View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Wisconsin-Milwaukee

University of Wisconsin Milwaukee
UWM Digital Commons

Theses and Dissertations

August 2017

Making Substitutions Explicit in SASyLf

Michael David Ariotti
University of Wisconsin-Milwaukee

Follow this and additional works at: https://dc.uwm.edu/etd

 Part of the Computer Sciences Commons

Recommended Citation
Ariotti, Michael David, "Making Substitutions Explicit in SASyLf" (2017). *Theses and Dissertations*. 1576.
https://dc.uwm.edu/etd/1576

This Thesis is brought to you for free and open access by UWM Digital Commons. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of UWM Digital Commons. For more information, please contact open-access@uwm.edu.

MAKING SUBSTITUTIONS EXPLICIT IN SASYLF

by

Michael D. Ariotti

A Thesis Submitted in

Partial Fulfillment of the

Requirements for the degree of

Master of Science

in Computer Science

at

The University of Wisconsin-Milwaukee

August 2017

ABSTRACT

MAKING SUBSTITUTIONS EXPLICIT IN SASYLF

by

Michael D. Ariotti

The University of Wisconsin-Milwaukee, 2017
Under the Supervision of Professor John Tang Boyland

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 thesis describes a new usability feature of SASyLF, "where" clauses, which make explicit previously hidden substitutions that arise through constructs in the proof code, primarily case analyses. An overview of SASyLF and logical frameworks is given, with motivating examples. 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.

© Copyright by Michael D. Ariotti, 2017
All Rights Reserved

To

my wife and son

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

There are many who deserve my appreciation and acknowledgement, without whom this work would not have been possible. Foremost is my research advisor, John Tang Boyland. It has been a pleasure to work with him on a software tool which both offers theoretical challenges and has real classroom application. Dr. Boyland has given me countless hours of his time and expertise; I hope that my contributions in this work are put to good use for future students.

I would like to thank my other committee members Christine Cheng and Ethan Munson for both their support and their advice in many areas.

I would also like to thank my wife Meghan for giving me unending support and encouragement. Many thanks also to Meghan again, and also to her parents Bruce and Cathy for taking care of my son Matty, and allowing me many days, nights, and weekends to read, write, and code while knowing he was in loving hands.

1 Introduction

SASyLF [1] (**S**econd-order **A**bstract **Sy**ntax **L**ogical **F**ramework) is an open-source, interactive proof assistant whose goal is to teach students how to write sound proofs. Like many other proof assistants, SASyLF's intended domain is programming language meta-theory, though it can be used to write proofs about many languages and logics. What sets SASyLF apart from other proof assistants is that the time required to learn to use it is typically much less, for two reasons. First, SASyLF's syntax (the form of its code) is designed to be readable by novices: "[language description and proofs] should be written as closely as possible to the way it is done on paper" [1, p. 31]. Second, when the user makes a mistake, SASyLF generates errors that are as descriptive and local to the cause as possible, and often offers suggestions for correction. SASyLF's ability to generate helpful errors is a key part of its design, and is facilitated by the fine granularity of its verification implementation.

The benefits for students using such a tool rather than writing proofs by hand is explained by its designers: "[when writing proofs, students make many mistakes] at a much lower level, e.g. skipping a step in a proof or applying an inference rule when the facts used do not match the rule's premises. Students may not even recognize they have made a mistake, and so do not seek out help" [1, p. 31].

On the other hand, SASyLF does little automatically, without user input. It does not write proofs on its own, and this is by design. The designers write, "we could add ... automated proof search, but ... novices could use it as a shortcut and thereby avoid learning the basic details of proofs" [1, p. 32]. Thus, SASyLF strikes a balance between providing guidance to students without doing their work for them, much as a human teacher would.

SASyLF was originally designed and developed as a command-line tool by Jonathan Aldrich and others. It is currently maintained[1] by John Tang Boyland, who uses the assistant with students to teach courses on type systems.

Since its initial development, many extensions have been implemented, in accordance with its educational design philosophy. Perhaps the most impactful extension is a plugin for Eclipse, which allows users to write proofs in an integrated development environment (IDE). This environment colors keywords and other constructs of the proof code; displays an outline of current syntax, judgments, and theorems; and highlights any errors and warnings on the code itself. Similarly interactive IDEs are offered with other proof assistants, including Coq and Isabelle.

This thesis describes another extension to SASyLF, called "where" clauses [2]. Beyond the original paper, this thesis gives a full description of its example language, as well as a brief overview of logical frameworks. In the chapters that follow, correctness is defined formally both for "where" clauses and for rule cases in which they typically appear. Finally, in Chapter 5, future work regarding the new feature is discussed.

[1] Both the interactive and command-line versions of SASyLF, along with their documentation and source, are available at `http://github.com/boyland/sasylf`.

2 An Overview of SASyLF with Where Clauses

This chapter gives a brief overview of the services SASyLF provides and how the assistant can be used. In particular, a language is introduced in SASyLF code, which will be referenced throughout this thesis. Additionally, an example theorem is given, and it is shown how "where" clauses can help write a proof for it.

2.1 Logical Frameworks and Language Descriptions

A *logical framework* is a means of storing or encoding logical information in such a way that a computer can reason about it [9, p. 17]. The information to study is a specification for a language or logic, typically a programming language.

SASyLF stores logical information in the dependently-typed lambda calculus called simply LF [3]; SASyLF's older brother Twelf [7] uses LF for this purpose as well.

A language being represented in a logical framework is called an *object language*—the object of study. In this representation, meta-theoretic properties (e.g., type soundness) can be proven about an object language in a computer-assisted or even fully-automated fashion. The beauty of this technique is that if (1) the logical framework is sound, and (2) the object language is adequately represented in it, then properties proven about the object language within the logical framework must hold for the language elsewhere.

As described by Pfenning [5], the formal description of a programming language often consists of three parts:

(1) *Abstract syntax*: typically defined in Backus-Naur form (BNF), this imposes inductive structure on programs written in the language. This is not to be confused with the *concrete syntax* of the language, which refers to the character-based tokens which are parsed from source code.

(2) *Type system*: in short, a classification system for terms in the language. A type system generally restricts the operations which can be legally performed on a given term, for the sake of detecting programmer errors.

(3) *Operational semantics*: also known as evaluation rules, these define the behavior of correctly-written programs in the language.

Parts (2) and (3) above are often defined via a deductive system of *inference rules*. An inference rule looks like ([5, p. 5])

$$\frac{J_1 \ \ldots \ J_n}{J}$$

where the J's represent *judgments*, units of logical information which can be true or false. An inference rule represents a logical implication: if all of the judgments J_1 through J_n (the

premises) are true, then we can *infer* judgment J (the *conclusion*) is also true. Inference rules with no premises (i.e., where $n = 0$) are sometimes called *axioms*.

Inference rules are usually *schematic* [5]. This means the judgments within the rule are not fixed—or *ground*—but are allowed to vary through the use of meta-variables. In this way, a schematic inference rule acts as a template; it is not meant only to hold for a specific set of object terms, but *for all* sets of object terms which fit the pattern of the rule. This allows them to describe relations concisely.

In SASyLF, both abstract syntax and inference rules appear in the code much as they would on paper. The abstract syntax of an object language is written with BNF rules. Judgments are each introduced with a form, followed by a set of inference rules. An example language is described both on paper and in SASyLF in the next section.

2.1.1 An Example Language

Figure 2.1 presents the language $\lambda_{\rightarrow\mathbf{B}}$, which is the simply-typed λ-calculus with the addition of booleans; this language mostly comes from Pierce [8], though the original SASyLF paper [1] used the simply-typed λ-calculus for its code examples as well. For this thesis, $\lambda_{\rightarrow\mathbf{B}}$ will provide examples for many of the topics of discussion.

The language description in Figure 2.1 contains the three components listed in the previous section: an abstract syntax, a type system, and evaluation rules (in the style of Pierce [8]). Terms `t` and types `T` are defined, as well as values `v` (which are also terms). The symbol Γ represents a context containing variables `x` marked with their types. The typing and evaluation relations are denoted with forms, followed by inference rules which define the relations. The notation $[\mathtt{v}_2/\mathtt{x}]\mathtt{t}_1$ located in the conclusion of rule E-AppAbs refers to substitution—that is, replacing all appearances of `x` in the term $\mathtt{t}_1$ with $\mathtt{v}_2$. This rule also defines β-reduction for $\lambda_{\rightarrow\mathbf{B}}$.

Figures 2.2, 2.3, and 2.4 give the same descriptions in SASyLF. The SASyLF versions closely resemble their "paper" equivalents. There are a few differences:

(1) The terminals of the language (`lam`, `Bool`, etc.) are listed explicitly for the aid of the generated parser, and the student user. (The terminals `lam` and `dot`, as well as the operator `->` could be written with Unicode characters in SASyLF, to be even closer to the Pierce text. In this thesis they are written with ASCII for the sake of formatting.)

(2) The definition of λ-abstractions includes `[x]` in the body. For the object language, the notation `t[x]` means that variable `x` may appear free in term `t`, and is the very same variable bound in the abstraction. Of course, `x` can be free in `t` in the paper version as well; it is just not explicit there.

(3) Rather than define values with syntax, "value" has become a judgment (a new relation on terms); examples of this judgment appear in rules E-App2 and E-AppAbs. This value judgment is defined in Figure 2.5.

(4) The typing relation in SASyLF `assumes Gamma`. Essentially, this means that the result of the relation may depend on variables in the context (and their types). This is present in the paper version of the relation, but in the SASyLF version, the dependency must be explicit.

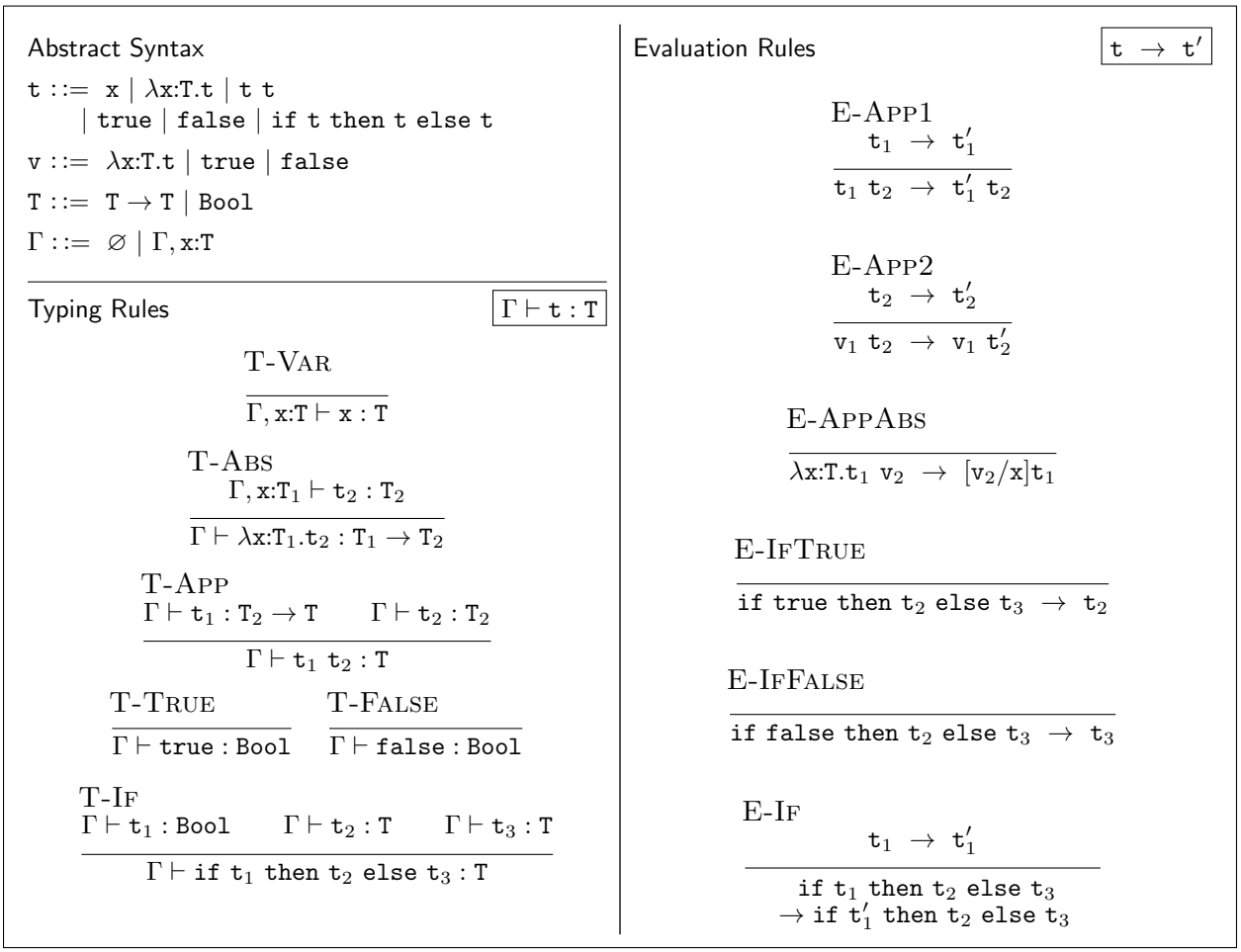

Figure 2.1: An example language, $\lambda_{\rightarrow \mathbf{B}}$ (on "paper")

(5) Using a similar notation to (2), the substitution defined by $[\mathtt{v}_2/\mathtt{x}]\mathtt{t}_1$ in the paper description is defined in SASyLF as `t1[t2]` (where `t2` is a value per the premise of rule E-AppAbs).

The next section discusses how theorems are written in SASyLF.

2.2 Theorems and Proofs

Once an object language has been described in SASyLF, theorems (and lemmas) can be written and proven about the meta-theory of the language. SASyLF will then verify the statement of each theorem, as well as its proof.

In SASyLF, each theorem $\mathcal{T}$ has an ordered list of *logical inputs*

$$x_1 : \tau_1,\ x_2 : \tau_2,\ \ldots,\ x_n : \tau_n$$

These inputs are *universal*—denoted in SASyLF by the `forall` keyword—meaning that the theorem holds true for every possible set of inputs with correct type(s). These inputs

```
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 2.2: An abstract syntax for $\lambda_{\rightarrow \mathbf{B}}$ in SASyLF

```
judgment typing: Gamma |- t : T
assumes Gamma

------------------- T-Var
Gamma, x:T |- x : T

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

Gamma |- t1 : T2 -> T
Gamma |- t2 : T2
-------------------- T-App
Gamma |- t1 t2 : T
```

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

-------------------- T-False
Gamma |- false : Bool

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

Figure 2.3: Typing rules for $\lambda_{\rightarrow \mathbf{B}}$ in SASyLF

represent initial assumptions, and form a local context Γ in $\mathcal{T}$, similarly to how formal parameters are treated as (the first) local variables in a programmatic function.

Assumptions in SASyLF are represented in LF, and since LF is dependently typed, often the type of an input x_i depends on the types of one or more inputs previous to x_i in the list. Thus, some inputs are syntactic constructs (which have no dependencies), while others can be schematic judgments on those constructs. Syntax inputs which appear in explicit schematic input judgments do not need to be explicitly listed themselves as input to the theorem, unless induction is performed on them during the proof.

Theorem $\mathcal{T}$ also has an ordered list of *logical outputs*

$$y_1 : \tau_{(1)},\ y_2 : \tau_{(2)},\ \ldots,\ y_n : \tau_{(m)}, y : \tau$$

where parentheses around the type subscripts denote that the types may be different from the types of the inputs. In SASyLF, the last output $y : \tau$ is the only one that appears in the

```
judgment eval: t -> t

t1 -> t1'
---------------- E-App1
t1 t2 -> t1' t2

t1 value
t2 -> t2'
---------------- E-App2
t1 t2 -> t1 t2'

t2 value
-------------------------------- E-AppAbs
(lam x:T dot t1[x]) t2 -> t1[t2]
```

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

------------------------------- E-IfFalse
if false then t2 else t3 -> t3

t1 -> t1'
------------------------ E-If
if t1 then t2 else t3 ->
if t1' then t2 else t3
```

Figure 2.4: Evaluation rules for $\lambda_{\rightarrow\mathbf{B}}$ in SASyLF

```
judgment value: t value

---------- V-True
true value

----------- V-False
false value

---------------------- V-Abs
lam x:T dot t[x] value
```

Figure 2.5: The value judgment for $\lambda_{\rightarrow\mathbf{B}}$ in SASyLF

code[1], following the keyword `exists`. The type of this last output is dependent on each of the previous types $\tau_{(j)}$ in the list (which means that all but the last output must be syntax), and may be dependent on syntactic construct types τ_i in the list of inputs as well.

These outputs of $\mathcal{T}$ are *existential*, meaning that for every possible set of inputs, at least one set of outputs can be produced. In fact, this is how $\mathcal{T}$ is proved: via a total recursive function whose output has type τ, usually defined by cases on the inputs. All the SASyLF constructs presented so far are represented internally in LF: the language syntax, the inference rules, even the inputs and outputs of theorems. The function which proves a theorem is represented separately, in a form which closely resembles Schürmann's meta-language $\mathcal{M}_2^+$ [9].

To prove a theorem, its proof-as-a-function must produce a derivation d with the same

[1]The newest versions of SASyLF allow derivation constructs with `and` and `or`, which can be used to simulate multiple outputs in a theorem; these constructs are mostly outside of the scope of this thesis.

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 expression which has that type, and the method of its creation, are in the hands of the proof function.

The next section describes techniques a SASyLF user can employ to produce an output $d{:}\tau$ from inputs $x_i{:}\tau_i$.

2.2.1 Proof Techniques

To prove a theorem, the SASyLF user has the following techniques available to produce derivations, and ultimately a derivation of the type τ matching the type of the theorem's output. Most of these correspond well to proof techniques described by Schürmann [9].

(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 [6]: `by weakening`, `by exchange`, and `by substitution`.

The semantics of proof by case analysis (2) is the subject of this thesis. In particular it will be shown, as it is by Schürmann [9], that a case analysis represents a simultaneous substitution applied to all assumptions in the context Γ. The new feature of the SASyLF language and the contribution of this thesis, namely "where" clauses, is designed to make these substitutions explicit.

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[2].

The example theorem in the following sections illustrates this.

[2]The 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.

2.2.2 An Example Theorem

The proofs for the type soundness of $\lambda_{\rightarrow\mathbf{B}}$ (progress and preservation) can be easily written in SASyLF, following those written from Pierce [8]. In fact, many of the language meta-theory proofs in Pierce's book use only the techniques described in §2.2.1. (This makes SASyLF particularly well-suited to be used by students learning topics from Pierce.)

Pierce's theorem of type preservation [8, p. 107] for the simply-typed λ-calculus is stated succinctly in English: If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{t} \rightarrow \mathtt{t}'$, then $\Gamma \vdash \mathtt{t}' : \mathtt{T}$. The natural-language ("paper") proof of this theorem for the segments of the language $\lambda_{\rightarrow\mathbf{B}}$ proceeds by structural induction on the typing derivation $\Gamma \vdash \mathtt{t} : \mathtt{T}$, via case analysis on the final rule applied to produce this judgment. (This is a straightforward application of technique (2) from the previous section.) This form of induction means that during the proof, this preservation theorem is assumed to hold true for subderivations of $\Gamma \vdash \mathtt{t} : \mathtt{T}$, which varies from case to case. Following is a rendering of this proof in Pierce's style, for these language segments.

Case T-VAR: $\mathtt{t} = \mathtt{x}$ $\Gamma = \Gamma', \mathtt{x}:\mathtt{T}$

> This case cannot occur, but the reason is subtle. Because $\mathtt{t}$ appears in a judgment without Γ, namely the evaluation judgment $\mathtt{t} \rightarrow \mathtt{t}'$, this means that $\mathtt{t}$'s type does not depend on any assumptions contained in Γ. In other words, the term $\mathtt{t}$ is *closed* with respect to Γ, though Γ is allowed to be non-empty. If the last rule in $\mathtt{t}$'s typing derivation is T-VAR, then $\mathtt{t}$'s type $\mathtt{T}$ *does* depend on Γ, rewritten as $\Gamma', \mathtt{x}:\mathtt{T}$ above, which is a contradiction.

Case T-ABS: $\mathtt{t} = \lambda\mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2$ $\mathtt{T} = \mathtt{T}_1 \rightarrow \mathtt{T}_2$ $\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2$

> This case cannot occur, because abstraction terms (by themselves) do not evaluate.

Case T-APP: $\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$ $\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_2 \rightarrow \mathtt{T}$ $\Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2$

> Given that $\mathtt{t}$ has the form of an application $\mathtt{t}_1\ \mathtt{t}_2$, there are three rules by which $\mathtt{t}$ can evaluate to $\mathtt{t}'$: E-APP1, E-APP2, E-APPABS. Each rule must be addressed with its own case.
>
> *Subcase* E-APP1: $\mathtt{t}' = \mathtt{t}_1'\ \mathtt{t}_2$ $\mathtt{t}_1 \rightarrow \mathtt{t}_1'$
>
> > By applying the inductive hypothesis to the judgments $\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_2 \rightarrow \mathtt{T}$ and $\mathtt{t}_1 \rightarrow \mathtt{t}_1'$ the judgment $\Gamma \vdash \mathtt{t}_1' : \mathtt{T}_2 \rightarrow \mathtt{T}$ is obtained. Applying rule T-APP to the judgments $\Gamma \vdash \mathtt{t}_1' : \mathtt{T}_2 \rightarrow \mathtt{T}$ and $\Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2$ results in the judgment $\Gamma \vdash \mathtt{t}_1'\ \mathtt{t}_2 : \mathtt{T}$, which is what needed to be shown for this case.
>
> *Subcase* E-APP2: $\mathtt{t}' = \mathtt{v}_1\ \mathtt{t}_2'$ $\mathtt{t}_1$ is a value $\mathtt{v}_1$ $\mathtt{t}_2 \rightarrow \mathtt{t}_2'$
>
> > Similar to the E-APP1 case: apply the induction hypothesis, then rule T-APP to the result to obtain $\Gamma \vdash \mathtt{v}_1\ \mathtt{t}_2' : \mathtt{T}$.
>
> *Subcase* E-APPABS: $\mathtt{t}' = [\mathtt{v}_2/\mathtt{x}]\mathtt{t}_{11}$ $\mathtt{t}_1 = \lambda\mathtt{x}{:}\mathtt{T}_2.\mathtt{t}_{11}$ $\mathtt{t}_2$ is a value $\mathtt{v}_2$
>
> > Proving this case requires the "substitution" lemma—that is, that types are preserved through substitution. More formally, the lemma states [8, p. 106] that if $\Gamma, \mathtt{x}{:}\mathtt{T}_4 \vdash \mathtt{t}_3 : \mathtt{T}_3$ and $\Gamma \vdash \mathtt{t}_4 : \mathtt{T}_4$, then $\Gamma \vdash [\mathtt{t}_4/\mathtt{x}]\mathtt{t}_3 : \mathtt{T}_3$. (The subscripts in the

definition have been chosen so as not to interfere with the theorem being proved.) The proof of this lemma is omitted here for brevity.

To use this lemma, first combine the judgment $\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_2 \to \mathtt{T}$ (from the beginning of the case) and the knowledge that $\mathtt{t}_1 = \lambda \mathtt{x}{:}\mathtt{T}_2.\mathtt{t}_{11}$ to form the judgment $\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_2.\mathtt{t}_{11} : \mathtt{T}_2 \to \mathtt{T}$. Then, invert this judgment through rule T-Abs (the only typing rule which types abstractions) to obtain $\Gamma, \mathtt{x}{:}\mathtt{T}_2 \vdash \mathtt{t}_{11} : \mathtt{T}$. Finally, since $\mathtt{v}_2$ is the same as $\mathtt{t}_2$ and $\Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2$ (again from the beginning of the case), $\Gamma \vdash [\mathtt{v}_2/\mathtt{x}]\mathtt{t}_{11} : \mathtt{T}$ by the substitution lemma.

Case T-True: $\mathtt{t} = \mathtt{true}$ $\mathtt{T} = \mathtt{Bool}$

This case cannot occur, because the term `true` does not evaluate.

Case T-False: $\mathtt{t} = \mathtt{false}$ $\mathtt{T} = \mathtt{Bool}$

This case cannot occur, for the same reason as T-True.

Case T-If: $\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$ $\Gamma \vdash \mathtt{t}_1 : \mathtt{Bool}$ $\Gamma \vdash \mathtt{t}_2 : \mathtt{T}$ $\Gamma \vdash \mathtt{t}_3 : \mathtt{T}$

There are three rules by which $\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$ can evaluate: E-IfTrue, E-IfFalse, and E-If. Each rule must be addressed in its own case.

Subcase E-IfTrue: $\mathtt{t}_1 = \mathtt{true}$ $\mathtt{t}_2 = \mathtt{t}'$

The term $\mathtt{t}_1$ must be `true` in this case, which means $\mathtt{t}'$ is the same as $\mathtt{t}_2$. Furthermore, the type of $\mathtt{t}_2$ is known to be `T`, from a subderivation of `t`'s typing derivation for this case.

Subcase E-IfFalse: $\mathtt{t}_1 = \mathtt{false}$ $\mathtt{t}_3 = \mathtt{t}'$

Similar to the previous case, except that $\mathtt{t}_1$ is `false` and $\mathtt{t}'$ is $\mathtt{t}_3$. The type of $\mathtt{t}_3$ is known to be `T` for the same reason.

Subcase E-If: $\mathtt{t}' = \mathtt{if}\ \mathtt{t}'_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$ $\mathtt{t}_1 \to \mathtt{t}'_1$

Applying the inductive hypothesis to $\Gamma \vdash \mathtt{t}_1 : \mathtt{Bool}$ and $\mathtt{t}_1 \to \mathtt{t}'_1$ yields $\Gamma \vdash \mathtt{t}'_1 : \mathtt{Bool}$. Reapplying rule T-If to this result, along with the typing subderivations for both $\mathtt{t}_2$ and $\mathtt{t}_3$ yields $\Gamma \vdash \mathtt{if}\ \mathtt{t}'_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}$. ■

In this example proof, only the techniques (1) and (2) are used, including all three versions of the former. Within each case, term equalities are listed, followed by a (possibly empty) list of judgments. The latter represent the premises of the inference rule being addressed by the case, rewritten in terms of known information in the proof. The listed equalities represent restrictions imposed by assuming the inference rule was the last one used in the derivation of $\Gamma \vdash \mathtt{t} : \mathtt{T}$ (or $\mathtt{t} \to \mathtt{t}'$ for the subcases). The significance of these equality relations, and the importance of making them explicit, is the core of this thesis. To help explain, Figures 2.6 and 2.7 show the same proof in SASyLF.

There are two explicit inputs to the SASyLF formulation of 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

```
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
  _: Gamma, x:T1 |- t1[x] : T2
  ---------------------------------- T-Abs
  _: Gamma |- lam x:T1 dot t1[x] :
       T1 -> T2
is
  proof by contradiction on e
end case
case rule
  d1: Gamma |- t1 : T2 -> T
  d2: Gamma |- t2 : T2
  ------------------------- T-App
  _: Gamma |- t1 t2 : T
is
  proof by case analysis on e:
    case rule
      e1: t1 -> t1'
      ------------------ E-App1
      _: t1 t2 -> t1' t2
    is
      d1': Gamma |- t1' : T2 -> T
        by induction hypothesis on d1, e1
      _: Gamma |- t1' t2 : T
        by rule T-App on d1', d2
    end case
    case rule
      _: t1 value
      e2: t2 -> t2'
      ------------------ E-App2
      _: t1 t2 -> t1 t2'
    is
      d2': Gamma |- t2' : T2
        by induction hypothesis on d2, e2
      _: Gamma |- t1 t2' : T
        by rule T-App on d1, d2'
    end case
    case rule
      v2: t2 value
      -------------------------- E-AppAbs
      _: (lam x:T2' dot t11[x]) t2 ->
           t11[t2]
    is
      d11: Gamma, x:T2' |- t11[x] : T
        by inversion of rule T-Abs on d1
      _: Gamma |- t11[t2] : T
        by substitution on d11, d2
    end case
  end case analysis
end case
...
```

Figure 2.6: The preservation proof for $\lambda_{\rightarrow \mathbf{B}}$ in SASyLF (Part 1)

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.

Theorem `preservation` in Figure 2.6 has a single output type, `Gamma |- t' : T`. As with all theorems in SASyLF, the nature of this output is existential (as explained in §2.2). However, in this theorem the output's dependencies—`t'` and `T`—are not existential, because they are inputs.

The proof in Figure 2.6 begins by declaring it will use induction on derivation `d`. Semantically this signifies structural induction[3], which means that the user is allowed to apply the

[3]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 thesis.

```
...
case rule
  ----------------------- T-True
  _: Gamma |- true : Bool
is
  proof by contradiction on e
end case
case rule
  ----------------------- T-False
  _: Gamma |- false : 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
    case rule
      ------------------------- E-IfFalse
      _: if false then t2 else t' -> t'
    is
      proof by d3
    end case
    case rule
      e1: t1 -> t1'
      ---------------------------- E-If
      _: if t1 then t2 else t3 ->
            if t1' then t2 else t3
    is
      d1': Gamma |- t1' : Bool
        by induction hypothesis on d1, e1
      _: Gamma |-
            if t1' then t2 else t3 : T
        by rule T-If on d1', d2, d3
    end case
  end case analysis
end case
end case analysis
end theorem
```

Figure 2.7: The preservation proof for $\lambda_{\rightarrow \mathbf{B}}$ in SASyLF (Part 2)

theorem being proved, during its proof, to a subderivation of `d` with similar LF type.

The proof proceeds via case analysis[4] on the typing derivation `d`, just as the paper proof does. The cases which must be addressed by the SASyLF proof are the same as the paper one: one for each typing rule. (For readability, the cases are listed in the same order in both proofs.) The rule T-Var is notably absent from the SASyLF proof, however. This is due to the form of the rule in Figure 2.3. The judgment `Gamma, x:T |- x : T` (not the same `Gamma` as in the theorem) implies that the term being typed depends on the hypothetical context, because it is a member of that context. However, it has been shown above that this cannot be. And so this rule does not need to be addressed with a case.

As in the paper proof, many of these cases in the SASyLF proof lead to an immediate contradiction (e.g., the T-True case), 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. This is in accordance

[4]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:`.

with the reasoning for a lack of automation, described in Chapter 1.

In the case for T-If in Figure 2.7, `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 §3.2.

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; the rules which apply are E-IfTrue, E-IfFalse, and E-If.

The proof for the first of these three rules, E-IfTrue, is completed in a single step. Notably, none of the techniques from §2.2.1 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 2.4.

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 much of the remainder of this thesis, bring these substitutions to light.

2.2.3 A Where Clause Example

Figures 2.8 and 2.9 show the same SASyLF proof 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.

Another notable substitution takes place when the rule T-Abs is inverted during the (sub)case for E-AppAbs. Here, the input type of `t1`, which must be an abstraction, is forced to match the left-hand side of its arrow type `T2 -> T` given by the judgment `d1`. This is required to apply the SASyLF construct `by substitution` in the next step of the proof.

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. Interestingly, these clauses appear, after a fashion, in the paper proof as well, and even in Pierce's original [8]. In this way then, having these clauses appear in the SASyLF code bring it even closer to the paper version.

```
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
  _: Gamma, x:T1 |- t1[x] : T2
  ------------------------------- T-Abs
  _: Gamma |- lam x:T1 dot t1[x] :
       T1 -> T2
  where t := lam x:T1 dot t1[x]
  and T := T1 -> T2
is
  proof by contradiction on e
end case
case rule
  d1: Gamma |- t1 : T2 -> T
  d2: Gamma |- t2 : T2
  ------------------------ T-App
  _: Gamma |- t1 t2 : T
  where t := t1 t2
is
  proof by case analysis on e:
    case rule
      e1: t1 -> t1'
      ----------------- E-App1
      _: t1 t2 -> t1' t2
      where t' := t1' t2
    is
      d1': Gamma |- t1' : T2 -> T
        by induction hypothesis on d1, e1
      _: Gamma |- t1' t2 : T
        by rule T-App on d1', d2
    end case
    case rule
      _: t1 value
      e2: t2 -> t2'
      ----------------- E-App2
      _: t1 t2 -> t1 t2'
      where t' := t1 t2'
    is
      d2': Gamma |- t2' : T2
        by induction hypothesis on d2, e2
      _: Gamma |- t1 t2' : T
        by rule T-App on d1, d2'
    end case
    case rule
      v2: t2 value
      ------------------------ E-AppAbs
      _: (lam x:T2' dot t11[x]) t2 ->
           t11[t2]
      where t1 := lam x:T2' dot t11[x]
      and t' := t11[t2]
    is
      d11: Gamma, x:T2' |- t11[x] : T
        by inversion of rule T-Abs on d1
          where T2 := T2'
      _: Gamma |- t11[t2] : T
        by substitution on d11, d2
    end case
  end case analysis
end case
...
```

Figure 2.8: Where clauses added to the SASyLF proof (Part 1)

```
...
case rule
  ----------------------- T-True
  _: Gamma |- true : Bool
 where t := true
 and T := Bool
is
  proof by contradiction on e
end case
case rule
  ----------------------- T-False
  _: Gamma |- false : Bool
 where t := false
 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
    case rule
      -------------------------- E-IfFalse
      _: if false then t2 else t' -> t'
     where t1 := false
     and t3 := t'
    is
      proof by d3
    end case
    case rule
      e1: t1 -> t1'
      ------------------------------ E-If
      _: if t1 then t2 else t3 ->
           if t1' then t2 else t3
     where t' := if t1' then t2 else t3
    is
      d1': Gamma |- t1' : Bool
        by induction hypothesis on d1, e1
      _: Gamma |-
           if t1' then t2 else t3 : T
        by rule T-If on d1', d2, d3
    end case
  end case analysis
end case
end case analysis
end theorem
```

Figure 2.9: Where clauses added to the SASyLF proof (Part 2)

3 Definitions of Correctness

The examples in the previous chapter 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.

3.1 LF Representation

In LF terms, the object language description consists of term constructors c and type constructors a, both LF constants. The `syntax` declaration (from Figure 2.2)

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

corresponds to the LF declarations[1]

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

There is no constructor for variables `x`; SASyLF simply associates the name prefix `x` with the syntax type $a_{\texttt{t}}$. (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

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

[1] The arrow $\to$ is used here, and in the example type constructor $a_{\texttt{eval}}$, 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.

A SASyLF inference rule R is stored as

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

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 all three typing rules with premises in Figure 2.3; in each rule, both the premise(s) and the conclusion have constructor $a_{\texttt{typing}}$. The evaluation rules for $\lambda_{\rightarrow\mathbf{B}}$ in Figure 2.4 which contain premises are similar.

3.2 Case Analyses

As mentioned in §2.2, a SASyLF theorem, together with its proof, is internally represented as a function. A theorem has inputs $x_1 : \tau_1$, $x_2 : \tau_2$, ..., $x_n : \tau_n$ 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[2] as τ; if different, the proof will continue after the case analysis is finished.

When performed on a derivation[3] $d : a_J\ \{\overline{\eta}\}_d$, 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. Hereafter this 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 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[4] 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

$$\sigma_d(a_J\ \{\overline{\eta}\}_f) = \sigma_d(\sigma(a_J\ \{\overline{\eta}\}_d)) \tag{3.1}$$

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

[2] 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 proof in Figures 2.6 and 2.7.

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

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

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.

A rule case analysis is complete if all such rules are addressed[5], and the proof function produces a derivation with type τ' within each case. These observations answer questions (1) and (2) from the beginning of this section.

Suppose a unifier σ_d exists for fresh version R_f of inference rule R, satisfying equation (3.1). Given a user-written rule case R' addressing rule R, to be sure that R' is sound, σ_d must not map any free variables of the conclusion of R', hereafter referred to as the *rule case conclusion* (RCC). The substitution σ_d can be altered to comply, if it does not already, in a process described in §4.1. 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.

In more formal terms, for R' to be correct, there must exist a "bijection" unifier

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

such that

$$R' = \sigma_c(\sigma_d(R_f)) \tag{3.2}$$

where every u_i is a free variable in $\sigma_d(R_f)$ and w_i is the corresponding free variable in R'.

The free variables w_i in the codomain of σ_c must not share names with other members of the local context Γ, for 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.

[5]It is possible that no rule conclusions unify with $a_J\ \{\overline{\eta}\}_d$. When this happens, the complete case analysis has no cases. This is what occurred, for example, in the rule case T-TRUE in Figure 2.7. In SASyLF, `proof by contradiction on e` is syntactic sugar for an empty case analysis on `e`.

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.

Suppose rule case R' is written correctly to address inference rule R, and so a bijection unifier σ_c exists. By equation (3.2), considering only the conclusions of R' and $\sigma_d(R_f)$, the RCC can be described as

$$\text{RCC} = \sigma_c(\sigma_d(a_J\ \{\overline{\eta}\}_f)) \tag{3.3}$$

where $a_J\ \{\overline{\eta}\}_f$ is the conclusion of R_f. Equations (3.1) and (3.3) then combine to form

$$\text{RCC} = \sigma_c(\sigma_d(\sigma(a_J\ \{\overline{\eta}\}_d))) = (\sigma_c \circ \sigma_d)(\sigma(a_J\ \{\overline{\eta}\}_d)) \tag{3.4}$$

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

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

where each v is a free variable of the CAS, is a one-way unifier from the CAS to the RCC. 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 $\sigma_u\ \Gamma$. In other words, inside the inner case, the substitution $\sigma'_u \circ \sigma_u$ is applied to all elements of Γ. In general, cases in nested case analyses represent a succession of composed unifiers

$$\sigma \triangleq \sigma_u^k \circ \cdots \circ \sigma_u^2 \circ \sigma_u^1 \tag{3.5}$$

applied to each member of Γ, where k case analyses are nested, and σ_u^1 represents the substitution for the case at outermost scope.

Therefore, σ as it appeared in equations (3.1) and (3.4) 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 2.7 requires cases only for rules E-IfTrue, E-IfFalse, and E-If, and not for all of the evaluation rules of $\lambda_{\rightarrow\mathbf{B}}$.

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,

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

To show the meaning of requirement (3b), consider the RCC for the inner rule case T-If in Figure 2.7

```
_: 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.

3.3 Where Clauses

The notion of "where" clauses benefits from this more formal description. Specifically, these clauses should 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.

3.3.1 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 (3.5), 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 one implemented in the new version of SASyLF. For example, the nested case E-IfTrue in Figure 2.9 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 that clauses represent only the newest mappings:

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

3.3.2 SASyLF Syntax

Chief among remaining considerations is how correct "where" clauses for rule cases should fit into SASyLF's abstract syntax, including the form of the clauses themselves. In Figures 2.8 and 2.9, they immediately follow an RCC and precede `is`; this syntax is generalized[6] in Figure 3.1. This is the ideal location for the clauses in a rule case, because they describe substitutions which occur as a result of the RCC; in particular, the right-hand sides of the clauses should 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.

3.3.3 Familiarity

Another consideration for "where" clauses for rule cases is that they should only ever list variables and expressions which have already been seen in the proof text. They should 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 should correspond to the "unparsed" (concrete syntax) versions of LF expressions v and η_v, respectively.

3.3.4 First- vs. Second-Order Left-Hand Sides

For first-order "where" clauses, the left-hand side should simply be the concrete name represented by v. SASyLF includes support for second-order[7] (and no higher) free variables,

[6] The syntax shown in Figures 3.1, 3.3, and 3.4 is adapted and (greatly) simplified from SASyLF's parsing specification.

[7] Recall that SASyLF stands for **S**econd-order **A**bstract **Sy**ntax **L**ogical **F**ramework.

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

Figure 3.1: The abstract syntax of a SASyLF rule case analysis, including the addition of "where" clauses

and "where" clauses describing substitutions on them are slightly more verbose. Figure 3.2, showing the beginning of a familiar lemma[8], 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 §2.1.1). Internally, this `[]` notation is represented with an LF application with v at the head. The left-hand side of v's "where" clause should 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[9]. Then

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

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

[8] Recall that the so-called "substitution lemma" [8] is not actually required to prove complete type preservation for $\lambda_{\rightarrow \mathbf{B}}$ in SASyLF (see Figure 2.6); the `by substitution` construct may be used instead.

[9] This could occur in the substitution lemma, in the case for rule `T-Abs` (not shown).

```
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 3.2: The beginning of a lemma with second-order free variables

```
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. Neither is

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

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

3.3.5 Optional Presence

A final consideration regarding "where" clause correctness is that their presence in the code should 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.

3.3.6 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:

```
(<ID> ":" <EXPR> | <PROOF>) <BY>
  <INVERSION> <OF> [<RULE>] <RULENAME> <ON> <ID>
    (<WHERE> <LHS> ":=" <RHS>
      (<AND> <LHS> ":=" <RHS>)*)?
```

Figure 3.3: The abstract syntax of a SASyLF `by inversion` derivation, including the addition of "where" clauses

```
<USE> <INVERSION> <OF> [<RULE>] <RULENAME> <ON> <ID>
  (<WHERE> <LHS> ":=" <RHS>
    (<AND> <LHS> ":=" <RHS>)*)?
```

Figure 3.4: The abstract syntax of a SASyLF `use inversion` construct, including the addition of "where" clauses

(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 $(v \mapsto \eta_v) \in \sigma_u$ should 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 should always yield errors, but mappings in σ_u which lack clauses should only yield errors if an option making the clauses mandatory is enabled.

3.3.7 Where Clauses for Inversions

Inversions are another SASyLF construct which can impose substitutions on members of the local context; thus, they require "where" clauses. Figure 3.3 describes the syntax for justification `by inversion`, with the clauses added. This construct represents a rule case analysis with a single applicable case, which is performed in-line; that is, there is no nesting. This means that the alterations to the local context via composition with σ occur immediately, and are in effect until the end of the given enclosing case of a proof. This behavior is even more reminiscent of a "let" construct than rule cases.

Figure 3.4 describes the syntax for the `use inversion` construct. This construct is similar to `by inversion`, except that no particular derivation is being justified. The semantic effect of `use inversion` is limited to changes in σ.

A key difference between inversions and typical rule cases is that inversions do not list an RCC. (A derivation justified with `by inversion` corresponds to one of the premises of the original rule. Multiple such derivations may be linked with `and`.) This makes "where" clauses even more important for them.

A complication which arises for inversions due to a lack of an RCC is that the user may write correct clauses which contain new information; in other words, familiarity (from §3.3.3) does not apply to inversions.

The requirements (detailed in §3.3.6) for how "where" clauses appear in the code, and that they represent mappings in $\sigma_u \circ \sigma$, do not change for inversions. What does change is that the user is allowed to affect σ_u with "where" clauses in these constructs, as opposed to rule cases, where the clauses are strictly passive witnesses to substitutions already computed and assumed.

4 Implementation in SASyLF

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[1] and case analysis verification.

4.1 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 §3.2. 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 §3.2. 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

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

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 $\varnothing$, 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.

4.2 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 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.

5 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.

5.1 Current Limitations

One major limitation to the current "where" clause implementation is that SASyLF does not verify "where" clauses when changes occur in the hypothetical context from a CAS to the RCC of a rule case. 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.

Another limitation involves the way the new version of SASyLF verifies "where" clauses for inversions. Currently, the user is not allowed to supply their own terms for the right-hand of clauses for inversions. That is, the user cannot write mappings which affect the local substitution; SASyLF can only verify "where" clauses for inversions for which both the left- and right-hand sides have already appeared in the code.

This has two limiting implications on the new feature for inversions. First, the mappings which are allowed is dictated solely by the particular σ_u which is produced through unification; alternative unification solutions are not allowed. Second, if the right-hand side of a "where" clause internally includes generated variables (originating from "fresh" copies of inference rules), the user currently cannot write the correct clause.

5.2 Possible Extension

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].

Bibliography

[1] Aldrich, J., Simmons, R. J., and Shin, K. SASyLF: An educational proof assistant for language theory. In *Proceedings of the 2008 international workshop on Functional and declarative programming in education*, pp. 31–40. ACM, 2008.

[2] Ariotti, M. and Boyland, J. T. Making substitutions explicit in SASyLF, 2017. Submitted to LFMTP 2017 (Accepted).

[3] Harper, R., Honsell, F., and Plotkin, G. A framework for defining logics. *Journal of the ACM (JACM)*, 40(1):143–184, 1993.

[4] Nipkow, T. Functional unification of higher-order patterns. In *Proceedings of Sixth International Workshop on Unification Schloss Dagstuhl, Germany*, p. 77, 1992.

[5] Pfenning, F. Computation and deduction, 2001.

[6] Pfenning, F. and Elliott, C. Higher-order abstract syntax. In *ACM SIGPLAN Notices*, volume 23, pp. 199–208. ACM, 1988.

[7] Pfenning, F. and Schürmann, C. System description: Twelf-a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction*, pp. 202–206. Springer-Verlag, 1999.

[8] Pierce, B. C. *Types and programming languages*. MIT press, Cambridge, Massachusetts, USA and London, England, 2002.

[9] Schürmann, C. *Automating the Meta Theory of Deductive Systems*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, 2000.