University of Udine

Department of Mathematics and Computer Science

PREPRINT

Work-in-Progress Proceedings of the Workshop on "Logical Frameworks and Meta-Languages: Theory and Practice"

Marino Miculan, Florian Rabe

Preprint nr.: 5/2017

Work-in-Progress Proceedings of the Workshop on

Logical Frameworks and Meta-Languages: Theory and Practice

held in Oxford on 2017-09-08
at the Conference on
Formal Structures for Computation and Deduction

Organizers and program chairs:
Marino Miculan, University of Udine, Italy
Florian Rabe, Jacobs University Bremen, Germany

Preface

This volume contains a selection of papers presented at LFMTP 2017, held in Oxford, September 8th, 2017

We received 8 submissions, each of which was formally peer-reviewed by three reviewers. Eventually the program committee chose 7 submissions for presentation at the workshop. Regular papers are published in the formal proceedings on the ACM Digital Library, and available at http://dl.acm.org/citation.cfm?id=3130261; work-in-progress papers are published in these proceedings.

In addition, the program included two invited talks: *Verifiable C, a Higher-order Impredicative Concurrent Separation Logic in Coq* by Andrew Appel, and *Names, Places, and Things; fragments of a partial intellectual biography of Randy Pollack* by James McKinna. The latter was part of a special session celebrating the 70th birthday of Randy Pollack, author of the LEGO proof assistant among many other contributions to this field.

We would like to thank the many people who helped make LFMTP 2017 a success. We thank the organizers of FSCD for their support. We are grateful to the program committee members for their thoughtful reviews and discussions. Finally, we thank the authors, the invited speakers, and the attendees for making this workshop an enjoyable and productive event.

Program Committee

- Thorsten Altenkirch (University of Nottingham, UK)
- Kaustuv Chaudhuri (INRIA, France)
- Gilles Dowek (ENS Cachan, France)
- Amy Felty (University of Ottawa, Canada)
- Andrzej Filinski (University of Copenhagen, Denmark)
- Marino Miculan (DMIF, University of Udine, Italy), co-chair
- Florian Rabe (Jacobs University Bremen, Germany), co-chair
- Wilmer Ricciotti (LFCS, University of Edinburgh, UK)
- Claudio Sacerdoti Coen (University of Bologna, Italy)
- Kristina Sojakova (Appalachian State University, USA)

Contents

Making Substitutions Explicit in SASyLF[*]

Michael D. Ariotti & John Tang Boyland
Department of EE & Computer Science
College of Engineering and Applied Science
University of Wisconsin–Milwaukee
Milwaukee, Wisconsin, USA
{ariotti, boyland}@uwm.edu

ABSTRACT

SASyLF is an interactive proof assistant whose goal is to teach: about type systems, language meta-theory, and writing proofs in general. This software tool stores user-specified languages and logics in the dependently-typed LF, and its internal proof structure closely resembles $\mathcal{M}_2^+$. This paper describes a new usability feature of SASyLF, "where" clauses, which make explicit previously hidden substitutions that arise from case analyses within a proof. The requirements for "where" clauses are discussed, including a formal definition of correctness. The feature's implementation in SASyLF is outlined, and future extensions are discussed.

KEYWORDS

SASyLF, proof assistant, education, unification, LF, $\mathcal{M}_2^+$

1 INTRODUCTION

SASyLF [1] is an interactive[1] proof assistant whose goal is to teach: about type systems, language meta-theory, and writing proofs in general. Originally designed and developed by Jonathan Aldrich and others, it is currently maintained by John Tang Boyland, who uses the assistant with students to teach courses on type systems.

To facilitate its purpose, SASyLF's language is close to what would be written in language descriptions and proofs on paper. Furthermore, the errors it generates are as descriptive and local to the cause as possible, often offering suggestions for correction.

Like its older brother Twelf [6], SASyLF stores logical information in the dependently-typed LF [2]. Unlike Twelf, SASyLF's internal proof structure closely resembles $\mathcal{M}_2^+$ [8]. SASyLF owes much to these previous works.

The SASyLF user describes a language or logic via an abstract syntax and a set of judgments, where each judgment is defined with a form and a set of inference rules.

After an object language is thus described, the user can write theorems about its meta-theory. Each theorem is represented with the form $\forall x_1 : \tau_1 . \forall x_2 : \tau_2 . \cdots . \forall x_n : \tau_n . \exists y : \tau$, and must be proven with a total recursive function, defined by cases on the inputs—as described by Schürmann [8].

For a given theorem, these inputs $x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n$ represent initial assumptions, forming a local context Γ to the theorem, similar to how formal parameters are treated as local variables in a programmatic function. For a theorem to be applicable to the entire object language, these inputs should be written with meta-variables.

Assumptions in SASyLF are represented in LF, and since LF is dependently typed, often τ_i depends one or more previous inputs $x_j : \tau_j$, where $j < i$. Thus, some inputs are syntactic constructs (which have no dependencies), while others can be schematic judgments on those constructs.

To prove a theorem, its proof-as-a-function must produce a derivation d with the same LF type τ as y for every possible set of inputs. (Again like a programmatic function, only the type of the output is enforced. The form of the LF term which has that type, and the method of its creation, are in the hands of the proof function.) The SASyLF proof-writer has the following techniques[2] available for employ on the path to producing d:

(1) The construction of a derivation via application of (a) inference rules in the language description, (b) lemmas or theorems proven prior to this one, or (c) this theorem, through induction. The arguments to such an application must be assumptions in the local context Γ. In the case of (c), the arguments must be "smaller" than the current inputs, in a technical sense familiar to those proving the termination of recursive functions.
(2) The construction of a derivation via case analysis of a syntax construct or derivation in scope. This technique is often applied to an input of the theorem, but a derivation constructed in the proof can be a case analysis subject as well. Also, a case analysis need not be the final construction of a proof; the proof can continue after the analysis is finished.
(3) For a theorem which allows hypothetical contexts—i.e., its local context Γ can be assumed to contain other assumptions than those explicitly listed as input—its proof is allowed to extend Γ with further hypothetical assumptions, as opposed to the explicit constructions described in (1) and (2), and (4).
(4) Related to (3), the construction of derivations through manipulation of the hypothetical context, taking advantage of the fact that object variables are represented internally by LF variables, via HOAS [5]: **`by weakening`**, **`by exchange`**, and **`by substitution`**.

The semantics of proof by case analysis (2) is the subject of this paper. In particular it will be shown, as it is by Schürmann [8], that a case analysis represents a simultaneous substitution applied to all assumptions in the context. Described here is a new feature of the SASyLF language, "where" clauses, which makes these substitutions explicit.

[*]The authors would like to thank the anonymous reviewers for their helpful comments.
[1]SASyLF was not originally interactive. It has been made so through a plugin for Eclipse. This plugin, along with the command-line version of the tool and the documentation and source for both, are available at `http://github.com/boyland/sasylf`.

[2]Most of these correspond well to proof techniques described by Schürmann [8].

```
terminals lam dot value
          true false if then else Bool

syntax
t ::= x | lam x:T dot t[x] | t t
    | true | false | if t then t else t

T ::= T -> T
    | Bool

Gamma ::= * | Gamma, x:T
```

Figure 1: An abstract syntax for $\lambda_{\rightarrow B}$

In the simplest terms, a theorem is proven along a given branch of its proof whenever some $d : \tau \in \Gamma$ (although, sometimes d has to be explicitly pointed out). However, substitutions resulting from case analyses can alter what τ means. By extension, then, "where" clauses can also make what remains to be proven more transparent[3].

The remainder of this section will describe an example language which will motivate "where" clauses. Section 2 details the requirements for the feature, including when "where" clauses are correct, and section 3 outlines an implementation to fulfill them. Section 4 discusses limitations of the current implementation, along with other avenues for future work.

1.1 An Example Language

Figure 1 shows the SASyLF description of the abstract syntax (in familiar BNF) for the simply-typed λ-calculus with the addition of booleans[4], $\lambda_{\rightarrow B}$. Terminals of the language are listed explicitly for the aid of the parser, and the student user. Terms `t` and types `T` are defined. The notation `t[x]` appearing in the abstraction production `lam x:T dot t[x]` means that object variable `x` may appear free in term `t`, and is the very same variable bound in the abstraction. To use variables, a syntax production must be provided for them. Furthermore, a hypothetical context `Gamma` (although the name can be different) must be defined to contain free variables. Judgments and theorems which refer to `Gamma`—the latter corresponding to (3) in the previous section—do so explicitly, with **`assumes`** `Gamma`.

Figure 2 shows relations written as judgments in SASyLF, the first describing the operational semantics, and the second describing the type system, of $\lambda_{\rightarrow B}$. Each judgment is given a name and a form, followed by a set of inference rules, each of which defines an instance of the judgment in its conclusion. The typing judgment in particular depends on the hypothetical context `Gamma`, though only `T-Abs` adds assumptions to `Gamma` in this language. The remainder of the rules have been omitted for brevity.

[3]The first author used SASyLF as a student, and wrote "where" clauses as comments in every proof for these stated benefits, even before SASyLF could parse or verify them.
[4]This example language is a reformulation of one from the original SASyLF paper [1], in combination with languages from Pierce [7].

```
judgment eval: t -> t

  ----------------------------- E-IfTrue
  if true then t2 else t3 -> t2
...

judgment typing: Gamma |- t : T
assumes Gamma

  -------------------- T-True
  Gamma |- true : Bool

  Gamma |- t1 : Bool
  Gamma |- t2 : T
  Gamma |- t3 : T
  ---------------------------------- T-If
  Gamma |- if t1 then t2 else t3 : T

  Gamma, x:T1 |- t2[x] : T2
  -------------------------------------- T-Abs
  Gamma |- lam x:T1 dot t2[x] : T1 -> T2
...
```

Figure 2: Some evaluation and typing rules for $\lambda_{\rightarrow B}$

1.2 A Proof Example

The proofs for the type soundness of this language—progress and preservation—can be easily written in SASyLF, following those written from Pierce [7]. In fact, many of the language meta-theory proofs in Pierce's book use only the techniques described in §1.

Figure 3 shows the beginning of a proof of type preservation for $\lambda_{\rightarrow B}$. There are two explicit inputs to the theorem, the derivations `d: Gamma |- t : T` and `e: t -> t'`. There are also three *implicit* inputs—`t`, `T`, and `t'`—which `d` and `e` depend upon. The arbitrary hypothetical context `Gamma` is not an input to the theorem, but a repository of hypothetical assumptions which may be extended during the proof. An oddity in this theorem is that `e` does not mention or assume `Gamma`, so in fact `t` and `t'` do not depend on it. Some proofs of preservation for the simply-typed λ-calculus are written for closed terms—`d: * |- t : T` in this SASyLF representation—but writing the theorem with an arbitrary `Gamma` makes it easier to apply.

The proof in Figure 3 begins by declaring it will use induction on derivation `d`. Semantically this signifies structural induction[5], which means that the user is allowed to apply the theorem being proved, during its proof, to a subderivation of `d` with similar LF type.

The proof proceeds via case analysis[6] on `d`. When a case analysis is performed on a derivation, each the inference rules in the language description which could have produced that derivation

[5]SASyLF has different options to allow induction on multiple derivations at once, but this flexibility is not needed here and is outside of the scope of this paper.
[6]The two lines **`use induction on`** `d` and **`proof by case analysis on`** `d:` could be combined with the syntactic sugar **`proof by induction on`** `d:`.

```
theorem preservation:
assumes Gamma
  forall d: Gamma |- t : T
  forall e: t -> t'
  exists Gamma |- t' : T.
use induction on d
proof by case analysis on d:
case rule
  -------------------------- T-True
  _: Gamma |- true : Bool
is
  proof by contradiction on e
end case
case rule
  d1: Gamma |- t1 : Bool
  d2: Gamma |- t2 : T
  d3: Gamma |- t3 : T
  ------------------------------------- T-If
  _: Gamma |- if t1 then t2 else t3 : T
is
  proof by case analysis on e:
    case rule
      -------------------------------- E-IfTrue
      _: if true then t' else t3 -> t'
    is
      proof by d2
    end case
...
```

Figure 3: The beginning of a preservation proof for $\lambda_{\rightarrow\mathbf{B}}$

must be addressed with a case. Here, d is a typing derivation, so all of the language's typing rules potentially provide cases. Since the term and type mentioned in d are written without any particular form, any typing rule could apply[7]. Many of these cases lead to an immediate contradiction (such as the T-True case shown), because of derivation e, also present in the context. This derivation says t must evaluate, so cases where t is a normal form—such as an abstraction or true—do not apply to the proof. SASyLF does not "look ahead" in any part of the proof, however, and these normal form cases must still be written out.

In the case for T-If, t is the if-expression if t1 then t2 else t3 (not a normal form), where t1, t2, and t3 are new terms in the context, with types given by the premises of the rule case. (These premises are added to the context as well.) Unlike t, the rule case does not impose any further restrictions on T; the meaning behind these restrictions are discussed in §2.

The proof immediately proceeds with a case analysis on e, the evaluation derivation. Again, a case must appear for every evaluation inference rule that applies. But the t in e: t -> t' has changed since the theorem began. It can no longer be *any* term, it must have the form if t1 then t2 else t3. As a result, not all the evaluation rules could have produced e here; from Pierce [7], the rules which apply are E-IfTrue, E-IfFalse, and E-If.

The proof shows the case for the first of the three rules, and completes the proof of that case in a single step. Notably, none of the techniques from §1 are used. This is because the required derivation is already in the context; it just needed to be pointed out.

It may not be clear why derivation d2 proves this case. The theorem requires that t' has type T, but d2 gives the type of t2. But term t2 *became* t' in the inner rule case, E-IfTrue. This was required for e to match—i.e., to *unify* with—the conclusion of the original rule E-IfTrue in Figure 2.

This lack of clarity—of just what exactly needs to be proven, and how to get there—stems from the various substitutions going on "under the hood" of the proof. "Where" clauses, detailed in the remainder of this paper, bring these substitutions to light.

1.3 A Where Clause Example

Figure 4 shows the same SASyLF proof segment with "where" clauses added. The single clause for the rule case T-If is not surprising: if rule T-If provides the type for t, then t must be an if-expression. The evaluation rule E-IfTrue, however, describes a particular evaluation which imposes further restrictions on t, and notably relating t2 and t'. The added "where" clauses make these restrictions clear. From them it can be seen that all previous derivations that mention t' are also talking about t2, and vice versa.

Thus, "where" clauses are a usability feature which require implicit information to be made explicit, for the sake of learning how to write proofs. As such, they align with SASyLF's original design philosophy.

Coq [3] (along with other tactic-based proof systems) generate some relevant substitutions from an inversion or induction and make them visible during the interactive proof process. Unlike this extension of SASyLF, such systems typically do not leave this information in the proof script stored with the proof.

2 DEFINING CORRECTNESS

These examples have been written to be as clear as possible. In the wild, the user can write proofs in many correct ways. The burden is on SASyLF to judge between what is dubious (and try to nudge the user in a better direction), and what is just wrong (and tell them to try again).

What does it mean for a "where" clause to be correct? It turns out this is closely related to what it means for a case analysis to be correct, and the latter is really three questions:

(1) Which cases apply?
(2) Have all cases been covered?
(3) Are the cases which need to be addressed written correctly in the proof?

Answering these questions requires a more formal presentation of SASyLF's internals than has been given so far, and will lead to how to determine "where" clause correctness.

[7] With the possible exception of T-Var (not shown), depending on how it is written.

```
theorem preservation:
assumes Gamma
  forall d: Gamma |- t : T
  forall e: t -> t'
  exists Gamma |- t' : T.
use induction on d
proof by case analysis on d:
case rule
  -------------------------- T-True
  _: Gamma |- true : Bool
  where t := true
  and T := Bool
is
  proof by contradiction on e
end case
case rule
  d1: Gamma |- t1 : Bool
  d2: Gamma |- t2 : T
  d3: Gamma |- t3 : T
  ------------------------------------ T-If
  _: Gamma |- if t1 then t2 else t3 : T
  where t := if t1 then t2 else t3
is
  proof by case analysis on e:
    case rule
      -------------------------------- E-IfTrue
      _: if true then t' else t3 -> t'
      where t1 := true
      and t2 := t'
    is
      proof by d2
    end case
...
```

Figure 4: Where clauses added to the proof segment

2.1 LF Representation

In LF terms, the object language description consists of term constructors c and type constructors a, both LF constants. The **syntax** declaration

```
t ::= x | lam x:T dot t[x] | t t
    | true | false | if t then t else t
```

corresponds to the LF declarations[8]

$$a_{\mathtt{t}} :: \mathtt{type} \qquad c_{\mathtt{true}} : a_{\mathtt{t}}$$
$$c_{\mathtt{lam}} : a_{\mathtt{T}} \to (a_{\mathtt{t}} \to a_{\mathtt{t}}) \to a_{\mathtt{t}} \qquad c_{\mathtt{false}} : a_{\mathtt{t}}$$
$$c_{\mathtt{app}} : a_{\mathtt{t}} \to a_{\mathtt{t}} \to a_{\mathtt{t}} \qquad c_{\mathtt{if}} : a_{\mathtt{t}} \to a_{\mathtt{t}} \to a_{\mathtt{t}} \to a_{\mathtt{t}}$$

There is no constructor for variables x; SASyLF simply associates the name prefix x with the syntax type $a_{\mathtt{t}}$. (Of course, this is entirely separate from the object typing system, which is internally represented as LF dependent types.) In more general terms, a SASyLF syntax declaration creates a type constructor a of LF base kind type, along with a term constructor c_i for every non-variable production i.

A **judgment** declaration J, on the other hand, creates a type constructor a_J which is typically not kind type, because the form of a judgment usually contains meta-variables. For example, the judgment eval: t -> t creates the constructor

$$a_{\mathtt{eval}} :: a_{\mathtt{t}} \to a_{\mathtt{t}} \to \mathtt{type}$$

A SASyLF inference rule R is stored as

$$\frac{a_i\ \{\overline{\eta}\}_i :: \mathtt{type} \qquad a_j\ \{\overline{\eta}\}_j :: \mathtt{type} \qquad \ldots}{a_J\ \{\overline{\eta}\}_J :: \mathtt{type}}\ R$$

where a_J in the conclusion matches the type constructor in the judgment declaration. If R has any premises (many inference rules do not), they are also instances of judgments, and not syntax constructs on their own. Each set $\{\overline{\eta}\}$ represents a full list of arguments to its type constructor, hence each derivation has kind type. The constructors for the premise and conclusion derivations need not be different (which would correspond to mutually dependent relations). In fact, they are often all the same, as is the case for both the typing rules with premises in Figure 2; in each rule, both the premise(s) and the conclusion have constructor $a_{\mathtt{typing}}$. The omitted evaluation rules for $\lambda_{\to\mathrm{B}}$ which contain premises would be similar.

2.2 Case Analysis Correctness

As mentioned in §1, a SASyLF theorem, together with its proof, is internally represented as a function. A theorem has inputs $x_1 : \tau_1$, $x_2 : \tau_2, \ldots, x_n : \tau_n$ contained in a local context Γ, and an output type τ.

A case analysis can be performed on any single syntax construct or derivation $d \in \Gamma$. A case analysis also has an output derivation type τ' which need not be the same[9] as τ; if different, the proof will continue after the case analysis is finished.

When performed on a derivation[10] $d : a_J\ \{\overline{\eta}\}_d$, the cases which need to be covered are all inference rules in the language description whose conclusions unify with $a_J\ \{\overline{\eta}\}_d$. These rules represent all of the possible final steps in the derivation (or proof) of $a_J\ \{\overline{\eta}\}_d$. In general, the context of the case analysis may already imply some substitutions σ as explained presently; these are applied to the derivation's type before unification.

Definition 2.1 (Case Analysis Subject) *Let $d : a_J\ \{\overline{\eta}\}_d \in \Gamma$ be a derivation on which a case analysis is performed in a SASyLF theorem. Let σ be a set of substitutions in effect at the location of the case analysis. Then the application $\sigma(a_J\ \{\overline{\eta}\}_d)$ is referred to as the* case analysis subject (CAS).

It is possible that the derivations in an inference rule R, as they are written by the user, share free variable names with the CAS. Such name clashes carry no semantic meaning, but could interfere

[8]The arrow $\to$ is used here, and in the example type constructor $a_{\mathtt{eval}}$, because there are no dependencies between the types of the inputs. In general, Π-notation is needed to describe LF types and kinds which are functions.

[9]All of the case analyses in the example preservation proof begin with **proof by case analysis**. The keyword **proof** is syntactic sugar for spelling out a derivation with the output type for the theorem, such as _: Gamma |- t' : T for the one in Figure 3.

[10]Of course, a case analysis can be performed on a syntax construct as well, such as t or T from $\lambda_{\to\mathrm{B}}$. Syntax case analyses are outside the scope of this paper, because where clauses for them would be trivial and redundant.

with unification, and so should be avoided. To check whether R needs to be addressed in a case analysis on d then, a copy R_f should be should be made of R which contains only fresh[11] free variables. If R_f's conclusion is $a_J\ \{\overline{\eta}\}_f$, then R must be addressed if there exists a (unifier) substitution σ_d such that

$$\sigma_d(a_J\ \{\overline{\eta}\}_f) = \sigma_d(\sigma(a_J\ \{\overline{\eta}\}_d)) \qquad (1)$$

In general, unifying with R_f's conclusion will impose restrictions on the free variables of the CAS; these are implied by supposing that the CAS's proof finishes via R. It is possible that the CAS is also more specific in some ways than the conclusion of R_f. Thus, such a unifier σ_d is not always one-directional.

Definition 2.2 (Case Analysis Completeness)
Let $\sigma(a_J\ \{\overline{\eta}\}_d)$ be the CAS of a rule case analysis with output type τ'. Let R_f be a "fresh" copy of inference rule R, such that no variable names are shared between R_f and the CAS.
The rule case analysis is complete *if:*

(i) Every rule R is addressed within, such that the conclusion of its fresh version R_f unifies with the CAS.
(ii) The proof function produces a derivation with type τ' within each case.

It is possible that no "fresh" rule conclusions unify with $a_J\ \{\overline{\eta}\}_d$; this means that the complete case analysis has no cases. This is what occurred in the rule case `T-True` in Figure 3. In SASyLF, **`proof by contradiction on`** `e` is syntactic sugar for an empty case analysis on `e`. ∎

This definition answers questions (1) and (2) from the beginning of this section. What about question (3)? When is a rule case itself written correctly? As it turns out, there are many ways to write them incorrectly—that is, in such a way as would introduce unsoundness into the proof.

Definition 2.3 (Rule Case Conclusion)
Suppose a unifier σ_d exists for fresh version R_f of inference rule R, satisfying equation (1). Given a user-written rule case R' addressing rule R, the conclusion of R' is referred to as the rule case conclusion *(RCC).*

To be sure that R' is sound, σ_d must not map any free variables of the RCC. The substitution σ_d can be altered to comply, if it does not already, in a process described in §3.1. But if σ_d cannot be made to comply with this requirement, this means that the RCC includes free variables which σ_d is about to substitute away, and this is an error.

Given a rule case R' and unifier σ_d which do not exhibit this error, a correct rule case for R in a case analysis on d can be computed with $\sigma_d(R_f)$—that is, R_f with σ_d applied to all of its premises and conclusion. For a user-written rule case R' addressing rule R to be correct, then, it must be written in exactly the same way as $\sigma_d(R_f)$, except that free variables can be renamed from one to the other in a one-to-one fashion.

Definition 2.4 (Rule Case Correctness)
Let R' be a user-written rule case addressing rule R in a case analysis. Let σ_d be a unifier of the CAS and the conclusion of fresh version R_f of R.
The rule case R' is correct *if:*

(i) None of the free variables of the RCC are mapped by σ_d.
(ii) There exists a "bijection" unifier

$$\sigma_c = \{u_1 \mapsto w_1, u_2 \mapsto w_2, \cdots, u_m \mapsto w_m\}$$

such that

$$R' = \sigma_c(\sigma_d(R_f)) \qquad (2)$$

where every u_i is a free variable in $\sigma_d(R_f)$ and w_i is the corresponding free variable in R'.
(iii) The free variables w_i in the codomain of σ_c do not share names with other members of the local context Γ. (This would imply relationships between R' and those members which may not be sound.)

The meaning behind the bijection unifier σ_c is that a correctly-written rule case R' represents exactly the level of restriction on the free variables of the CAS which is required by supposing inference rule R is the last rule applied in the CAS's proof.

If a unifier σ_c exists, but it is not a bijection, it is either because R' is "too general" (it does not impose enough restrictions on the free variables of the CAS), or because R' is "too strict" (it imposes too many). The former occurs if R' contains free variables which are not needed—that is, they stand for elements of $\sigma_d(R_f)$ which are already known to be more specific than the variable chosen. This includes when multiple free variables in R' are used to stand for a single free variable in $\sigma_d(R_f)$. On the other hand, R' is "too strict" when a free variable should have been used in R', to allow flexibility in what the variable stands for in $\sigma_d(R_f)$, but it was not. This includes when the same variable is used twice in R', when two different variables should have been used. If R' is too strict, an error is generated; if too general, a warning. Both possibilities can occur in one incorrectly written rule case; if this occurs, SASyLF reports the error.

If no unifier σ_c exists at all between R_f and R', then rule case R' does not address inference rule R, and SASyLF asks the user to try again. ∎

This definition of correct rule cases sheds light on the nature of the substitutions which arise from them.

Suppose rule case R' is written correctly to address inference rule R, and so a bijection unifier σ_c exists. By equation (2), considering only the conclusions of R' and $\sigma_d(R_f)$, the RCC can be described as

$$\text{RCC} = \sigma_c(\sigma_d(a_J\ \{\overline{\eta}\}_f)) \qquad (3)$$

where $a_J\ \{\overline{\eta}\}_f$ is the conclusion of R_f. Equations (1) and (3) then combine to form

$$\text{RCC} = \sigma_c(\sigma_d(\sigma(a_J\ \{\overline{\eta}\}_d))) = (\sigma_c \circ \sigma_d)(\sigma(a_J\ \{\overline{\eta}\}_d)) \qquad (4)$$

Recall that $\sigma(a_J\ \{\overline{\eta}\}_d)$ is none other than the CAS.

Definition 2.5 (Rule Case Substitutions)
The set of substitutions imposed by a correctly written rule case R' whose conclusion satisfies equation (4) is

$$\sigma_u \triangleq \{\overline{v \mapsto \eta_v}\} \subseteq (\sigma_c \circ \sigma_d)$$

where each v is a free variable of the CAS, is a one-way unifier from the CAS to the RCC.

[11] An easy way to obtain such variables is to create names for them which the user cannot write.

This unifier σ_u represents the restrictions imposed on free variables v of the CAS as a consequence of addressing inference rule R particularly with rule case R'. The composition $\sigma_c \circ \sigma_d$ may contain mappings from free variables of R_f, but these are irrelevant to the unification of the RCC and the CAS, and by extension the remainder of the proof; because of this, these mappings are not included in σ_u.

The restrictions described by σ_u do not only affect the CAS; they affect every member of the local context Γ containing free variables in σ_u's domain. In essence, $\sigma_u = \{\overline{v \mapsto \eta_v}\}$ instantiates all appearances of every free variable v across members of Γ with the more specific expression η_v; as a consequence, the v's should "disappear" inside the scope of rule case R'.

Furthermore, this substitution effect is cumulative with successive, nested case analyses. If a case analysis is performed inside the first, another unifier σ'_u is exists for each case, and σ'_u is applied to all elements of σ_u Γ. In other words, inside the inner case, the substitution $\sigma'_u \circ \sigma_u$ is applied to all elements of Γ.

Definition 2.6 (**Local Context Substitutions**)
Let Γ be the initial local context of a SASyLF theorem (i.e., the theorem's inputs). At a location L of the theorem's proof, assume

$$\sigma_u^k, \cdots, \sigma_u^2, \sigma_u^1$$

are substitutions imposed by k nested case analyses whose syntactic context encompasses L, where σ_u^1 represents the substitution for the case at outermost scope. These nested case analyses imply a succession of composed unifiers

$$\sigma \triangleq \sigma_u^k \circ \cdots \circ \sigma_u^2 \circ \sigma_u^1 \tag{5}$$

which is applied to each member of Γ, as well as to any new member of the local context which yet remains.

Therefore, σ as it appeared in equations (1) and (4) represents the successive composition of substitutions implied by all cases, or other statements (such as inversions) that cause variable substitution, whose syntactic context encompasses rule case R'.

The presence of such outer-scope substitutions is why, for example, the inner case analysis on `e` in Figure 3 requires cases only for rules `E-IfTrue`, `E-IfFalse`, and `E-If`, and not for all of the evaluation rules of $\lambda_{\rightarrow\mathrm{B}}$. ■

It is noted above that the RCC must not mention any free variables mapped by σ_d. Furthermore, the existence of a bijection unifier σ_c as defined above implies that the RCC does not mention any free variables mapped by σ_c, either. Additionally, the RCC must not reuse any free variables mapped by σ; if it does, this is always an error, for σ cannot be altered.

In summary, following are the requirements for a correct rule case analysis, written as answers to the questions posed at the beginning of the section. In accordance with the observation above, a substitution σ is assumed to be in effect due to (enclosing) case analyses currently in scope; $\sigma = \varnothing$ at the outset of a proof. Also assume a local context Γ. Finally, assume the subject of the case analysis is derivation $d : a_J\ \{\overline{\eta}\}_d \in \Gamma$, and the output of the case analysis is of type τ'.

(1) An inference rule R (as opposed to syntax productions, for a syntax case analysis) applies to the case analysis if the conclusion of a "fresh" version R_f unifies with $\sigma(a_J\ \{\overline{\eta}\}_d)$ via unifier σ_d.
(2) All cases are covered when each rule from (1) has a correctly written rule case, followed by the production of the target derivation being proved by the case analysis.
(3) A rule case R', addressing inference rule R, is written correctly if:
   (a) There exists a "bijection" unifier σ_c which maps free variables of $\sigma_d(R_f)$ to free variables in R'. The codomain of σ_c must be disjoint from Γ.
   (b) The conclusion of R' (the RCC) does not mention any free variables which have been substituted away by enclosing cases, *including R' itself.* That is,

$$\mathrm{RCC} = \sigma(\mathrm{RCC}) = \sigma_d(\mathrm{RCC}) = \sigma_c(\mathrm{RCC})$$

To show the meaning of requirement (3b), consider the RCC for the inner rule case `T-If` in Figure 3

```
_: if true then t' else t3 -> t'
```

If this RCC had been written either as

```
_: if true then t2 else t3 -> t'
```

or as

```
_: t -> t'
```

neither would satisfy this last requirement. The term `t` was substituted away in an outer case (via σ), while `t2` is about to be substituted away in this case (via $\sigma_u \subseteq \sigma_c \circ \sigma_d$).

Interestingly, requirement (3) allows the user to rename free variables of the CAS when writing the RCC, as long as the new names are not already members of Γ.

2.3 Where Clause Correctness

Before correctness for "where" clauses is defined, it is important to note that unlike case analysis correctness, "where" clauses have no effect on the semantics or soundness of the proof in which they appear. That is, if correctness for these clauses is incorrectly or insufficiently defined or implemented, the soundness of current and future SASyLF proofs are not affected. An exception to this is the **`inversion`** construct; "where" clauses associated with inversions could have semantic effect on the remainder of the proof. For now, "where" clauses for inversions are left to future work.

The notion of "where" clauses benefits from the more formal description of case analyses in the previous section. Specifically, these clauses must be written to make explicit the restrictions imposed by substitutions σ. There are several considerations which complicate the requirements for "where" clauses. They are addressed in the following sections.

2.3.1 *Nested Case Analyses.*

For a single case analysis, there is only one σ_u in the composition σ. For nested case analyses, however, there are multiple substitutions in play; which should correct "where" clauses represent? Looking back at the definition of σ (5), there are two viable options.

The clauses could represent the full substitution σ. However, they are more succinct if they describe only σ_u^k, the last substitution imposed by a case. In other words, the latter version of "where" clauses describes only the most recent restrictions, as opposed to repeating old information. Thus, this more succinct version is the

```
(<CASE> <RULE>
  (<ID> ":" <EXPR>)* // premises
       <BAR>
   <ID> ":" <EXPR> // conclusion
  (<WHERE> <LHS> ":=" <RHS>
    (<AND> <LHS> ":=" <RHS>)*)?
<IS>
  (<DERIVATION>)+ // continuation of proof
<END> <CASE>)*
```

Figure 5: The abstract syntax of cases in a rule case analysis, including the addition of "where" clauses

one implemented in the new version of SASyLF. For example, the nested case `E-IfTrue` in Figure 4 could have (only) the clause

```
where t := if true then t' else t3
```

which reflects the entire composition σ of substitutions for this rule case; but it is more useful to require clauses that represent only the newest mappings:

```
where t1 := true and t2 := t'
```

2.3.2 *SASyLF Syntax.*

Chief among remaining considerations is how correct "where" clauses should fit into SASyLF's abstract syntax, including the form of the clauses themselves. In Figure 4, they immediately follow an RCC and precede **is**; this syntax is generalized[12] in Figure 5. This is the ideal location for the clauses in the code, because they describe substitutions which occur as a result of the RCC; in particular, the right-hand sides of the clauses must all appear in the RCC. Furthermore, the "where" clauses are listed just before the section of the proof affected by the substitutions they describe, similarly to way "let"-bindings appear in other languages.

2.3.3 *Familiarity.*

Another consideration is that "where" clauses must only ever list variables and expressions which have already been seen in the proof text. They must never introduce anything new; the clauses should decrease confusion, not increase complexity. "Where" clauses are intended to describe $\sigma_u = \{\overline{v \mapsto \eta_v}\}$, where the v's are free variables in the CAS. Therefore, the left- and right-hand sides (`<LHS>`, `<RHS>`) of a correct clause must correspond to the "unparsed" (concrete syntax) versions of LF expressions v and η_v, respectively.

2.3.4 *First- vs. Second-Order Left-Hand Sides.*

For first-order "where" clauses, the left-hand side must simply be the concrete name represented by v. SASyLF includes support for second-order[13] (and no higher) free variables, and "where" clauses describing substitutions on them are slightly more verbose. Figure 6,

```
lemma substitution-preserves-typing:
assumes Gamma
  forall d1: Gamma |- t1 : T1
  forall d2: Gamma, x:T1 |- t2[x] : T2
  exists Gamma |- t2[t1] : T2.
  proof by induction on d2:
case rule
  ----------------------------- T-True
  _: Gamma, x:T1 |- true : Bool
  where t2[x] := true
  and T2 := Bool
is
  proof by rule T-True
end case
...
```

Figure 6: A lemma with second-order free variables

showing the beginning of a familiar lemma[14], also shows a simple second-order "where" clause:

```
where t2[x] := true
```

Whenever a second-order free variable v appears in SASyLF's syntax, it is immediately followed by explicit arguments, each enclosed in `[]`. At the object language level, if such an argument is a bound variable `x`, it acts as a visual marker that the bound variable `x` may be free in the object term represented by v (as described in §1.1). Internally, this `[]` notation is represented with an LF application with v at the head. The left-hand side of v's "where" clause must list v's arguments as they appear in the CAS, modulo α-equivalence of the *whole* clause and the original mapping $v \mapsto \eta_v$. In the above example, it would be inaccurate to allow

```
where t2 := true
```

letting the `[x]` be forgotten.

For a less simple example, suppose the LF mapping

$$\texttt{t2} \mapsto \lambda y{:}a_{\texttt{t}}.(c_{\texttt{lam}}\ \texttt{T1'}\ \lambda z{:}a_{\texttt{t}}.(\texttt{t21}\ y\ z))$$

is present in σ_u for a given rule case[15]. Then

```
where t2[x] := lam x':T1' dot t21[x][x']
```

is a correct "where" clause representing this mapping. By α-equivalence,

```
where t2[x'] := lam x:T1' dot t21[x'][x]
```

is also correct, but

```
where t2[x] := lam x':T1' dot t21[x'][x]
```

is not, because this right-hand expression is not the same as the LF expression above (`t21` $z\ y$ is not the same as `t21` $y\ z$, all else being equal). Neither is

```
where t2[x] := lam x:T1' dot t21[x][x]
```

correct, because the LF bound variables in the mapping are distinct.

[12]The syntax shown in Figure 5 is adapted and (greatly) simplified from SASyLF's parsing specification.
[13]SASyLF stands for **S**econd-order **A**bstract **Sy**ntax **L**ogical **F**ramework.
[14]The so-called "substitution lemma" [7] is not actually required to prove complete type preservation for $\lambda_{\to B}$ in SASyLF; the **by substitution** construct may be used instead.
[15]This could occur in the substitution lemma, in the case for rule `T-Abs` (not shown).

2.3.5 *Optional Presence.*

A final consideration regarding "where" clause correctness is that their presence in the code must be optional. Proofs for complex object languages can be lengthy, with many nested case analyses; not every "where" clause in these proofs may be helpful, especially for the advanced user writing them. For novice users, however, being forced to write correct "where" clauses is a boon. For these users, writing the clauses demonstrates their understanding of the substitutions they describe, and having this information visible in the code makes continuing the proof more straightforward.

2.3.6 *Summary of Requirements.*

In summary, given a set of substitutions σ in effect at the beginning of a case analysis, a (correct) rule case R' in that analysis, and a set of new restrictions σ_u imposed by R', "where" clauses for R' are correct if:

(1) Each clause represents a distinct mapping in σ_u, instead of a mapping from the combined substitution $\sigma_u \circ \sigma$.
(2) The left-hand and right-hand sides of a clause representing a mapping $(v \mapsto \eta_v) \in \sigma_u$ must be the concrete syntax representations of LF expressions v and η_v, respectively. For second-order free variables v, a list of arguments each enclosed in `[]` must follow v's name on the left-hand side. Clauses representing mappings α-equivalent to $v \mapsto \eta_v$ are allowed.

In addition, incorrectly written "where" clauses must always yield errors, but mappings in σ_u which lack clauses should only yield errors if an option making the clauses mandatory is enabled.

3 IMPLEMENTATION

Much of the infrastructure needed to verify "where" clauses was already present in the SASyLF system prior to the feature's addition. This includes LF-expression unification[16] and case analysis verification.

3.1 Rule Case Verification

Case analysis verification in SASyLF includes tracking and applying CAS-RCC unifiers σ_u to members of contexts Γ as necessary. To accomplish this, SASyLF parses an abstract syntax (sub)tree (AST) from a theorem and proof in the source, which is traversed in depth-first fashion, visiting children in the order they appear in the source. The root of proof subtree P is associated with an empty substitution σ. Every case analysis in the proof represents a subtree of P. When a case node is entered, a new substitution $\sigma \longleftarrow \sigma_u \circ \sigma$ is created for that node. After verification on the case node is complete, its parent's σ is restored. When $x\!:\!\tau \in \Gamma$ are accessed at any node of the proof, the σ associated with that node is applied to τ first. All of this machinery was in place before "where" clauses were conceived; these substitutions play a critical role in SASyLF's proof verification process.

A side effect of adding the new feature to SASyLF was looking more closely at this implementation; the results of this research are summarized in §2.2. Errors were found in the verification of rule cases, in particular relating to the use of free variables. Prior to this work, cases which were "too general" or which included free variables about to be substituted away sometimes went undetected.

Following is a description of the of new process for rule case verification, which refers to the work in §2.2. For this process, assume that once an error is reported, the procedure is finished; further errors are not sought. When verifying a rule case R' addressing inference rule R, the first step is to check that $R' = \sigma(R')$, where σ is the composition of substitutions in effect at the outset of the case analysis. If this equality fails, the error is reported.

Next, σ_d is computed by unifying the CAS (to which σ has already been applied) and the conclusion of a fresh instance R_f of the rule R. If this unification fails, it is reported that R' is unnecessary. Otherwise, σ_d is "rotated" to preserve (not map) free variables of the RCC (the conclusion of R').

This rotation of a substitution is generalized in an algorithm called SELECTUNAVOIDABLE. This algorithm takes as input a substitution σ and a set of free variables V. Each free variable $v \in V$ is checked if it can be "avoided" by σ—i.e., removed from the domain of σ, if present there. For each v. this is possible (1) if v is not in the domain of σ to begin with, or (2) if $\sigma(v) = \eta_v$ is η-equivalent to a free variable $z \notin V$. In the latter case, the mapping $v \mapsto \eta_v$ is "rotated" to become $z \mapsto v$, altering σ as a side effect. This rotation is nontrivial in general, and can affect the other mappings in σ via composition with the new one. After all $v \in V$ have been checked in this way, the algorithm returns a set of free variables $S \subseteq V$, those which could not be avoided.

The specific rotation of σ_d above is achieved by gathering the free variables of the RCC into a set V and executing SELECTUNAVOIDABLE(σ_d, V). If the resultant set S is not empty, R' is unsound. Otherwise, the substitution σ_d resulting from this operation is applied to produce the correct rule case candidate $\sigma_d(R_f)$. Unification is attempted with this candidate and R'. If it fails entirely, R' does not correctly address R. If a unifier σ_c is found, SELECTUNAVOIDABLE is executed on it twice to establish a bijection (the order of the two executions matters): first avoiding the free variables of R_f, then avoiding the free variables of R'. If the resultant set S from the first execution is non-empty, then R' is "too strict." If S from the second execution is non-empty, then R' is "too general." If both executions return empty sets, the codomain of σ_c is intersected with the local context Γ; if the result not $\emptyset$, an error is generated. Otherwise, R' is correctly written, and σ_u is the set of all mappings in $\sigma_c \circ \sigma_d$ which act on free variables of the CAS.

3.2 Where Clause Verification

To verify "where" clauses, the new version of SASyLF parses each of the user-written clauses into two LF expressions (the left and right sides). It then matches them, via LF expression equality, to mappings in σ_u. (If the rule case is not correct and σ_u does not exist, "where" clauses for that case are not verified.)

For second-order "where" clauses, arguments in `[]` are parsed from the left-hand side into a list of variable bindings; these are made available when parsing the right-hand side, as if bound on that side. The user's right-hand LF expression is then wrapped with lambda abstractions corresponding to the left-hand arguments; the last argument forms the first wrapping, and so on. The right-hand side is then verified via LF expression equality just as with a

[16] SASyLF implements Nipkow's unification algorithm [4], with additional conservative heuristics for unifying non-pattern applications.

first-order clause, and α-equivalence is allowed. If the user gives non-variable arguments, not enough arguments, or too many, appropriate errors are given. A special error is generated if there are arguments on the left-hand side of a first-order clause.

4 FUTURE WORK

The primary avenue for future work with "where" clauses should be usability testing with actual users, preferably students learning to use SASyLF and to write sound proofs. The feature seems worthy of inclusion (and has led to many interesting subproblems and bug fixes), but it is not currently known whether student users will find "where" clauses helpful or obtrusive.

4.1 Current Limitation

There is one major limitation to the current "where" clause implementation: SASyLF does not verify "where" clauses when changes occur in the hypothetical context from a CAS to the RCC. This is due the way these contexts are internally represented, via additional abstractions wrapped around an LF expression in the context. There are potential plans to revamp this representation, which would also change the way these clauses are handled.

4.2 Extensions

The new version of SASyLF parses the user's "where" clauses to LF, and verifies them at that level. An extension of this feature is to produce the concrete clauses internally and insert them into the user's code; this can be accomplished with an Eclipse "Quick Fix" option. The cases for a case analysis can already be generated and inserted in this way, which is similar to a feature described in the original SASyLF paper [1].

Another extension for "where" clauses lies with inversions, a feature in SASyLF which allows the proof-writer to perform a case analysis with exactly one applicable case in-line. This construct immediately alters the local substitution σ, and this alteration remains in effect until the end of the given case in a proof. In this way, the `inversion` construct behaves similarly to a "let" construct in other languages. Because of the alterations to σ, inversions should include "where" clauses just as rule cases do. In fact, the clauses may be even more important for inversions, since they do not explicitly list an RCC in their syntax.

5 CONCLUSION

"Where" clauses in SASyLF provide a means of making previously hidden substitutions in a proof explicit to the user. These substitutions represent restrictions that occur when answering a case in a case analysis, and have a pervasive effect on the remainder of the proof within that case. Making these substitutions explicit should make learning to write proofs with the assistant easier for student users, and thus aligns with SASyLF's education-focused design philosophy.

REFERENCES

[1] Jonathan Aldrich, Robert J Simmons, and Key Shin. 2008. SASyLF: An educational proof assistant for language theory. In *Proceedings of the 2008 international workshop on Functional and declarative programming in education*. ACM, 31–40.
[2] Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A framework for defining logics. *Journal of the ACM (JACM)* 40, 1 (1993), 143–184.
[3] The Coq development team. 2016. *The Coq proof assistant reference manual*. LogiCal Project. `http://coq.inria.fr` Version 8.6.1.
[4] Tobias Nipkow. 1992. Functional Unification of Higher-Order Patterns. In *Proceedings of Sixth International Workshop on Unification Schloss Dagstuhl, Germany*. 77.
[5] F. Pfenning and C. Elliott. 1988. Higher-order Abstract Syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*. ACM, New York, NY, USA, 199–208. `https://doi.org/10.1145/53990.54010`
[6] Frank Pfenning and Carsten Schürmann. 1999. System Description: Twelf-A Meta-Logical Framework for Deductive Systems. In *Proceedings of the 16th International Conference on Automated Deduction*. Springer-Verlag, 202–206.
[7] Benjamin C Pierce. 2002. *Types and programming languages*. MIT press, Cambridge, Massachusetts, USA and London, England.
[8] Carsten Schürmann. 2000. *Automating the Meta Theory of Deductive Systems*. Ph.D. Dissertation. School of Computer Science, Carnegie Mellon University.

Kripke-Style Contextual Modal Type Theory

Yuito Murase
The University of Tokyo
Tokyo, Japan
murase@lyon.is.s.u-tokyo.ac.jp

ABSTRACT

Under the Curry-Howard isomorphism, modal operators correspond to the type of closed code. Nanevski et al. generalized this result and proposed the contextual modal type theory. They introduced the notion of context that corresponds to free variables of code. Therefore the contextual modal type theory treats open code.

This paper provides another formulation of contextual modal type theory: Kripke-style contextual modal type theory. Our type system is based on the Kripke-style formulation of modal logic, whereas the original system is based on the dual-context formulation. The resulting system has Lisp-like quasiquotation, and hence we expect that KCMTT is adequate for the basis of syntactical metaprogramming.

KEYWORDS

Contextual Modal Type Theory, Lambda Calculus, Modal Logic, Metaprogramming

1 INTRODUCTION

The theory of modal calculi, which corresponds to intuitionistic modal logic through the Curry-Howard correspondence [11], have been studied since 1990s [1, 6, 8, 9]. It is known that some modalities correspond to types of closed code, that is, code without free variables. For example, the type $\Box A$ represents closed code that will be evaluated to the value of type A. From this perspective, modal calculi have been studied as a foundation for staged computation and run-time code generation [3].

The main restriction of modal calculi is that they can manipulate only closed codes. Nanevski et al [7] proposed a solution to this problem, as Contextual Modal Type Theory(CMTT). Contextual modal types are a generalized notion of modal types. They are allowed to have an environment in a modal operator. For example, the type $[x\colon A, y\colon B]C$ represents code that will be evaluated to the value of type C, *under the environment* $x\colon A, y\colon B$. As you can see, modal types are the special case of contextual modal types, where environments are always empty (and therefore code is closed). CMTT is based on Pfenning and Davies' dual-context modal type system [8] (we borrow the name 'dual-context' from Kavvos [4]). Their modal type system corresponds to S4 modal logic, and therefore CMTT corresponds to S4 modal logic.

In this paper, we propose another type system for CMTT. To distinguish from the original CMTT, we call our type system as Kripke-style CMTT (KCMTT). As the name shows, KCMTT is a generalization of the Kripke-style modal type system[3, 6, 9], where contexts form stack and terms have Lisp-like quasi-quotation. As a result, KCMTT provides four variations that correspond to K, T, K4, S4 respectively. KCMTT is different from the original CMTT at this point. The following table shows the position of KCMTT among related work.

	Modal Type	Contextual Modal Type
Dual-Context	[8]	[7]
Kripke-Style	[6, 9]	KCMTT

The paper is structured as follows. In Section 2, we provide Kripke-style contextual modal logic, which is the logic part of KCMTT. In section 3, we give the definition KCMTT in detail and show fundamental properties. Finally, we discuss future work and our motivation for KCMTT.

2 KRIPKE-STYLE CONTEXTUAL MODAL LOGIC

Before the definition of the type system, we introduce Kripke-style natural deduction for contextual modal logic(KCML). KCML is a natural extension of Pfenning and Davies' [8] Kripke-style modal logic. The fundamental idea of KCML and Kripke-style modal logic is the Kripke-style judgment, which has a stack of context.

First, we explain the notion of Kripke-style hypothetical judgment and then construct natural deduction system. We also show that our system is well-defined, that is, introduction and elimination rules for contextual modality satisfy local-soundness and local completeness [8].

2.1 Kripke-Style Hypothetical Judgment

First, we introduce Kripke-style hypothetical judgment, a generalization of hypothetical judgment. The idea of Kripke-style hypothetical judgment is not new: Martini and Masini [6] and Pfenning and Wong [9] initially proposed Kripke-style judgment around the same time, to construct modal calculi.

In a Kripke-style hypothetical judgment, hypotheses form a stack, where semicolons separate contexts. We write $A, B, \ldots$ for propositions, Γ for hypotheses, and Ψ for a stack of hypotheses.

$$\Gamma_m; \Gamma_{m-1}; \ldots; \Gamma_1 \vdash A$$

Informally, this judgment states the following fact from the viewpoint of Kripke semantics: for arbitrary world sequence $w_m \to w_{n-1} \to \ldots \to w_1$, the proposition A holds at the world w_1 if Γ_m holds in w_m, Γ_{m-1} holds in w_{m-1}, and so on. When the context stack has single context, it is equivalent to hypotetical judgment.

In the rest of this section, we construct a natural deduction system on the Kripke-style hypothetical judgment. First, we define the following hyp rule, as we can use the assumption in the current world as the conclusion.

$$\text{hyp}\ \frac{A \in \Gamma}{\Psi; \Gamma \vdash A}$$

We construct natural deduction system of KCML in the following way. First, we define structural properties that Kripke-style judgment should satisfy. Afterward, we add rules for logical connectives so that those principles are formally proved as metatheory.

First, we define the following substitution principle. The substitution principle states that assumptions in a level can be replaced with other assumptions when we can conclude all of the former assumptions from the latter ones. This principle generalizes the usual substitution principle which substitutes a single variable. This style is useful when we reason quotations in Section 3.

Substitution Principle If $\Psi; A_1, \ldots, A_m; \Psi' \vdash B$ and $\Psi; \Gamma \vdash A_i$ holds for all $1 \leq i \leq n$, then $\Psi; \Gamma; \Psi' \vdash B$.

In addition to substitution principle, we can add two structural principles imposing some properties of world relations: reflexivity and transitivity.

As we said before, a stack of context is corresponds to a sequence of worlds. When the world relation satisfies reflexsivity, any two adjacent worlds in the stack can be same. Therefore it is natural to assume that we can merge them.

Reflexive Principle If $\Psi; \Gamma; \Gamma'; \Psi' \vdash A$, then $\Psi; \Gamma, \Gamma'; \Psi' \vdash A$.

Same discussion applies when the world relation satisfies transitivity. In this case, we can insert contexts between adjacent contexts.

Transitive Principle If $\Psi; \Gamma; \Psi' \vdash A$, then $\Psi; \ldots; \Gamma; \Psi' \vdash A$.

As a result, we have four variations of logic depending on whether we assume reflexivity and transitivity of the world relation. In classical modal logic [5], it is known that K, T, K4, and S4 modal logic correspond to those properties of the world relation. Therefore we identify symbols K, T, K4, S4 with those variations. In the rest of this paper, we write $\Psi \vdash_K A$ when we assume no properties of the world relation. We write $\Psi \vdash_T A$ when we assume reflexivity, $\Psi \vdash_{K4} A$ when we assume transitivity, and $\Psi \vdash_{S4} A$ when we assume both. We just write $\Psi \vdash A$ when we do not assume those conditions.

2.2 Kripke-Style Natural Deduction

Now we are ready to construct a natural deduction system for KCML. For simplicity, we consider the fragment with implication and contextual modality. Let us denote propositional variables with $P, Q, \ldots$. Propositions in KCML are inductively defined as follows.

Context $\Gamma ::= \cdot \mid A, \Gamma$

Propositions $A, B ::= P \mid A \to B \mid [\Gamma]A$

For a contextual modality $[\Gamma]A$, we call the formar part *context part*, and the latter *body part*.

Let us define introduction and elimination rules for logical connectives. For implications, their introduction and elimination rules are almost same as usual hypothetical judgment.

$$\to \mathrm{I}\ \frac{\Psi; \Gamma, A \vdash B}{\Psi; \Gamma \vdash A \to B} \quad \to \mathrm{E}\ \frac{\Psi; \Gamma \vdash A \to B \qquad \Psi; \Gamma \vdash A}{\Psi; \Gamma \vdash B}$$

Rules for implications are concerned with only the current world. Other worlds in the context stack are used only when we use contextual modal operator.

The introduction rule for contextual modality is defined as follows.

$$[\,]\mathrm{I}\ \frac{\Psi; \Gamma \vdash A}{\Psi \vdash [\Gamma]A}$$

Kripke's multiple world semantics justifies this rule. Let us think of a special case where $\Psi; \Gamma; \cdot \vdash A$. The current world corresponds to the arbitrary world next to Γ, and we can interpret "A holds for any world next to Γ". By the definition of modal operator in Kripke's multiple world semantics, we conclude that "$\Box A$ holds at Γ". Contextual modality generalizes modal operator to have context.

When we assume neither reflexivity nor transitivity, the corresponding elimination rule is defines as follows. This rule states that A holds in the next world assuming Γ when $[B_1 \ldots B_m]A$ holds at the current world and B_i holds in the next world for each i assuming Γ. As you can see, introduction / elimination rules for modal operator interact with context stack by pushing and popping.

$$[\,]\mathrm{E}\ \frac{\Psi \vdash [B_1 \ldots B_m]A \qquad \Psi; \Gamma \vdash B_i \text{ for } 1 \leq i \leq m}{\Psi; \Gamma \vdash A}$$

We can generalize this elimination rules to support reflexivity and transitivity as follows. Assuming reflexivity, we can identify the current world as the next world, and this corresponds to the case $l = 0$. Assuming transitivity, the lth next world is also the next world for $l > 1$.

$$[\,]\mathrm{E}_l\ \frac{\Psi \vdash [B_1, \ldots, B_m]A \qquad \Psi; \Gamma_l; \ldots; \Gamma_1 \vdash B_i \text{ for } 1 \leq i \leq m}{\Psi; \Gamma_l; \ldots; \Gamma_1 \vdash A}$$

$$\text{where} \begin{cases} l = 1 & \text{for K} \quad l = 0, 1 \quad \text{for T} \\ l \geq 1 & \text{for K4} \quad l \geq 0 \quad \text{for S4} \end{cases}$$

Pfenning and Davies[8] stated that the elimination rule should not be too strong or too weak concerning the introduction rule, and proposed two conditions that introduction/elimination rules should satisfy: local soundness and local completeness. We should confirm that introduction/elimination rules for contextual modal types satisfy these conditions.

Let us think of the case of S4. The same discussion holds for K, T, and K4. Local soundness is the property that an elimination rule is not too strong with respect to the introduction rule. This property is shown by the following local reduction pattern where $n \geq 0$. This pattern demonstrates that we can omit introduction followed by elimination. $\mathcal{D}'$ is generated from $\mathcal{D}$ and $\mathcal{E}$, with substitution, reflexive, and transitive principle.

$$[\,]\mathrm{E}_l\ \frac{[\,]\mathrm{I}\ \dfrac{\overset{\mathcal{D}}{\Psi; A_1, \ldots, A_m \vdash B}}{\Psi \vdash [A_1, \ldots, A_m]B} \qquad \overset{\mathcal{E}}{\Psi; \Gamma_l; \ldots; \Gamma_1 \vdash A_i \text{ for } 1 \leq i \leq m}}{\Psi; \Gamma_l; \ldots; \Gamma_1 \vdash B}$$

$$\Downarrow R$$

$$\overset{\mathcal{D}'}{\Psi; \Gamma_l; \ldots; \Gamma_1 \vdash B}$$

On the other hand, local completeness is the property that an elimination rule is not too weak with respect to the introduction rule. This property is shown by the following local expansion pattern. This pattern demonstrates that original judgment (in this pattern, $\Psi \vdash [A_1, \ldots, A_m]B$) can be restored after elimination.

$$\begin{array}{c}\mathcal{D}\\ \Psi \vdash [A_1,\ldots,A_m]B\end{array}$$

$$\Downarrow E$$

$$[\,]\mathrm{E}_1\,\dfrac{\begin{array}{c}\mathcal{D}\\ \Psi\vdash[A_1,\ldots,A_m]B\end{array} \qquad \text{hyp}\,\dfrac{}{\Psi;A_1,\ldots,A_m\vdash A_i}\ \ \text{for } 1\le i\le m}{[\,]\mathrm{I}\,\dfrac{\Psi;A_1,\ldots,A_m\vdash B}{\Psi\vdash[A_1,\ldots,A_m]B}}$$

We call this natural deduction system KCML, which consists of Kripke-style judgment, the hyp rule, and the introduction and elimination rules for implication and contextual modality.

2.3 Fundamental Properties

Theorem 2.1. *(1)* $\Psi \vdash_K A \Rightarrow \Psi \vdash_X A$ *for* $X \in \{T, K4, S4\}$
(2) $\Psi \vdash_T A \Rightarrow \Psi \vdash_{S4} A$
(3) $\Psi \vdash_{K4} A \Rightarrow \Psi \vdash_{S4} A$

Proof. For (1), it easy to show that derivation tree of $\Psi \vdash_K A$ is also derivation tree of $\Psi \vdash_X A$ for $X \in \{T, K4, S4\}$. Same discussion for (2) and (3). □

Finally, we formally prove that KCML satisfies the substitution, reflexive, and transitive principles.

Theorem 2.2. *(1) For* $X \in \{K, T, K4, S4\}$, *if* $\Psi; A_1 \ldots A_m; \ldots \vdash_X B$ *and* $\Psi; \Gamma \vdash_X A_i$ *holds for all* $1 \le i \le m$, *then* $\Psi; \Gamma; \ldots \vdash_X B$.
(2) For $X \in \{T, S4\}$, *if* $\Psi; \Gamma; \Gamma'; \ldots \vdash_X B$, *then* $\Psi; \Gamma, \Gamma'; \ldots \vdash_X B$.
(3) For $X \in \{K4, S4\}$, *if* $\Psi; \Gamma; \ldots \vdash_X B$, *then* $\Psi; \ldots; \Gamma; \ldots \vdash_X B$.

Proof. By induction on the derivation rules. □

2.4 Examples

We show some examples provable in KCML.

(1) $\vdash_K [C](A \to B) \to [C]A \to [C]B$
(2) $B \vdash_T [B]A \to A$
(3) $\vdash_{K4} [C]A \to [D][C]A$ (6) $\vdash_K [A]A$
(4) $\vdash_K [C]A \to [C, D]A$ (7) $\vdash_K [A]B \to [\,](A \to B)$
(5) $\vdash_K [C, C]A \to [C]A$ (8) $\vdash_K [\,](A \to B) \to [A]B$

In these examples, you can see that contextual modality generalizes modality. Each of (1), (2), (3) corresponds to the contextual version of axioms K, T, and 4. Note that (2) is derivable assuming reflexivity, and (3) assuming transitivity. Figure 1 gives the derivation tree of (1). We omit derivation trees for other examples.

$$\to\mathrm{I}\,\dfrac{\to\mathrm{I}\,\dfrac{[\,]\mathrm{I}\,\dfrac{\to\mathrm{E}\,\dfrac{[\,]\mathrm{E}_1\,\dfrac{\text{hyp}\,\dfrac{}{\Gamma\vdash_K [C](A\to B)}\qquad \text{hyp}\,\dfrac{}{\Gamma;C\vdash_K C}}{\Gamma;C\vdash_K A\to B}\qquad \begin{array}{c}\vdots\\ \Gamma;C\vdash_K A\end{array}}{\Gamma;C\vdash_K B}}{\Gamma\vdash_K [C]B}}{[C](A\to B)\vdash_K [C]A\to[C]B}}{\vdash_K [C](A\to B)\to[C]A\to[C]B}$$

where $\Gamma := [C](A \to B), [C]A$

Figure 1: Example of a Derivation Tree

With the introduction rule for contextual modality, we can identify the proposition $[\Gamma]A$ with the hypothetical judgment $\Gamma \vdash A$ (not Kripke-style hypothetical judgment). The propositions (4), (5) and (6) represent weakening, contraction, and the hyp rule, demonstrating this idea.

The propositions (7) and (8) show that contextual modality is equivalent to modality with implication. In this sense, contextual modal logic is not stronger than modal logic. However, we think that contextual modality enables us a finer analysis of some notions. For example, our motivation for contextual modality is to reason binding manipulation on open code. We give detail in Section 4.

3 KRIPKE-STYLE CONTEXTUAL MODAL TYPE THEORY

In this section, we construct Kripke-style contextual modal type theory (KCMTT), a typed lambda calculus which corresponds to KCML through the Curry-Howard correspondence [11].

3.1 Type System

We write $x, y, \ldots$ for variables, τ for base types, $l, m, n \ldots$ for non-negative integers. The syntax of KCMTT is defined as follows.

Types	S, T	$::= \tau \mid S \to T \mid [S_1, \ldots, S_m]T$
Terms	M, N, L	$::= x \mid \lambda x{:}\,T.M \mid MN$
		$\mid$ ‘$\langle x_1{:}T_1,\ldots,x_m{:}T_m\rangle M$ $\mid$,$_l\langle N_1,\ldots,N_m\rangle M$
Context	Γ	$::= \cdot \mid \Gamma, x{:}\,T$
Context Stack	Ψ	$::= \cdot \mid \Psi; \Gamma$
Judgment	J	$::= \Psi \vdash_X M{:}\,T \ (X \in \{K, K4, T, S4\})$

For a context $\Gamma = x_1 : T_1 \ldots x_m : T_m$, we define *the domain of* Γ as $dom(\Gamma) = x_1, \ldots, x_m$ and *the range of* Γ as $rg(\Gamma) = T_1, \ldots, T_m$. For a context stack $\Psi = \Gamma_1; \ldots; \Gamma_m$, we also define the domain of Ψ as $dom(\Psi) = dom(\Gamma_1), \ldots, dom(\Gamma_m)$. Let us think of a Kripke-style judgment $\Psi \vdash M : T$. For the case of K and K4, it is enough to assume that variables in the range of Γ are distinct for each context Γ in Ψ. However, in the case of T and S4 adjacent contexts can be merged by the reflexive principle, and therefore we assume that all variables in the domain of Ψ are distinct.

In KCMTT, two constructs are added to simply typed lambda calculus [11]: *quotation* and *unquotation*. From the viewpoint of staged computation, a quotation ‘$\langle\Gamma\rangle M$ can be interpreted as "a code of M under the environment Γ". On the other hand, unquotation can be seen as "evaluation of the code M through l stages, giving the environmt $N_1, \ldots, N_m$". As a special case, 0-level unquotation can be interpreted as `eval` function in Lisp.

The typing rules of KCMTT are defined as follows. Those rules correspond to deduction rules of KCML in Section 2. In the rest of this paper, we assume that all terms are typed: for any term M, there exists a type judgment $\Psi \vdash M : T$. We also identify terms under α-equivalence in Definition 3.5, and hence we can rename bound variables.

$$(\mathrm{Var})\,\dfrac{x : T \in \Gamma}{\Psi; \Gamma \vdash x : T} \qquad (\mathrm{Abs})\,\dfrac{\Psi; \Gamma, x{:}\,T \vdash M : S}{\Psi; \Gamma \vdash \lambda x{:}\,T.M : T \to S}$$

$$(\mathrm{App})\,\dfrac{\Psi; \Gamma \vdash M : T \to S \qquad \Psi; \Gamma \vdash N : T}{\Psi; \Gamma \vdash MN : S}$$

$$(\text{Quo})\ \frac{\Psi;\Gamma \vdash M : T}{\Psi \vdash \text{`}\langle\Gamma\rangle M : [rg(\Gamma)]T}$$

$$(\text{Unq})_l\ \frac{\Psi \vdash M : [T_1,\dots,T_m]S \qquad \begin{array}{c}\Psi;\Gamma_l;\dots;\Gamma_1 \vdash N_i : T_i\\ \text{for } 1 \le i \le m\end{array}}{\Psi;\Gamma_l;\dots;\Gamma_1 \vdash \ ,_l\langle N_1,\dots,N_m\rangle M : S}$$

$$\text{where}\begin{cases} l = 1 & \text{for K} & l = 0,1 & \text{for T}\\ l \ge 1 & \text{for K4} & l \ge 0 & \text{for S4}\end{cases}$$

In KCMTT, free variables have levels which correspond to the depth of the context stack. For $l \ge 1$, the set of level-l free variables in a term M, $FV_l(M)$, is defined as follows.

$$\begin{aligned}
FV_l(x) &= \begin{cases}\{x\} & \text{when } l = 1\\ \emptyset & \text{otherwise}\end{cases}\\
FV_l(\lambda x{:}\,T.M) &= \begin{cases}FV_l(M) - \{x\} & \text{when } l = 1\\ FV_l(M) & \text{otherwise}\end{cases}\\
FV_l(MN) &= FV_l(M) \cup FV_l(N)\\
FV_l(\text{`}\langle\Gamma\rangle M) &= FV_{l+1}(M)\\
FV_l(\,,_k\langle N_1,\dots,N_m\rangle M)&\\
&= \begin{cases}\bigcup_{1\le i\le m} FV_l(N_i) & \text{when } l \le k\\ FV_{l-k}(M) \cup \bigcup_{1\le i\le m} FV_l(N_i) & \text{otherwise}\end{cases}
\end{aligned}$$

For a judgment $\Gamma_m;\dots;\Gamma_1 \vdash M : T$, $FV_l(M)$ corresponds to the l-th context Γ_l. We can formally state this idea by the following lemma.

Lemma 3.1. *If $\Psi;\Gamma_l;\dots;\Gamma_1 \vdash M{:}\,T$, then $FV_l(M) \subseteq dom(\Gamma_l)$.*

Proof. By induction on the derivation. □

A quotation $\text{`}\langle\Gamma\rangle M$ corresponds to the $[\,]$-introduction rule in KCML. Γ is required to include all level-1 free variables in M by the (Quo) rule, and therefore there are no ill-formed codes with "undeclared variables". An unquotation $,_l\langle N_1,\dots,N_m\rangle M$ corresponds to the $[\,]$-elimination rule in KCML. In contrast to quotation, it substitutes all level-1 free variables in M with $N_1,\dots,N_m$.

3.2 Substitution

For $l \ge 1$, a substitution $[N_1/x_1,\dots,N_m/x_m]_l$ is a meta operation that maps a term to a term. It substitutes level-l free variables $x_1,\dots,x_m$ with terms $N_1,\dots,N_m$, respectively.

Substitution is inductively defined as follows. We denote σ for a content of substitution. For $\sigma = N_1/x_1,\dots,N_m/x_m$, we define $FV_l(\sigma) = \bigcup_{1\le i\le m} FV_l(N_i)$ and $dom(\sigma) = x_1,\dots,x_m$. We assume that all variables in $dom(\sigma)$ are distinct.

$$\begin{aligned}
x[\sigma]_l &= \begin{cases}N & \text{when } l = 1 \text{ and } N/x \in \sigma\\ x & \text{otherwise}\end{cases}\\
(MN)[\sigma]_l &= (M[\sigma]_l)(N[\sigma]_l)\\
(\lambda x{:}\,A.M)[\sigma]_l &= \begin{cases}\lambda x{:}\,A.(M[\sigma]_l) & \text{when } l = 1,\\ & x \notin dom(\sigma)\\ & \text{and } x \notin FV_1(\sigma)\\ \lambda x{:}\,A.(M[\sigma]_l) & \text{when } l > 1\end{cases}\\
(\text{`}\langle\Gamma\rangle M)[\sigma]_l &= \text{`}\langle\Gamma\rangle(M[\sigma]_{l+1})\\
(\,,_k\langle N_1\dots N_m\rangle M)[\sigma]_l &= \begin{cases},_k\langle N_1[\sigma]_l\dots N_m[\sigma]_l\rangle M & \text{when } l \le k\\ ,_k\langle N_1[\sigma]_l\dots N_m[\sigma]_l\rangle M[\sigma]_{l-k} & \text{otherwise}\end{cases}
\end{aligned}$$

Substitution corresponds to rewriting proof trees with the substitution principle in KCML. The following substitution lemma formally states the substitution principle in KCMTT.

Lemma 3.2 (Substitution Lemma). *If $\Psi;x_1{:}\,S_1,\dots,x_m : S_m;\Gamma_{l-1};\dots;\Gamma_1 \vdash M{:}\,T$ and $\Psi;\Gamma \vdash N_i : S_i$ for all $1 \le i \le m$, then $\Psi;\Gamma;\Gamma_{l-1};\dots;\Gamma_1 \vdash M[\sigma]_l{:}\,T$, where $\sigma = N_1/x_1,\dots,N_m/x_m$.*

Proof. By induction on the derivation rules. □

Note that this substitution is capture-avoiding one, though there is apparently no collision check for quotation. It works because the substitution and the bindings of the quotation are at different levels. As a result of substitution lemma, we can state that weakening, exchange, and single substitution preserves types.

3.3 Level Substitution

For $l \ge 1$ and $m \ge 0$, a level substitution $\uparrow_l^m$ is a meta operation that maps a term to a term.

$$\begin{aligned}
x \uparrow_l^m &= x\\
(MN) \uparrow_l^m &= (M \uparrow_l^m)(N \uparrow_l^m)\\
(\lambda x{:}\,A.M) \uparrow_l^m &= \lambda x{:}\,A.(M \uparrow_l^m)\\
(\text{`}\langle\Gamma\rangle M) \uparrow_l^m &= \text{`}\langle\Gamma\rangle(M \uparrow_{l+1}^m)\\
(\,,_k\langle N_1,\dots,N_n\rangle M) \uparrow_l^m &= \begin{cases},_{k+m-1}\langle N_1\uparrow_l^m,\dots,N_n\uparrow_l^m\rangle M & \text{when } l \le k\\ ,_k\langle N_1\uparrow_l^m,\dots,N_n\uparrow_l^m\rangle M \uparrow_{l-k}^m & \text{otherwise}\end{cases}
\end{aligned}$$

The idea of the level substitution may not be intuitive. Proof theoretically, it corresponds to rewriting proof-trees with reflexive/transitive principles. The following lemmas formally state those principles.

Lemma 3.3 (Level Substitution Lemma). *(i) For $X \in \{T, S4\}$, if $\Psi;\Gamma_{l+1};\Gamma_l;\dots;\Gamma_1 \vdash_X M{:}\,T$, then $\Psi;\Gamma_{l+1},\Gamma_l;\dots;\Gamma_1 \vdash_X M \uparrow_l^0{:}\,T$.*
(ii) For $X \in \{K4, S4\}$ and $m > 1$, if $\Psi;\Gamma_{l+1};\Gamma_l;\dots;\Gamma_1 \vdash_X M{:}\,T$, then $\Psi;\Gamma_{l+1};\Psi';\Gamma_l;\dots;\Gamma_1 \vdash_X M \uparrow_l^m{:}\,T$ where Ψ' is size-$(m-1)$ stack of empty contexts.

Proof. By induction on the derivation rules. □

Note that a level substitution $\uparrow_1^1$ is identity on terms. Therefore the level substituion lemma for the K variant is trivial.

3.4 Equivalence on Terms

Now we are ready to define α-equivalence, β-reduction and η-expansion rules.

Definition 3.4. Let $\sim$ be a binary relation on terms. $\sim$ is *compatible* iff it satisfies the following conditions.

$$M_1 \sim M_2 \Rightarrow \lambda x{:}\,T.M_1 \sim \lambda x{:}\,T.M_2$$
$$M_1 \sim M_2 \Rightarrow (M_1N) \sim (M_2N)$$
$$M_1 \sim M_2 \Rightarrow (NM_1) \sim (NM_2)$$
$$M_1 \sim M_2 \Rightarrow \text{`}\langle\Gamma\rangle M_1 \sim \text{`}\langle\Gamma\rangle M_2$$

$$M_1 \sim M_2 \Rightarrow\ ,l\langle N_1 \dots N_m\rangle M_1 \sim\ ,l\langle N_1 \dots N_m\rangle M_2$$
$$M_1 \sim M_2 \Rightarrow\ ,l\langle L_1 \dots L_m, M_1, L'_1 \dots L'_n\rangle N \sim\ ,l\langle L_1 \dots L_m, M_2, L'_1 \dots L'_n\rangle N$$

Definition 3.5. α-equivalence $=_\alpha$ is the least transitive, reflexive, and compatible relation on terms satisfying the following:

$$\lambda x{:}\, T.M =_\alpha \lambda y{:}\, T.(M[x/y]_1)$$
$$`\langle x_1{:}T_1, \dots, x_m{:}T_m\rangle M =_\alpha\ `\langle y_1{:}T_1, \dots, y_m{:}T_m\rangle(M[y_1/x_1, \dots, y_m/x_m]_1)$$

Definition 3.6. β-reduction $\overset{R}{\Rightarrow}$ and η-expansion $\overset{E}{\Rightarrow}$ are the least compatible relations on terms which satisfy the following:

$$(\lambda x{:}\, A.M)N \overset{R}{\Rightarrow} M[N/x]_1$$
$$,k\langle N_1, \dots, N_m\rangle `\langle x_1{:}T_1, \dots, x_m{:}T_m\rangle M \overset{R}{\Rightarrow} M \uparrow_1^k [N_1/x_1, \dots, N_m/x_m]_1$$
$$M \overset{E}{\Rightarrow} \lambda x{:}\, T.Mx \quad \text{when } \Psi \vdash M : T \to S$$
$$M \overset{E}{\Rightarrow}\ `\langle x_1{:}T_1, \dots, x_m{:}T_m\rangle(\ ,1\langle \vec{x}\rangle M) \quad \text{when } \Psi \vdash M : [T_1, \dots, T_m]S$$

Finally, we get subject reduction and expansion.

Theorem 3.7 (Subject Reduction/Expansion). *(i) If* $\Psi \vdash M{:}\, T$ *and* $M \overset{R}{\Rightarrow} N$*, then* $\Psi \vdash N{:}\, T$ *.*
(ii) If $\Psi \vdash M{:}\, T$ *and* $M \overset{E}{\Rightarrow} N$*, then* $\Psi \vdash N{:}\, T$ *.*

Proof. By induction on the definition of $\overset{R}{\Rightarrow}$ and $\overset{E}{\Rightarrow}$. The base cases are proved by Lemma 3.2 and 3.3. □

3.5 Examples

The following examples show KCMTT type judgments which correspond to examples in Section 2. We use X, Y for higher level variables, and a, b for lower ones.

(1) $\vdash_K \lambda X{:}\, [C](A \to B).\lambda Y{:}\, [C]A.`\langle a{:}C\rangle(\ ,1\langle a\rangle X)(\ ,1\langle a\rangle Y)$
$: [C](A \to B) \to [C]A \to [C]B$
(2) $a : B \vdash_T \lambda X{:}\, [B]A.\ ,0\langle a\rangle X : [B]A \to A$
(3) $\vdash_{K4} \lambda X{:}\, [C]A.`\langle a{:}D\rangle `\langle b{:}C\rangle\ ,2\langle b\rangle X : [C]A \to [D][C]A$
(4) $\vdash_K \lambda X{:}\, [C]A.`\langle a{:}C, b{:}D\rangle\ ,1\langle a\rangle X : [C]A \to [C, D]A$
(5) $\vdash_K \lambda X{:}\, [C, C]A.`\langle a{:}C\rangle\ ,1\langle a, a\rangle X : [C, C]A \to [C]A$
(6) $\vdash_K\ `\langle a{:}A\rangle a : [A]A$
(7) $\vdash_K \lambda X{:}\, [A]B.`\langle\rangle(\lambda a{:}\, A.\ ,1\langle a\rangle X) : [A]B \to [\,](A \to B)$
(8) $\vdash_K \lambda X{:}\, [\,](A \to B).`\langle a{:}A\rangle(\ ,1\langle\rangle X)a : [\,](A \to B) \to [A]B$

4 FUTURE WORK

In this paper, we introduced the overview of KCMTT.

There are many problems to be solved. This paper only provides subject reduction/expansion and does not prove confluency and strong normalization. We expect that proofs in previous work of Kripke-style modal type theory may be helpful. Comparison between S4 KCMTT and the original CMTT is also necessary. We expect that they have equal expressiveness, but otherwise the difference can be interesting. Davies and Pfenning[3] provide mutual translation between S4 Kripke-style modal type theory and dual-context modal type theory. This translation may help us to solve the problem.

Our goal is to construct a type system for syntactical metaprogramming such as macro system in Common Lisp [12] or Template Haskell [10]. Although those implementations give a great extensibility to programming languages, they are known not to be type-safe. In other words, code with syntactic extension, even if it is well-typed, may extend to ill-typed code. Therefore we want a type system for syntactical metaprogramming that guarantees type-safety of syntactic extension.

The basic idea of such metaprogramming is quasiquotation syntax, which enables programmers to construct code. There are some formal systems which provide Lisp-like quasiquotation such as Kripke-style modal calculi [3] and linear temporal calculi [2]. However, they are not capable of binding manipulation: Kripke-style modal calculi only treat closed code, and linear temporal calculi do not allow access to free variables in open code. We think KCMTT can be the basis for type system of syntactical metaprogramming because it provides Lisp-like quasiquotation and allows access to free variables in open code.

With comparison to CMTT, we think that KCMTT is preferable as the type system for this purpose. First, it has quasiquotation constructs. Second, it is sufficiently *weak* as a logical system. We do not need runtime code evaluation for syntactical metaprogramming. It is known that the T axiom corresponds to runtime code evaluation, and therefore we may omit assumption on the reflexivity. KCMTT provides K and K4 variants, and we think those variants perceive the nature of syntactical metaprogramming.

ACKNOWLEDGMENTS

We thank Yuichi Nishiwaki and members of Hagiya Laboratory for valuable discussion. We also appreciate anonymous referees for helpful comments and suggestions.

REFERENCES

[1] Gianluigi Bellin, Valeria de Paiva, and Eike Ritter. 2001. Extended Curry-Howard correspondence for a basic constructive modal logic. In *Proceedings of Methods for Modalities*. http://ukpmc.ac.uk/abstract/CIT/485479
[2] Rowan Davies. 2017. A Temporal Logic Approach to Binding-Time Analysis. *J. ACM* 64, 1 (2017), 1–45.
[3] Rowan Davies and Frank Pfenning. 2001. A Modal analysis of Staged Computation. *J. ACM* 48, 3 (2001), 555–604. https://doi.org/10.1145/382780.382785
[4] G. A. Kavvos. 2017. Dual-Context Calculi for Modal Logic. In *Logic in Computer Science*. Reykjavik. arXiv:1602.04860 http://arxiv.org/abs/1602.04860
[5] Saul A. Kripke. 1963. Semantical Analysis of Modal Logic I Normal Modal Propositional Calculi. *Mathematical Logic Quarterly* 9, 5-6 (1963), 67–96. https://doi.org/10.1002/malq.19630090502
[6] Simone Martini and Andrea Masini. 1996. A Computational Interpretation of Modal Proofs. In *Proof Theory of Modal Logic*. 213–241. https://doi.org/10.1007/978-94-017-2798-3_12
[7] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Transactions on Computational Logic* 9, 3 (2008), 1–49. https://doi.org/10.1145/1352582.1352591
[8] Frank Pfenning and Rowan Davies. 2001. A Judgmental Reconstruction of Modal Logic. *Mathematical Structures in Computer Science* 11, 4 (2001), 511–540. https://doi.org/10.1017/S0960129501003322
[9] Frank Pfenning and H. C. Wong. 1995. On a Modal λ-Calculus for S4. *Electronic Notes in Theoretical Computer Science* 1, C (1995), 515–534. https://doi.org/10.1016/S1571-0661(04)00028-3
[10] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. *SIGPLAN Not.* 37, 12 (Dec. 2002), 60–75. https://doi.org/10.1145/636517.636528
[11] Morten Heine Sørensen and Pawel Urzyczyn. 2006. *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)*. Elsevier Science Inc., New York, NY, USA.
[12] Guy L. Steele, Jr. 1990. *Common LISP: The Language (2Nd Ed.)*. Digital Press, Newton, MA, USA.

Property-Based Testing via Proof Reconstruction

Work-in-progress

Roberto Blanco
Dale Miller
INRIA Saclay — Île-de-France & LIX/École polytechnique
Palaiseau, France
roberto.blanco,dale.miller@inria.fr

Alberto Momigliano
DI, Università degli Studi di Milano, Italy
momigliano@di.unimi.it

ABSTRACT

Property-based testing is a technique for validating code against an executable specification by automatically generating test-data. From its original use in programming languages, this technique has now spread to most major proof assistants to complement theorem proving with a preliminary phase of conjecture testing. We present a proof theoretical reconstruction of this style of testing for relational specifications (such as those used in the semantics of programming languages) and employ the Foundational Proof Certificate framework to aid in describing test generators. We do this by presenting certain kinds of “proof outlines” that can be used to describe the shape and size of the generators for the conditional part of a proposed property. Then the testing phase is reduced to standard logic programming search. After illustrating our techniques on simple, first-order (algebraic) data structures, we lift it to data structures containing bindings using λ-tree syntax. The λProlog programming language is capable of performing both the generation and checking of tests. We validate this approach by tackling benchmarks in the metatheory of programming languages coming from related tools such as PLT-Redex.

1 INTRODUCTION

In this brief paper, we examine *property-based testing (PBT)* from a proof theory point-of-view and explore some of the advantages that result from exploiting this perspective.

1.1 Generate-and-test as bipoles

Imagine that we wish to write a relational specification for reversing lists. There are, of course, many ways to write such a specification but in every case, the formula

$$\forall L\colon (list\ int)\ \forall R\colon (list\ int)\ [rev\ L\ R \supset rev\ R\ L]$$

stating that *rev* is idempotent should be a theorem. In fact, we might wish to prove a number of formulas of the form $\forall x\colon \tau\ [P(x) \supset Q(x)]$ where both P and Q are formulae over given relational specifications and a single variable (for brevity). Occasionally, it can be important in this setting to move the type judgment $x\colon \tau$ into the logic by turning the type into a predicate: $\forall x[(\tau(x) \wedge P(x)) \supset Q(x)]$. Proving such formulas can often be difficult since their proof may involve the clever invention of prior lemmas and induction invariants. In many practical settings, such formulas are, in fact, not theorems since the relational specifications in P and/or Q can contain errors. It can be therefore valuable to first attempt to find counterexamples to such formulas prior to pursue a proof. That is, we might try to prove formulas of the form $\exists x[(\tau(x) \wedge P(x)) \wedge \neg Q(x)]$ instead. If a term t of type τ can be discovered such that $P(t)$ holds while $Q(t)$ does not, then one can return to the specifications in P and Q and revise them using the concrete evidence in t about how the specifications are wrong. The process of writing and revising relational specifications could be aided if such counterexamples are discovered quickly and automatically.

The literature contains at least two ways to view Horn clause-style relational specifications in proof-theoretic terms. For example, specifications such as

```
nat z.               lst nl.
nat (s N) :- nat N.  lst (cns N Ns) :- nat N, lst Ns.

app nl Xs Xs.
app (cns X Xs) Ys (cns X Zs) :- app Xs Ys Zs.
```

can be viewed as a set of first-order Horn clauses: one of these formulas would be the universal closure of

$$[app\ Xs\ Ys\ Zs \supset app\ (cns\ X\ Xs)\ Ys\ (cns\ X\ Zs)].$$

The proof search approach to encoding Horn clause computation results in the structuring of proofs with repeated switchings between a *goal-reduction* phase and a *backchaining* phase [19]. The notion of *focused proof systems* generalizes this view of proof construction in the sense that goal-reduction corresponds to the *negative* phase: during this phase, the conclusion-to-premise construction of proofs proceeds without needing to make any choices (no backtracking). At the same time, the backchaining phase corresponds to the *positive* phase: during this phase, proof construction generally needs to consume some information from, say, an oracle or to allow for some nondeterminism. The combination of a positive phase and a negative phase is called a *bipole*. In this view of logic programming, proof search involves proofs with arbitrary numbers of bipoles. Comprehensive focusing systems exist for linear, intuitionistic and classical logics [15].

A different approach to the proof theory of Horn clauses involves encoding them as *fixed points*. For example, the Prolog-style specifications above of *nat* and *app* can be written instead as the following fixed point definitions.

$$nat = \mu\lambda N\lambda n\ (n = \mathbf{z} \vee \exists n'(n = \mathbf{s}\ n' \wedge^+ N\ n'))$$
$$app = \mu\lambda A\lambda xs\lambda ys\lambda zs\ ((xs = \mathbf{nl} \wedge^+ ys = zs) \vee$$
$$\exists x'\exists xs'\exists zs'(xs = \mathbf{cns}\ x'\ xs' \wedge^+ zs = \mathbf{cns}\ x'\ zs' \wedge^+ A\ xs'\ ys\ zs'))$$

When using a focused proof system for logic extended with fixed points, such as is employed in Bedwyr [2] and described in [1, 13], proofs of formulas such as $\exists x\colon \tau\ [P(x) \wedge^+ \neg Q(x)]$ are a single bipole:

when reading a proof bottom up, a positive phase is followed on all its premises by a single negative phase that completes the proof. In particular, the positive phase corresponds to the *generation* phase and the negative phase corresponds to the *testing* phase. From this description, it is conceptionally easy (as one would expect) to construct an implementation of the testing phase while it can be difficult to steer the generation phase through a (possibly) great deal of nondeterminism. For example, the blind exhaustive enumeration of possible counterexamples is generally known to be ineffective. Significant sophistication may go into crafting generators and assembling them.

1.2 Flexible test case generation via proof reconstruction

The *foundational proof certificate (FPC)* framework was proposed in [9] as a means of defining proof structures used in a range of different theorem provers (e.g., resolution refutations, Herbrand disjuncts, tableaux, etc). The FPC framework was designed using focused proof systems as a kind of protocol: during the construction of a positive phase, the proof checker could request specific information from a proof certificate. In the general setting, proof certificates do not need to contain all the details required to complete a formal proof. In those cases, a proof checker would need to perform proof *reconstruction*. For example, FPCs can be used as *proof outlines* [5] since they can describe some of the general shape of a proof: e.g., apply the obvious induction invariant and complete the proof via the enumeration of all remaining cases. The proof checker would attempt to fill in the missing details, either obtaining a proof of the described shape or failing to do so.

In this paper, we propose to use FPCs as a language for *describing generators*. We have experimented with writing proof checkers in both OCaml (as an extension to Abella [3]) and λProlog, which could be used to check proof certificates and in the process steer the proof of the expression $P(x)$, and the corresponding typing expression, say, $\tau(x)$.

As we shall illustrate, we have defined certificates that describe families of proofs that are limited either by the number of inference rules that they contain, by their height, or by both. Using similar techniques, it is possible to define FPCs that target specific types for specific treatment: for example, when generating integers, only (user-defined) small integers would be produced. Using a proof reconstructing checker (such as is easy to do with a logic programming system), the search space of proofs that a FPC describes for a specific formula of the form $\exists x\ [\tau(x) \wedge^+ P(x) \wedge^+ \neg Q(x)]$ can be directly translated into a description of the range of possible witness terms for this quantifier.

1.3 Lifting PBT to treat λ-tree syntax

Describing a computational task using proof theory often allows researchers to lift descriptions based on first-order (algebraic) terms to descriptions based on *λ-tree syntax* (a specific approach to higher-order abstract syntax). For example, once logic programming was given a proof search description, it was natural to generalize the usual approaches to logic programming from the manipulation of first-order terms (Prolog) to the manipulation of λ-terms (λProlog) [17]. Similarly, once certain model checking and inductive theorem provers were presented using sequent calculus in a first-order logic with fixed points [1, 13], it was possible to incorporate λ-terms syntax in generalizations of model checkers, as in the Bedwyr system [2], and of theorem provers, as in Abella [3].

The full treatment of λ-tree syntax in a logic with fixed points is usually accommodated with the addition of the ∇-quantifier [12, 18]. While the ∇-quantifier has had significant impact in several reasoning tasks (for example, in the formalized metatheory of the π-calculus and λ-calculus) an important result about ∇ is the following: if fixed point definitions do not contain implications and negations, then exchanging occurrences of $\forall$ and ∇ does not affect what atomic formulas are proved [18, Section 7.2]. Since we shall be limiting ourselves to Horn-like recursive definitions, the λProlog implementation of $\forall$ will also implement ∇.

This direct treatment of λ-terms within the PBT setting allows us to apply property-based testing to a number of metaprogramming tasks. After describing some more details of how PBT can be encoded in proof theory (and logic programming) in the next section, we discuss in Section 3 the treatment of metaprogramming.

2 BASIC APPROACH

The setup follows [16]; we introduce a simple specification logic, which drives the derivation of our object logic. In this case it is basically the usual Prolog vanilla meta-interpreter, save for interpreting ∇ as Π; the "program" is represented as Horn-like clauses by a two-place predicate `prog` relating heads and bodies, built out of object-level logical constants (`tt`, `or`, `and`, `nabla`) and user-defined constructors for predicates. For example, to generate lists of `as` and `bs` and compute the reverse a list, we have the following `prog` clauses, where we omit the code for `append`:

```
prog (is_elt a) tt.
prog (is_elt b) tt.
prog (is_eltlist nl) tt.
prog (is_eltlist (cns X Xs))
     (and (is_elt X) (is_eltlist Xs)).
prog (rev nl nl) tt.
prog (rev (cns X Xs) Rs)
     (and (rev Xs Sx) (append Sx (cns X nl) Rs)).
```

Suppose we want to falsify the assertion that the reverse of a list is equal to itself. The generation phase is steered by the predicate `check`, which uses a certificate (its first argument) to produce candidate lists up to a certain bound, in this case the *height* of a proof of being a list. The testing phase performs deterministic computation with the meta-interpreter `interp` and then negates the conclusion using negation-as-failure (NAF):

```
cexrev Xs Ys :- check (qgen (qheight 3)) (is_eltlist Xs),
                interp (rev Xs Ys), not (Xs = Ys).
```

Note that the call to NAF is safe since, by the totality of `rev`, `Ys` will be ground.

The FPC kernel is presented in Figure 1. Each object-level connective is interpreted as λProlog code, and user-defined constructors are looked up in `prog` and unfolded. This is driven by the meta-interpreter `interp` (omitted). To it, `check` adds a certificate term and calls to *expert* predicates on said term (except `nabla`, which is transparent to the experts). Experts decide when the computation proceeds — producing certificates for the continuations — and when

```
check Cert tt          :- tt_expert Cert.
check Cert (and G1 G2) :- and_expert Cert Cert1 Cert2, check Cert1 G1, check Cert2 G2.
check Cert (or G1 G2)  :- or_expert Cert Cert' LR, ((LR = left, check Cert' G1); (LR = right, check Cert' G2)).
check Cert (nabla G)   :- pi x\ check Cert (G x).
check Cert A           :- unfold_expert Cert Cert', prog A G, check Cert' G.

tt_expert     (qgen (qsize In In)).
tt_expert     (qgen (qheight _)).
or_expert     (qgen (qsize In Out)) (qgen (qsize In Out)) _.
or_expert     (qgen (qheight H))    (qgen (qheight H))    _.
and_expert    (qgen (qsize In Out)) (qgen (qsize In Mid)) (qgen (qsize Mid Out)).
and_expert    (qgen (qheight H))    (qgen (qheight H))    (qgen (qheight H)).
unfold_expert (qgen (qsize In Out)) (qgen (qsize In' Out)) :- In > 0, In' is In - 1.
unfold_expert (qgen (qheight H))    (qgen (qheight H'))    :- H  > 0, H'  is H  - 1.
```

Figure 1: Kernel for expert-driven term generation

it fails. The first argument of an expert, e.g., `and_expert`, refers to the conclusion of the corresponding rule and the remaining ones, if any, to the premises. Here the complexity of generated candidates is bound by limiting unfoldings, either by height (`qheight`, producing shallow terms), number of constructors (`qsize`, producing small terms), or both by *pairing* (not shown here, but see [6]).

3 PBT FOR METAPROGRAMMING

To showcase the ease with which we handle searching for counterexamples in binding signatures, we encode a simply-typed λ-calculus augmented with constructors for integers and lists, following the PLT-Redex benchmark from http://docs.racket-lang.org/redex/benchmark.html. The language is as follows:

Types	A, B	::=	$int \mid ilist \mid A \to B$
Terms	M	::=	$x \mid \lambda x{:}A.\ M \mid M_1\ M_2 \mid c \mid err$
Constants	c	::=	$n \mid plus \mid nil \mid cons \mid hd \mid tl$
Values	V	::=	$c \mid \lambda x{:}A.\ M \mid plus\ V$
			$\mid cons\ V \mid cons\ V_1\ V_2$

The rules for the dynamic and static semantics are given in Figure 2, where the latter assumes a signature Σ with the obvious type declarations for constants. Rules for *plus* are omitted for brevity.

The encoding in λProlog is pretty standard and also omitted: we declare constructors for terms, constants and types, while we carve out values via an appropriate predicate. A similar predicate characterizes the threading in the operational semantics of the *err* expression, used to model run time errors such as taking the head of an empty list. We follow this up (see the bottom of Figure 2) with the static semantics (predicate `wt`), where constants are typed via a table `tcc`. Note that we have chosen an *explicitly* context-ed encoding of typing as opposed to one based on hypothetical judgments such as in [16]: this choice avoids using implications in the body of the typing predicate and, as a result, allows us to use λProlog's universal quantifier to implement the reasoning level ∇-quantifier.

Now, this calculus enjoys the usual property of subject reduction and progress, where the latter means "being either a value, an error, or able to make a step." And in fact we can fairly easily prove those results in a theorem prover such as Abella. However, the case distinction in the progress theorem does require some care: were it to be unprovable given a mistake in the specification, it would not be immediate to localize where the problem may be. On the other hand, one could wonder whether our calculus enjoys the *subject expansion* property — the alert reader will undoubtedly realize that this is highly unlikely, but rather than wasting time in a proof attempt, we search for a counterexample and find:

```
cexsexp M M' A :- check (qgen (qsize 8 _)) (step M M'),
                  interp (wt null M' A),
                  not (interp (wt null M A)).
A  = listTy
M' = c nl
M  = app (c hd) (app (app (c cns) (c nl)) (c _))
```

Other queries we can ask: are there *untypable* terms, or terms that do not converge to a value?

As a more comprehensive validation we addressed the nine mutations proposed by the PLT-Redex benchmark, to be spotted as a violation of either the preservation or progress properties. For example, the first mutation introduces a bug in the typing rule for application, matching the range of the function type to the type of the argument:

$$\frac{\Gamma \vdash_\Sigma M_1 : A \to B \quad \Gamma \vdash_\Sigma M_2 : B}{\Gamma \vdash_\Sigma M_1\ M_2 : B}\ \texttt{T-APP-B1}$$

The given mutation makes both properties fail:

```
cexprog M A :- check (qgen (qsize 6 _)) (wt null M A),
               not (interp (progress M)).
A = intTy
M = app (c hd) (c (toInt zero))

cexpres M M' A :- check (qgen (qsize 8 _)) (wt null M A),
                  interp (step M M'),
                  not (interp (wt null M' A)).
A  = funTy listTy intTy
M' = lam (x\ c hd) listTy
M  = app (lam (x\ lam (y\ x) listTy) intTy) (c hd)
```

Table 1 reports the tests, performed under Ubuntu 16.04 on a Intel Core i7-870 CPU, 2.93GHz with 8GB RAM. We time-out the computation when it exceeds 300 seconds. We list the results obtained by λProlog (λP) under Teyjus [20], the counterexample found, and a brief description of the bug together with Redex's difficulty rating (shallow, medium, unnatural). The column αC lists the time taken by αCheck [7] using NAF, which is not always the best technique [8],

$$\frac{}{hd\ (cons\ M_1\ M_2) \longrightarrow M_1}\ \texttt{E-HD} \qquad \frac{}{tl\ (cons\ M_1\ M_2) \longrightarrow M_2}\ \texttt{E-TL}$$

$$\frac{}{\lambda x : A.\ M\ V \longrightarrow [x \mapsto V]M}\ \texttt{E-ABS} \qquad \frac{M_1 \longrightarrow M_1'}{M_1\ M_2 \longrightarrow M_1'\ M_2}\ \texttt{E-APP1} \qquad \frac{M \longrightarrow M'}{V\ M \longrightarrow V\ M'}\ \texttt{E-APP2}$$

$$\frac{}{\vdash_\Sigma err : A}\ \texttt{T-ER} \qquad \frac{\Sigma(c) = A}{\vdash_\Sigma c : A}\ \texttt{T-K} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash_\Sigma x : A}\ \texttt{T-VAR} \qquad \frac{\Gamma, x : A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x : A.\ M : A \to B}\ \texttt{T-ABS} \qquad \frac{\Gamma \vdash_\Sigma M_1 : A \to B \quad \Gamma \vdash_\Sigma M_2 : A}{\Gamma \vdash_\Sigma M_1\ M_2 : B}\ \texttt{T-APP}$$

```
prog (wt _ err _)           tt.
prog (wt _ (c M) A)         (tcc M A).
prog (wt Gamma M A)         (memb (bind M A) Gamma).
prog (wt Gamma (lam M Ax)   (funTy Ax A)) (nabla x\ wt (cons (bind x Ax) Gamma) (M x) A).
prog (wt Gamma (app M N) A) (and (wt Gamma M (funTy B A)) (wt Ga N B)).
```

Figure 2: Static and dynamic semantics of the *Stlc* language.

bug	check	αC	λP	cex	Description/Rating
1	preservation	0.3	0.05	(λx:*int*. λy:*ilist*. x) *hd*	range of function in app rule
	progress	0.1	0.02	*hd 0*	matched to the arg. (S)
2	progress	0.27	0.06	*(cons 0) nil*	value *(cons v) v* omitted (M)
3	preservation	0.04	0.01	(λx:*int*. *cons*) *cons*	order of types swapped
	progress	0.1	0.04	*hd 0*	in function pos of app (S)
4	progress	t.o.	207.3	*(plus 0) ((cons 0) nil)*	the type of cons return *int* (S)
5	preservation	t.o.	0.67	*tl ((cons 0) nil)*	tail reduction returns the head (S)
6	progress	24.8	0.4	*hd ((cons 0) nil)*	hd reduction on part. applied cons (M)
7	progress	1.04	0.1	*hd* ((λx:*ilist*. *err*) *nil*)	no eval for argument of app (M)
8	preservation	0.02	0.01	(λx:*ilist*. x) *nil*	lookup always returns int (U)
9	preservation	0.1	0.02	(λx:*ilist*. *cons*) *nil*	vars do not match in lookup (S)

Table 1: *Stlc* benchmark

but it corresponds very closely to the architecture of the present paper. Of course, αCheck sits on top of an interpreted (prototype) language, whereas Teyjus is a compiler: however, one can argue that the two level themselves out, since we use meta-interpretation for test generation. The results are essentially indistinguishable, save for bugs 4, 5 and 6: in the first, which is surprisingly hard to find, αCheck times out, while we comfortably beat the time limit. αCheck flunks number 5, which is immediate for us. Finally in bug 6 αCheck's fixed integrative deepening strategy needs to explore the search space up to level 11, while we can leverage the FPC ability to use the `qsize` metric.

4 RELATED WORK

Property-based testing is a technique for validating code against an executable specification by automatically generating test-data, typically in a random and/or exhaustive fashion. From its original use in programming languages [10], this technique has now spread to most major proof assistants [4, 22] to complement theorem proving with a preliminary phase of conjecture testing. We do not have the space for a comprehensive review, for which we refer to [7], but we mention two of the main players w.r.t. metatheory model checking: *PLT-Redex* [11] is an executable DSL for mechanizing semantic models built on top of *DrRacket* with support for random testing à la QuickCheck; its usefulness has been demonstrated in several impressive case studies [14]. However, Redex has limited support for relational specifications and none whatsoever for binding signature. This is where αCheck [7] comes in. The tool adds on top of the nominal logic programming language αProlog a checker for relational specifications as we do here. One of the implementation techniques is based as well on NAF, as far as testing of the conclusion is concerned. The generation phase is instead “wired in” via iterative-deepening search, based on derivation height. In this sense αCheck is less flexible than the FPC-based architecture that we propose here, since it can be seen as a fixed choice of experts.

Finally, more distant cousins in the logic programming world are *declarative debugging* [21] and the Logic-Based Model Checking project at Stony Brook (http://www.cs.sunysb.edu/~lmc).

5 CONCLUSION AND FUTURE WORK

We have described some work-in-progress that uses standard logic programming techniques and some recent developments in proof theory to design a flexible framework for PBT. Given its proof theoretic pedigree, it was immediate to extend PBT to the metaprogramming setting.

Figure 1 specifies only two certificate formats: one that limits the size and one that limits the height of a proof. We have also implemented another certificate format that implements both restrictions at the same time. It is easy to code other certificates: by reading random bits from an external source of entropy, certificates can describe randomly organized proofs (and, hence, witness terms). Certificates can also be organized to consider only allowing small proofs for one type but random for another type: thus, one could easily design a certificate that would explore randomly generated lists containing just, say, the integers 0 and 1.

While λProlog is used here to discover counterexamples, one does not actually need to trust the logical soundness of λProlog (negation-as-failure makes this a complex issue). Any counterexample that is discovered can be output and used within, say, Abella to formally prove that it is indeed a counterexample. In fact, we plan to integrate our take on PBT in Abella, in order to support both proofs and *disproofs*.

Acknowledgments The work of Blanco and Miller was funded by the ERC Advanced Grant ProofCert.

REFERENCES

[1] D. Baelde. Least and greatest fixed points in linear logic. *ACM Trans. on Computational Logic*, 13(1), Apr. 2012.

[2] D. Baelde, A. Gacek, D. Miller, G. Nadathur, and A. Tiu. The Bedwyr system for model checking over syntactic expressions. In F. Pfenning, editor, *21th Conf. on Automated Deduction (CADE)*, number 4603 in LNAI, pages 391–397, New York, 2007. Springer.

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

[4] J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic proof and disproof in Isabelle/HOL. In C. Tinelli and V. Sofronie-Stokkermans, editors, *FroCoS*, volume 6989 of *Lecture Notes in Computer Science*, pages 12–27. Springer, 2011.

[5] R. Blanco and D. Miller. Proof outlines as proof certificates: a system description. In I. Cervesato and C. Schürmann, editors, *Proceedings First International Workshop on Focusing*, volume 197 of *Electronic Proceedings in Theoretical Computer Science*, pages 7–14. Open Publishing Association, Nov. 2015.

[6] R. Blanco, Z. Chihani, and D. Miller. Translating between implicit and explicit versions of proof. In L. de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction*, volume 10395 of *LNCS*, pages 255–273. Springer, 2017.

[7] J. Cheney and A. Momigliano. αCheck: A mechanized metatheory model checker. *Theory and Practice of Logic Programming*, 17(3):311âĂŞ352, 2017.

[8] J. Cheney, A. Momigliano, and M. Pessina. Advances in property-based testing for αProlog. In B. K. Aichernig and C. A. Furia, editors, *Tests and Proofs - 10th International Conference, TAP 2016, Vienna, Austria, July 5-7, 2016, Proceedings*, volume 9762 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2016.

[9] Z. Chihani, D. Miller, and F. Renaud. A semantic framework for proof evidence. *J. of Automated Reasoning*, 2016.

[10] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, pages 268–279. ACM, 2000.

[11] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.

[12] A. Gacek, D. Miller, and G. Nadathur. Combining generic judgments with recursive definitions. In F. Pfenning, editor, *23th Symp. on Logic in Computer Science*, pages 33–44. IEEE Computer Society Press, 2008.

[13] Q. Heath and D. Miller. A proof theory for model checking: An extended abstract. In I. Cervesato and M. Fernández, editors, *Proceedings Fourth International Workshop on Linearity (LINEARITY 2016)*, volume 238 of *EPTCS*, Jan. 2017.

[14] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 285–296, New York, NY, USA, 2012. ACM.

[15] C. Liang and D. Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009.

[16] R. McDowell and D. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. on Computational Logic*, 3(1):80–136, 2002.

[17] D. Miller and G. Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.

[18] D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, Oct. 2005.

[19] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[20] G. Nadathur and D. J. Mitchell. System description: Teyjus — A compiler and abstract machine based implementation of λProlog. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in LNAI, pages 287–291, Trento, 1999. Springer.

[21] L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

[22] Z. Paraskevopoulou, C. Hritcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2015.

POPLMark Reloaded

Andreas Abel
Department of Computer Science and Engineering, Gothenburg University / Chalmers, Sweden
andreas.abel@gu.se

Alberto Momigliano
DI, Università degli Studi di Milano, Italy
momigliano@di.unimi.it

Brigitte Pientka
School of Computer Science, McGill University, Montreal, Canada
bpientka@cs.mcgill.ca

ABSTRACT

As a follow-up to the POPLMark Challenge, we propose a new benchmark for machine-checked metatheory of programming languages: establishing strong normalization of a simply-typed lambda-calculus with a proof by Kripke-style logical relations. We believe that this case-study overcomes some of the limitations of the original challenge and highlights, among others, the need of native support for context reasoning and simultaneous substitutions.

1 INTRODUCTION

The usefulness of sets of benchmarks has been recognized in many areas of computer science, and in particular in the theorem proving community, for stimulating progress or at least taking stocks of what the state of the art is — *TPTP* [Sutcliffe 2009] is one shining example. The situation is less satisfactory for proof assistants, where each system comes with its own set of examples/libraries, some of them gigantic; this is not surprisingly, since we are potentially addressing the whole realm of mathematics.

In a more limited setting, some 12 years ago, a group of renowned programming language theorists came together and issued the so-called *POPLMark Challenge* [Aydemir et al. 2005] (PC, in short), with the aim of fostering the collaboration between the PL community and researchers in proofs assistants/logical frameworks to bring about:

> "[...] a future where the papers in conferences such as POPL and ICFP are routinely accompanied by mechanically checkable proofs of the theorems they claim" (page 51 op. cit.)

As we know, the challenge revolved around the meta-theory of $\mathsf{F}_{<:}$, which, requiring induction over *open* terms, was an improvement over the gold standard of mechanized meta-theory in the nineties: type soundness. Yet, the spotlight of the PC was still on

> "type preservation and soundness theorems, unique decomposition properties of operational semantics, proofs of equivalence between algorithmic and declarative versions of type systems, etc." (*ibidem*)

Further, the authors made paramount "the problem of representing and reasoning about inductively-defined structure with *binders*" (our emphasis), while providing a balanced criticism of de Bruijn indexes as an encoding technique. That focus was understandable, since at that time the only alternative to concrete representations was higher-order abstract syntax (HOAS), mostly in the rather peculiar Twelf setting, the implementation of nominal logic being in its infancy.

While the response of the theorem-proving community was impressive with more than 15 (partial) solutions submitted (https://www.seas.upenn.edu/~plclub/poplmark/), one can argue whether the envisioned future has became our present — according to Sewell's POPL 2014 Program Chair's Report (https://www.cl.cam.ac.uk/~pes20/popl2014-pc-chair-report.pdf) "Around 10% of submissions were completely formalised, slightly more partially formalised". It is also debatable whether the challenge had a direct impact on the development of proof assistants and logical frameworks: specialized systems such as Abella [Baelde et al. 2014] and Beluga [Pientka and Cave 2015] were born out of independent research of the early 2000. To be generous, we could impute Abella's generalization of its specification logic to higher-order [Wang et al. 2013] to this Twelf POPLMark solution [Pientka 2007], but development in mainstream systems such as Coq, Agda, and (Nominal) Isabelle were largely driven by other (internal) considerations.

In a much more modest setting, but in tune with the goal of the PC, [Felty et al. 2017] recently presented some benchmarks with the intention of going beyond the issue of representing binders, whose pro and cons they consider well-understood. Rather, the emphasis was on 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. These are more or less supported in systems featuring some form of hypothetical and parametric reasoning, but the same issues occur in first-order representation as well; in this setting, typically, they are not recognized as crucial, rather they are considered one of the prices one has to pay when reasoning over open terms. This set of benchmarks was accompanied by a preliminary design of a common language and open repository [Felty et al. 2015b], which is fair to say did not have a resounding impact so far.

In the mean time, the PL world did not stand still, obviously. One element that we have picked on is the multiplication of the use of proofs by *logical relations* [Statman 1985] — not coincidentally, those featured in [Aydemir et al. 2005]'s section "Beyond the challenge". From the go-to technique to prove normalization of certain calculi, proofs by logical relations are now used to attack problems in the theory of complex languages models, with applications to issues in equivalence of programs, compiler correctness, representation independence and even more intensional properties such as non-interference, differential privacy and secure multi-language inter-operability, to cite just a few [Ahmed 2015; Bowman and Ahmed 2015; Neis et al. 2015].

Picking up on PC's final remark "We will issue a small number of further challenges [...]", we propose, as we detail in Section 2.3 a new challenge that we hope it will move the bar a bit forward. We suggest Strong Normalization (SN) for the simply-typed lambda-calculus proven via logical relations in the Kripke style formulation, see [Coquand 1991] for an early use. We discuss the rationale in the next Section.

2 THE CHALLENGE

2.1 Problem Selection

(Strong) normalization by Tait's method is a well-understood and reasonably circumscribed problem that has been a cornerstone of mechanized PL theory, starting from [Altenkirch 1993]. There are of course many alternative ways to prove SN for a lambda-calculus, see for example the inductive approach of [Joachimski and Matthes 2003], partially formalized in [Abel 2008], or by reduction from strong to weak normalization [Sorensen 1997]. For that matter, a SN proof via logical relations for the simply-typed lambda calculus can be carried out (see for a classic example [Girard et al. 1990]) without appealing to a Kripke definition of reducibility, at the cost, though, of a rather cavalier approach to "free" variables. However, the Kripke technique is handy in establishing SN for richer theories such as dependently typed ones, as well as for proving stronger results, for example about equivalence checking [Crary 2005; Harper and Pfenning 2005].

We claim that mechanizing such a proof is indeed challenging since:

- It focus on reasoning on *open* terms and on relating different contexts or *worlds*, taking seriously the Kripke analogy. The quantification over *all* extensions of the given world may be problematic for frameworks where contexts are only implicitly represented, or, on the flip side, may require several boring weakening lemmas in first-order representations.
- The definition of *reducibility* requires a sophisticated notion of inductive definition, which must be compatible with the binding structures, but also be able to take into account *stratification*, to tame the negative occurrence of the defined notion.
- Simultaneous substitutions and their equational theory (composition, commutation etc.) are central in formulating and proving the main result. For example, in the proof of the Fundamental Theorem 2.8, we need to push substitutions through (binding) constructs.

In this sense, this challenge goes well beyond the original PC, where the emphasis was on binder representations, proofs by structural induction and operational semantics animation.

Previous formalizations of strong normalization usually follows Girard's approach, see for example [Donnelly and Xi 2007] carried out in ATS/LF, or the one available in the Abella repository (abella-prover.org/~normalization/). Less frequent are formalizations following the Kripke discipline: both [Cave and Pientka 2015] and [Narboux and Urban 2008] encode [Crary 2005]'s account of decision procedures for term equivalence in the STLC, in Beluga and Nominal Isabelle respectively; the latter was then extended in [Urban et al. 2011] to formalize the analogous result for LF [Harper and Pfenning 2005]. See [Abel and Vezzosi 2014] for a SN Kripke-style proof for a more complex calculus and [Rabe and Sojakova 2013] for another take to handling dependent types — this paper also contains many more references to the literature.

The choice of a Kripke-style proof of SN for the STLC may sound contentious on several grounds and hence we will try to motivate it further:

- We acknowledge that SN is not the most exciting application of logical relations, some of which we have mentioned in the previous Section. Still, it is an important topic in type theory, in particular w.r.t. logical frameworks' meta-theory, see for example [Altenkirch and Kaposi 2016], and in this sense dear to our hearts. It is a well-known textbook example, which uses techniques that should be familiar to the community of interest in the simplest possible setting.
- Yes, the STLC is the prototypical toy language, while a POPL paper will address richer PL theory aspects. For one, adding more constructs, say in the PCF direction, perhaps with an iterator, would make the proof of the fundamental theorem longer, but not more interesting. Secondly, we think that a good benchmark should be simple enough that it could be tried out almost immediately if one is acquainted with proof-assistants. Conversely, it should encourage a PL theorist to start playing with proof assistants. Finally, we do suggest extensions of our challenge in the next Section.
- The requirement of the "Kripke-style" may seem overly constrictive, especially since this may not be strictly needed for the STLC. However, as we have argued before, this is meant as a springboard for more complex case studies, where this technique is forced on us. Remember that we are interested in *comparing* solutions. A more ambitious challenge may not solicit enough solutions, if the problem is too exotic or simply too lengthy.

2.2 Evaluation Criteria

One of the limitations of the PC experiment was in the *evaluation* of the solutions, although it is not easy to avoid the "trip to the zoo" effect, well-known from trying to comparing programming languages: there is no theory underlying the evaluation; criteria tend to be rather qualitative, and finally, the comparison itself may be lengthy [Felty et al. 2015a]. Within these limitations, of the proposed solutions we will take into consideration the:

- Size of the necessary infrastructure for defining the base language: binding, substitutions, renamings, contexts, together with substitution and other infrastructural lemmas.
- Size of the main development versus the main theorems in the on-paper proof, in particular, number of technical lemmas not having a direct counterpart in the on-paper proof.

More qualitatively, we will try to assess the:

- Ease of using the infrastructure for supporting binding, contexts, etc. How easy is it to apply the appropriate lemmas in the main proof? For example, does applying the equational theory of substitutions require low-level rewriting, or is it automatic?
- Ease of development of the overall proof; what support is present for proof construction, when not for proof and counterexample search?

2.3 The Challenge, Explained

Let us recall the definition of the STLC, starting with the grammar of terms, types, contexts and substitutions:

$$\begin{array}{lll} \text{Terms} & M, N & ::= x \mid \lambda x{:}T.M \mid M\ N \\ \text{Types} & T, S & ::= B \mid T \rightarrow S \\ \text{Context} & \Gamma & ::= \cdot \mid \Gamma, x{:}T \\ \text{Subs} & \sigma & ::= \epsilon \mid \sigma, N/x \end{array}$$

The static and dynamic semantics are standard and are depicted in Figure 1. Since we want to be very upfront about the fact that evaluation goes under a lambda and thus involves open terms, we make the context explicit even in the reduction rules, contrary to what, say, Barendregt would do. Note that, because of rule E-Abs, we do not need to assume that the base type is inhabited by a constant. We denote with $[\sigma]M$ the application of the simultaneous substitution σ to M and with $[\sigma_1]\sigma_2$ their composition.

We now define the set of *strongly-normalizing* terms as pioneered by [Altenkirch 1993] and by now usual:

$$\frac{\forall M'.\ \Gamma \vdash M \longrightarrow M' \quad \Gamma \vdash M' \in \mathsf{SN}}{\Gamma \vdash M \in \mathsf{SN}}\ SN-WF$$

expressing that the set of strongly normalizing terms is the well-founded part of the reduction relation. A more explicit formulation of strong normalization is allowed, see for example [Joachimski and Matthes 2003], but then an equivalence proof should be provided. Note that reasoning with the above rule *SN-WF* cannot proceed by structural induction, since it is not the case that M' is a sub-term of M.

The logical predicates have the following structure:

- $\Gamma \vdash M \in \mathcal{R}_T$, and
- $\Gamma' \vdash \sigma \in \mathcal{R}_\Gamma$.

We use a Kripke-style logical relations definition where we *witness* the context extension using a *weakening* substitution ρ. This can be seen as a *shift* in de Bruijn terminology, while other encodings may use different (or no particular) implementation techniques for handling context extensions.

Definition 2.1 (Reducibility Candidates).

- $\Gamma \vdash M \in \mathcal{R}_B$ iff $\Gamma \vdash M : B$ and $\Gamma \vdash M \in \mathsf{SN}$:
- $\Gamma \vdash M \in \mathcal{R}_{T \rightarrow S}$ iff $\Gamma \vdash M : T \rightarrow S$ and for all N, Δ such that $\Gamma \leq_\rho \Delta$, if $\Delta \vdash N \in \mathcal{R}_T$ then $\Delta \vdash ([\rho]M)\ N \in \mathcal{R}_S$.

As usual, we lift reducibility to substitutions:

Definition 2.2 (Reducible Substitutions).

- $\Gamma' \vdash \epsilon \in \mathcal{R}$.
- $\Gamma' \vdash \sigma, N/x \in \mathcal{R}_{\Gamma, x:T}$ iff $\Gamma' \vdash \sigma \in \mathcal{R}_\Gamma$ and $\Gamma' \vdash N \in \mathcal{R}_T$.

We now give an outline of the proof as a sequence of lemmas — the reader will find all the details in the forthcoming full version of this paper.

Lemma 2.3 (Semantic Function Application). *If* $\Gamma \vdash M \in \mathcal{R}_{T \rightarrow S}$ *and* $\Gamma \vdash N \in \mathcal{R}_T$ *then* $\Gamma \vdash M\ N \in \mathcal{R}_S$.

Proof. Immediate, by definition. □

Lemma 2.4 (SN Closure under Weakening). *If* $\Gamma_1 \leq_\rho \Gamma_2$ *and* $\Gamma_1 \vdash M \in \mathsf{SN}$ *then* $\Gamma_2 \vdash [\rho]M \in \mathsf{SN}$.

Proof. By induction on the derivation of $\Gamma_1 \vdash M \in \mathsf{SN}$. □

Lemma 2.5 (Closure of Reducibility under Weakening). *If* $\Gamma_1 \leq_\rho \Gamma_2$ *and* $\Gamma_1 \vdash M \in \mathcal{R}_T$ *then* $\Gamma_2 \vdash [\rho]M \in \mathcal{R}_T$.

Proof. By cases on the definition of reducibility using the above Lemma 2.4 and weakening for typing. □

Lemma 2.6 (Weakening of Reducible Substitutions). *If* $\Gamma_1 \leq_\rho \Gamma_2$ *and* $\Gamma_1 \vdash \sigma \in \mathcal{R}_\Phi$ *then* $\Gamma_2 \vdash [\rho]\sigma \in \mathcal{R}_\Phi$.

Proof. By induction on the derivation of $\Gamma_1 \vdash \sigma \in \mathcal{R}_\Phi$ using Closure of Reducibility under Weakening. □

Lemma 2.7 (Closure under Beta Expansion). *If* $\Gamma \vdash N \in \mathsf{SN}$ *and* $\Gamma \vdash [N/x]M \in \mathcal{R}_S$ *then* $\Gamma \vdash (\lambda x{:}T.M)\ N \in \mathcal{R}_S$.

Proof. By induction on S after a suitable generalization. □

Theorem 2.8 (Fundamental Theorem). *If* $\Gamma \vdash M : T$ *and* $\Gamma' \vdash \sigma \in \mathcal{R}_\Gamma$ *then* $\Gamma' \vdash [\sigma]M \in \mathcal{R}_T$.

Proof. By induction on $\Gamma \vdash M : T$. In the case for functions, we use Closure under Beta Expansion (Lemma 2.7) and Weakening of reducible substitution (Lemma 2.6). □

3 BEYOND THE CHALLENGE

There is an ongoing tension between weak and strong logical frameworks [de Bruijn 1991], with which we can encode our benchmarks. Weak frameworks are designed to accommodate advanced infrastructural features for binders (HOAS/nominal syntax etc.) and for judgments (hypothetical and parametric), but may struggle on other issues, such as facilities for computation or higher-order quantification/impredicativity. There are at least two coordinates in which we can directly extend our benchmark, to further highlight this dilemma:

- Logical relations for dependent types [Abel and Vezzosi 2014; Rabe and Sojakova 2013], up to the Calculus of Constructions. Here we need to go beyond first-order quantification, which is typically what is on offer in weak frameworks.
- Proof by logical relation via *step-indexing* [Appel and McAllester 2001]. Here we have two issues:
  1. the logical relation may even be harder to be accepted by the meta-language as an inductive definition than with simple types; in fact, the work around the negative occurrence of the defined relation cannot be based on structural induction on types, but it has to use some form of course-of-value induction.
  2. It involves a limited amount of arithmetic reasoning:
     "definitions and proofs have a tendency to become cluttered with extra indices and even arithmetic, which are really playing the role of construction line." ([Benton and Hur 2010]).

  This latter point may be problematic for frameworks such as Abella and Beluga, which do not (yet) have extensive libraries, nor computational mechanisms (rewriting, reflection) for those tasks.

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

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ u \qquad \frac{\Gamma, x : T \vdash M : S}{\Gamma \vdash (\lambda x{:}T.M) : (T \to S)}\ \text{T-Abs}^{x} \qquad \frac{\Gamma \vdash M : (T \to S) \quad \Gamma \vdash N : T}{\Gamma \vdash (M\ N) : S}\ \text{T-App}$$

$\boxed{\Gamma \vdash M \longrightarrow M'}$ Term M steps to term M' in context Γ

$$\frac{\Gamma, x{:}T \vdash M \longrightarrow M'}{\Gamma \vdash \lambda x{:}T.M \longrightarrow \lambda x{:}T.M'}\ \text{E-Abs}^{x} \qquad \frac{}{\Gamma \vdash (\lambda x{:}T.M)\ N \longrightarrow [N/x]M}\ \text{E-App-Abs} \qquad \frac{\Gamma \vdash M \longrightarrow M'}{\Gamma \vdash M\ N \longrightarrow M'\ N}\ \text{E-App2} \qquad \frac{\Gamma \vdash N \longrightarrow N'}{\Gamma \vdash M\ N \longrightarrow M\ N'}\ \text{E-App1}$$

Figure 1: Typing and reduction rules for the STLC

4 CALL FOR ACTION

We ask the community to submit solutions and we plan to invite everyone who does so to contribute towards a joint paper discussing trade-offs between them. The authors commit themselves to produce solutions in Agda, Abella and Beluga. To resurrect the slogan from the PC, a small step (excuse the pun) for us, a big step for bringing mechanized meta-theory to the masses!

REFERENCES

A. Abel. Normalization for the simply-typed lambda-calculus in twelf. *Electronic Notes in Theoretical Computer Science*, 199:3 – 16, 2008. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/j.entcs.2007.11.009. Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004).

A. Abel and A. Vezzosi. A formalized proof of strong normalization for guarded recursive types. In *APLAS*, volume 8858 of *Lecture Notes in Computer Science*, pages 140–158. Springer, 2014.

A. Ahmed. Verified compilers for a multi-language world. In *SNAPL*, volume 32 of *LIPIcs*, pages 15–31. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

T. Altenkirch. A formalization of the strong normalization proof for system F in LEGO. In *TLCA*, volume 664 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 1993.

T. Altenkirch and A. Kaposi. Type theory in type theory using quotient inductive types. In *POPL*, pages 18–29. ACM, 2016.

A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, Sept. 2001. ISSN 0164-0925. doi: 10.1145/504709.504712.

B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLMARK challenge. In *Eighteenth International Conference on Theorem Proving in Higher Order Logics*, volume 3603 of *LNCS*, pages 50–65. Springer, 2005.

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

N. Benton and C. Hur. Step-indexing: The good, the bad and the ugly. In *Modelling, Controlling and Reasoning About State*, volume 10351 of *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2010.

W. J. Bowman and A. Ahmed. Noninterference for free. In *ICFP*, pages 101–113. ACM, 2015.

A. Cave and B. Pientka. A case study on logical relations using contextual types. In *LFMTP*, volume 185 of *EPTCS*, pages 33–45, 2015.

T. Coquand. An algorithm for testing conversion in type theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.

K. Crary. Logical relations and a case study in equivalence checking. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. The MIT Press, 2005.

N. de Bruijn. A plea for weaker frameworks. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 40–67. Cambridge University Press, 1991.

K. Donnelly and H. Xi. A formalization of strong normalization for simply-typed lambda-calculus and system F. *Electr. Notes Theor. Comput. Sci.*, 174(5):109–125, 2007.

A. Felty, A. Momigliano, and B. Pientka. The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 2—a survey. *Journal of Automated Reasoning*, 55(4):307–372, 2015a. ISSN 0168-7433. doi: 10.1007/s10817-015-9327-3.

A. Felty, A. Momigliano, and B. Pientka. Benchmarks for reasoning with syntax trees containing binders and contexts of assumptions. *MATHEMATICAL STRUCTURES IN COMPUTER SCIENCE*, 2017. doi: 10.1017/S0960129517000093.

A. P. Felty, A. Momigliano, and B. Pientka. An open challenge problem repository for systems supporting binders. In *LFMTP*, volume 185 of *EPTCS*, pages 18–32, 2015b.

J.-Y. Girard, Y. Lafont, and P. Tayor. *Proofs and types*. Cambridge University Press, 1990.

R. Harper and F. Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Transactions on Computational Logic*, 6(1):61–101, 2005.

F. Joachimski and R. Matthes. Short proofs of normalization for the simply- typed λ-calculus, permutative conversions and Gödel's T. *Archive for Mathematical Logic*, 42(1):59–87, Jan 2003. ISSN 1432-0665. URL http://dx.doi.org/10.1007/s00153-002-0156-9.

J. Narboux and C. Urban. Formalising in nominal isabelle crary's completeness proof for equivalence checking. *Electr. Notes Theor. Comput. Sci.*, 196:3–18, 2008.

G. Neis, C. Hur, J. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis. Pilsner: a compositionally verified compiler for a higher-order imperative language. In *ICFP*, pages 166–178. ACM, 2015.

B. Pientka. Proof pearl: The power of higher-order encodings in the logical framework LF. In *Twentieth International Conference on Theorem Proving in Higher-Order Logics*, LNCS, pages 246–261. Springer, 2007.

B. Pientka and A. Cave. Inductive beluga: Programming proofs. In *CADE*, volume 9195 of *Lecture Notes in Computer Science*, pages 272–281. Springer, 2015.

F. Rabe and K. Sojakova. Logical relations for a logical framework. *ACM Trans. Comput. Logic*, 14(4):32:1–32:34, Nov. 2013. doi: 10.1145/2536740.2536741.

M. H. Sorensen. Strong normalization from weak normalization in typed λ-calculi. *Information and Computation*, 133(1):35 – 71, 1997. ISSN 0890-5401. doi: http://dx.doi.org/10.1006/inco.1996.2622. URL http://www.sciencedirect.com/science/article/pii/S089054019692622X.

R. Statman. Logical relations and the typed lambda-calculus. *Information and Control*, 65(2/3):85–97, 1985.

G. Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

C. Urban, J. Cheney, and S. Berghofer. Mechanizing the metatheory of lf. *ACM Trans. Comput. Logic*, 12(2):15:1–15:42, Jan. 2011. ISSN 1529-3785. doi: 10.1145/1877714.1877721.

Y. Wang, K. Chaudhuri, A. Gacek, and G. Nadathur. Reasoning about higher-order relational specifications. In *Fifteenth International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 157–168. ACM Press, 2013.