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]

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

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

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

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

$$R' = \sigma_c(\sigma_d(R_f)) \tag{2}$$

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

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

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

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

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,

[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.

```
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.

[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 $\sigma \leftarrow \sigma_u \circ \sigma$ 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.