. In *LFMTP'12*, pages 33–42. ACM, 2012.
18. A. Momigliano, B. Pientka, and D. Thibodeau. A case study in programming coinductive proofs: Howe's method. *MSCS, in press*, 2018.
19. Z. Paraskevopoulou, C. Hritcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. In *ITP*, volume 9236 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2015.
20. M. Patrignani, A. Ahmed, and D. Clarke. Formal approaches to secure compilation. *ACM Comput. Surv.*, 51(6):125:1–125:36, Feb. 2019.
21. A. M. Pitts. Operationally Based Theories of Program Equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, 1997.
22. A. M. Pitts. Howe's method for higher-order languages. In D. Sangiorgi and J. Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, volume 52, chapter 5, pages 197–232. Cambridge University Press, Nov. 2011.
23. M. Sozeau and N. Oury. First-class type classes. In *TPHOLs*, volume 5170 of *LNCS*, pages 278–293. Springer, 2008.