Under consideration for publication in J. Functional Programming

POPLMark Reloaded: Mechanizing Proofs by Logical Relations

ANDREAS ABEL
Department of Computer Science and Engineering, Gothenburg University, Gothenburg, Sweden

GUILLAUME ALLAIS
iCIS, Radboud University, Nijmegen, Netherlands

ALIYA HAMEER and BRIGITTE PIENTKA
School of Computer Science, McGill University, Montreal, Canada

ALBERTO MOMIGLIANO
Department of Computer Science, Università degli Studi di Milano, Milan, Italy

STEVEN SCHÄFER and KATHRIN STARK
Saarland Informatics Campus, Saarland University, Saarland, Germany

Abstract

We propose a new collection of benchmark problems in mechanizing the metatheory of programming languages, in order to compare and push the state of the art of proof assistants. In particular, we focus on proofs using logical relations and propose establishing strong normalization of a simply-typed lambda-calculus with a proof by Kripke-style logical relations as a benchmark. We give a modern view of this well-understood problem by formulating our logical relation on well-typed terms. Using this case study, we share some of the lessons learned tackling this problem in different dependently-typed proof environments. In particular, we consider the mechanization in Beluga, a proof environment that supports higher-order abstract syntax encodings and contrast it to the development and strategies used in general purpose proof assistants such as Coq and Agda.The goal of this paper is to engage the community in discussions on what support in proof environments is needed to truly bring mechanized metatheory to the masses and engage said community in the crafting of future benchmarks.

1 Introduction

Programming languages (PL) design and implementations are often tricky to get right. To understand the subtle and complex interactions of language concepts and avoid flaws, we must rely on rigorous language definitions and on proofs so that we can ensure that languages are safe, robust, and trustworthy. However, writing such proofs by hand is very often complicated and error prone, not because of any particularly challenging insights involved, but simply because of the overhead required in keeping track of all the relevant

details and possible cases. These difficulties increase significantly when languages grow larger and proofs grow correspondingly longer; when working with a real-world programming language, carrying out such proofs by hand becomes unmanageable and in the end it is rarely done. In principle, proof assistants provide a way of formalizing programming languages definitions and writing machine-verified proofs about their properties, and they would seem a promising solution to this problem — and there is by now a ever-extending list of significant achievements in the area (Klein & Nipkow, 2006; Lee *et al.*, 2007; Leroy, 2009; Kumar *et al.*, 2014). Note that setting up the necessary infrastructure common to the design and verification of a PL model may incur in significant overhead, typically for modeling variables, substitutions, typing contexts, etc. In fact, with the exception of (Lee *et al.*, 2007), in all the cited papers, the authors went out of their way to avoid dealing with those issues and chose to work with "raw" terms, at the cost of losing α-equivalence and capture-avoiding substitutions, when not being pushed to use techniques such as closure-based operational semantics. This has arguably hindered the full adoption of this technology by the working semanticist, as the comments in (Rossberg *et al.*, 2014) further explain.[1] Over 10 years ago, Aydemir et al. (2005) proposed as a challenge the mechanization of part of the metatheory of $\mathtt{F}_{<:}$, which featured second-order polymorphism, subtyping, and records. In order to understand the state of the art in the field, the problems were chosen to emphasize aspects of programming languages that were known to be difficult to formalize: in particular, the challenge placed most of its focus on dealing with *binding* structures, such as functions, let-expressions, and pattern matching constructs. The challenge allowed the PL community to survey existing techniques for reasoning about variable bindings and popularized, to a certain extent, the use of proof assistants in their daily work (Pierce & Weirich, 2012).

We believe the time is ripe to revisit the question: How close are we to a world where every paper on programming languages is accompanied by an electronic appendix with machine checked proofs? To do this, we need to go well beyond the original POPLMark challenge. In this paper, we propose a new collection of benchmark problems in the mechanization of the metatheory of programming languages for comparing, understanding, and especially pushing the state of the art of proof assistants. In particular, we focus on proofs using *logical relations* (LR), an elegant and versatile proof technique with applications ubiquitous in programming languages theory. Initially introduced by Tait (1967), the technique has been applied to logical problems such as strong normalization (Girard, 1972), λ-definability (Plotkin, 1973), and the semantics of computational type theory (Werner, 1992; Geuvers, 1995). In recent years, the LR proof technique has become a staple for programming languages researchers (see, e.g., Ahmed (2013)). There is also a long tradition (almost a gold standard) in using proofs by logical relations to showcase the power of a given proof assistant; for early examples see (Altenkirch, 1993) and (Coquand, 1993).

The core idea of logical relations is straightforward: say we want to use LR for proving that all well-typed terms have a property P; then we define, by recursion on types, a relational model for those types. Thus, for instance, (well-typed) pairs consist of components

[1] Concerning the use of the *locally nameless* representation for binders: 'Out of a total of around 550 lemmas, approximately 400 were tedious "infrastructure" lemmas; only the remainder had direct relevance to the metatheory of F_ω or elaboration".

that are related pairwise, while (well-typed) terms at function type take logically related arguments to related results. Using the relational model for functions, e.g.,in the case of function application, so requires that (well-typed) logically related arguments indeed exist. While for unscoped terms, we can always assume a new, free variable, for scoped or well-typed syntax we have to guarantee that such a variable actually does exist. Here, *Kripke-style* LR (Mitchell & Moggi, 1991) come in play: we may think of the context in which an open term is meaningful as a *world* and then consider the term in all possible context/world extensions. In this extended context we can assume the existence of such a variable. Kripke-style logical relations hence provide a powerful proof technique that particularly applies to reasoning about *open* programs.

They also often play an essential role in analyzing the meta-theoretic properties of dependently typed systems (Urban *et al.*, 2011; Abel *et al.*, 2018).

In general LR proofs may be difficult to mechanize as they require a careful set-up for reasoning about open terms, modeling contexts of assumptions and context extensions, using recursive definitions, and reasoning about (simultaneous) substitutions. As Altenkirch (1993) remarked:

> *"I discovered that the core part of the proof (here proving lemmas about CR [reducibility candidates]) is fairly straightforward and only requires a good understanding of the paper version. However, in completing the proof I observed that in certain places I had to invest much more work than expected, e.g., proving lemmas about substitution and weakening."*

In practice, developing logical relation proofs for realistic languages can require an almost heroic mechanization effort in existing general-purpose proof assistants.

We propose the benchmark of establishing strong normalization of the simply-typed λ-calculus (STLC), extended to disjoint sums, with a proof by Kripke-style logical relations. We make three main contributions:

Contribution 1: A tutorial on strong normalization proof(s) using logical relations We give a modular tutorial proof of strong normalization for STLC via logical relations. Following Goguen (1995), we define reduction in a type-directed manner, as this simplifies the analysis and eventual formalization of the metatheory and it is a common technique found in dependently-typed systems.

We motivate our choice of this challenge in Section 2 before moving on to our exposition of the problem in Section 3. We start (Section 3.2) by relating the traditional approach of describing normalization using the *accessibility* relation, that is the well-founded part of the reduction relation, with the "modern" view of describing the set of (strongly) normalizing terms by rule induction (van Raamsdonk & Severi, 1995; Joachimski & Matthes, 2003). In Section 3.6, we prove strong normalization of STLC using the latter and then (Section 3.7) we extend the challenge to disjoint sums and briefly mention other extensions in Section 3.8. We omit most of the proof details from the main text, but we give the proofs of all lemmas and theorems in detail in the accompanying appendix.

Contribution 2: Challenge problem(s) These challenge problems seek to highlight various aspects that commonly arise when mechanizing PL semantics and its metatheory

in a proof assistant, only some of which have been covered by the original POPLMark challenge: What are the costs/benefits of a certain variable binding representation? Do we represent well-typed terms extrinsically, that is with a separate typing judgment, or intrinsically, exploiting some form of dependent types in the logical framework? Is there built-in support for (simultaneous) substitutions and their equational theory? If not, how do we support working with substitutions? How do we reason about contexts of assumptions and their structural properties, such as exchange, weakening, and strengthening? Is it convenient to use notions such as renaming and weakening substitutions to witness relations among contexts? What is the support for more sophisticated versions of induction, such as well-founded or lexicographic? How hard is it to represent well-founded (Noetherian) inductive definitions such as accessibility relations? Is it problematic to deal with recursive definitions that are not strictly positive, but are recursively defined based on the structure of types? The answers to these questions can help us to guide our mechanizations and avoid some possible pitfalls.

Contribution 3: Lessons learned We have tackled the challenge problem(s) in different *dependently-typed* proof environments using a variety of techniques: higher-order abstract syntax and first-class contexts, substitutions and renamings in Beluga (Section 4.1); well-typed de Bruijn encoding in Coq (Section 4.2); and well-typed de Bruijn encoding in Agda using the generic-syntax library of (Allais *et al.*, 2018) (Section 4.3). The formalizations can be browsed at `https://poplmark-reloaded.github.io` and downloaded from `https://github.com/poplmark-reloaded/poplmark-reloaded`.

In all these systems, we have mechanized the soundness of the inductive definition of strong normalization with respect to the one in terms of accessibility, and then the strong normalization proof for STLC. We have also completed the extension to disjoint sums. Just as a software testing suite only makes the grade if it finds bugs, a benchmark ought to stress the limitations of the technology underlying current proof assistants, and we comment for each mechanization on some of the critical lessons learned; in particular, we discuss how the benchmark highlighted shortcomings in each of the systems we have employed. Further, we summarize and compare how we deal with some of the key challenges in these systems (Section 5).

We hope that our benchmark will serve as a baseline for measuring the progress in developing proof environments and/or libraries, and that others will be motivated to submit solutions. At the same time, we want to emphasize that this benchmark is but one dimension along which we can evaluate and compare proof assistants, and there are many other aspects for which we should aim to craft additional challenges (especially for problems that deal with resources, state, coinduction, etc.). We will deem this paper a success if we manage to engage the community in discussing what support proof environments should provide to truly bring mechanized metatheory to the masses.

2 Motivation

A natural question one might ask upon reading the description of the benchmark is, why this specific problem? Certainly, starting a conversation about proof assistant support for mechanizing logical relations proofs is a worthy goal, given their recent ubiquity in pro-

gramming languages research; still, why restrict ourselves to an elementary property such as strong normalization, and on such a small calculus as the STLC? And why resort to Kripke LR for its normalization, when, after all, this can be established by more elementary syntactical means, for example using hereditary substitutions (Watkins *et al.*, 2002)?

We would like to stress that this problem is a *benchmark*, as opposed to a *grand* challenge (Hoare, 2003), with the aim of providing a common ground for comparison between current proof assistants while pushing the baseline further than the original POPLMark challenge. We consider this problem as a *first* step in evaluating the capabilities of different proof assistants for mechanizing proofs by logical relations; there is certainly potential for follow-up problems to be formulated by which to emphasize other categories of logical relations proofs and related issues. At the same time, strong normalization, even restricted to the STLC, is trickier than it looks; one can easily get it wrong, especially since there are no detailed textbook descriptions. With this in mind, we have chosen a challenge problem that satisfies two major criteria:

- *The problem should be easily accessible, while still being worthwhile.* In order to effectively engage the PL community, the challenge problem should be one that is easily understood and acted upon. It should be small enough in scale that it can be mechanized in a reasonable amount of time, and ideally it should be doable by a graduate student[2]. We would like the problem to be a suitable way to get acquainted with a particular proof assistant and with mechanizing a particular proof technique. Strong normalization of the lambda calculus, being a fundamental and widely-studied problem, fits all these criteria well, and keeping to the STLC keeps the problem size small while still exhibiting the technical challenges we wish to emphasize that arise in more complex proofs. This motivates also the use of a Kripke-style proof, since this a technique that cannot be avoided in other more advanced settings, such as equivalence checking (Crary, 2005) or more generally studying the metatheory of dependently typed systems (Abel *et al.*, 2018).
- *The problem should serve as a first step to survey the state of the art, compare proof assistants, and encourage development to make them more robust.* There are, of course, other problems that may meet (at least partially) our accessibility criterion: for instance proving parametricity properties for System F; or mechanizing properties of systems where *resources* play a significant role. And indeed these problems might serve well as follow-ups to this one in a benchmarks suite. For a first benchmark in mechanizing proofs by logical relations, however, we feel they are not as useful. The former requires impredicativity, which would limit the scope of our comparison; in fact, proof assistants such as Agda and Beluga do not support it, the former by design, the latter as dictated by its current foundations. Proofs over resources (as an example see the termination proof in (Ahmed *et al.*, 2007), where the presented system involves linear types and memory management) involves reasoning with structures such as *heaps*, and requires users to implement their own infrastructure, as existing systems do not typically provide any specific support for

[2] In fact, this is exactly what happened with Sebastian Sturm's master thesis (Sturm, 2018), where he gives a solution of the benchmark in *F**.

heaps. In this case, we would be comparing the user's cleverness in representing the relevant structures more than the features of the systems themselves, and this would provide a less useful basis, we believe, for comparison of the features of existing proof assistants. By choosing the problem of strong normalization, with contexts as Kripke worlds, we wish first and foremost to highlight aspects of mechanizing metatheory that have seen a lot of growth in the past several years, and for which there exists a variety of different approaches. We discuss possible future benchmarks that may build upon this one in Section 7.

This paper seeks to pose and hopefully answer the question: How far have we come since the original POPLMark challenge? In their proposal, Aydemir et al. identified four key features that they wished their challenge problem to emphasize: *binding*, *complex inductions*, *experimentation*, and *component reuse*. We focus here on pushing the baseline in the first two of these areas:

- *Binding.* A central issue in the POPLMark challenge was reasoning about variable binding. In the past decade, significant progress has been made in implementing libraries and tools for working with binders in general-purpose proof assistants such as Coq and Agda, (e.g., (Chlipala, 2008; Aydemir & Weirich, 2010; Felty & Momigliano, 2012; Lee *et al.*, 2012; Schäfer *et al.*, 2014)), exploring different foundations (see, e.g., (Licata & Harper, 2009; Pouillard & Pottier, 2010)), as well as extending the power of HOAS-based systems such as Abella and Beluga (e.g., (Cave & Pientka, 2012; Cave & Pientka, 2013; Wang *et al.*, 2013)). Our benchmark aims to see how these systems fare when presented with a greater variety of issues that occur in the presence of binders, beyond the mere syntactical representation of the binding structures: we refer here to context extensions, properties of contexts such as weakening and exchange, simultaneous substitutions and renamings. And in fact, the process of mechanizing our benchmark in Agda, Beluga, and Coq has led to progress in all three of the tested systems and libraries in this regard.
- *Complex inductions.* The POPLMark challenge focused on syntactic inductive proofs, in particular type soundness. In contrast, our benchmark involves the more challenging principles of lexicographic induction and induction that is stable under exchange of variables. The accessibility definition of strong normalization used in the first half of the challenge requires an inductive type with infinite branching. Finally, formalizing the logical relation involves a non-strictly positive, yet *stratified* definition, that is, a relation defined inductively on one of the indices.

Thus we feel that our choice of challenge problem, while seemingly small and simple, in fact emphasizes several issues that must be dealt with in order to mechanize the more complicated logical relations proofs that come up in practice. We seek not to directly advance the state of the art in PL theory, but instead to stimulate discussion between the communities for different proof assistants, to encourage sharing of ideas between different systems, and to make the core of our systems more robust.

$\boxed{\Gamma \vdash M : A}$ Term M has type A in the context Γ

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : A \Rightarrow B} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash M : A \Rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

$\boxed{\Gamma' \vdash \sigma : \Gamma}$ Substitution σ maps variables from Γ to terms in the context Γ'

$$\frac{}{\Gamma' \vdash \cdot : \cdot} \qquad \frac{\Gamma' \vdash \sigma : \Gamma \qquad \Gamma' \vdash M : A}{\Gamma' \vdash \sigma, M/x : \Gamma, x{:}A}$$

Fig. 1. Typing for STLC, and well-typed substitutions

3 Strong normalization for the λ-calculus

Proving (strong) normalization of the λ-calculus (STLC) using logical relations is a well-known result; however, modern tutorials on the subject are largely missing. There are two textbook resources one might refer to (Pierce, 2002) (Chapter 11) and (Girard *et al.*, 1989) (Chapter 6), neither of them being satisfactory for our purposes. Pierce (2002) only discusses weak normalization and on top of it he only considers reductions for closed terms, i.e., there is no evaluation inside a λ-abstraction. Girard et al. (1989) present a high-level account of proving strong normalization where many of the more subtle issues are skipped — in fact, it is over in less than five pages.

3.1 Simply typed λ-calculus with type-directed reductions

Our challenge problem is centered around the simply-typed λ calculus with a base type i. The grammar for the main syntactical entities are:

Terms	M, N	$::=$	$x \mid \lambda x{:}A.M \mid M\ N$
Types	A, B	$::=$	$A \Rightarrow B \mid \mathsf{i}$
Contexts	Γ	$::=$	$\cdot \mid \Gamma, x{:}A$
Substitutions	σ	$::=$	$\cdot \mid \sigma, M/x$

The typing rules for terms and substitutions are standard, but we include them here for completeness in Fig. 1. Note that only variables will inhabit the base type.

Since the notion of reducibility is Kripke-based, we will be extra careful about *contexts* and operations over them, such as weakening; we have learned the hard way how leaving such apparently trivial notions under-specified may come back to haunt us during the mechanization. We view a context as an ordered list without repetitions that grows on the right (as opposed to at arbitrary positions within the context). Although *renamings* ρ are merely substitutions that map variables to variables, we write $\Gamma' \leq_\rho \Gamma$ for well-typed renamings instead of $\Gamma' \vdash \rho : \Gamma$. In the case that ρ is an identity substitution, i.e., maps variables to themselves, Γ' is an extension of Γ modulo reordering. This is one motivation for our notation $\Gamma' \leq_\rho \Gamma$, the other being the analogy with record subtyping.

We define the application of the simultaneous substitution σ to a term M below and write $\sigma(x)$ to denote variable lookup. Note that inherent in this definition is that σ provides instantiations for all the free variables in M, as $\sigma(x)$ would otherwise not be defined.

Type-directed reduction : $\boxed{\Gamma \vdash M \longrightarrow N : A}$

$$\frac{\Gamma \vdash \lambda x{:}A.M : A \Rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x{:}A.M)\ N \longrightarrow [N/x]M : B}\ \text{S-}\beta$$

$$\frac{\Gamma \vdash M \longrightarrow M' : A \Rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN \longrightarrow M'N : B}\ \text{S-App-L} \qquad \frac{\Gamma \vdash M : A \Rightarrow B \quad \Gamma \vdash N \longrightarrow N' : A}{\Gamma \vdash MN \longrightarrow MN' : B}\ \text{S-App-R}$$

$$\frac{\Gamma, x{:}A \vdash M \longrightarrow M' : B}{\Gamma \vdash \lambda x{:}A.M \longrightarrow \lambda x{:}A.M' : A \Rightarrow B}\ \text{S-Abs}$$

Type-directed multi-step reduction : $\boxed{\Gamma \vdash M \longrightarrow^* N : A}$

$$\frac{}{\Gamma \vdash M \longrightarrow^* M : B}\ \text{M-Refl} \qquad \frac{\Gamma \vdash M \longrightarrow N : B \quad \Gamma \vdash N \longrightarrow^* M' : B}{\Gamma \vdash M \longrightarrow^* M' : B}\ \text{M-Trans}$$

Fig. 2. Type-directed reductions

$$\begin{array}{lcl} [\sigma](\lambda x{:}A.B) & = & \lambda x{:}A.[\sigma, x/x]M \\ [\sigma](M\ N) & = & [\sigma]M\ [\sigma]N \\ [\sigma](x) & = & \sigma(x) \end{array} \qquad \begin{array}{lcl} [\sigma](\cdot) & = & \cdot \\ [\sigma](\sigma', M/x) & = & [\sigma]\sigma',\ [\sigma]M/x \\ [\sigma]([\sigma_1]\sigma_2) & = & [[\sigma]\sigma_1]\sigma_2 \\ [N/x]([\sigma, x/x]M) & = & [\sigma, N/x]M \end{array}$$

Furthermore, we often silently exploit in this tutorial standard equational properties about well-typed simultaneous substitutions (which also naturally hold for renamings), such as composition and application. For convenience, we also use the single substitution operation $[N/x]M$, whose definition is the usual capture-avoiding one, and show how single and simultaneous substitution compose.

We describe single step reductions in Figure 2. Following (Goguen, 1995), we describe reductions in a type-directed manner. While types only play an essential role when we include rules such as η-expansion or in the presence of the unit type, which we do not consider here, we follow his approach as it scales to defining more complex notions such as type-directed equivalence (see also Sec. 3.8). There is another, more pedagogical advantage to the typed view: the scope of variables is made explicit, making them either bound by lambda-abstraction or in the typing context Γ. On top of the single step reduction relation, we define multi-step reductions.

Our benchmark relies on a number of basic properties about typing and typed reductions whose proofs are straightforward.

Lemma 3.1 (Reductions Preserve Typing)**.** *If* $\Gamma \vdash M \longrightarrow N : A$*, then* $\Gamma \vdash M : A$ *and* $\Gamma \vdash N : A$*.*

Proof. By induction on the given derivation. □

Lemma 3.2 (Weakening and Exchange for Typing and Typed Substitutions)**.**

- *If* $\Gamma, y{:}A, x{:}A' \vdash M : B$*, then* $\Gamma, x{:}A', y{:}A \vdash M : B$*.*
- *If* $\Gamma \vdash M : B$*, then* $\Gamma, x{:}A \vdash M : B$*.*
- *If* $\Gamma' \vdash \sigma : \Gamma$*, then* $\Gamma', x{:}A \vdash \sigma : \Gamma$*.*

Proof. By induction on the given derivation; the second property relies on the first.

□

Corollary 3.1 (Weakening of Renamings)**.** *If* $\Gamma' \leq_\rho \Gamma$*, then* $\Gamma', x{:}A \leq_\rho \Gamma$.

Lemma 3.3 (Anti-Renaming of Typing)**.** *If* $\Gamma' \vdash [\rho]M : A$ *and* $\Gamma' \leq_\rho \Gamma$*, then* $\Gamma \vdash M : A$.

Proof. By induction on the given typing derivation taking into account the equational theory of substitutions. □

Lemma 3.4 (Weakening and Exchange of Typed Reductions)**.**

- *If* $\Gamma \vdash M \longrightarrow N : B$*, then* $\Gamma, x{:}A \vdash M \longrightarrow N : B$.
- *If* $\Gamma, y{:}A, x{:}A' \vdash M \longrightarrow N : B$*, then* $\Gamma, x{:}A', y{:}A \vdash M \longrightarrow N : B$.

Proof. By mutual induction on the given derivation. □

Lemma 3.5 (Substitution Property of Typed Reductions)**.** *If* $\Gamma, x{:}A \vdash M \longrightarrow M' : B$ *and* $\Gamma \vdash N : A$*, then* $\Gamma \vdash [N/x]M \longrightarrow [N/x]M' : B$.

Proof. By induction on the first derivation, using the usual properties of composition of substitutions as well as weakening and exchange. □

We will also rely on some standard properties about multi-step reduction.

Lemma 3.6 (Properties of Multi-Step Reductions)**.**

1. *If* $\Gamma \vdash M_1 \longrightarrow^* M_2 : B$ *and* $\Gamma \vdash M_2 \longrightarrow^* M_3 : B$*, then* $\Gamma \vdash M_1 \longrightarrow^* M_3 : B$.
2. *If* $\Gamma \vdash M \longrightarrow^* M' : A \Rightarrow B$ *and* $\Gamma \vdash N : A$*, then* $\Gamma \vdash M\ N \longrightarrow^* M'\ N : B$.
3. *If* $\Gamma \vdash M : A \Rightarrow B$ *and* $\Gamma \vdash N \longrightarrow^* N' : A$*, then* $\Gamma \vdash M\ N \longrightarrow^* M\ N' : B$.
4. *If* $\Gamma, x{:}A \vdash M \longrightarrow^* M' : B$*, then* $\Gamma \vdash \lambda x{:}A.M \longrightarrow^* \lambda x{:}A.M' : A \Rightarrow B$.
5. *If* $\Gamma, x{:}A \vdash M : B$ *and* $\Gamma \vdash N \longrightarrow N' : A$*, then* $\Gamma \vdash [N/x]M \longrightarrow^* [N'/x]M : B$.

Proof. Properties 1, 2, 3, and 4 are proven by induction on the given multi-step relation. Property 5 is proven by induction on $\Gamma, x{:}A \vdash M : B$ using weakening and exchange (Lemma 3.4). □

Remark One may choose to formulate the weakening and substitution properties using simultaneous substitutions—and in fact, we will need those properties in the subsequent development.

Lemma 3.7 (Closure under Simultaneous Substitution and Renaming)**.** *Let* $\Gamma \vdash M \longrightarrow N : A$.

1. *If* $\Gamma' \vdash \sigma : \Gamma$*, then* $\Gamma' \vdash [\sigma]M \longrightarrow [\sigma]N : A$.
2. *If* $\Gamma' \leq_\rho \Gamma$*, then* $\Gamma' \vdash [\rho]M \longrightarrow [\rho]N : A$.

3.2 Defining strong normalization

Classically, a term M is strongly normalizing if there are no infinite reduction sequences from M. Constructively, we can define strong normalization $\Gamma \vdash M : A \in \mathsf{sn}$ as an *accessibility* relation:

$$\frac{\forall N.\ \Gamma \vdash M \longrightarrow N : A \Longrightarrow \Gamma \vdash N : A \in \mathsf{sn}}{\Gamma \vdash M : A \in \mathsf{sn}}$$

Since we follow a type-directed reduction strategy, we have added typing information to accessibility.

We can unwind $\Gamma \vdash M : A \in \mathsf{sn}$ along a multi-step reduction:

Strongly normalizing neutral terms : $\boxed{\Gamma \vdash M : A \in \mathsf{SNe}}$

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash x : A \in \mathsf{SNe}} \qquad \frac{\Gamma \vdash R : A \Rightarrow B \in \mathsf{SNe} \quad \Gamma \vdash M : A \in \mathsf{SN}}{\Gamma \vdash RM : B \in \mathsf{SNe}}$$

Strongly normalizing terms : $\boxed{\Gamma \vdash M : A \in \mathsf{SN}}$

$$\frac{\Gamma, x{:}A \vdash M : B \in \mathsf{SN}}{\Gamma \vdash \lambda x{:}A.M : A \Rightarrow B \in \mathsf{SN}} \qquad \frac{\Gamma \vdash R : A \in \mathsf{SNe}}{\Gamma \vdash R : A \in \mathsf{SN}} \qquad \frac{\Gamma \vdash M \longrightarrow_{\mathsf{SN}} M' : A \qquad \Gamma \vdash M' : A \in \mathsf{SN}}{\Gamma \vdash M : A \in \mathsf{SN}}$$

Strong head reduction : $\boxed{\Gamma \vdash M \longrightarrow_{\mathsf{SN}} N : A}$

$$\frac{\Gamma \vdash N : A \in \mathsf{SN} \quad \Gamma, x{:}A \vdash M : B}{\Gamma \vdash (\lambda x{:}A.M)\, N \longrightarrow_{\mathsf{SN}} [N/x]M : B} \qquad \frac{\Gamma \vdash M \longrightarrow_{\mathsf{SN}} M' : A \Rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN \longrightarrow_{\mathsf{SN}} M'N}$$

Fig. 3. Inductive definition of Strong Normalization

Lemma 3.8 (Multi-step Strong Normalization)**.** *If* $\Gamma \vdash M \longrightarrow^* M' : A$ *and* $\Gamma \vdash M : A \in \mathsf{sn}$*, then* $\Gamma \vdash M' : A \in \mathsf{sn}$.

Proof. Induction on $\Gamma \vdash M \longrightarrow^* M' : A$. □

While the above definition is appealing in its simplicity and has been widely used (see for example (Altenkirch, 1993)), it is not without defects: it involves reasoning about reduction sequences and positions of terms, so much that "the reduct analysis becomes increasingly annoying in normalization proofs for more and more complex systems" (Joachimski & Matthes, 2003). This applies both to pen-and-paper and mechanized proofs.

As pioneered in (van Raamsdonk & Severi, 1995) and further developed in (Joachimski & Matthes, 2003), we can easily characterize strongly normalizing terms inductively, building on the standard definitions of normal and neutral terms. Raamsdonk and Severi (1995) observed that a term M is in the set of (strongly) normalizing terms, if either it is a normal form or it can be obtained as the result of a series of expansions starting from a normal form. However, to preserve strong normalization during expansions, we have to be careful about terms "lost" via substitution for a variable with no occurrence. In particular, strong normalization of $[N/x]M$ does not imply strong normalization of $(\lambda x{:}A.M)\, N$ or of N, when x does not occur in M. This motivates the concept of *strong head reduction*, which is a weak head reduction $(\lambda x{:}A.M)\, N N_1 \ldots N_n \longrightarrow [N/x]M N_1 \ldots N_n$ with the extra condition that N is strongly normalizing. Our inductive definition given in Fig. 3 follows (van Raamsdonk & Severi, 1995), augmented to track typing information.

Many proofs, not only those involving normalization, become simpler thanks to the inductive definition, since it allows us to prove properties by structural induction: we reduce the task of checking all one-step reducts to analyzing no more than one standard reduct and some subterms. The reduct analysis is localized to the proof that the inductive notion of normalization entails the accessibility version, a property we refer to as "soundness" (Theorem 3.1).

3.3 Challenge 1a: Properties of sn

To establish soundness, we start with the following:

Lemma 3.9 (Properties of Strongly Normalizing Terms)**.**

1. $\Gamma \vdash x : A \in \mathsf{sn}$ *for all variables* $x : A \in \Gamma$.
2. *If* $\Gamma, x{:}A \vdash M : B \in \mathsf{sn}$, *then* $\Gamma \vdash \lambda x{:}A.M : A \Rightarrow B \in \mathsf{sn}$.
3. *If* $\Gamma \vdash [N/x]M : B \in \mathsf{sn}$ *and* $\Gamma \vdash N : A$, *then* $\Gamma, x{:}A \vdash M : B \in \mathsf{sn}$.
4. *If* $\Gamma \vdash M\ N : B \in \mathsf{sn}$, *then* $\Gamma \vdash M : A \Rightarrow B \in \mathsf{sn}$ *and* $\Gamma \vdash N : A \in \mathsf{sn}$.

Proof. The first property is immediate, as there are no reductions from variables. Properties (2), (3), and (4) are proven by induction on the given derivation. □

Lemma 3.10 (Weak Head Expansion)**.** *If* $\Gamma \vdash N : A \in \mathsf{sn}$ *and* $\Gamma \vdash [N/x]M : B \in \mathsf{sn}$, *then* $\Gamma \vdash (\lambda x{:}A.M)\ N : B \in \mathsf{sn}$.

Proof. By lexicographic induction on $\Gamma \vdash N : A \in \mathsf{sn}$ and $\Gamma, x{:}A \vdash M : B \in \mathsf{sn}$ (which is entailed by $\Gamma \vdash [N/x]M : B \in \mathsf{sn}$ due to Lemma 3.9), analysing the possible reductions of $(\lambda x{:}A.M)\ N$. □

Remark We might again generalize Properties 3.9 (3) and 3.10 in terms of simultaneous substitutions.

To make the proof of soundness more modular, it is useful to characterize terms that are "blocked" by a variable, i.e., that are of the form $x\ M_1 \ldots M_n$. Those terms will not trigger a weak head reduction and correspond intuitively to those characterized by SNe. We call these terms *neutral*, following (Altenkirch *et al.*, 1995); in the normalization proof, they play a similar role as Girard's neutrals (Girard *et al.*, 1989). They are defined by the judgment $\Gamma \vdash R : A\ \mathsf{ne}$:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A\ \mathsf{ne}} \qquad \frac{\Gamma \vdash R : A \Rightarrow B\ \mathsf{ne} \qquad \Gamma \vdash N : A}{\Gamma \vdash R\ N : B\ \mathsf{ne}}$$

Now, we can show that neutral terms are forward closed and that if R is neutral and strongly normalizing together with N being strongly normalizing, then $R\ N$ is strongly normalizing.

Lemma 3.11 (Closure Properties of Neutral Terms)**.**

1. *If* $\Gamma \vdash R : A\ \mathsf{ne}$ *and* $\Gamma \vdash R \longrightarrow R' : A$, *then* $\Gamma \vdash R' : A\ \mathsf{ne}$.
2. *If* $\Gamma \vdash R : A \Rightarrow B\ \mathsf{ne}$ *and* $\Gamma \vdash R : A \Rightarrow B \in \mathsf{sn}$, *and* $\Gamma \vdash N : A \in \mathsf{sn}$, *then* $\Gamma \vdash R\ N : B \in \mathsf{sn}$.

Proof. Property (1) follows by induction on $\Gamma \vdash R : A\ \mathsf{ne}$. Property (2) is proven by lexicographic induction on $\Gamma \vdash R : A \Rightarrow B \in \mathsf{sn}$ and $\Gamma \vdash N : A \in \mathsf{sn}$, using Property (1). □

We further introduce the notion of sn-*strong head reduction* $\longrightarrow_{\mathsf{sn}}$, which is analogous to (SN-)strong head reduction $\longrightarrow_{\mathsf{SN}}$.

$$\frac{\Gamma \vdash N : A \in \mathsf{sn} \quad \Gamma, x{:}A \vdash M : B}{\Gamma \vdash (\lambda x{:}A.M)\ N \longrightarrow_{\mathsf{sn}} [N/x]M : B} \qquad \frac{\Gamma \vdash M \longrightarrow_{\mathsf{sn}} M' : A \Rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N \longrightarrow_{\mathsf{sn}} M'\ N : B}$$

Lemma 3.12 (Confluence of sn)**.** *If* $\Gamma \vdash M \longrightarrow_{\mathsf{sn}} N : A$ *and* $\Gamma \vdash M \longrightarrow N' : A$ *then either* $N = N'$ *or there* $\exists Q$ *s.t.* $\Gamma \vdash N' \longrightarrow_{\mathsf{sn}} Q : A$ *and* $\Gamma \vdash N \longrightarrow^* Q : A$.

Proof. By induction on $\Gamma \vdash M \longrightarrow_{\mathsf{sn}} N : A$. □

Lemma 3.13 (Backward Closure of sn)**.**

1. *If* $\Gamma \vdash N : A \in \mathsf{sn}$, $\Gamma \vdash M : A \Rightarrow B \in \mathsf{sn}$, $\Gamma \vdash M \longrightarrow_{\mathsf{sn}} M' : A \Rightarrow B$ *and* $\Gamma \vdash M'\ N : B \in \mathsf{sn}$, *then* $\Gamma \vdash M\ N : B \in \mathsf{sn}$.
2. *If* $\Gamma \vdash M \longrightarrow_{\mathsf{sn}} M' : A$ *and* $\Gamma \vdash M' : A \in \mathsf{sn}$, *then* $\Gamma \vdash M : A \in \mathsf{sn}$.

Proof. Property (1) is proven by lexicographic induction on $\Gamma \vdash N : A \in \mathsf{sn}$ and $\Gamma \vdash M : A \Rightarrow B \in \mathsf{sn}$. Property (2) is proven by induction on $\Gamma \vdash M \longrightarrow_{\mathsf{sn}} M' : A$, using Property (1) in the inductive case.

We here sketch the use of the Confluence lemma and lexicographic induction in the proof of property (1); the full proof can be found in the electronic appendix.

To show that $\Gamma \vdash M\ N : B \in \mathsf{sn}$, we assume $\Gamma \vdash M\ N \longrightarrow Q : B$, and establish that $\Gamma \vdash Q : B \in \mathsf{sn}$. By case analysis on $\Gamma \vdash M\ N \longrightarrow Q : B$, two cases remain:

1. If $\Gamma \vdash N \longrightarrow N' : A$ and $Q = M\ N'$, the claim follows directly with the inductive hypothesis for $\Gamma \vdash N' : A \in \mathsf{sn}$.
2. If $\Gamma \vdash M \longrightarrow M'' : A \Rightarrow B$ and $Q = M''\ N$, we require confluence to establish any relation between M' and M'': Either $M' = M''$, in which case the goal follows directly from our assumption that $\Gamma \vdash M'\ N : B \in \mathsf{sn}$; or M' and M'' converge to the same term P, where we can establish $\Gamma \vdash P\ N : B \in \mathsf{sn}$ by Lemmas 3.6(2) and 3.8. Then we require the inductive hypothesis for $\Gamma \vdash M'' : A \Rightarrow B \in \mathsf{sn}$ to establish $\Gamma \vdash M''\ N : B \in \mathsf{sn}$.

Note how a mere structural induction is not enough: the lexicographic induction is necessary, to later have the particular inductive hypothesis available. □

Challenging aspects Proving confluence about sn (Lemma 3.12, sometimes referred to as the *weak standardization* lemma), requires a detailed and tedious case analysis and has previously not been stated as clearly.[3] The introduction of sn-strong head reduction $\longrightarrow_{\mathsf{sn}}$, which mirrors strong head reduction $\longrightarrow_{\mathsf{SN}}$, helps structure and simplify the proofs. However, the proofs about backward closure of sn require a detailed analysis of the reduction relation, for example, in Property (1) of Lemma 3.13. This analysis can be quite tricky as we rely on simultaneous well-founded induction.

3.4 Challenge 1b: Soundness of inductive definition of strongly normalizing terms

We now have all the properties about sn that are necessary to establish soundness save for the simple property that all terms inductively characterized by SNe are neutral.

Lemma 3.14. *If* $\Gamma \vdash M : A \in \mathsf{SNe}$, *then* $\Gamma \vdash M : A$ ne.

[3] For example, Abel and Vezzosi (2014) state a variant of this lemma that makes some unnecessary assumptions about strong normalization.

Proof. By induction on $\Gamma \vdash M : A \in \mathsf{SNe}$. □

Theorem 3.1 (Soundness of SN)**.**

1. *If* $\Gamma \vdash M : A \in \mathit{SN}$*, then* $\Gamma \vdash M : A \in \mathsf{sn}$.
2. *If* $\Gamma \vdash M : A \in \mathit{SNe}$*, then* $\Gamma \vdash M : A \in \mathsf{sn}$.
3. *If* $\Gamma \vdash M \longrightarrow_{\mathit{SN}} M' : A$*, then* $\Gamma \vdash M \longrightarrow_{\mathsf{sn}} M' : A$.

Proof. By mutual structural induction on the given derivations using the closure properties (Lemmas 3.9 (1), 3.9 (2), 3.11 (2), 3.13 (2), 3.14). □

Challenging aspects Given our set-up, the final soundness proof is straightforward. This is largely due to the fact that we have factored out and defined the $\longrightarrow_{\mathsf{sn}}$ relation in analogy to the $\longrightarrow_{\mathsf{SN}}$ relation.

Additional twist One might want to consider a formulation of neutral terms that relies on *evaluation contexts* instead of a predicate. Using evaluation contexts is elegant and might be advantageous as the language grows. Evaluation contexts are defined inductively as either a hole or the application of an evaluation context C to a term M.

$$\text{Evaluation Contexts } C \quad ::= \quad \_ \mid C\ M$$

Definition 3.2 (Typing Rules for Evaluation Contexts)**.** *We write* $\Gamma \mid A_0 \vdash C : A$ *to state that the evaluation context* C *lives in scope* Γ*, has a hole of type* A_0 *and gives rise to an overall expression of type* A*. This relation is specified by the two following typing rules:*

$$\frac{}{\Gamma \mid A_0 \vdash \_ : A_0} \qquad \frac{\Gamma \mid A_0 \vdash C : A \Rightarrow B \quad \Gamma \vdash M : A}{\Gamma \mid A_0 \vdash C\,M : B}$$

We extend the notion of reduction as usual and of sn (respectively SN) to evaluation contexts by demanding that all the terms appearing in such a context are sn (respectively SN). For that, we need the usual notion of *filling* a hole.

Lemma 3.15 (Properties of Evaluation Contexts)**.**

1. *If* $\Gamma \mid A_0 \vdash C : A$ *and* $\Gamma \vdash M : A_0$*, then* $\Gamma \vdash C[M] : A$.
2. *If* $\Gamma \mid A_0 \vdash C : A \in \mathsf{sn}$ *and* $x{:}A_0 \in \Gamma$*, then* $\Gamma \vdash C[x] : A \in \mathsf{sn}$.
3. *If* $\Gamma \vdash M : A \in \mathit{SNe}$*, then there exists* A_0*,* C *and* $x{:}A_0 \in \Gamma$ *such that* $M = C[x]$ *and* $\Gamma \mid A_0 \vdash C : A \in \mathit{SN}$.

Proof. (1), (2) by induction on C. (3) by induction on $\Gamma \vdash M : A \in \mathsf{SNe}$. □

Lemma 3.16 (Closure Properties of Strongly Normalizing Evaluation Contexts)**.**

1. *If* $\Gamma \mid A_0 \vdash C : B \in \mathsf{sn}$*,* M *is not an abstraction, and* $\Gamma \vdash C[M] \longrightarrow N : B$*, then*
   either $\exists M'$ *s.t.* $N = C[M']$ *and* $\Gamma \vdash M \longrightarrow M' : A_0$
   or $\exists C'$ *s.t.* $N = C'[M]$ *and* $\forall M''.\ \Gamma \vdash C[M''] \longrightarrow C'[M''] : B$.
2. *If* $\Gamma, x{:}A \vdash M : A_0 \in \mathsf{sn}$*,* $\Gamma \vdash N : A \in \mathsf{sn}$*,* $\Gamma \mid A_0 \vdash C : B \in \mathsf{sn}$*,* $\Gamma \vdash C[\,[N/x]M\,] : B \in \mathsf{sn}$ *then* $\Gamma \vdash C[\,(\lambda x{:}A.\ M)\,N\,] : B \in \mathsf{sn}$.
3. *If* $\Gamma \vdash M \longrightarrow_{\mathsf{sn}} M' : A_0$ *and* $\Gamma \vdash C[M'] : A \in \mathsf{sn}$ *then* $\Gamma \vdash C[M] : A \in \mathsf{sn}$.

Proof. (1) by induction on C and case analysis on the reduction step. (2) is proven by lexicographic induction on the sn assumptions, by assuming $C[\ (\lambda x{:}A.M)N\ ]$ steps to a reduct and using (1) to decide whether the reduction happens in C, M, N or $((\lambda x{:}A.M)N)$. Finally (3) is proven by induction on the reduction step either using (2) or growing the evaluation context. □

Theorem 3.2 (Soundness of SN using Evaluation Contexts)**.**

1. *If* $\Gamma \vdash M : A \in \mathit{SN}$*, then* $\Gamma \vdash M : A \in \mathsf{sn}$.
2. *If* $\Gamma \mid A_0 \vdash C : A \in \mathit{SN}$*, then* $\Gamma \mid A_0 \vdash C : A \in \mathsf{sn}$.
3. *If* $\Gamma \vdash M \longrightarrow_{\mathit{SN}} M' : A$*, then* $\Gamma \vdash M \longrightarrow_{\mathsf{sn}} M' : A$.

Proof. By mutual induction on the respective derivations using 3.15 (3, 1) to deal with the SNe case 3.16 (3) to deal with the $\longrightarrow_{\mathsf{SN}}$ case. □

3.5 Challenge 2a: Properties of strong normalization

Before we describe the main proof of strong normalization, we need to establish a number of properties about the inductive definition of SN.

In particular, we express two key properties about context extensions over strongly normalizing terms in terms of *renaming* substitutions as witnesses for the context extension. Lemma 3.17 states that we can transport any well-typed term that is strongly normalizing in the context Γ to the extended context Γ'. The second property is the dual and states that given a well-typed term $[\rho]M$ in a context extension Γ' we can recover M in the context Γ.

Lemma 3.17 (Renaming)**.**

1. *If* $\Gamma \vdash M : A \in \mathit{SN}$ *and* $\Gamma' \leq_\rho \Gamma$*, then* $\Gamma' \vdash [\rho]M : A \in \mathit{SN}$.
2. *If* $\Gamma \vdash M : A \in \mathit{SNe}$ *and* $\Gamma' \leq_\rho \Gamma$*, then* $\Gamma' \vdash [\rho]M : A \in \mathit{SNe}$.
3. *If* $\Gamma \vdash M \longrightarrow_{\mathit{SN}} N : A$ *and* $\Gamma' \leq_\rho \Gamma$*, then* $\Gamma' \vdash [\rho]M \longrightarrow_{\mathit{SN}} [\rho]N : A$.

Proof. By mutual induction on the given derivations using the Weakening Lemma 3.2.

□

Lemma 3.18 (Anti-Renaming)**.**

1. *If* $\Gamma' \vdash [\rho]M : A \in \mathit{SN}$ *and* $\Gamma' \leq_\rho \Gamma$*, then* $\Gamma \vdash M : A \in \mathit{SN}$
2. *If* $\Gamma' \vdash [\rho]M : A \in \mathit{SNe}$ *and* $\Gamma' \leq_\rho \Gamma$*, then* $\Gamma \vdash M : A \in \mathit{SNe}$
3. *If* $\Gamma' \vdash [\rho]M \longrightarrow_{\mathit{SN}} N' : A$ *and* $\Gamma' \leq_\rho \Gamma$*, then there exists* N *s.t.* $\Gamma \vdash M \longrightarrow_{\mathit{SN}} N : A$ *and* $[\rho]N = N'$.

Proof. By mutual induction on the given derivations, in each case exploiting that ρ is a renaming. For example, we might encounter that $\Gamma' \vdash [\rho]M : A \in \mathsf{SNe}$ via the application rule. Since $[\rho]M$ is an application, we need to conclude that M is also an application to proceed with the proof.

The proof moreover takes into account several equational properties of substitutions:

Consider for example the case of term abstraction, where we have the premise

$$\Gamma' \vdash \lambda x{:}A.[\rho, x/x]M : A \Rightarrow B \in \mathsf{SN}$$

and require weakening (Lemma 3.1) to apply the inductive hypothesis.

Next, in the case of beta reduction for strong head reduction,we have the premise

$$\Gamma' \vdash [\rho]((\lambda x{:}A.M)\ N) \longrightarrow_{\mathsf{SN}} [\rho, [\rho]N/x]M : B$$

and have to handle composition of renamings and substitutions appropriately to proceed.

Full proof details can be found in the appendix. □

This proof challenges support for binders in various ways: For one, it requires the systems to have an appropriate notion of renamings — the claim is not true for full substitutions. In addition, the proof requires convenient handling of equations for both renamings and substitutions.

Lastly, we need extensionality of SN for function types, which will be crucial in the proof of Girard's property CR1:

Lemma 3.19 (Extensionality of SN)**.** *If $x{:}A \in \Gamma$ and $\Gamma \vdash M\ x : B \in SN$, then $\Gamma \vdash M : A \Rightarrow B \in SN$.*

Proof. By induction on $\Gamma \vdash M\ x : B \in \mathsf{SN}$. □

Remark One can also define extensionality of SN as:

If $\Gamma \vdash M : A \Rightarrow B$ and $\Gamma, x{:}A \leq_\rho \Gamma$ and $\Gamma, x{:}A \vdash [\rho]M\ x : B \in \mathsf{SN}$, then $\Gamma \vdash M : A \Rightarrow B \in \mathsf{SN}$.

The proof of this formulation of extensionality is similar, but relies more heavily on the anti-renaming lemma.

Challenging aspects The main issue here is doing case analysis modulo the equational theory of renamings in the proof for Lemma 3.18 when considering all possible cases for $\Gamma' \vdash [\rho]M : A \in \mathsf{SN}$.

3.6 Challenge 2b: Proving strong normalization with logical relations

We now turn to the definition of our logical predicate. Since we are working with well-typed terms, we define the semantic interpretation of $\Gamma \vdash M \in \mathscr{R}_{A \Rightarrow B}$ considering all extensions of Γ (described by $\Gamma' \leq_\rho \Gamma$) in which we may use M.

- $\Gamma \vdash M \in \mathscr{R}_{\mathsf{i}}$ iff $\Gamma \vdash M : \mathsf{i} \in \mathsf{SN}$
- $\Gamma \vdash M \in \mathscr{R}_{A \Rightarrow B}$ iff for all $\Gamma' \leq_\rho \Gamma$ such that $\Gamma' \vdash N : A$, if $\Gamma' \vdash N \in \mathscr{R}_A$, then $\Gamma' \vdash ([\rho]M)\ N \in \mathscr{R}_B$.

Rather than attempt the proof of the main result directly and then extract additional lemmas one might need about the semantic types, we follow Girard's technique and characterize some key properties that our logical predicate needs to satisfy.

Theorem 3.3.

1. *CR1: If $\Gamma \vdash M \in \mathscr{R}_A$, then $\Gamma \vdash M : A \in SN$.*
2. *CR2: If $\Gamma \vdash M \longrightarrow_{SN} M' : A$ and $\Gamma \vdash M' \in \mathscr{R}_A$, then $\Gamma \vdash M \in \mathscr{R}_A$, i.e., backward closure.*
3. *CR3: If $\Gamma \vdash M : A \in SNe$, then $\Gamma \vdash M \in \mathscr{R}_A$.*

Proof. The properties are proven by mutual induction on the structure of A. □

We prove that if a term is well-typed, then it is strongly normalizing in two steps:

Step 1 If $\Gamma \vdash M \in \mathscr{R}_A$, then $\Gamma \vdash M : A \in \mathsf{SN}$.
Step 2 If $\Gamma \vdash M : A$ and $\Gamma' \vdash \sigma \in \mathscr{R}_\Gamma$, then $\Gamma' \vdash [\sigma]M \in \mathscr{R}_A$.

Step 1 is satisfied by the fact that by CR 1 all terms in $\mathscr{R}_A$ are strongly normalizing. We now prove the second step, which is often referred to as the *Fundamental Lemma*. It states that if M has type A and we can provide "good" instantiation σ, yielding terms which are themselves normalizing for all the free variables in M, then $\Gamma \vdash [\sigma]M \in \mathscr{R}_A$. The definition of such instantiations σ is as follows:

Definition 3.3 (Semantic Substitutions)**.**

$$\frac{}{\Gamma' \vdash \cdot \in \mathscr{R}_\cdot} \qquad \frac{\Gamma' \vdash \sigma \in \mathscr{R}_\Gamma \qquad \Gamma' \vdash M \in \mathscr{R}_A}{\Gamma' \vdash \sigma, M/x \in \mathscr{R}_{\Gamma,x:A}}$$

Lemma 3.20 (Fundamental lemma)**.** *If* $\Gamma \vdash M : A$ *and* $\Gamma' \vdash \sigma \in \mathscr{R}_\Gamma$ *then* $\Gamma' \vdash [\sigma]M \in \mathscr{R}_A$.

Proof. By induction on $\Gamma \vdash M : A$ using the renaming lemma for $\Gamma' \vdash \sigma \in \mathscr{R}_\Gamma$. □

Corollary 3.4. *If* $\Gamma \vdash M : A$*, then* $\Gamma \vdash M : A \in \mathsf{SN}$*.*

Proof. Using the fundamental lemma with the identity substitution $\Gamma \vdash \mathsf{id} \in \mathscr{R}_\Gamma$, we obtain $\Gamma \vdash M \in \mathscr{R}_A$. By CR1, we know $\Gamma \vdash M \in \mathsf{SN}$. □

Challenging aspects The definition of the logical predicate is not strictly positive, but is well-founded as it is defined by recursion on the structure of the type. The proofs of CR1 through CR3 rely extensively on reasoning about context extensions and in turn about renamings. Similarly, the fundamental lemma requires a good handling of simultaneous substitutions.

3.7 Extension: Disjoint sums

How well do our proof method and formal mechanization cope with language extensions? In this section, we augment our language of terms with the constructors and eliminators for disjoint sums:

$$\begin{array}{llll} \text{Terms} & M,N & ::= & \cdots \mid \mathsf{inl}\, M \mid \mathsf{inr}\, M \mid \mathsf{case}\, M\, \mathsf{of}\, \mathsf{inl}\, x \Rightarrow N_1 \mid \mathsf{inr}\, y \Rightarrow N_2 \\ \text{Types} & A,B & ::= & \cdots \mid A + B \end{array}$$

The new reduction rules are given in Figure 4. For reasons of brevity, the congruence rules for inl, inr and case are omitted; the full set of reduction rules can be found in the Appendix. We also extend our definitions of SN, SNe, and $\longrightarrow_{\mathsf{SN}}$ (Figure 5). The extension to the definition of sn-strong head reduction ($\longrightarrow_{\mathsf{sn}}$) is analogous to that of strong head reduction ($\longrightarrow_{\mathsf{SN}}$), using sn instead of SN, and can also be found in the Appendix.

Disjoint sums add several features that we need to account for in our proofs:

Type-directed reduction : $\boxed{\Gamma \vdash M \longrightarrow N : A}$

$$\frac{\Gamma \vdash M : A \quad \Gamma, x{:}A \vdash N_1 : C \quad \Gamma, y{:}B \vdash N_2 : C}{\Gamma \vdash \mathsf{case}\,(\mathsf{inl}\,M)\,\mathsf{of}\,\mathsf{inl}\,x \Rightarrow N_1 \mid \mathsf{inr}\,y \Rightarrow N_2 \longrightarrow [M/x]N_1 : C}\ \text{E-CASE-INL}$$

$$\frac{\Gamma \vdash M : B \quad \Gamma, x{:}A \vdash N_1 : C \quad \Gamma, y{:}B \vdash N_2 : C}{\Gamma \vdash \mathsf{case}\,(\mathsf{inr}\,M)\,\mathsf{of}\,\mathsf{inl}\,x \Rightarrow N_1 \mid \mathsf{inr}\,y \Rightarrow N_2 \longrightarrow [M/y]N_2 : C}\ \text{E-CASE-INR}$$

Fig. 4. Type-directed reduction, extended with disjoint sums

Strongly normalizing neutral terms : $\boxed{\Gamma \vdash M : A \in \mathsf{SNe}}$

$$\frac{\Gamma \vdash M : A + B \in \mathsf{SNe} \quad \Gamma, x{:}A \vdash N_1 : C \in \mathsf{SN} \quad \Gamma, y{:}B \vdash N_2 : C \in \mathsf{SN}}{\Gamma \vdash \mathsf{case}\,M\,\mathsf{of}\,\mathsf{inl}\,x \Rightarrow N_1 \mid \mathsf{inr}\,y \Rightarrow N_2 : C \in \mathsf{SNe}}$$

Strongly normalizing terms : $\boxed{\Gamma \vdash M : A \in \mathsf{SN}}$

$$\frac{\Gamma \vdash M : A \in \mathsf{SN}}{\Gamma \vdash \mathsf{inl}\,M : A + B \in \mathsf{SN}} \qquad \frac{\Gamma \vdash M : B \in \mathsf{SN}}{\Gamma \vdash \mathsf{inr}\,M : A + B \in \mathsf{SN}}$$

Strong head reduction : $\boxed{\Gamma \vdash M \longrightarrow_{\mathsf{SN}} N : A}$

$$\frac{\Gamma \vdash M : A \in \mathsf{SN} \quad \Gamma, x{:}A \vdash N_1 : C \in \mathsf{SN} \quad \Gamma, y{:}B \vdash N_2 : C \in \mathsf{SN}}{\Gamma \vdash \mathsf{case}\,(\mathsf{inl}\,M)\,\mathsf{of}\,\mathsf{inl}\,x \Rightarrow N_1 \mid \mathsf{inr}\,y \Rightarrow N_2 \longrightarrow_{\mathsf{SN}} [M/x]N_1 : C}$$

$$\frac{\Gamma \vdash M : B \in \mathsf{SN} \quad \Gamma, x{:}A \vdash N_1 : C \in \mathsf{SN} \quad \Gamma, y{:}B \vdash N_2 : C \in \mathsf{SN}}{\Gamma \vdash \mathsf{case}\,(\mathsf{inr}\,M)\,\mathsf{of}\,\mathsf{inl}\,x \Rightarrow N_1 \mid \mathsf{inr}\,y \Rightarrow N_2 \longrightarrow_{\mathsf{SN}} [M/x]N_2 : C}$$

$$\frac{\Gamma \vdash M \longrightarrow_{\mathsf{SN}} M' : A + B \quad \Gamma, x{:}A \vdash N_1 : C \quad \Gamma, y{:}B \vdash N_2 : C}{\Gamma \vdash \mathsf{case}\,M\,\mathsf{of}\,\mathsf{inl}\,x \Rightarrow N_1 \mid \mathsf{inr}\,y \Rightarrow N_2 \longrightarrow_{\mathsf{SN}} \mathsf{case}\,M'\,\mathsf{of}\,\mathsf{inl}\,x \Rightarrow N_1 \mid \mathsf{inr}\,y \Rightarrow N_2 : C}$$

Fig. 5. Inductive definition of Strong Normalization, extended with disjoint sums

Well-typed terms and reduction rules With the addition of injections and case-expressions the number of rules for reduction more than doubles. We also need to extend weakening and exchange for typing (Lemma 3.2), weakening of typed reductions (Lemma 3.4) and substitution properties for typed reductions (Lemma 3.5), in addition to the proof that reductions preserve typing (Lemma 3.1) and anti-renaming of typing (Lemma 3.3). The new cases are standard, as well as our lemmas about multi-step reductions. In particular, we need the following:

Lemma 3.21 (Properties of Multi-Step Reductions)**.**

1. *If* $\Gamma \vdash M \longrightarrow^* M' : A$*, then* $\Gamma \vdash \mathit{inl}\,M \longrightarrow^* \mathit{inl}\,M' : A + B$.
2. *If* $\Gamma \vdash M \longrightarrow^* M' : B$*, then* $\Gamma \vdash \mathit{inr}\,M \longrightarrow^* \mathit{inr}\,M' : A + B$.
3. *If* $\Gamma \vdash M \longrightarrow^* M' : A + B$*, then* $\Gamma \vdash \mathit{case}M\ \mathit{of}\ \mathit{inl}x \Rightarrow N_1 \mid \mathit{inr}y \Rightarrow N_2 \longrightarrow^* \mathit{case}M'\ \mathit{of}\ \mathit{inl}x \Rightarrow N_1 \mid \mathit{inr}y \Rightarrow N_2 : C$.
4. *If* $\Gamma, x{:}A \vdash N_1 \longrightarrow^* N_1' : C$*, then* $\Gamma \vdash \mathit{case}M\ \mathit{of}\ \mathit{inl}x \Rightarrow N_1 \mid \mathit{inr}y \Rightarrow N_2 \longrightarrow^* \mathit{case}M\ \mathit{of}\ \mathit{inl}x \Rightarrow N_1' \mid \mathit{inr}y \Rightarrow N_2 : C$.
5. *If* $\Gamma, y{:}B \vdash N_2 \longrightarrow^* N_2' : C$*, then* $\Gamma \vdash \mathit{case}M\ \mathit{of}\ \mathit{inl}x \Rightarrow N_1 \mid \mathit{inr}y \Rightarrow N_2 \longrightarrow^* \mathit{case}M\ \mathit{of}\ \mathit{inl}x \Rightarrow N_1 \mid \mathit{inr}y \Rightarrow N_2' : C$.

Neutral terms Following the definition of SNe, *neutral terms* (Definition 3.3) contain a new inhabitant:

$$\frac{\Gamma \vdash M : A + B \;\mathsf{ne} \qquad \Gamma, x : A \vdash N_1 : C \qquad \Gamma, y : B \vdash N_2 : C}{\Gamma \vdash \mathsf{case}\, M \,\mathsf{of}\, \mathsf{inl}\, x \Rightarrow N_1 \mid \mathsf{inr}\, y \Rightarrow N_2 : C \;\mathsf{ne}}$$

This comes with the need for a new closure property for neutral terms, which we will state shortly in Lemma 3.25.

3.7.1 Extension: Challenge 1

In the previous section, we described our extensions for syntax, reduction, and strong normalization. To prove soundness, we need to extend our repertoire of properties about strong normalization accordingly. The major statements, i.e. confluence (Lemma 3.12), backward closure (Lemma 3.13), and soundness, follow immediately from additional helper lemmas as described in this section.

First, the extended language requires additional subterm properties of strong normalization.

Lemma 3.22 (Properties of Strongly Normalizing Terms)**.**

1. *If* $\Gamma \vdash M : A \in \mathsf{sn}$*, then* $\Gamma \vdash \mathit{inl}\, M : A + B \in \mathsf{sn}$.
2. *If* $\Gamma \vdash M : B \in \mathsf{sn}$*, then* $\Gamma \vdash \mathit{inr}\, M : A + B \in \mathsf{sn}$.
3. *If* $\Gamma \vdash \mathit{case} M \mathit{\ of\ inl} x \Rightarrow N_1 \mid \mathit{inr} y \Rightarrow N_2 : C \in \mathsf{sn}$*, then* $\Gamma \vdash M : A + B \in \mathsf{sn}$ *and* $\Gamma, x{:}A \vdash N_1 : C \in \mathsf{sn}$ *and* $\Gamma, y{:}B \vdash N_2 : C \in \mathsf{sn}$.

The new reduction rules E-CASE-INL and E-CASE-INR play a similar role to β-reduction for abstractions. They thus invoke additional instances of weak head expansion and a new case of the backward lemma.

Lemma 3.23 (Weak Head Expansion)**.**

1. *If* $\Gamma \vdash M : A \in \mathsf{sn}$ *and* $\Gamma \vdash [M/x]N_1 : C \in \mathsf{sn}$ *and* $\Gamma, y{:}B \vdash N_2 : C \in \mathsf{sn}$*, then* $\Gamma \vdash \mathit{case}(\mathit{inl}\, M) \mathit{\ of\ inl} x \Rightarrow N_1 \mid \mathit{inr} y \Rightarrow N_2 \in \mathsf{sn}$.
2. *If* $\Gamma \vdash M : B \in \mathsf{sn}$ *and* $\Gamma, x{:}A \vdash N_1 : C \in \mathsf{sn}$ *and* $\Gamma \vdash [M/y]N_2 : C \in \mathsf{sn}$*, then* $\Gamma \vdash \mathit{case}(\mathit{inr}\, M) \mathit{\ of\ inl} x \Rightarrow N_1 \mid \mathit{inr} y \Rightarrow N_2 \in \mathsf{sn}$.

Lemma 3.24 (Backward Closure of sn)**.** *If* $\Gamma \vdash M : A + B \in \mathsf{sn}$*,* $\Gamma, x{:}A \vdash N_1 : C \in \mathsf{sn}$*,* $\Gamma, y{:}B \vdash N_2 : C \in \mathsf{sn}$*,* $\Gamma \vdash M \longrightarrow^* M' : A + B$*, and* $\Gamma \vdash \mathit{case} M' \mathit{\ of\ inl} x \Rightarrow N_1 \mid \mathit{inr} y \Rightarrow N_2 \in \mathsf{sn}$*, then* $\Gamma \vdash \mathit{case} M \mathit{\ of\ inl} x \Rightarrow N_1 \mid \mathit{inr} y \Rightarrow N_2 \in \mathsf{sn}$.

Together with these extensions, backward closure (Lemma 3.13) follows as before.

While the preservation of neutral terms by reduction (Lemma 3.14 (1)) extends immediately, the corresponding case for the soundness proof requires the following addition to the closure property (Lemma 3.14 (2)):

Lemma 3.25 (Closure Properties of Neutral Terms)**.** *If* $\Gamma \vdash M : A + B \in \mathsf{sn}$*,* $\Gamma \vdash M : A + B \;\mathsf{ne}$*,* $\Gamma, x{:}A \vdash N_1 : C \in \mathsf{sn}$*, and* $\Gamma, y{:}B \vdash N_2 : C \in \mathsf{sn}$*, then* $\Gamma \vdash \mathit{case} M \mathit{\ of\ inl} x \Rightarrow N_1 \mid \mathit{inr} y \Rightarrow N_2 \in \mathsf{sn}$.

The soundness lemma itself (Lemma 3.1) follows immediately with the previous definitions.

3.7.2 Extension: Challenge 2

Finally, we extend our definition of the logical predicate $\mathscr{R}_A$ to disjoint sums. This is not as simple as it may seem: the naïve definition of $\Gamma \vdash \mathsf{inl}\, M \in \mathscr{R}_{A+B}$ iff $\Gamma \vdash M \in \mathscr{R}_A$ and dual is insufficient as it fails to satisfy CR2 and CR3. For example, we have $y{:}A \vdash \mathsf{inl}\, y \in \mathscr{R}_{A+B}$ according to our definition. But although $y{:}A \vdash (\lambda x{:}A.\mathsf{inl}\, x)\, y \longrightarrow \mathsf{inl}\, y : A+B$, our definition fails to prove that $y{:}A \vdash (\lambda x{:}A.\mathsf{inl}\, x)\, y \in \mathscr{R}_{A+B}$.

To be a reducibility candidate, we thus need to improve upon our definition of the logical relation such that it satisfies CR2 and CR3. A *closure* $\overline{\mathscr{R}_A}$ of $\mathscr{R}_A$ ensures exactly this:

$$\frac{\Gamma \vdash M \in \mathscr{R}_A}{\Gamma \vdash M \in \overline{\mathscr{R}_A}} \qquad \frac{\Gamma \vdash M : A \in \mathsf{SNe}}{\Gamma \vdash M \in \overline{\mathscr{R}_A}} \qquad \frac{\Gamma \vdash M : \overline{\mathscr{R}_A} \qquad \Gamma \vdash N \longrightarrow_{\mathsf{SN}} M : A}{\Gamma \vdash N : \overline{\mathscr{R}_A}}$$

The logical predicate $\mathscr{R}_{A+B}$ is then defined as follows:

$$\Gamma \vdash M \in \mathscr{R}_{A+B} \quad \text{iff} \quad \Gamma \vdash M \in \overline{\{\mathsf{inl}\, M' \mid \Gamma \vdash M' \in \mathscr{R}_A\} \cup \{\mathsf{inr}\, M' \mid \Gamma \vdash M' \in \mathscr{R}_B\}}$$

Showing that $\mathscr{R}_{A+B}$ is a reducibility candidate is straightforward. The latter two properties are immediate by the definition of the closure. The proof of CR1 requires an inner induction on the structure of the closure, which is straightforward from the definitions of $\mathscr{R}_{A+B}$ and SN.

Finally, we need to update the fundamental lemma (Lemma 3.20): The cases involving inl and inr are straightforward, and the case expression requires an inner induction similar to the proof of CR1. See the Appendix for full details.

Overall the approach to defining strong normalization inductively is remarkably modular and we hope that this extension to disjoint sums provides further evidence of the value of the inductive definition of strong normalization.

3.8 Further extensions

The current benchmark problem focuses on a unary logical relation (or predicate). A binary logical relation is typically used to prove decidability of conversion in the presence of type-directed equality rules, such as in a typed lambda-calculus with η for function, product, and unit types. In this setting, term equality is checked in a type-directed way interleaving weak-head evaluation with head comparison (Harper & Pfenning, 2005; Goguen, 2005; Abel & Scherer, 2012). Crary (Crary, 2005) explains for the simply-typed lambda-calculus how to set up a suitable Kripke logical relation to prove that algorithmic equality is complete, in particular, transitive and compatible with application. Formalizing the simply-typed version by Crary (2005) would be a logical extension of the *POPLMark Reloaded* challenge.

4 Solutions

We have completed the challenge problem using different encodings and proof environments. We will briefly survey them below.

4.1 Solution A: Using higher-order abstract syntax with Beluga

Beluga (Pientka & Dunfield, 2010) is a proof environment built on top of the (contextual) logical framework LF (Harper *et al.*, 1993; Nanevski *et al.*, 2008). Proofs are implemented as recursive programs based on the Curry-Howard correspondence. The type checker then verifies that we manipulate well-typed derivations, while the totality checker certifies that functions cover all cases and are terminating.

Definitions Beluga allows the user to encode well-typed lambda-terms directly using higher-order abstract syntax (HOAS) where binders in the object language (i.e., in our case STLC) are represented using the binders in the meta-language (i.e., in our case LF) (Pientka, 2007). Such encodings inherit not only α-renaming and substitution from the meta-language, but also weakening/exchange and (single) substitution lemmas. We exploit dependent types in LF to characterize well-typed terms. Hence typing invariants are tracked implicitly using the LF type checker. The type family `tm` is indexed by `ty`, which describes our object language types and has two LF constants `lam` for STLC abstractions and `app` for applications. Note that there is no case for variables — instead we define `lam` as taking in an LF function `tm A → tm B` as an argument. In other words, we re-use the LF function space to model the scope of variables in STLC abstractions.

The reduction relation is modelled using the type family `step`. This is where we take advantage of HOAS. In encoding the β-reduction rule, we simply step `(app (lam M) N)` to `(M N)`. Substitution in STLC is modelled using LF application `(M N)`. Similarly, when reducing the body of an STLC abstraction, we handle renaming by generating a new LF variable `x` and step `(M x)` to some result `(M' x)` (see LF constant `rlam`).

```
LF ty : type =
  | base : ty
  | arr  : ty → ty
    → ty;
```

```
LF tm : ty → type =
  | lam : (tm A → tm B) → tm (arr A B)
  | app : tm (arr A B) → tm A → tm B;
```

```
LF step : tm A → tm A → type =
  | rbeta : step (app (lam _ M) N) (M N)
  | rlam  : (Πx:tm A. step (M x) (M' x)) → step (lam M) (lam M')
  | rappl : step M M' → step (app M N) (app M' N)
  | rappr : step N N' → step (app M N) (app M N');
```

The encoding of (intrinsically) typed terms and reduction rules is just 19 lines of Beluga code.

We obtain basic properties about typed reductions (such as Lemma 3.1), weakening and exchange properties for well-typed terms (Lemma 3.2), and anti-renaming properties of well-typed terms (Lemma 3.3) for free. In addition, we obtain also for free the weakening and exchange property of typed reductions (Lemma 3.4) and the substitution property for typed reductions (Lemma 3.5).

We can model a well-typed term together with its typing context using contextual objects and contextual types offered by contextual LF (Pientka, 2008). Contexts, substitutions, and renamings are also internalized and programmers can work with them directly in Beluga. Strong normalization as accessibility can then be directly encoded in Beluga using *indexed inductive types* (Cave & Pientka, 2012).

```
schema cxt = tm A;

inductive Sn : (Γ : cxt) {M : [Γ ⊢ tm A[]]} type =
| Acc : {Γ : cxt}{A:[ ⊢ ty]}{M : [Γ ⊢ tm A[]]}
        ({M' : [Γ ⊢ tm A[]]} {S : [Γ ⊢ step M M']} Sn [Γ ⊢ M'])
     → Sn [Γ ⊢ M]
```

We write here curly braces for universally quantified variables; we write round parentheses for implicit arguments in Beluga that merely provide a type annotation to some of the parameters. As mentioned before, contexts are first-class citizens and we can declare their "type" by giving a schema definition which in our example simply states that a context of schema `cxt` consists of declarations `tm A` for some type `A`. The contextual type `[Γ ⊢ tm A[]]` describes terms `M` that have type `A` in the context Γ. Further, we want to express that the type `A` is closed and cannot refer to declarations in Γ. We hence associate it with a weakening substitution (written as `A[]`), which transports an object from the empty context to the context Γ.

Kripke-style logical relations about well-typed terms are represented using *indexed stratified types* (Jacob-Rao *et al.*, 2018), which are recursive types that are defined by well-founded recursion over an index. Here, we define Red as a stratified type for well-typed terms based on the type of the term. Context extensions are witnessed by renamings that have type `[Γ' ⊢# Γ]` and are again first-class citizens in Beluga.

```
stratified  Red : (Γ : cxt) {A : [ ⊢ ty ]} {M : [Γ ⊢ tm A[]]} type =
  | RBase : SN [Γ ⊢ M] → Red [ ⊢ base ] [Γ ⊢ M]
  | RArr  : ({Γ' : cxt} {ρ : [Γ' ⊢# Γ]} {N : [Γ' ⊢ tm A[]]}
              Red [⊢ A] [Γ' ⊢ N] → Red [⊢ B] [Γ' ⊢ app M[ρ] N])
            → Red [ ⊢ arr A B ] [Γ ⊢ M]  ;
```

Lemmas and proofs The main advantages of supporting well-typed terms, contexts, and substitutions comes into play when building and type-checking proofs as programs. In particular, Beluga compares well-typed (contextual) objects not only modulo $\beta\eta$, but also modulo the equational theory of simultaneous substitutions described on page 8, taking into account composition, decomposition, and associativity properties of substitutions. This avoids cluttering our proofs with them and leads to a direct, elegant, and compact mechanization of the challenge problem. This is particularly evident in the proof of anti-renaming (Lemma 3.18), which can directly be implemented by pattern matching on the definition of SN, SNe and $\longrightarrow_{\mathsf{SN}}$. We show a few cases illustrating the idea in Fig. 6.

The type of the Beluga program directly corresponds to the statement of the anti-renaming Lemma 3.18 . We leave some arguments such as the type `A` implicit and do not quantify over them; Beluga's type reconstruction algorithm will infer the type. The totality declaration states that the subsequent program can be viewed as an inductive proof that proceeds over `SN [Γ' ⊢ M[ρ]]`. This is verified by the totality checker. We introduce all the arguments that are explicitly quantified using curly braces via the keyword **mlam**; subsequently we introduce the variable `s` that stands for `SN [Γ' ⊢ M[ρ]]`. We then use pattern matching on `s` and consider three possible cases: `SAbs s'`, `SNeu s'` and `SRed r' s'`. Note that in the informal proof we took into account the equational theory of renamings; in Beluga, this is directly incorporated into equivalence checking, unification, and matching. Otherwise the program proceeds as the informal proof. In the case for `SAbs`, weakening of the

```
rec anti_renameSN : {Γ : cxt}{Γ' : cxt} {ρ : [Γ' ⊢# Γ]}{M : [Γ ⊢ tm A[]]}
                    SN [Γ' ⊢ M[ρ]] → SN [Γ ⊢ M]  =
    / total s (anti_renameSN Γ Γ' A ρ M s) /
mlam Γ, Γ', ρ, M ⇒ fn s ⇒ case s of
| SAbs s' ⇒
    SAbs (anti_renameSN [Γ, x:tm _] [Γ', x:tm _ ] [Γ', x:tm _ ⊢ ρ[…], x
    ] [Γ, x:tm _ ⊢ _] s')
| SNeu s' ⇒  SNeu (anti_renameSNe [Γ' ⊢ ρ] [Γ ⊢ M] s')
| SRed r' s' ⇒
   let SNRed' [Γ'] [Γ]  [Γ ⊢ N ] r = anti_renameSNRed [_] [_] [Γ' ⊢ ρ] [_
     ⊢ _ ] r' in
   let s'' = anti_renameSN [Γ] [Γ'] [Γ' ⊢ ρ] [Γ ⊢ N ] s' in
     SRed r s''

and anti_renameSNe  : (Γ : cxt)(Γ' : cxt) {ρ : [Γ' ⊢# Γ]}{M : [Γ ⊢ tm A
    []]}
                      SNe [Γ' ⊢ M[ρ]] → SNe [Γ ⊢ M]  =
    / total s (anti_renameSNe Γ Γ' A ρ M s) /
mlam ρ, M ⇒ fn s ⇒ case s of
|  SVar [Γ' ⊢ _ ] ⇒  SVar [ _ ⊢ _ ]
|  SApp r s ⇒
   let r' = anti_renameSNe [_ ⊢ ρ] [_ ⊢ _] r in
   let s' = anti_renameSN [_] [_] [_ ⊢ ρ] [_ ⊢ _] s in
    SApp r' s'

and anti_renameSNRed: ….  ;
```

Fig. 6. Implementation of the anti-renaming lemma in Beluga

renaming ρ is accomplished by associating ρ with the weakening substitution written as `[…]`. Type reconstruction in Beluga is quite powerful and we can often write an underscore for arguments that are uniquely determined and can be reconstructed.

Size The encoding of the strong normalization proof (Sec. 3.6) is 97 lines of Beluga code; encoding some of the basic properties stated in Sec. 3.1 takes up 39 LOC out of 75 LOC. The encoding of the soundness proof of the inductive strong normalization definition (Sec. 3.3 and Sec. 3.4) is about 192 LOC (see also Fig. 15).

Scalability As the proofs written in Beluga closely resemble their informal versions, the significant increase in length for the pen-and-paper proofs resulting from the addition of several reduction rules for disjoint sums was reflected in the mechanization of the extended soundness proof. This was particularly evident in the Confluence lemma, which had to be extended with an additional 13 cases; and several of these were repetitive. The growing size is also reflected in the size of the soundness proof which is roughly double in size (see again Fig. 15). On the other hand, as the proofs using the inductive definition of SN are quite modular and scale easily without the need for additional lemmas (in particular, they are completely independent of any new reduction rules), the extension of the proof using the inductive definition was correspondingly short, taking fewer than 100 additional lines of code.

```
Inductive tm (Γ : ctx) : ty → Type :=
| var  A : Var Γ A → tm Γ A
| app  A B : tm Γ (A ⇒ B) → tm Γ A → tm Γ B
| lam  A B : tm (A :: Γ) B → tm Γ (A ⇒ B).

Fixpoint Var Γ A :=
  match Γ with
  | []  ⇒ False
  | B :: Γ' ⇒ (A = B) + Var Γ' A
  end.
```

Fig. 7. Type declaration for well-typed de Bruijn in Coq

Lessons learned In general, the Beluga programs correspond directly to the informal proofs and in fact writing out the Beluga programs helped clarify the overall structure of the former. The given benchmark problems were instrumental in driving the implementation and testing of first-class renamings in Beluga. The benchmark also motivated the developers to generalize the totality checker, which was overly conservative in some places, and extended it with lexicographic orders following Twelf's (Pientka, 2005). However, this benchmark also outlines some current limitations:

1. The structural subterm ordering employed by Beluga's totality checker does not take into account exchange; hence, we cannot certify the mechanized proof of the substitution property of multi-step reductions (Lemma 3.6 (5)) where exchange is needed. One might be able to work with a more generalized form of the substitution lemma; possibly more natural would be to extend the implementation and allow a more general structural subterm ordering that allows for exchange of variables.
2. From a practical perspective, one would like to have more support for constructing proofs interactively. This would make it easier for the programmer to tackle larger mechanizations.
3. Type reconstruction in dependently typed systems is still a challenging and not well-understood problem; in Beluga this is exacerbated by the need to infer contexts, substitutions, and renamings. As a consequence, we write out more explicit arguments than in comparable dependently typed systems.

4.2 Solution B: Using well-typed de Bruijn encoding in Coq

The Coq proof assistant is based on an expressive type theory, but offers no special support for reasoning about languages with binders. Different encoding strategies exist to represent variable bindings and the necessary infrastructure ranging from de Bruijn (De Bruijn, 1972), locally nameless (Pollack, 1994; Charguéraud, 2012), weak or parametric HOAS (Despeyroux *et al.*, 1995; Chlipala, 2008), full HOAS (Felty & Momigliano, 2009; Felty & Momigliano, 2012) to named representations (Anand, 2016). Even for de Bruijn, the exact implementation choices (e.g., substitutions as list or as functions, single-scoped de Bruijn or parallel substitutions) have to be carefully chosen and later proofs can differ widely.

In the present work, we focus on a well-typed variant of the de Bruijn representation (Benton *et al.*, 2012) together with parallel substitutions.

In the well-typed de Bruijn encoding, terms are encoded with an inductive family of types `tm` (Figure 7). The type `tm Γ A` stands for terms of type A in context Γ, where a context is a list of types. We write `Var Γ A` for the type of positions of type A in the context Γ, which is defined by recursion on Γ. Unlike in LF, terms have an explicit variable constructor which lifts positions in the context to the corresponding terms.

```
Inductive step {Γ} : ∀ {A}, tm Γ A → tm Γ A → Prop :=
| step_beta  A B (M : tm (A :: Γ) B) (N : tm Γ A) : step  (app (lam M) N) (M[N/])
| step_abs  A B (M M' : tm (A :: Γ) B) : step  M M' → step (lam M) (lam M')
| step_appL A B s1 s2 t: step  s1 s2 → step (app s1 t) (app s2 t)
| step_appR A B s  t1 t2 : step  t1 t2 → step (app s t1) (app s  t2)
```

Fig. 8. Definition of the step relation in Coq

```
Fixpoint Red (Γ : ctx) (A : ty) : tm Γ A → Prop :=
  match A with
  | Base ⇒ fun M ⇒ SN M
  | (A ⇒ B) ⇒ fun M ⇒
      ∀ Γ' (ρ : ren Γ Γ') (N : tm Γ' A), Red M → Red (app (M <ρ>) N)
  end.
```

Fig. 9. Definition of the logical relation in Coq

With these definitions, a (simultaneous) renaming is a function ρ mapping positions in one context Γ to positions in another context Γ', i.e., ρ: ∀ A, Var Γ A → Var Γ' A. A substitution σ is a function mapping context positions to terms in another context, i.e., σ: ∀ A, Var Γ A → tm Γ' A. We write `M<ρ>` and `M[σ]` for the (capture avoiding) instantiation of a term M under a renaming or substitution, respectively.

Their definitions are standard using the primitives suggested in (Abadi *et al.*, 1991), adapted to a well-typed setting. The primitives suffice to express single substitution, denoted as $M[N/]$. To ensure termination, our implementation requires us to define instantiation of renamings and substitutions separately.

Using these definitions entails a range of additional lemmas stating the connection between the different primitives; for example lemmas about composition, decomposition, and associativity of substitutions. These lemmas form a complete (Schäfer *et al.*, 2015) and convergent rewriting system (Curien *et al.*, 1992). As a consequence, all further assumption-free substitution lemmas of the form $s = t$, where s and t contain term constructors, substitution, and the substitution primitives, are proven automatically by rewriting based on a variant of the σ-calculus (Abadi *et al.*, 1991).

This approach is specific to de Bruijn representations and typically not available under other encodings.

Definitions Our definitions are as expected: We implement reduction and strong normalization with inductive types and the logical relation is defined by structural recursion on types.

In contrast to the Beluga solution, we implement beta-reduction using the defined notion of substitution (Figure 8). The logical relation uses the notion of renaming (Figure 9). In general, cases where renamings or substitutions occur, such as for the logical relation in the case of a function type, will later require the rewriting system of the σ-calculus for the corresponding proofs.

Having well-typed (or scoped) terms proves incredibly useful for avoiding errors in the definitions.

Lemmas and proofs The mechanization closely follows the informal proofs. No additional statements were needed. Whenever a statement holds for names but not for de Bruijn, we use the equational theory of the σ-calculus. Repetitive proofs, like the renaming or anti-renaming lemmas, are solved by using Coq's tactic support and writing small scripts to handle the case analysis.

Similar to Beluga, anti-renaming (Lemma 3.18) requires additional care. As an induction can only be performed if the arguments are variables, we have to generalize the goal accordingly. For example, for SN we prove the following equivalent statement:

$\forall\ \Gamma\ A\ (M{:}\ \mathrm{tm}\ \Gamma\ A).\ \mathrm{SN}\ M \to \forall\ \Gamma'\ M'\ (\rho{:}\ \mathrm{ren}\ \Gamma'\ \Gamma).\ M = M'\langle\rho\rangle \to \mathrm{SN}\ M'$

This uses an induction relying on several inversion lemmas.

Size Our solution is concise (Figure 15), in large part due to Coq's automation. Setting up simultaneous renaming and substitution together with the before-mentioned lemmas requires around 150 lines of technical boilerplate.

Scalability The extension to disjoint sums contains twice as many constructors for both the inductive term type and reducibility. Still, our approach scales well: The automation scripts carried over for many lemmas, with at most minor adaptions in the binder cases. Of course, we still have to prove the additional lemmas and handle more cases. These follow the informal proofs. The confluence lemma proves tiresome, as the already high number of cases explodes; maybe evaluation contexts could make these proofs more modular. As can be seen by the line numbers, the framework and reductions require fewer adaptions, while the additional lemmas for strong normalization and soundness are responsible for most additional code.

Variation: Scoped syntax In scoped syntax, terms are $\mathbb{N}$-indexed sets and variables are numbers bounded by that index (see Fig. 10). Analogous to well-typed syntax, types enforce correct *scoping*, but are weaker in that correct *typing* is not guaranteed. The scoped de Bruijn encoding technique is part of the folklore, being already proposed in (Altenkirch, 1993) and used in several case studies, e.g., in Adams' Coq formalization of pure type system (Adams, 2006). A second version of the proofs formalises the challenge with this representation.

In contrast to the well-typed setting, the Autosubst library automatically generates the required definitions and proofs concerning substitution and renaming. These are generated automatically from the corresponding HOAS signature of terms (Kaiser *et al.*, 2017) using the primitives suggested in (Abadi *et al.*, 1991).

As in many cases the typing judgments are superfluous, the proofs translate almost one-to-one from well-typed syntax. The only notable difference happens in the Fundamental lemma (Lemma 3.20), where the induction is on $\Gamma \vdash M : A$ and we thus need an explicit typing statement.

Lessons learned The challenge encouraged us to rethink what a library such as Autosubst is missing: Is it possible to prove substitutivity lemmas automatically? What about renaming or anti-renaming-lemmas? How can we support more advanced (object) types? The

```
Inductive tm (n : ℕ) : Type :=
| var : fin n → tm n
| app : tm n → tm n → tm n
| lam : tm (S n) → tm n.
```

```
Fixpoint fin (n: ℕ) : Type :=
  match n with
  | 0 ⇒ False
  | S n ⇒ option (fin n)
  end.
```

Fig. 10. The declaration for scoped de Bruijn terms in Coq

challenge problem provided the motivation to extend and generalize the Autosubst library to also support well-typed syntax and proofs of substitution and renaming lemmas. This will reduce the boilerplate infrastructure we have built manually. It is unclear if one can also automatically generate proofs of the anti-renaming lemmas, as they do not hold in general.

Comparison The proofs were developed in parallel with the ones in Beluga. Together with our self-imposed restriction to similar lemma statements, difficulties arose in the same places — notwithstanding the different approach to variable binding. We were thus surprised not by the differences but by the similarities between the solutions in Beluga and Coq. This similarity might disappear for more complex syntax, where good practices are less established, as well as for different approaches to binders, such as single-point substitutions or nominal logic.

4.3 Solution C: Using well-typed de Bruijn encoding in Agda

Agda (Norell, 2009) is a dependently-typed programming language based on Martin-Löf Type Theory (Martin-Löf, 1982) with inductive families (Dybjer, 1994), induction-recursion (Dybjer & Setzer, 1999), copattern-matching (Abel *et al.*, 2013) and sized types (Abel, 2010). It offers no special support for representing syntaxes with binding. However, remembering that generic programming is everyday programming in type theory (Altenkirch & McBride, 2003; Benke *et al.*, 2003), we can rely on external libraries providing general tools to represent and manipulate such syntaxes.

In this solution we use the `generic-syntax` library (Allais *et al.*, 2018); this is a generalisation of the observation that a large class of scope-and-type preserving traversals can be seen as instances of the same notion of semantics (Allais *et al.*, 2017). Roughly speaking, this semantics is a scope-aware *fold* over terms of a syntax. Recalling previous results on folds (Malcolm, 1990), we know that once this shared structure is exposed, it can be exploited to state and prove generic lemmas e.g., fusion lemmas, or even the fundamental lemmas of logical relations.

4.3.1 A syntax for syntaxes with binding

The `generic-syntax` library offers a description language for the user to specify what one "layer" of intrinsically scoped and typed syntax with binding looks like. Terms are then obtained as a fixed point of such a description. This construction is analogous to the definition of the universe of indexed families in (Chapman *et al.*, 2010), except that we now deal with binding.

Specification This description language is directly inspired by the careful analysis of the structure of syntaxes with binding.

Our first task is to identify what it means for a term to be well scoped and well sorted. We call I –Scoped a type constructor of kind $(I \to \mathsf{List}\ I \to \mathsf{Set})$. The first index characterises the sort of the overall expression, while the second is the context listing the sorts of the various variables currently in scope. This will be the kind of our terms.

Our second task is to identify the key aspects which one needs in order to describe a term in a syntax with binding. We note three fundamental building blocks:

1. the ability to **store values**, e.g., the type of a let-bound variable cannot be inferred from the type of the overall expression and thus needs to be stored in the `let` node;
2. the ability to **have substructures that may or may not bind extra variables**, e.g., the body of a lambda is a substructure with an extra variable in scope while the function in an application node is a substructure in the same scope as the overall expression;
3. the ability to **enforce that the sort of a term is a given *i* in *I***, e.g., a lambda abstraction *will* have a function type whilst an application node *will* have a type matching its functional argument's co-domain.

From this specification, we can derive an actual implementation of the description language and its semantics.

Implementation We define a datatype Desc of descriptions of well scoped-and-typed syntaxes with binding. It is parameterised by the set I of the sorts that the variables of that syntax may have. A description is given a semantics as an endofunction on I –Scoped (i.e., $(I \to \mathsf{List}\ I \to \mathsf{Set})$ as defined earlier) by the function $[\![\cdot]\!]$. Formally, this corresponds to the following declarations of an inductive type and a recursive function over it:

$$\underline{\mathsf{data}}\ \mathsf{Desc}\ (I : \mathsf{Set}) : \mathsf{Set}_1\ \underline{\mathsf{where}} \qquad\qquad [\![\cdot]\!] : \mathsf{Desc}\ I \to I\text{–}\mathsf{Scoped} \to I\text{–}\mathsf{Scoped}$$

The Desc datatype has three constructors corresponding to the three fundamental building blocks that we have highlighted above. We introduce each constructor together with its interpretation.

1. $`\sigma$ takes two arguments: a Set A and a family of descriptions d indexed over A. It corresponds to a Σ-type and typically uses A as a set of tags marking distinct language constructs (see e.g., TermC). The scope is unchanged when interpreting the rest of the description.

$$`\sigma : (A : \mathsf{Set})\ (\mathsf{d} : A \to \mathsf{Desc}\ I) \to \mathsf{Desc}\ I$$
$$[\![\ `\sigma\ A\ d\ ]\!]\ X\ i\ \Gamma = \Sigma_{a:A}\ ([\![\ d\ a\ ]\!]\ X\ i\ \Gamma)$$

2. $`\mathsf{X}$ takes three arguments: $\Delta$ a List $I$, $j$ an $I$ and $d$ a Description. It marks the presence of a recursive substructure of sort $j$ with extra variables of sort $\Delta$ in scope (see e.g., $(`\mathsf{X}\ (A :: []))\ B$) declaring the body of a λ-abstraction of type $A \Rightarrow B$). The argument d corresponds to the rest of the description and is evaluated in an untouched scope.

$$`\mathsf{X} : \mathsf{List}\ I \to I \to \mathsf{Desc}\ I \to \mathsf{Desc}\ I$$
$$[\![\ `\mathsf{X}\ \Delta\ j\ d\ ]\!]\ X\ i\ \Gamma = X\ (\Delta +\!\!+ \Gamma)\ j \times [\![\ d\ ]\!]\ X\ i\ \Gamma$$

3. ‘■ takes one argument j of type I. This token finishes the description and enforces that the current branch is of sort j (see e.g., ‘■ $(A \Rightarrow B)$ ensuring that the λ-abstraction branch has a function type). This constraint is translated as an equality constraint between the sort we are forcing the term to have and the one that comes as an input.

$$‘\blacksquare : I \to \mathsf{Desc}\ I$$
$$[\![\ ‘\blacksquare\ j\ ]\!]\ X\ i\ \Gamma = i \equiv j$$

Example: Un(i)typed lambda calculus The untyped lambda calculus can be seen as a well scoped and well sorted calculus where the notion of sort is the unit type $\top$. It has two constructors of interest: application and lambda-abstraction.

1. An application node has exactly two substructures, each of which living in the same context as the node itself (hence we use the empty list ([]) as the appropriate notion of context extension). Using _ as the name of the unique value of type $\top$, this gives us the description: ‘X [] _ (‘X [] _ (‘■ _)).
2. A lambda-abstraction node has exactly one substructure with precisely one newly-bound variables. This translates to this description: ‘X (_ :: []) _ (‘■ _)

Finally, we can produce the complete definition by storing (using ‘σ) a tag of type Bool allowing us to pick one of the two constructors defined above. We obtain the final description:

```
‘σ Bool λ b → if b
                then (‘X [] _ (‘X [] _ (‘■ _)))
                else (‘X (_ :: []) _ (‘■ _))
```

4.3.2 Terms as free relative monads on descriptions

We have now seen descriptions and their formal semantics as endofunctions on I–Scoped. The endofunctions thus obtained have a lot of structure: they all are strictly positive endofunctors. Their formal definition and study is outside the scope of this paper, but the practical consequence is that we can take their fixed point.

Terms Concretely, this means that for any description d, we can define the inductive family of well-scoped and well-sorted terms (Tm d) as follows. The two constructors witness two alternatives: we can either have a variable, or we can have one ‘layer’ of term (whose shape is given by $[\![\ d\ ]\!]$) where substructures are terms themselves.

$$\underline{\mathsf{data}}\ \mathsf{Tm}\ (d : \mathsf{Desc}\ I)\ (i : I)\ (\Gamma : \mathsf{List}\ I) : \mathsf{Set}\ \underline{\mathsf{where}}$$
$$‘\mathsf{var} : i \in \Gamma \to \mathsf{Tm}\ d\ i\ \Gamma$$
$$‘\mathsf{con} : [\![\ d\ ]\!]\ (\mathsf{Tm}\ d)\ i\ \Gamma \to \mathsf{Tm}\ d\ i\ \Gamma$$

Our `generic-syntax` library defines generic functions acting on (Tm d) such as parallel renaming and substitution. It also provides proofs of identity and fusion lemmas for these traversals.

A categorical detour This fixed point is the free relative monad (Altenkirch *et al.*, 2014; Altenkirch *et al.*, 2015) with respect to the indexed functor of well scoped-and-typed de Bruijn indices. Informally, a functor T (our terms) is said to be a monad *relative* to a functor V (our variables) if we have two well-behaved polymorphic functions return (our ability to embed variables into terms) and bind (our parallel substitution) with the following types.

$$\begin{aligned}&\mathsf{return} : V(A) \to T(A)\\ &\mathsf{bind} \quad : (V(A) \to T(B)) \to T(A) \to T(B)\end{aligned}$$

By well-behaved we mean that both an identity and a fusion law should hold. That is to say that on one hand (bind return) should be equal to the identity function and that on the other (bind $f \circ$ bind g) should be equal to (bind (bind $f \circ g$)). These are precisely two of the results provided generically by the library.

The Coq solution (Section 4.2) demonstrated that choosing to implement *parallel* renaming and substitution are convenient pragmatic *choices* for a programmer to make. Here the choice of these definitions arise naturally from the framework.

Other generic results Our library provides more advanced results such as a generic statement and proof of the fundamental lemma of logical relations. It is implemented in two steps: we first compute from the syntax description a set of constraints that the relation has to satisfy and then prove that whenever these constraints are met, the fundamental lemma holds. It is a real advantage in solving this challenge.

It is important for the reader to keep in mind that all of the results described so far are obtained by generic programming over the universe of syntaxes with binding and *not* code generation. The output of a code generator cannot be trusted: a bug in the generator can lead it to produce invalid proofs. By contrast, a generic proof is always guaranteed to work.

4.3.3 Our solution to the Challenge

Definitions Now that we have seen how the library is organised, we can define the simply-typed λ-calculus by first giving the datatype of simple types Type and a set of tags corresponding to the language's constructs: Lam represents λ-abstraction nodes while App stands for application; both constructors carry two types.

```
data Type : Set where
  α       : Type
  _⇒_     : Type → Type → Type

data TermC : Set where
  Lam : Type → Type → TermC
  App : Type → Type → TermC
```

Fig. 11. Types and constructor tags for STLC

We then define the (Type $\times$ List Type)-indexed functor underlying STLC as an element of Desc Type. It offers a choice of two constructors (λ and application respectively) by using a sigma type taking a TermC tag first and a description of the corresponding subterms second defined by pattern-matching on the TermC value.

In the Lam A B case, we expect a recursive substructure of type B living in a context extended by a newly-bound variable of type A. This ensures the term thus defined has type $A \Rightarrow B$. In the App A B case, we expect two recursive substructures living in the same

contexts: one with a function type $A \Rightarrow B$ and one with an argument type A. This gives rise to a term of type B.

```
TermD : Desc Type
TermD = ‘σ TermC λ where
  (Lam A B) → ‘X (A :: []) B (‘■ (A ⇒ B))
  (App A B) → ‘X [] (A ⇒ B) (‘X [] A (‘■ B))
```

Fig. 12. Description of STLC as a syntax with binding

If we write STLC for (Tm TermD) we have, modulo isomorphism, three constructors:

- ‘var taking proofs that A is in Γ to STLC A Γ
- ‘λ (lambda abstraction) of type STLC B $(A :: \Gamma) \to$ STLC $(A \Rightarrow B)$ Γ
- ‘$\bullet$ (infix application) of type STLC $(A \Rightarrow B)$ $\Gamma \to$ STLC A $\Gamma \to$ STLC B Γ

Thanks to Agda's support for pattern-synonyms (Pickering *et al.*, 2016) that are refolded in goal types, the fact that we are using an encoding is practically invisible from the reader's point of view. The only troublesome point is the lack of support for interactive case-analysis based on the defined pattern synonyms: case-splits have to be hand-written.

Lemmas and proofs The formalisation closely follows the informal proofs, always generalising the statements involving single-variable substitutions to parallel substitutions. Most inductive definitions are indexed by an extra size argument and the related lemmas are proven to be size preserving, which helps tremendously in getting Agda to check that our recursive definitions are indeed total.

Compared to Coq, we cannot rely on tactics and have to write all proof terms by hand. However, the expressiveness of dependently-typed pattern-matching, the power of size-based termination checking and the consequent library on which we are relying means that our proofs are just as short as tactics-based ones (cf. section 5.4).

Differently from Beluga and similarly to Coq, the theory of renaming and substitution is not internalised. This means that we sometimes have to introduce a level of indirection when stating a lemma so that pattern-matching will not get stuck on unification problems involving renaming. This is particularly apparent in the anti-renaming lemma that Beluga tackles head-on (cf. Figure 6), while we have to use a dummy argument, which is then constrained via an equality, similarly to the Coq solution. We show a few cases below in Figure 13 to illustrate the problem (we quantify over the unbound variables implicitly and use the absurd pattern () to dismiss impossible cases).

The type theory that Agda implements supports large elimination, which means that the definition of the logical predicate for the proof of strong normalization is a run-of-the-mill recursive function.

Our biggest departure from the other formalisations is in the soundness proof, where we opted for evaluation context-based proofs (see Section 3.4 for a description of the twist); this allows us to avoid proving confluence of the reduction relations. Evaluation contexts are especially useful for the proof-term-driven Agda or Beluga, as they make the description of reductions and the corresponding proofs more compact and fewer cases need to be written. However, this additional abstraction makes less of a difference in a

```
th^-1^SN : ∀ M ρ → M′ ≡ ren ρ M → Δ ⊢SN A ∋ M′ → Γ ⊢SN A ∋ M
th^-1^SN M          ρ eq  (neu pr) = neu (th^-1^SNe M ρ eq pr)
th^-1^SN (`λ M)     ρ refl (lam pr) = lam (th^-1^SN M _ refl pr)
th^-1^SN (`var x)   ρ ()   (lam pr)
th^-1^SN (M `• N)   ρ ()   (lam pr)
th^-1^SN M          ρ refl (red r pr) =
  let (M′ , eq , r′) = th^-1^⇝SN M ρ r in red r′ (th^-1^SN M′ ρ eq pr)

th^-1^SNe : ∀ M ρ → M′ ≡ ren ρ M → Δ ⊢SNe A ∋ M′ → Γ ⊢SNe A ∋ M
th^-1^SNe (`var x)   ρ refl (var _)    = var x
th^-1^SNe (M `• N)   ρ refl (app p q)  = app (th^-1^SNe M ρ refl p) (th^-1^SN N ρ refl q)

th^-1^⇝SN : ∀ M ρ → Δ ⊢ A ∋ ren ρ M ⇝SN N′ → ∃ λ N → N′ ≡ ren ρ N × Γ ⊢ A ∋ M ⇝SN N
```

Fig. 13. Implementation of the anti-Renaming lemma in Agda

```
_⊢ℛ_∋_ : ∀ Γ A → Term A Γ → Set
Γ ⊢ℛ α       ∋ M = Γ ⊢SN α ∋ M
Γ ⊢ℛ A ⇒ B  ∋ M = ∀ {Δ} ρ {N} → Δ ⊢ℛ A ∋ N → Δ ⊢ℛ B ∋ ren ρ M `• N
```

Fig. 14. Logical predicate for Strong Normalization

tactic-driven proof assistant like Coq where additional cases can be discharged (mostly) automatically.

Size The proof relies on `generic-syntax` that is approximately 1600 lines of code, as well as (a small part of) the standard library most notably for the definition and properties of the reflexive transitive closure of a relation. The mechanization of the main challenge problems (STLC) is then only 425 LOC (excluding comments and blank lines) as we can take advantage of some of the generic properties established in the library. The mechanization of the challenge extended with disjoint sums (STLC+) is 784 LOC. Both solutions include not only the soundness but also the completeness of the inductive definition of strongly normalizing terms. Figure 15 shows a breakdown section-by-section of the cost of the formalization effort.

Scalability Between STLC and STLC+, the number of constructors in the language more than doubled and the total size of solution is slightly less than twice as big.

To keep things tractable in STLC+ we were forced to introduce new abstractions to structure the completeness proof nicely.

Lessons learned While attempting to solve the challenge, we had to implement new generic results in `generic-syntax` that we had overlooked because of how "boring" the results are (e.g., renaming with the identity is the identity).

This formalisation effort has also revealed some possible extensions of our library: congruence closure of a relation ought to be a generic operation derived from a language's syntax and we ought to be able to prove it well-behaved with respect to (anti-)renaming and (anti-)substitution as long as the original relation is. Similarly the notions of evaluation

contexts and term plugging should be defined once and for all by building on the work studying the derivatives of regular types (Abbott *et al.*, 2005).

5 Critical summary and comparison

Our three mechanizations are remarkably similar in the overall structure, although this may be a factor of referring to a very detailed common informal proof. We will hereby discuss some of the key points among these different solutions.

5.1 Representation of well-typed terms

We explored two different representations of bindings in our mechanizations, one based on HOAS, the other modeled via (well-typed) de Bruijn representations. The HOAS representation is, perhaps not surprisingly, the most compact one; it only requires 19 resp. 30 LOC for defining well-typed terms and well-typed reductions.

In our Coq solution we use well-typed de Bruijn terms. The definition of the inductive type itself is concise, but establishing renaming and substitution (see the next subsection) require approx. 120 LOC for the basic definition of STLC and approx. 150 LOC for the extension with disjoint sums. Here, we can (potentially) exploit code generation via the Autosubst framework to ease the definition of the initial infrastructure. In fact, contextual objects may be viewed as an abstraction of well-typed de Bruijn encodings (Hofmann, 1999) and can actually be compiled to well-typed de Bruijn (Ferreira *et al.*, 2013). Thus one could approximate some of Beluga's features through code generation (Kaiser *et al.*, 2017) and Coq's rewriting facilities. Conversely, this case study can be seen as a step towards understanding how to support HOAS via code generation and generic programming.

In the Coq solution based on scoped terms we use a separate typing judgment (extrinsic typing). In this case, the differences in the proof structure are negligible and the developments are very similar — of course, we have to prove preservation of typing. However, this similarity may depend on this proof in particular, or on the fact that we have *ported* it from the typed to the untyped setting, rather than starting one from scratch. Or it can be a factor of the SN proof itself, since anecdotal evidence, e.g., (Momigliano, 2012), suggests otherwise: in that case study roughly $1/4$ of the theorems involved maintaining type invariants.

We feel that the jury is still out regarding the intrinsic vs. extrinsic types encoding debate: on one hand, the intrinsic type discipline when it works — and recent results ((Poulsen *et al.*, 2018)) shows that it applies outside the usual suspects — yields very compact and economic encodings. On the other, extrinsic typing scales more easily to dependent types (Danielsson, 2007; Chapman, 2009; Altenkirch & Kaposi, 2016) and see, e.g., the Agda formalization in (Abel *et al.*, 2018).

Our Agda solution uses a library (Allais *et al.*, 2018) that provides us with a universe of well scoped-and-typed syntaxes with binding in which to encode our languages. The definition of STLC takes 10 LOC while that of STLC extended with disjoint sums takes 15 LOC. From these definitions we get renaming, substitution as well as (among other things) identity and fusion lemmas. Previous projects like GMeta (Lee *et al.*, 2012) expected users to define their own inductive types and then deploy isomorphisms to embed them into

the universe the generic programs are defined over; we instead expect users to use the universe directly. Agda supports pattern synonyms thus allowing us to hide the encoding. The following example defines a pattern synonym for a λ-abstraction constructor. Agda de-sugars the synonyms we write *and* re-sugars the ones it prints in goal buffers.

```
pattern ‘λ b = ‘con (Lam _ _ , b , refl)
```

We add 9 LOC (resp. 15 LOC) of Agda's pattern synonyms and display pragmas make the user experience nicer. The only downside to working directly in the encoded syntax is that Agda does not yet offer a way to configure its automated case-splitting machinery to generate patterns using a set of pattern synonyms. We currently have to write our pattern-matching definitions by hand.

5.2 Renamings and substitutions

In Beluga, (simultaneous) substitution and renamings are built into the underlying type-theoretic foundation. Checking whether two objects are convertible or unifiable is done modulo the equational theory of substitutions. Thus, weakening and anti-renaming properties such as Lemma 3.2 and 3.3 come for free: the first inherited from LF, the latter from Beluga's theory of substitutions.

In the Coq solution, we use simultaneous substitution as an operation where we implement substitutions as functions that maps variables to terms. Renamings are represented as functions from variables to variables. Coq's rewriting facilities take care of solving equational properties of substitutions automatically.

In the Agda solution, a general notion of scoped-and-typed $\mathscr{V}$-valued environments inherited from (Allais *et al.*, 2018) supersedes both renaming and substitution. Given an I –Scoped relation $\mathscr{V}$, a $\mathscr{V}$-environment from context Γ to context Δ, associates to each variable $x{:}\sigma$ in Γ a value of type $\mathscr{V}\,\sigma\,\Delta$.

Taking $\mathscr{V}$ to be de Bruijn variables yields renamings while using terms yields substitutions. This is a generalisation of (McBride, 2006), which shows how to combine renaming and substitution into a single parametrized operation, cutting down on the boilerplate proofs: instead of four composition lemmata for renamings and substitutions, there is a single parameterized one. However, proofs of weakening and anti-renaming tend to be more tedious than in Beluga, requiring the use of inversion lemmas. A caveat: The use of proper functions requires function extensionality in the meta theory, which is by default absent in the intensional type theories of Agda and Coq. Alternatively, a special pointwise equality can be defined for substitutions.

To support matching on terms under a renaming, both our Coq and Agda solution rely on equality constraints. An alternative is to use an inductive relation between the renaming ρ, a term M and a term M' s.t. $[\rho]M = M'$. This allows us to reason by induction on this relation, as suggested e.g., in (Abel & Vezzosi, 2014). We can show that this inductive relation is equivalent to the algorithmic (functional) specification.

5.3 Induction and recursive definitions

While in the strong normalization proof we only need fairly straightforward arguments by (mutual) structural induction, the proofs of results such as weak head expansion (Lemma 3.10) and backward closure of sn (Lemma 3.13) require lexicographic induction. This resulted in the extension of the totality checker in Beluga to verify termination for such proofs.

In Coq, the user decides via tactics in which order inductions are done (and deals with the consequences). For mutually recursive types (e.g., for the soundness proof), we had to replace the standard induction principle with an (automatically generated) mutual induction principle.

In Agda, we rely on sized types or on structural subterm ordering. Agda will infer a suitable termination measure, if possible.

The proofs of substitution properties for typed reductions (Lemma 3.5) and multi-step reductions (Lemma 3.6 (5)), formulated using single substitutions, require an induction principle that takes into account exchange of variables. As Beluga's totality checker only considers direct subterms, it is currently unable to verify such proofs. This, again, could motivate further extensions to Beluga's totality checker. The Coq and Agda mechanizations instead generalize these lemmas to simultaneous substitutions, which avoids the need to reason about exchange.

The accessibility relation, while being an infinitary inductive definition, does not pose any problem to its encoding via inductive types in any of the considered systems. The logical predicate definition can be encoded in Agda and Coq using a recursive type; in Beluga, we use a *stratified* type; both are ways to define predicates (relations) inductively on one of the indices. In contrast to inductive definitions, such definitions do not give rise to induction principles.

5.4 Lines of code

We summarize in Fig. 15 the lines of code (LOC) necessary to implement the challenge problems considering the definitions and lemmas from each section in Sec. 3. Of course we are well aware that such comparisons are only partially meaningful — especially since in Coq we use proof scripts to construct proofs and in Beluga and Agda we write proofs as programs. Even between Beluga and Agda there are some fundamental differences: Agda supports simultaneous pattern matching similar to Haskell, while in Beluga we usually split on one variable at a time following more the OCaml tradition and the pen-and-paper development of a proof. These different styles in writing proofs impact the LOC count.

However, the LOC counts illustrate some fundamental points: proving the main challenges seems rather similar in size and scale: the main difference lies in how we support the definition of binding syntax. Since Beluga supports binding, contexts, substitutions, renamings, etc. intrinsically, it requires no additional lemmas w.r.t. syntax representation and has therefore the leanest set up. In Coq, we need to support all the above when setting up typed terms, together with the substitution statements of the σ-calculus. The size of the set-up is thus up to three times larger than in Beluga. The generic syntax library is, compared to the rest of the mechanization, huge — it is 2 to 4 times the size of the code necessary to solve the actual challenge problem(s). This is partly due to the fact that it

	Beluga		Coq (Scoped)		Agda	
	STLC	**STLC+**	**STLC**	**STLC+**	**STLC**	**STLC+**
Library (generic-syntax)					1600	1600
Generated Code			0 (167)	0 (218)		
Sec 3.1: Typed Terms & Reduction	75	174	202 (74)	265 (132)	85	144
Sec 3.2: Defining Strong Norm.	30	48	22 (22)	30 (30)	34	47
Twist: Evaluation Contexts					38	57
Sec 3.3: Properties of sn	163	355	97 (97)	214 (212)	97	219
Sec 3.4: Soundness of SN	29	50	17 (17)	24 (24)	12	18
Sec 3.5: Properties of SN	69	136	43 (47)	67 (71)	42	65
Sec 3.6: SN proof using a Log. Pred.	80	155	55 (72)	115 (136)	70	134
Total	446	918	446 (329)	715 (605)	378	684

Fig. 15. Lines of code for Challenge problem 1 and 2

provides more than is necessary for the discussed challenge problems and, of course, it is an effort for which you pay only once.

6 Related work

Normalization proofs by logical relations (Tait, 1967; Friedman, 1975) have been used early in the design and implementation of proof assistants to demonstrate the power of a given system (Coquand, 1993). In fact there is a wide range of approaches and implementations — they all differ from our benchmark.

Altenkirch (1993) presented the first and very influential published encoding of strong normalization, namely for System F in Lego following (Girard *et al.*, 1989)'s proof. Given the lack of inductive types in Lego at the time he had to encode them using recursors. He restricts to untyped lambda terms represented by standard de Bruijn indices — a choice he later in the paper regrets and calls "actually a little bit of cheating" in the accompanying technical report (Altenkirch, 1992), where he also mentions intrinsically-typed representations. System F types are represented with scoped terms. He introduces the accessibility definition of strong normalization and defines reducibility candidates on untyped terms.

Barras (1997) revisited the problem for the Calculus of Constructions and adapted Altenkirch's proof. Girard's strong normalization proof has been also implemented in ATS/LF (Xi, 2004) and translated to Abella (Gacek, 2010). The mechanization introduces a constant inhabiting any type to avoid working with open terms, bringing in lemmas that have no correspondence in the informal proof. This is arguably a shortcut that made sense for ATS/LF (which is constitutionally incapable to reason about open terms), but much less for Abella (which in principle shines in this regard). This trick is in fact absent from the encoding of the same proof in Isabelle Nominal (Group, 2009).

Berger et al. (2006) present formalizations in Minlog, Coq and Isabelle/HOL of a Tait-style proof with an emphasis on extracting variants of the normalization-by-evaluation algorithm. Only the Coq formalization is reported complete, based on a de Bruijn encoding and recursively defining (closed) reducibility by strong elimination.

Abel and Vezzosi (2014) show strong normalization in a more complex setting, namely a simply-typed lambda calculus with guarded recursive types; the Agda mechanization uses intrinsically well-typed terms over coinductively defined object types and unifies

renaming and substitution following (McBride, 2006). The operational semantics is based on evaluation contexts, and the Kripke-style logical relation takes into account context extensions and antitonicity of the recursion depth. From this formalization, a solution of the *POPLMark Reloaded* challenge can be extracted.

There are also a number of mechanizations of weak normalization using reducibility candidates, such as the one in Beluga (Cave & Pientka, 2015; Cave & Pientka, 2018), which also uses a Kripke-style logical relation via context extensions. Another example is (Abel *et al.*, 2018), which covers also dependent types, but with extrinsic typing. Intrinsically well-typed terms are used by (Altenkirch & Kaposi, 2017) in form of a quotient inductive-inductive type.

Further, there are formalizations of normalization following the combinatorial proof in (Joachimski & Matthes, 2003). One such example is the mechanization in Isabelle by Berghofer (2004); another is Abel's (2008) in Twelf. Using the inductive definition of normalization one can in fact avoid using logical predicates at all and prove normalization by elementary means. However, this proof technique does not scale to strong normalization in a proof-theoretically weak system such as Twelf. In fact, we cannot directly represent the accessibility relation: a way to circumvent this problem is to follow Schürmann and Sarnat (2008)'s approach of building an intermediate logic where to express the logical predicate, see also (Rasmussen & Filinski, 2013). This, to our knowledge, has not been done yet.

Last, there have been mechanizations of the meta-theory of equivalence checking in the simply-typed lambda calculus and in LF (Narboux & Urban, 2008; Cave & Pientka, 2018; Urban *et al.*, 2011; Cheney & Momigliano, 2017), in particular showing completeness of the type-directed equivalence algorithm relying on a Kripke-logical relation in Nominal Isabelle and Beluga. The mechanization of this problem in Beluga follows to a large extent the same ideas highlighted in the solution for the strong normalization proof.

As far as other benchmarks for PL theory, Felty et al. (2015; 2018) recently presented some benchmarks emphasizing the all important and often neglected issue of reasoning within a *context of assumptions*, and the role that properties such as weakening, ordering, subsumption play in formal proofs.

7 Conclusion and future work

From our perspective, the proposed challenge problem has already been a success: it has led to a more modern tutorial-style presentation of proving strong normalization using Kripke-style logical relations. It has brought together different communities and developers: those working on logical frameworks, those building libraries for mechanizing meta-theory, and those who use a code generation approach to handling syntax and binding. It has stimulated progress in each of these systems and communities: for example, it has led to extensions to renamings and generalizations of the totality checker in Beluga. It has motivated future revisions in the design of Autosubst and it has led to the implementation of new results in the generic-syntax library that were previously overlooked. Last but not least, it has also helped us better understand the similarities and differences in each of the approaches and we hope there will be more cross-fertilization in the future.

Towards future benchmarks for mechanized metatheory This benchmark is one dimension along which we can compare proof assistants and there are many other aspects where existing proof technology remains ad-hoc and few abstractions exist. We mention a few areas where new benchmarks could and should be crafted. This is by no means an exhaustive enumeration.

One direction of interest is capturing in a direct way more exotic bindings (Cheney, 2005; Weirich *et al.*, 2011; Keuchel *et al.*, 2016; Urban & Kaliszyk, 2012). An obvious first step would be n-ary bindings, i.e. the binding of an arbitrary, previously unknown number of variables, which are needed to express nested pattern matching or mutual recursive let constructions, see for a recent example (Rizkallah *et al.*, 2018). In the context of pattern matching, these were already part of the less-known and even less tackled part B of the first POPLMark challenge. Strong normalization for a language with arbitrarily nested pattern matching would push the boundaries of existing binding techniques.

Reasoning with state and concurrency plays of course a prominent role in programming languages theory and therefore in related formalizations, which have mostly been carried out with ad-hoc methods. In fact, the development of sub-structural logical frameworks or libraries supporting reasoning with resources is lagging behind, with the very notable exception of *Iris* (`https://iris-project.org/`). The few case studies addressed in the literature (Cervesato & Pfenning, 2002; Mahmoud & Felty, 2019) concern issues akin to the first POPLMark challenge, i.e., type preservation. It is important to craft benchmarks that are not overly complex. As such, two suitable options are logical relations for mutable state (Ahmed, 2004) or topics in the meta-theory of session types such as (Pérez *et al.*, 2014).

Finally, another area that deserves new benchmarks is coinductive reasoning. While this has been a staple in proof assistants since the late 90's, most case studies regarded properties of bisimlarity in process and lambda-calculi, to name just a few recent papers (Tiu & Miller, 2010; Momigliano *et al.*, 2019; Bengtson *et al.*, 2016; Lenglet & Schmitt, 2018). These turned (a posteriori) not so challenging, since those coinductive proofs can be carried out more or less with the limited technology of guarded induction. Now that new approaches to the theory and implementation of coinduction have emerged, such as parametrized coinduction (Hur *et al.*, 2013), copatterns (Abel *et al.*, 2013) and *AmiCo* (Blanchette *et al.*, 2017), we seek new benchmarks, possibly something along the lines of (Abel & Chapman, 2014).

We often are driven by proving the next result about a given system, rather than reflecting upon the way we proved a given result. Developing benchmarks is not an easy task, but it enriches our understanding and helps us to develop better foundations and more robust tools for the future.

References

Abadi, Martín, Cardelli, Luca, Curien, Pierre-Louis, & Lévy, Jean-Jacques. (1991). Explicit substitutions. *Journal of functional programming*, **1**(4), 375–416.

Abbott, Michael, Altenkirch, Thorsten, McBride, Conor, & Ghani, Neil. (2005). ∂ for data: Differentiating data structures. *Fundamenta informaticae*, **65**(1-2), 1–28.

Abel, Andreas. (2008). Normalization for the simply-typed lambda-calculus in Twelf. *Pages 3–16 of:* Schürmann, Carsten (ed), *Logical Frameworks and Metalanguages (LFM 04)*. Electronic Notes in Theoretical Computer Science, vol. 199. Elsevier.

Abel, Andreas. (2010). MiniAgda: Integrating sized and dependent types. *Pages 14–28 of:* Bove, Ana, Komendantskaya, Ekaterina, & Niqui, Milad (eds), *Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers, PAR 2010, Edinburgh, UK, 15th July 2010.* EPTCS, vol. 43.

Abel, Andreas, & Chapman, James. (2014). Normalization by evaluation in the delay monad: A case study for coinduction via copatterns and sized types. *Pages 51–67 of: MSFP*. EPTCS, vol. 153.

Abel, Andreas, & Scherer, Gabriel. (2012). On irrelevance and algorithmic equality in predicative type theory. *Logical methods in computer science*, **8**(1:29), 1–36. TYPES'10 special issue.

Abel, Andreas, & Vezzosi, Andrea. (2014). A formalized proof of strong normalization for guarded recursive types. *Pages 140–158 of:* Garrigue, Jacques (ed), *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*. Lecture Notes in Computer Science, vol. 8858. Springer.

Abel, Andreas, Pientka, Brigitte, Thibodeau, David, & Setzer, Anton. (2013). Copatterns: Programming infinite structures by observations. *Pages 27–38 of:* Giacobazzi, Roberto, & Cousot, Radhia (eds), *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'13, Rome, Italy, January 23 - 25, 2013*. ACM Press.

Abel, Andreas, Öhman, Joakim, & Vezzosi, Andrea. (2018). Decidability of conversion for type theory in type theory. *Proceedings of the ACM on programming languages*, **2**(POPL), 23:1–23:29.

Adams, Robin. (2006). Formalized metatheory with terms represented by an indexed family of types. *Pages 1–16 of:* Filliâtre, Jean-Christophe, Paulin-Mohring, Christine, & Werner, Benjamin (eds), *Types for Proofs and Programs*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Ahmed, Amal. (2004). *Semantics of types for mutable state*. Ph.D. thesis, Princeton University.

Ahmed, Amal. (2013). *Logical relations*. Oregon Programming Languages Summer School (OPLSS).

Ahmed, Amal, Fluet, Matthew, & Morrisett, Greg. (2007). L^3: A linear language with locations. *Fundam. inform.*, **77**(4), 397–449.

Allais, Guillaume, Chapman, James, McBride, Conor, & McKinna, James. (2017). Type-and-scope safe programs and their proofs. *Pages 195–207 of: Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*. CPP 2017. New York, NY, USA: ACM.

Allais, Guillaume, Atkey, Robert, Chapman, James, McBride, Conor, & McKinna, James. (2018). A type and scope safe universe of syntaxes with binding: Their semantics and proofs. *Proc. ACM program. lang.*, **2**(ICFP), 90:1–90:30.

Altenkirch, Thorsten. (1992). *Brewing strong normalization proof with LEGO*. Tech. rept. 92-230. LFCS, Edinburgh. http://www.lfcs.inf.ed.ac.uk/reports/92/ECS-LFCS-92-230/ECS-LFCS-92-230.pdf.

Altenkirch, Thorsten. (1993). A formalization of the strong normalization proof for System F in LEGO. *Pages 13–28 of:* Bezem, Marc, & Groote, Jan Friso (eds), *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*. Lecture Notes in Computer Science, vol. 664. Springer.

Altenkirch, Thorsten, & Kaposi, Ambrus. (2016). Type theory in type theory using quotient inductive types. *Pages 18–29 of:* Bodík, Rastislav, & Majumdar, Rupak (eds), *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM Press.

Altenkirch, Thorsten, & Kaposi, Ambrus. (2017). Normalisation by evaluation for type theory, in type theory. *Logical methods in computer science*, **13(4:1)**(Oct.), 1–26.

Altenkirch, Thorsten, & McBride, Conor. (2003). Generic programming within dependently typed programming. *Pages 1–20 of: Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*. Deventer, The Netherlands, The Netherlands: Kluwer, B.V.

Altenkirch, Thorsten, Hofmann, Martin, & Streicher, Thomas. (1995). Categorical reconstruction of a reduction free normalization proof. *Pages 182–199 of:* Pitt, David H., Rydeheard, David E., & Johnstone, Peter (eds), *Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK, August 7-11, 1995, Proceedings*. Lecture Notes in Computer Science, vol. 953. Springer.

Altenkirch, Thorsten, Chapman, James, & Uustalu, Tarmo. (2014). Relative monads formalised. *Journal of Formalized Reasoning*, **7**(1), 1–43.

Altenkirch, Thorsten, Chapman, James, & Uustalu, Tarmo. (2015). Monads need not be endofunctors. *Logical methods in computer science*, **11**(1).

Anand, Abishek. (2016). *Exploiting uniformity in substitution: The Nuprl term model*. The 5th International Congress on Mathematical Software (ICMS 2016).

Aydemir, B., Bohannon, A., Fairbairn, M., Foster, J., Pierce, B., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., & Zdancewic, S. (2005). Mechanized metatheory for the masses: The POPLmark challenge. *Pages 50–65 of:* Hurd, Joe, & Melham, Thomas F. (eds), *Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*. Lecture Notes in Computer Science, vol. 3603. Springer.

Aydemir, Brian, & Weirich, Stephanie. (2010). *LNgen: Tool support for locally nameless representations*. Tech. rept. MS-CIS-10-24. University of Pennsylvania.

Barras, Bruno, & Werner, Benjamin. (1997). *Coq in Coq*. 30 pages, unpublished.

Bengtson, Jesper, Parrow, Joachim, & Weber, Tjark. (2016). Psi-calculi in isabelle. *Journal of automated reasoning*, **56**(1), 1–47.

Benke, Marcin, Dybjer, Peter, & Jansson, Patrik. (2003). Universes for generic programs and proofs in dependent type theory. *Nordic j. of computing*, **10**(4), 265–289.

Benton, Nick, Hur, Chung-Kil, Kennedy, Andrew, & McBride, Conor. (2012). Strongly typed term representations in Coq. *Journal of Automated Reasoning*, **49**(2), 141–159.

Berger, Ulrich, Berghofer, Stefan, Letouzey, Pierre, & Schwichtenberg, Helmut. (2006). Program extraction from normalization proofs. *Studia logica*, **82**(1), 25–49.

Berghofer, Stefan. (2004). Extracting a normalization algorithm in Isabelle/HOL. *Pages 50–65 of: TYPES*. Lecture Notes in Computer Science, vol. 3839. Springer.

Blanchette, Jasmin Christian, Bouzy, Aymeric, Lochbihler, Andreas, Popescu, Andrei, & Traytel, Dmitriy. (2017). Friends with benefits. *Pages 111–140 of: Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201*. New York, NY, USA: Springer-Verlag New York, Inc.

Cave, Andrew, & Pientka, Brigitte. (2012). Programming with binders and indexed data-types. *Pages 413–424 of:* Field, John, & Hicks, Michael (eds), *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. ACM Press.

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

Cave, Andrew, & Pientka, Brigitte. (2015). A case study on logical relations using contextual types. *Pages 33–45 of:* Cervesato, Iliano, & Chaudhuri, Kaustuv (eds), *Proceedings Tenth International Workshop on Logical Frameworks and Meta Languages: Theory and Practice, LFMTP 2015, Berlin, Germany, 1 August 2015.* EPTCS, vol. 185.

Cave, Andrew, & Pientka, Brigitte. (2018). Mechanizing proofs with logical relations – Kripke-style. *Mathematical structures in computer science*, **28**(9), 1606–1638.

Cervesato, I., & Pfenning, F. (2002). A linear logical framework. *Information and computation*, **179**(1), 19–75. cited By 52.

Chapman, James. (2009). Type theory should eat itself. *Electronic notes in theoretical computer science*, **228**, 21–36. Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP 2008).

Chapman, James, Dagand, Pierre-Évariste, McBride, Conor, & Morris, Peter. (2010). The gentle art of levitation. *Sigplan not.*, **45**(9), 3–14.

Charguéraud, Arthur. (2012). The locally nameless representation. *Journal of automated reasoning*, **49**(3), 363–408.

Cheney, James. (2005). Toward a general theory of names: binding and scope. *Pages 33–40 of:* Momigliano, Alberto, & Pollack, Randy (eds), *ACM SIGPLAN International Conference on Functional Programming, Workshop on Mechanized reasoning about languages with variable binding, MERLIN 2005, Tallinn, Estonia, September 30, 2005*. ACM.

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

Chlipala, Adam. (2008). Parametric higher-order abstract syntax for mechanized semantics. *Pages 143–156 of: ACM Sigplan Notices*, vol. 43. ACM.

Coquand, Catarina. (1993). From semantics to rules: A machine assisted analysis. *Pages 91–105 of:* Börger, Egon, Gurevich, Yuri, & Meinke, Karl (eds), *Computer Science Logic, 7th Workshop, CSL '93, Swansea, United Kingdom, September 13-17, 1993, Selected Papers*. Lecture Notes in Computer Science, vol. 832. Springer.

Crary, Karl. (2005). Logical relations and a case study in equivalence checking. Pierce, Bejamin C. (ed), *Advanced Topics in Types and Programming Languages*. The MIT Press.

Curien, Pierre-Louis, Hardin, Thérèse, & Ríos, Alejandro. (1992). Strong normalization of substitutions. *Pages 209–217 of:* Havel, Ivan M., & Koubek, Václav (eds), *Mathematical Foundations of Computer Science 1992, 17th International Symposium, MFCS'92, Prague, Czechoslovakia, August 24-28, 1992, Proceedings*. Lecture Notes in Computer Science, vol. 629. Springer.

Danielsson, Nils Anders. (2007). A formalisation of a dependently typed language as an inductive-recursive family. *Pages 93–109 of:* Altenkirch, Thorsten, & McBride, Conor (eds), *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*. Lecture Notes in Computer Science, vol. 4502. Springer.

De Bruijn, Nicolaas Govert. (1972). Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Pages 381–392 of: Indagationes Mathematicae (Proceedings)*, vol. 75. Elsevier.

Despeyroux, Joëlle, Felty, Amy P., & Hirschowitz, André. (1995). Higher-order abstract syntax in coq. *Pages 124–138 of:* Dezani-Ciancaglini, Mariangiola, & Plotkin, Gordon D. (eds), *Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA '95, Edinburgh, UK, April 10-12, 1995, Proceedings*. Lecture Notes in Computer Science, vol. 902. Springer.

Dybjer, Peter. (1994). Inductive families. *Formal aspects of computing*, **6**(4), 440–465.

Dybjer, Peter, & Setzer, Anton. (1999). A finite axiomatization of inductive-recursive definitions. *Pages 129–146 of:* Girard, Jean-Yves (ed), *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings*. Lecture Notes in Computer Science, vol. 1581. Springer.

Felty, Amy, & Momigliano, Alberto. (2012). Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *Journal of automated reasoning*, **48**(1), 43–105.

Felty, Amy P., & Momigliano, Alberto. (2009). Reasoning with hypothetical judgments and open terms in Hybrid. *Pages 83–92 of: 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP '09)*. New York, NY, USA: ACM.

Felty, Amy P., Momigliano, Alberto, & Pientka, Brigitte. (2015). The next 700 challenge problems for reasoning with higher-order abstract syntax representations - part 2 - A survey. *J. autom. reasoning*, **55**(4), 307–372.

Felty, Amy P., Momigliano, Alberto, & Pientka, Brigitte. (2018). Benchmarks for reasoning with syntax trees containing binders and contexts of assumptions. *Mathematical structures in computer science*, **28**(9), 1507–1540.

Ferreira, Francisco, Monnier, Stefan, & Pientka, Brigitte. (2013). Compiling contextual objects: bringing higher-order abstract syntax to programmers. *Pages 13–24 of: Proceedings of the 7th workshop on Programming languages meets program verification*. ACM.

Friedman, Harvey. (1975). Equality between functionals. *Pages 22–37 of:* Parikh, R. (ed), *Logic Colloquium*. Lecture Notes in Mathematics, vol. 453. Springer.

Gacek, Andrew. (2010). *Girard's proof of strong normalization of the simply-typed lambda-calculus calculus*. `http://abella-prover.org/examples/lambda-calculus/normalization/stlc-strong-norm.html`.

Geuvers, Herman. (1995). A short and flexible proof of strong normalization for the calculus of constructions. *Pages 14–38 of: Selected Papers from the International Workshop on Types for Proofs and Programs*. TYPES '94. London, UK, UK: Springer-Verlag.

Girard, J. Y. (1972). *Interprétation fonctionnelle et elimination des coupures de l'arithmétique d'ordre supérieur*. These d'état, Université de Paris 7.

Girard, Jean-Yves, Lafont, Yves, & Taylor, Paul. (1989). *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science, vol. 7. Cambridge University Press.

Goguen, Healfdene. (1995). Typed operational semantics. *Pages 186–200 of:* Dezani-Ciancaglini, Mariangiola, & Plotkin, Gordon (eds), *2nd International Conference on Typed Lambda Calculi and Applications (TLCA'95)*. Lecture Notes in Computer Science (LNCS 902). Springer.

Goguen, Healfdene. (2005). Justifying algorithms for $\beta\eta$ conversion. *Pages 410–424 of:* Sassone, Vladimiro (ed), *Foundations of Software Science and Computational Structures, 8th International Conference, FoSSaCS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Lecture Notes in Computer Science, vol. 3441. Springer.

Group, Nominal Methods. (2009). *Strong normalization for the simply typed lambda calculus*. `https://isabelle.in.tum.de/dist/library/HOL/HOL-Nominal-Examples/SN.html`.

Harper, Robert, & Pfenning, Frank. (2005). On equivalence and canonical forms in the LF type theory. *ACM transactions on computational logic*, **6**(1), 61–101.

Harper, Robert, Honsell, Furio, & Plotkin, Gordon D. (1993). A framework for defining logics. *Journal of the association of computing machinery*, **40**(1), 143–184.

Hoare, Tony. (2003). The verifying compiler: A grand challenge for computing research. *Pages 25–35 of:* Böszörményi, László, & Schojer, Peter (eds), *Modular Programming Languages*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Hofmann, Martin. (1999). Semantical analysis of higher-order abstract syntax. *Pages 204–213 of: Logic in Computer Science, 1999. Proceedings. 14th Symposium on*. IEEE.

Hur, Chung-Kil, Neis, Georg, Dreyer, Derek, & Vafeiadis, Viktor. (2013). The power of parameterization in coinductive proof. *Pages 193–206 of: POPL '13*. NY, USA: ACM.

Jacob-Rao, Rohan, Pientka, Brigitte, & Thibodeau, David. (2018). Index-stratified types. *Pages 19:1–19:17 of:* Kirchner, Hélène (ed), *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*. LIPIcs, vol. 108. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Joachimski, Felix, & Matthes, Ralph. (2003). Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and Gödel's T. *Archive of mathematical logic*, **42**(1), 59–87.

Kaiser, Jonas, Schäfer, Steven, & Stark, Kathrin. (2017). Autosubst 2: Towards reasoning with multi-sorted de Bruijn terms and vector substitutions. *Pages 10–14 of: Proceedings of the Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*. LFMTP '17. New York, NY, USA: ACM.

Keuchel, Steven, Weirich, Stephanie, & Schrijvers, Tom. (2016). Needle & Knot: Binder boilerplate tied up. *Pages 419–445 of: European Symposium on Programming*. Springer.

Klein, Gerwin, & Nipkow, Tobias. (2006). A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM trans. program. lang. syst.*, **28**(4), 619–695.

Kumar, Ramana, Myreen, Magnus O., Norrish, Michael, & Owens, Scott. (2014). CakeML: a verified implementation of ML. *Pages 179–192 of:* Jagannathan, Suresh, & Sewell, Peter (eds), *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. ACM Press.

Lee, Daniel K., Crary, Karl, & Harper, Robert. (2007). Towards a mechanized metatheory of standard ML. *Pages 173–184 of:* Hofmann, Martin, & Felleisen, Matthias (eds), *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*. ACM Press.

Lee, Gyesik, d. S. Oliveira, Bruno C., Cho, Sungkeun, & Yi, Kwangkeun. (2012). GMeta: A generic formal metatheory framework for first-order representations. *Pages 436–455 of:* Seidl, Helmut (ed), *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Lecture Notes in Computer Science, vol. 7211. Springer.

Lenglet, Sergueï, & Schmitt, Alan. (2018). Hoπ in Coq. *Pages 252–265 of: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2018. New York, NY, USA: ACM.

Leroy, Xavier. (2009). Formal verification of a realistic compiler. *Communications of the ACM*, **52**(7), 107–115.

Licata, Daniel R, & Harper, Robert. (2009). A universe of binding and computation. *Pages 123–134 of: ACM Sigplan Notices*, vol. 44. ACM.

Mahmoud, Mohamed Yousri, & Felty, Amy P. (2019). Formalization of metatheory of the quipper quantum programming language in a linear logic. *J. autom. reasoning*, **63**(4), 967–1002.

Malcolm, Grant. (1990). Data structures and program transformation. *Sci. comput. program.*, **14**(2-3), 255–279.

Martin-Löf, Per. (1982). Constructive mathematics and computer programming. *Studies in logic and the foundations of mathematics*, **104**, 153–175.

McBride, Conor. (2006). *Type-preserving renaming and substitution*. Unpublished draft.

Mitchell, John C., & Moggi, Eugenio. (1991). Kripke-style models for typed lambda calculus. *Ann. pure appl. logic*, **51**(1-2), 99–124.

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

Momigliano, Alberto, Pientka, Brigitte, & Thibodeau, David. (2019). A case study in programming coinductive proofs: Howe's method. *Mathematical structures in computer science*, **29**(8), 1309–1343.

Nanevski, Aleksandar, Pfenning, Frank, & Pientka, Brigitte. (2008). Contextual modal type theory. *ACM transactions on computational logic*, **9**(3), 1–49.

Narboux, Julien, & Urban, Christian. (2008). Formalising in Nominal Isabelle Crary's completeness proof for equivalence checking. *Electronic notes in theoretical computer science*, **196**, 3 – 18. Proceedings of the Second International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2007).

Norell, Ulf. (2009). Dependently typed programming in Agda. *Pages 230–266 of: AFP Summer School*. Springer.

Pérez, Jorge A., Caires, Luís, Pfenning, Frank, & Toninho, Bernardo. (2014). Linear logical relations and observational equivalences for session-based concurrency. *Inf. comput.*, **239**, 254–302.

Pickering, Matthew, Érdi, Gergö, Peyton Jones, Simon, & Eisenberg, Richard A. (2016). Pattern synonyms. *Pages 80–91 of: Proceedings of the 9th International Symposium on Haskell*. Haskell 2016. New York, NY, USA: ACM.

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

Pientka, Brigitte. (2007). Proof pearl: The power of higher-order encodings in the logical framework LF. *Pages 246–261 of:* Schneider, Klaus, & Brandt, Jens (eds), *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*. Lecture Notes in Computer Science, vol. 4732. Springer.

Pientka, Brigitte. (2008). A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. *Pages 371–382 of:* Necula, George C., & Wadler, Philip (eds), *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. ACM Press.

Pientka, Brigitte, & Dunfield, Joshua. (2010). Beluga: a framework for programming and reasoning with deductive systems (System Description). *Pages 15–21 of:* Giesl, Jürgen, & Haehnle, Reiner (eds), *5th International Joint Conference on Automated Reasoning (IJCAR'10)*. Lecture Notes in Artificial Intelligence (LNAI 6173). Springer.

Pierce, Benjamin C. (2002). *Types and programming languages*. MIT Press.

Pierce, Benjamin C., & Weirich, Stephanie. (2012). Introduction to the special issue on the POPLMark Challenge. *Journal of Automated Reasoning*, **49**(3), 301–302.

Plotkin, Gordon. (1973). *Lambda-definability and logical relations*. Memorandum sai-rm-4. University of Edinburgh.

Pollack, Robert. (1994). Closure under alpha-conversion. *Pages 313–332 of:* Barendregt, Henk, & Nipkow, Tobias (eds), *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*. Lecture Notes in Computer Science, vol. 806. Springer.

Pouillard, Nicolas, & Pottier, François. (2010). A fresh look at programming with names and binders. *Pages 217–228 of: ACM Sigplan Notices*, vol. 45. ACM.

Poulsen, Casper Bach, Rouvoet, Arjen, Tolmach, Andrew, Krebbers, Robbert, & Visser, Eelco. (2018). Intrinsically-typed definitional interpreters for imperative languages. *PACMPL*, **2**(POPL), 16:1–16:34.

Rasmussen, Ulrik, & Filinski, Andrzej. (2013). Structural logical relations with case analysis and equality reasoning. *Pages 43–54 of:* Momigliano, Alberto, Pientka, Brigitte, & Pollack, Randy (eds), *Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks & Meta-languages: Theory & Practice, LFMTP 2013, Boston, Massachusetts, USA, September 23, 2013*. ACM.

Rizkallah, Christine, Garbuzov, Dmitri, & Zdancewic, Steve. (2018). A formal equational theory for call-by-push-value. *Pages 523–541 of: International Conference on Interactive Theorem Proving*. Springer.

Rossberg, Andreas, Russo, Claudio, & Dreyer, Derek. (2014). F-ing modules. *Journal of functional programming*, **24**(5), 529–607.

Schäfer, Steven, Tebbi, Tobias, & Smolka, Gert. (2014). Autosubst: Automation for de Bruijn substitutions. *6th Coq Workshop (July 2014)*.

Schäfer, Steven, Smolka, Gert, & Tebbi, Tobias. (2015). Completeness and decidability of de Bruijn substitution algebra in Coq. *Pages 67–73 of: Proceedings of the 2015 Conference on Certified*

Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015. Berlin, Heidelberg: Springer-Verlag.

Schürmann, Carsten, & Sarnat, Jeffrey. (2008). Structural logical relations. *Pages 69–80 of:* Pfenning, Frank (ed), *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*. IEEE Computer Society Press.

Sturm, Sebastian. (2018). *Verification and theorem proving in F^**. M.Phil. thesis, LMU. `https://github.com/sturmsebastian/Fstar-master-thesis-code`.

Tait, William W. (1967). Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic*, **32**(2), 198–212.

Tiu, Alwen, & Miller, Dale. (2010). Proof search specifications of bisimulation and modal logics for the pi-calculus. *ACM trans. comput. log.*, **11**(2), 13:1–13:35.

Urban, Christian, & Kaliszyk, Cezary. (2012). General bindings and alpha-equivalence in nominal isabelle. *Logical methods in computer science*, **8**(2).

Urban, Christian, Cheney, James, & Berghofer, Stefan. (2011). Mechanizing the metatheory of LF. *ACM transactions on computational logic*, **12**(2), 15:1–15:42.

van Raamsdonk, Femke, & Severi, Paula. (1995). *On normalisation*. Tech. rept. 95/20. Technische Universiteit Eindhoven.

Wang, Yuting, Chaudhuri, Kaustuv, Gacek, Andrew, & Nadathur, Gopalan. (2013). Reasoning about higher-order relational specifications. *Pages 157–168 of: Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*. ACM.

Watkins, Kevin, Cervesato, Iliano, Pfenning, Frank, & Walker, David. (2002). *A concurrent logical framework I: Judgments and properties*. Tech. rept. CMU-CS-02-101. Department of Computer Science, Carnegie Mellon University.

Weirich, Stephanie, Yorgey, Brent A, & Sheard, Tim. (2011). Binders unbound. *Pages 333–345 of: ACM SIGPLAN Notices*, vol. 46. ACM.

Werner, Benjamin. (1992). A normalization proof for an impredicative type system with large elimination over integers. *Pages 341–357 of: International Workshop on Types for Proofs and Programs (TYPES)*.

Xi, Hongwei. (2004). Applied type system. *Pages 394–408 of: TYPES 2003*. Lecture Notes in Computer Science, vol. 3085. Springer.