



FORMALIZATION OF SASyLF

by

Sneha Elizabeth Popley

Submitted in partial fulfillment of the  
requirements for Departmental Honors in  
the Department of Computer Science  
Texas Christian University  
Fort Worth, Texas

May 3, 2010

FORMALIZATION OF SASyLF

Project Approved:

---

(Supervising Professor)

---

---

---

## TABLE OF CONTENTS

ACKNOWLEDGEMENTS .....	vi
INTRODUCTION .....	1
Contributions .....	2
OVERVIEW .....	3
CORE CALCULUS .....	4
Syntax & Judgments .....	5
Theorems and Proofs .....	8
Typing Rules .....	13
Case .....	13
Let-normal Form .....	14
TRANSLATION FROM CORE CALCULUS TO $M_2^+$ .....	14
Type Translation .....	14
Term Translation .....	15
FUTURE WORK .....	16
APPENDIX A .....	17
APPENDIX B .....	22
BIBLIOGRAPHY .....	23
ABSTRACT .....	24

## ACKNOWLEDGEMENTS

I want to start off by thanking Dr. Stephanie Weirich for introducing programming language theory to my sophomore mind. The summer after my sophomore year was eye-opening to say the least. I was selected as a participant of the Computing Research Association's (CRA) Distributed Research Experience for Undergraduates: a paid research internship in a graduate-school environment. I had the opportunity to work with Stephanie on functional programming and dependent types. It was my first time researching programming language theory, and I loved it. She has been my mentor since, even though she has been on a different continent for the past year.

Also, thank you to Dr. Jonathan Aldrich for taking an email from a random undergraduate from TCU seriously. I had decided that the summer after my junior year needed to give me more research experience in the field, and I wanted to explore research opportunities in other schools. After I had spent a few days of looking through different professors' websites, Jonathan's research in SASyLF and PLAID caught my attention. I had never met him before, and he wasn't even looking to hire interns for the summer, but I still asked him if he had an REU position for the summer. After a month's correspondence, I got the job! So, thank you again Jonathan, for the opportunity to immerse myself in the graduate-school environment for the summer.

A big thanks to Robert Simmons for working with me last summer. Jonathan tried to introduce us in my first week, but we could never get a hold of him. In the sixth week, I finally introduced myself to Rob, and some great discussions came about. I have fond

memories of meeting in the graduate lounge in Wean Hall, and, in an hour, covering the whiteboards around the room with our typing rules and derivations. I appreciate your role as an amazing and reliable mentor over the summer.

Finally, thanks to the PLAID and LF groups at CMU. I learned a lot discussing the project and other ideas with you. Presenting the project at the end of the summer to a supportive and knowledgeable crowd was an invaluable experience.

Continuing this project at TCU was hard for me; among other reasons, it was the first time I was working with someone who did not have an office down the hall. I also had five classes and was applying to graduate schools at the same time. I couldn't have reached this stage without help from my committee members: Dr. Rhonda Hatcher, Dr. Dick Rinewalt, and Dr. Antonio Sanchez. Thanks to Loren Spice for constantly giving me advice and putting up with my semi-breakdowns. Winning the Boller Award (given to best TCU Honors Presentation) would not have been possible without all their help.

Last but not least, I have to thank the TCU Honors Program for giving me an invaluable academic and emotional support system for the past four years. Dr. Peggy Watson and Dr. Ron Pitcock have encouraged me endlessly throughout my undergraduate career, and I don't know where I would be without them.

## INTRODUCTION

Formal programming language theory includes the study of the mathematical representation of the meaning of programming languages. It usually has a steep learning curve attached to it, mainly because it can be hard and tedious to describe the semantics of programming languages precisely. Immediate feedback and convenient, interactive tools can ease the process of learning as well as teaching programming languages.

Isabelle/HOL (Nipkow, Pauson, & Wenzel, 2002), Coq (Bertot & Castéran, 2004), and Twelf (Schürmann, 1998) are some proof assistants that support interactive theorem-proving (the process of developing proofs about programming language concepts interactively with a computer). However, they have a challenging learning curve due in part to nontrivial encoding techniques. SASyLF ("Sassy-elf") has paper-like syntax, explicit proofs, intuitive error messages, and the ability to reason about any system with variable binding (Aldrich, Simmons, & Shin, 2008). It has been used in introductory courses in programming language theory with promising initial results. SASyLF is based on LF, the logical framework that allows formalization of languages in higher order abstract syntax. Higher order abstract syntax is a way to represent programs, rules, and other objects in syntax in formal systems (Pfenning & Elliot, 1988). This higher order abstract syntax is a technique to represent the object language's binding constructs with the meta-language's binding constructs. However, for usability, SASyLF puts an added restriction of second order abstract syntax on LF. These and other simplifications allow it to provide clean, explicit syntax that resembles handwritten proofs.

The term “formal” implies “admits logical reasoning” in programming language theory (Norrish, 1998). A formal definition of a programming language gives a precise meaning to programs. This precise meaning is essential to successfully reason about a language and convince others about the soundness of its design. Standard ML and C are examples of some programming languages that have been formalized using previously mentioned tools such as Twelf (Lee, Crary, & Harper, 2007) and HOL (Norrish, 1998). In this paper, we formalize SASyLF in the logical framework  $M_2^+$  by showing SASyLF’s syntax in terms of  $M_2^+$ . We present a transition core calculus semantically similar to SASyLF. This calculus represents the meaning of programs written in SASyLF, while having syntax that is similar to the target language,  $M_2^+$ .  $M_2^+$  is a sound logic to reason about deductive systems while providing higher-order representation techniques and inductive definitions (Schürmann, 1998). The type theory that corresponds to the recursive function space in LF that encodes properties about deductive systems is  $M_2^+$ . Since SASyLF is LF-based, we decided to use  $M_2^+$  as our formalization language.

### Contributions

The contributions of this paper include:

- 1) A mathematical specification of the semantics of SASyLF in  $M_2^+$ .
- 2) Further analysis of SASyLF proofs, especially its heavy use of let-binding with respect to LF.
- 3) A mathematical foundation for further development of SASyLF, for example to incorporate conjunctions, etc.

## OVERVIEW

The core calculus of SASyLF serves as an intermediate in the semantic specification of SASyLF in  $M_2^+$ . The core calculus has syntax similar to  $M_2^+$  and semantics similar to SASyLF. The verification of type safety of the core calculus involved challenges arising from formalization of our language in LF. We needed a core calculus to overcome major semantic differences between SASyLF and  $M_2^+$ , and this core calculus was formulated to ensure that it was a convenient transition between the two. Like SASyLF, the core calculus uses second order syntax and judgments as types. SASyLF has the concept of variable binding built-in, to ensure that students can learn without having to implement it repeatedly (Aldrich, Simmons, & Shin, 2008). Since the core calculus resembles SASyLF semantically, it has embedded variable binding as well. This ensures we get binding for free, as expected in LF.

In LF, a context is defined as a list of assumptions (Schürmann, 1998). These can include LF expressions and variables are utilized in theorems and proofs. In SASyLF, each judgment in a theorem can have an attached context, as long as this doesn't produce any conflicting contexts. This differs from the handling of contexts in  $M_2^+$ , in which a theorem, not just its individual judgments, has an attached context. This difference was challenging to represent in  $M_2^+$ , and also brought up questions about the complex handling of contexts in SASyLF.

## CORE CALCULUS

terminals fn unit

$e ::= x$   
 | "(" ")"  
 | e e  
 | fn  $x : \tau \Rightarrow e[x]$

$\tau ::= \text{unit}$   
 |  $\tau \rightarrow \tau$

$\Gamma ::= *$   
 |  $\Gamma, x \text{ ok}$

judgment ok:  $\Gamma \vdash e \text{ ok}$   
 assumes  $\Gamma$

----- ok-unit  
 $\Gamma \vdash "(" ")" \text{ ok}$

----- ok-var  
 $\Gamma, x \text{ ok} \vdash x \text{ ok}$

$\Gamma \vdash e_1 \text{ ok}$   
 $\Gamma \vdash e_2 \text{ ok}$   
 ----- ok-app  
 $\Gamma \vdash e_1 e_2 \text{ ok}$

$\Gamma, x_1 \text{ ok} \vdash e[x_1] \text{ ok}$   
 ----- ok-fn  
 $\Gamma \vdash \text{fn } x : \tau \Rightarrow e[x] \text{ ok}$

The core calculus, as mentioned earlier, was designed to be syntactically similar to  $M_2^+$  but semantically similar to SASyLF. Its main role is to serve as a transition between SASyLF and  $M_2^+$  in the formalization through the representation of SASyLF semantics in  $M_2^+$ . It relies on the  $M_2^+$  algorithm for completeness (coverage and termination).

**Figure 1: Syntax & Judgments in SASyLF**

## Syntax & Judgments

SASyLF has syntax that resembles handwritten proofs. The syntax block declares a grammar in conventional BNF, followed by judgments. Figure 1 demonstrates this paper-like syntax along with some typing rules. This example shows the declaration of the simply typed lambda calculus and its associated typing rules. It starts off with the declaration of key words that will be used as terminals in the lambda calculus:  $\lambda$  and  $\text{unit}$ . This is followed by a definition of the syntax of the simply typed lambda calculus.

Expressions are denoted by  $e$  and types are denoted by  $\tau$ . The notation  $e\{x\}$  is used to indicate that the variable  $x$  is bound in expression  $e$ . We quote parentheses to denote their appearance in the target language, since parentheses are also used to parse expressions in SASyLF.

In figure 1, judgments follow the syntactic definitions of the simply typed lambda calculus in a SASyLF program. We name the judgment  $\text{ok}$  and describe its expected form  $\Gamma \vdash e \text{ ok}$ . Each rule is a series of premises (on new lines), a horizontal line and the name of the rule, and the conclusion. So, for instance, in the rule called  $\text{ok-app}$ , the premises are  $\Gamma \vdash e_1 \text{ ok}$  and  $\Gamma \vdash e_2 \text{ ok}$  and the conclusion is  $\Gamma \vdash e_1 e_2 \text{ ok}$ .  $e_1$  and  $e_2$  represent any expression, since rules are interpreted schematically (i.e. according to the specified judgment form).

In this case,  $\Gamma$  is the context which must represent a list of judgments and variables. The `assumes` keyword indicates the presence of a context, and SASyLF also requires the presence of recursive cases that bind one variable while using  $\Gamma$  in only one location (Aldrich, Simmons, & Shin, 2008). In our definition, this context can usually include a case such as  $\Gamma, x \text{ ok}$ . Additionally, each recursive case must have a rule similar to `t-var` (see Appendix B). The rule in this case is `ok-var`. With the `ok-var` rule, we can interpret  $x \text{ ok}$  in  $\Gamma$  to mean that any expression  $e$  that is bound to  $x$  must have a proof of the expression  $\Gamma \vdash e \text{ ok}$ . This justifies that `ok` judgment is preserved, even when substituting  $e$  for  $x$ .

A signature in the core calculus enumerates the possibilities of type constants, constructors, judgments, and judgment terms. Figure 2 shows the constructors for the lambda calculus in the core calculus and  $M_2^+$ . As in SASyLF, it starts off with the definition of the syntax of the simply typed lambda calculus: `tau` denotes types and `exp` denotes expressions. We also define `empty` which was the quoted set of parenthesis in SASyLF. `@` represents application in simply typed lambda calculus, and `lam` represents the variable bindings in the lambda calculus.

$$\begin{aligned}
\Sigma &= \text{tau} : \text{type}, \\
&\text{unit} : \text{tau}, \\
&\text{arrow} : \text{tau} \rightarrow \text{tau} \rightarrow \text{tau}, \\
&\text{exp} : \text{type}, \\
&\text{empty} : \text{exp}, \\
&\text{@} : \text{exp} \rightarrow \text{exp} \rightarrow \text{exp}, \\
&\text{lam} : \text{tau} \rightarrow (\text{exp} \rightarrow \text{exp}) \rightarrow \text{exp}, \\
& \\
&\text{ok} : \text{exp} \rightarrow \text{type}, \\
&\text{ok-empty} : \text{ok emp}, \\
&\text{ok-fn} : \Pi T : \text{tau}. \Pi e_1 : \text{exp}. \Pi (x : \text{exp} \text{ dx} : \text{ok } x)^L d1 : \text{ok } e_1 x. \\
&\quad \text{ok lam}([x : \text{exp}]e_1 x), \\
&\text{ok-app} : \Pi e_1 : \text{exp}. \Pi e_2 : \text{exp}. \Pi d1 : \text{ok } e_1. \Pi d2 : \text{ok } e_2. \text{ok } \text{@} e_1 e_2.
\end{aligned}$$

**Figure 2: Syntax and Judgments in the Core Calculus**

These definitions are followed by judgments similar to `ok` in the SASyLF example described initially. We define the form of `ok` and describe three possible rules with the lambda calculus: an empty expression (`ok-emp`), an expression with bound variables (`ok-fn`), and an expression that involves an application (`ok-app`). In the case of these typing rules, we have to explicitly declare the variables and their types before we use them to form premises or conclusions. In `ok-app`, we require the type-soundness of expressions  $e_1$  and  $e_2$  to state the conclusion `ok @ e1 e2`.

We demonstrate SASyLF's second order nature in the core calculus by restricting types to a second-order nature; for example,  $A_0$  represents a base-level type,  $A_1$  represents a

first-order type, and  $A_2$  represents a second-order type (see figure 3). We also split construction types ( $A_0$  to  $A_2$ ) from judgment types ( $DA_0$ ,  $DA_1$ , and  $DA_2$ ). They could be combined to represent more complex types, but we decided to separate the types of syntactic declarations (construction types) from the types of judgments. Due to SASyLF's second order syntax, neither syntax nor judgments can include judgments as parts.

<i>Signature</i>	$\Sigma ::= \Sigma, s:type \mid \Sigma, c:A_2 \mid \Sigma, j:K \mid \Sigma, r:DA_2$
<i>Type constants</i>	$s ::= \dots$
<i>Types of order 0</i>	$A_0 ::= s$
<i>Types of order 1</i>	$A_1 ::= s \mid A_0 \rightarrow A_1$
<i>Types of order 2</i>	$A_2 ::= s \mid A_1 \rightarrow A_2$
<i>Terms</i>	$M ::= c \mid M M \mid \lambda x:A_0.M$
<i>Kinds / Judgment Forms</i>	$K ::= \text{type} \mid A_0 \rightarrow K$
<i>Judgment constants</i>	$j ::= \dots$
<i>Atomic judgments</i>	$J ::= j \mid J M$
<i>Judgments of order 0</i>	$DA_0 ::= J$
<i>Judgments of order 1</i>	$DA_1 ::= J \mid DA_0 \rightarrow DA_1 \mid \Pi x:A_0.DA_1$
<i>Judgments of order 2</i>	$DA_2 ::= J \mid DA_1 \rightarrow DA_2 \mid \Pi x:A_1.DA_2$
<i>Derivations</i>	$DM ::= r \mid x \mid DM DM \mid DM M \mid \lambda x:A_0.DM \mid \lambda x:DA_0.DM$

**Figure 3: Syntax of the Core Calculus**

### Theorems and Proofs

SASyLF supports theorems of the form "for all «list of meta-variables and judgments» there exists «judgment»." While theorems of this form are limited, they are adequate for a lot of programming theory, corresponding to proofs that are naturally expressed by induction and case analysis on derivations.  $G$ , which stands for Generalized Formulas, represents the form of a theorem in the core calculus (see figure 4). In the core calculus, we state the contexts attached to each judgment as a single context attached to the theorem.

<i>Block contexts</i>	$C ::= \cdot \mid C, x:A_1$
<i>Block variables</i>	$\rho ::= \{x:A_0, d_x:DA_0\}$
<i>Block list</i>	$S ::= \cdot \mid S, B$
<i>Block declaration</i>	$B ::= \text{SOME } C \text{ BLOCK } \rho$
<i>Generalized Formula</i>	$G ::= \Box S.F^-$
<i>Formulas</i>	$F^- ::= \forall x:A_1.F^- \mid DA_1 \rightarrow F^- \mid F^+$
<i>Formulas</i>	$F^+ ::= \exists x:A_1.F^+ \mid DA_1$

**Figure 4: Syntax for Contexts and Formulas in the Core Calculus**

Figure 5 shows a theorem and its proof in SASyLF. The theorem statement (called `ok-thm`) states that for a given context  $\Gamma$ , if  $e$  `ok` (per the stated judgments), then  $e$  `ok`.

**theorem ok-thm : forall dt : Gamma |- e ok exists Gamma |- e ok.**

dv : Gamma |- e ok by induction on dt:

**case rule**

----- ok-var

dxt: Gamma', x ok |- x ok

is

dxt' : Gamma', x ok |- x ok by rule ok-var

**end case**

**case rule**

de1t : Gamma' |- e1 ok

de2t: Gamma' |- e2 ok

----- ok-app

det : Gamma' |- e1 e2 ok

is

de1t' : Gamma' |- e1 ok by induction hypothesis on de1t

de2t' : Gamma' |- e2 ok by induction hypothesis on de2t

det' : Gamma |- e1 e2 ok by rule ok-app on de1t', de2t'

**end case**

end induction

**end theorem**

**Figure 5: Theorem and its Proof in SASyLF**

This theorem's proof serves the purpose of introducing induction and case analysis in SASyLF. In SASyLF, a proof is a list of justified judgments. For instance, performing induction on  $dt$  gives us the four cases that correspond to `ok-unit`, `ok-var`, `ok-app`, and `ok-fn`. (only `ok-var` and `ok-app` are shown in figure 5) Each rule is stated using fresh meta-variables that are bound in the case. In the case related to `ok-app`, we assume  $de1t$  and  $de2t$ , as per the declaration of the judgment. We can then apply the induction

hypothesis to  $de1t$  and  $de2t$  to obtain  $de1t'$  and  $de2t'$  respectively.  $ok\text{-app}$  applied to  $de1t'$  and  $de2t'$  gives us  $det'$ , which matches our goal. SASyLF checks if the stated derivations are actually obtained through the specified application and induction hypotheses.

Thm :  $\square$  SOME. BLOCK  $x : \text{exp}, dx : ok\ x. \forall e_1 : \text{exp}. \forall d_1 : ok\ e_1. \exists d_2 : ok\ e_1$

Proof :

box  $S . \mu \text{allok } \epsilon F. \Lambda e_1 : \text{exp}.$

case( $d_1$ ) of

(  $e'_1 : \text{exp}, e'_2 : \text{exp}, d'_1 : ok\ e'_1, d'_2 : ok\ e'_2$   
 $\triangleright (ok\text{-app } d'_1\ d'_2 / d_1)$   
 $\rightarrow \text{let } \langle d1 \rangle = \text{allok } e'_1 \quad \text{in}$   
 $\quad \text{let } \langle d2 \rangle = \text{allok } e'_2 \quad \text{in}$   
 $\quad \langle ok\text{-app } d_1\ d_2 \rangle$ )

(  $x : \text{exp}, dx : x\ ok$   
 $\triangleright (dx / d_1)$   
 $\langle dx \rangle$ )

**Figure 6: Theorem and its Proof in the Core Calculus**

We can see the same semantics in the core calculus (figure 6). In the case of the core calculus, we do not have to initially name the theorem. However, to support inductive function calls, we call it  $allok$  at the beginning of the proof. We also take  $e_1$  as an argument. We then perform case analysis on  $d_1$  (the cases for  $ok\text{-emp}$  and  $ok\text{-fn}$  are in Appendix B). In the case of  $ok\text{-app}$ , we have to explicitly declare fresh variables  $e'_1, e'_2, d'_1,$  and  $d'_2$  that exist until the end of the case. We then go on to let-bind the values of  $d_1'$

and  $d_2'$  obtained through the inductive hypothesis, and, finally, apply `ok-app` (as declared earlier) to the let-bound variables.

While one has to explicitly mention substitutions for each case in a  $M_2^+$  proof, SASyLF does not require such explicit descriptions and uses pattern matching and let-bindings to determine the substitutions in each case. The proofs in SASyLF (and hence in the core calculus) are in a let-normal form, which allow SASyLF to give good error messages. This differs from  $M_2^+$ , which has let-binding, but does not implement the let-normal form as often. Additionally, even though the contexts in case terms are described using substitutions of a similar syntax to  $M_2^+$ , there is a difference in case analysis in SASyLF and  $M_2^+$ ; SASyLF supports nested, one-level-at-a-time case analysis. On the other hand,  $M_2^+$  supports case analysis many levels deep, but only at the top level of a logic definition. Essentially, proofs in SASyLF are represented differently: as functions in let-normal form that accept input derivations, perform case analysis as necessary, and produce output derivations, possibly by making recursive calls (through the induction hypothesis, as seen in figure 6). The core calculus representation is in let-normal form and resembles  $M_2^+$ . The checker for SASyLF follows  $M_2^+$  closely, benefitting from existing proof soundness for  $M_2^+$ .

### Typing Rules

The following rules define a typing relationship between terms and types.

$$\frac{(\mathbf{x} \in F^-) \in \Delta}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S}^- \mathbf{x} : F^-}$$

$$\frac{\Psi, x:A_1; \Delta \vdash_{\Sigma, \Gamma, S} P:F^-}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S}^- \Lambda x:A_1.P : \forall x:A_1.F^-}$$

$$\frac{\Psi, x:DA_1; \Delta \vdash_{\Sigma, \Gamma, S} P:F^-}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S}^- \Lambda x:DA_1.P : DA_1 \rightarrow F^-}$$

**Figure 7: Typing Rules in the Core Calculus**

The above typing rules are some of the basic rules in the core calculus (see appendices for a comprehensive list). They are similar to the types rules of  $M_2^+$ , with the following exceptions:

#### Case

In  $M_2^+$ , for each case, we need to explicitly define the substitutions for each available variable in the context. However, in the core calculus, which represents the semantics of SASyLF, we don't need to explicitly state every single substitution.

$$\frac{\Psi; \Delta \vdash_{\Sigma, \Gamma, S}^- \mathbf{x}:F_1^- \quad \Psi; \Delta \vdash_{\Sigma, \Gamma, S}^- \Omega:F_2^-}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S}^- \text{case } \mathbf{x} \text{ of } \Omega:F_2^-}$$

**Figure 8: Typing Rules for case**

For example induction on the church numeral  $n$  leads to two cases: case 0 and case  $s\ n'$ .

We need not take into account any other variables; SASyLF uses pattern matching and let-normal binding to complete the case.

Let-normal Form

$$\frac{\Psi; \Delta \vdash_{\Sigma, \Gamma, S} P : F_1^- \quad \Psi; \Delta, \mathbf{y} \in F_1^- \vdash_{\Sigma, \Gamma, S} P' : F_2^-}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S} \text{let } \mathbf{y} \in F_1^- = P \text{ in } P' : F_2^-}$$

$$\frac{\Psi; \Delta \vdash_{\Sigma, \Gamma, S} P : F_1^- \quad \Psi, x : A_1; \Delta, \mathbf{y} \in F_1^- \vdash_{\Sigma, \Gamma, S} P' : F_2^-}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S} \text{let} \langle x : A_1, \mathbf{y} \in F_1^- \rangle = P \text{ in } P' : F_2^-}$$

$$\frac{\Psi; \Delta \vdash_{\Sigma, \Gamma, S} P : F_1^- \quad \Psi, x : DA_1; \Delta \vdash_{\Sigma, \Gamma, S} P' : F_2^-}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S} \text{let} \langle x : DA_1 \rangle = P \text{ in } P' : F_2^-}$$

**Figure 9: Typing Rules for let**

Function calls in SASyLF, and hence the core calculus, have let-normal form. They define an inner context in which the results of the function call are available using a let-clause.

TRANSLATION FROM CORE CALCULUS TO  $M_2^+$

Type Translation

The type translation rules show the equivalence between types in the core calculus and the corresponding types in  $M_2^+$ . These translation rules show the relationship between possible types in the core calculus and  $M_2^+$ . Below are rules that show the correspondence between rules that involve the forall quantifier ( $\forall$ ).

$$\frac{\Psi, x:A_1 \vdash_{\Sigma, S} F_S^- \Rightarrow F_M}{\Psi \vdash_{\Sigma, S}^- \forall x:A_1 F_S^- \Rightarrow \forall x:A. F_M}$$

$$\frac{\Psi, x:DA_1 \vdash_{\Sigma, S} F_S^- \Rightarrow F_M}{\Psi \vdash_{\Sigma, S}^- DA_1 \rightarrow F_S^- \Rightarrow \forall x:A. F_M}$$

$$\frac{\Psi \vdash_{\Sigma, S}^+ F_S^+ \Rightarrow F_M}{\Psi \vdash_{\Sigma, S}^- F_S^+ \Rightarrow F_M}$$

Figure 10: Type Translation Rules

### Term Translation

Term translation rules show the relationship between theorems and their proofs in the core calculus and their counterparts in  $M_2^+$ . In the case below, the rule says that given a variable  $x$  of type  $A_1$  in context and corresponding proof terms in the core calculus and  $M_2^+$ , we can conclude that for all variables  $x$  of type  $A_1$ , the same relationship exists.

$$\frac{\Psi, \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^+ P_S:F_S^+ \Rightarrow P_M:F_M}{\Psi, \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^- P_S:F_S^+ \Rightarrow P_M:F_M}$$

$$\frac{\Psi, x:A_1; \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^- P_S:F_S^- \Rightarrow P_M:F_M}{\Psi; \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^- \Lambda x:A_1. P_S:\forall x:A_1. F_S^- \Rightarrow \Lambda x:A. P_M:\forall x:A. F_M}$$

Figure 11: Term Translation Rules

While the rules above are more basic rules that show the relationship between fundamental logic terms, the correspondence between a case statement in the core calculus and one in  $M_2^+$  demonstrates the complexities between the two (see Appendix A).

### FUTURE WORK

This work describes SASyLF's semantics in  $M_2^+$ . The mathematical analysis of SASyLF gives us a basis for future development of SASyLF. The goal is to incorporate pairs, mutual induction, and automatic theorem proving into SASyLF. There is a simple form of automatic theorem proving (with keyword `auto`) in SASyLF, which works for basic theorem completions, but it needs to be extended for more complex cases. Added support for pairs and mutual induction will allow SASyLF to be a more versatile tool in an introductory class to programming language theory.

Our choice of  $M_2^+$  was based on the similarities between a SASyLF proof and an  $M_2^+$  proof. Beluga is another functional programming language that supports dependent types. Delphin also supports higher-order abstract syntax and dependent types while using LF for representation. These meta-logics could provide alternative methods to formalize SASyLF.

## APPENDIX A

### 1 Syntax

<i>Signature</i>	$\Sigma ::= \Sigma, s:\text{type} \mid \Sigma, c:A_2 \mid \Sigma, j:K \mid \Sigma, r:DA_2$
<i>Type constants</i>	$s ::= \dots$
<i>Types of order 0</i>	$A_0 ::= s$
<i>Types of order 1</i>	$A_1 ::= s \mid A_0 \rightarrow A_1$
<i>Types of order 2</i>	$A_2 ::= s \mid A_1 \rightarrow A_2$
<i>Terms</i>	$M ::= c \mid M M \mid \lambda x:A_0.M$
<i>Kinds / Judgement Forms</i>	
<i>Kinds</i>	$K ::= \text{type} \mid A_0 \rightarrow K$
<i>Judgment constants</i>	$j ::= \dots$
<i>Atomic judgments</i>	$J ::= j \mid J M$
<i>Judgments of order 0</i>	$DA_0 ::= J$
<i>Judgments of order 1</i>	$DA_1 ::= J \mid DA_0 \rightarrow DA_1 \mid \Pi x:A_0.DA_1$
<i>Judgments of order 2</i>	$DA_2 ::= J \mid DA_1 \rightarrow DA_2 \mid \Pi x:A_1.DA_2$
<i>Derivations</i>	$DM ::= r \mid x \mid DM DM \mid DM M \mid \lambda x:A_0.DM \mid \lambda x:DA_0.DM$
<i>Block contexts</i>	
<i>Block contexts</i>	$C ::= \cdot \mid C, x:A_1$
<i>Block variables</i>	$\rho ::= \{x:A_0, d_x:DA_0\}$
<i>Block list</i>	$S ::= \cdot \mid S, B$
<i>Block declaration</i>	$B ::= \text{SOME } C \text{ BLOCK } \rho$
<i>Generalized Formula</i>	
<i>Formulas</i>	$G ::= \square S.F^-$
<i>Formulas</i>	$F^- ::= \forall x:A_1.F^- \mid DA_1 \rightarrow F^- \mid F^+$
<i>Formulas</i>	$F^+ ::= \exists x:A_1.F^+ \mid DA_1$
<i>General Proof terms</i>	$Q ::= \text{box } S.P$
<i>Proof terms</i>	$P ::= \mathbf{x} \mid PM \mid PDM \mid \Lambda x:DA_1.P \mid \Lambda x:A_1.P \mid \langle M, P \rangle \mid \langle DM \rangle$ $\mid \mu \mathbf{x} \in F.P \mid \nu \rho.P \mid \text{let } \mathbf{x} \in F = P \text{ in } P'$ $\mid \text{let } \langle x:A_1, y \in F \rangle = P \text{ in } P' \mid \text{case } \mathbf{x} \text{ of } \Omega$ $\mid \text{let } \langle x:DA_1 \rangle = P \text{ in } P'$
<i>Case terms</i>	
<i>Case terms</i>	$\Omega ::= \cdot \mid \Omega, (\Psi \triangleright \psi \mapsto P)$
<i>Generalized contexts</i>	
<i>Generalized contexts</i>	$\Psi ::= \cdot \mid \Psi, x:A_1 \mid \Psi, \rho^L \mid \Psi, x:DA_1$
<i>Contexts</i>	
<i>Contexts</i>	$\psi ::= \cdot \mid \psi, M/x \mid \psi, DM/x \mid \psi, \rho'/\rho$
<i>Lemma Contexts</i>	
<i>Lemma Contexts</i>	$\Gamma ::= \cdot \mid \Gamma, \mathbf{x} \in G$
<i>Meta-assumptions</i>	
<i>Meta-assumptions</i>	$\Delta ::= \cdot \mid \Delta, \mathbf{x} \in F$

### 2 Typing rules for Core SASyLF

#### 2.1 Type well-formedness rules

$$\boxed{\vdash_{\Sigma} G \text{ wf}}$$

$$\frac{\vdash_{\Sigma, S} F^- \text{ wf}}{\vdash_{\Sigma} \square S.F^- \text{ wf}}$$

$$\boxed{\Psi \vdash_{\Sigma, S} F^- \text{ wf}^-}$$

$$\frac{\Psi, x:A_1 \vdash_{\Sigma, S} F^-}{\Psi \vdash_{\Sigma, S} \forall x:A_1.F^- \text{ wf}^-}$$

$$\frac{\Psi \vdash_{\Sigma} DA_1 \text{ type} \quad \Psi \vdash_{\Sigma, S} F^-}{\Psi \vdash_{\Sigma, S} DA_1 \rightarrow F^- \text{ wf}^-}$$

$$\frac{\Psi \vdash_{\Sigma, S} F^+ \text{ wf}^+}{\Psi \vdash_{\Sigma, S} F^+ \text{ wf}^-}$$

$$\boxed{\Psi \vdash_{\Sigma, S} F^+ \text{ wf}^+}$$

$$\frac{\Psi, x:A_1 \vdash_{\Sigma, S} F^+}{\Psi \vdash_{\Sigma, S} \exists x:A_1. F^+ \text{ wf}^+}$$

$$\frac{\Psi \vdash_{\Sigma} DA_1 \text{ type}}{\Psi \vdash_{\Sigma, S} DA_1 \text{ wf}^+}$$

## 2.2 Typing rules

$$\boxed{\Psi \vdash_{\Sigma, \Gamma} Q:G}$$

$$\frac{\Psi, \Delta \vdash_{\Sigma, \Gamma, S} P:F^-}{\cdot \vdash_{\Sigma, \Gamma} \text{box } S.P : \square S.F^-}$$

$$\boxed{\Psi, \Delta \vdash_{\Sigma, \Gamma, S}^- P:F^-}$$

$$\frac{(\mathbf{x} \in F^-) \in \Delta}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S}^- \mathbf{x} : F^-}$$

$$\frac{\Psi, x:A_1; \Delta \vdash_{\Sigma, \Gamma, S} P:F^-}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S}^- \lambda x:A_1. P : \forall x:A_1. F^-}$$

$$\frac{\Psi, x:DA_1; \Delta \vdash_{\Sigma, \Gamma, S} P:F^-}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S}^- \lambda x:DA_1. P : DA_1 \rightarrow F^-}$$

$$\frac{\Psi; \Delta \vdash_{\Sigma, \Gamma, S}^+ P:F^+}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S}^- P:F^+}$$

$$\frac{\Psi \vdash_{\Sigma} M:A_1 \quad \Psi; \Delta, \vdash P:\Pi x:A_1 F^-}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S}^- PM:F^-}$$

$$\frac{\Psi \vdash_{\Sigma} DM:DA_1 \quad \Psi; \Delta, \vdash P:DA_1 \rightarrow F^-}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S}^- PDM:F^-}$$

$$\frac{\Psi; \Delta, \mathbf{x} \in F \vdash_{\Sigma, \Gamma, S}^- P:F^-}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S}^- \mu \mathbf{x} \in F. P:F^-}$$

$$\frac{\Psi; \Delta \vdash_{\Sigma, \Gamma, S}^- \mathbf{x}:F_1^- \quad \Psi; \Delta \vdash_{\Sigma, \Gamma, S}^- \Omega:F_2^-}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S}^- \text{case } \mathbf{x} \text{ of } \Omega:F_2^-}$$

$$\frac{\Psi; \Delta \vdash_{\Sigma, \Gamma, S} P: F_1^- \quad \Psi; \Delta, \mathbf{y} \in F_1^- \vdash_{\Sigma, \Gamma, S} P': F_2^-}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S} \text{let } \mathbf{y} \in F_1^- = P \text{ in } P': F_2^-}$$

$$\frac{\Psi; \Delta \vdash_{\Sigma, \Gamma, S} P: F_1^- \quad \Psi, x:A_1; \Delta, \mathbf{y} \in F_1^- \vdash_{\Sigma, \Gamma, S} P': F_2^-}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S} \text{let}(x:A_1, \mathbf{y} \in F_1^-) = P \text{ in } P': F_2^-}$$

$$\frac{\Psi; \Delta \vdash_{\Sigma, \Gamma, S} P: F_1^- \quad \Psi, x:DA_1; \Delta \vdash_{\Sigma, \Gamma, S} P': F_2^-}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S} \text{let}(x:DA_1) = P \text{ in } P': F_2^-}$$

$$\frac{\Psi; \Delta \vdash_{\Sigma, \Gamma, S} \Omega: F^-}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S} \Omega: F^-}$$

$$\boxed{\Psi, \Delta \vdash_{\Sigma, \Gamma, S}^+ P: F^+}$$

$$\frac{\Psi \vdash_{\Sigma, \Gamma, S} M: A_1 \quad \Psi, \Delta \vdash_{\Sigma, \Gamma, S} P: F^+[M/x]}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S}^+ \langle M, P \rangle : \exists x: A_1. F^+}$$

$$\frac{\Psi \vdash_{\Sigma, \Gamma, S} DM: DA_1}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S}^+ \langle DM \rangle : DA_1}$$

$$\frac{\rho = \Pi x: A_0. \Pi d_x: DA'_1 \quad \Psi, \rho; \Delta \vdash_{\Sigma, \Gamma, S} P: DA_1}{\Psi; \Delta \vdash_{\Sigma, \Gamma, S}^+ \nu \rho. P: \Pi x: A_0. \Pi d_x: DA'_1. DA_1}$$

### 3 Translation from Core SASyLF into $M_2^+$

Note:  $\overrightarrow{\Delta}$  is used as an abbreviation for  $\Delta_S \Rightarrow \Delta_M$ , and  $\overrightarrow{\Gamma}$  is used as an abbreviation for  $\Gamma_S \Rightarrow$

#### 3.1 Type translation

Note:  $A_0 \sim A$ ,  $A_1 \sim A$ ,  $DA_0 \sim A$ , and  $DA_1 \sim A$ .

$$\boxed{\vdash_{\Sigma} G_S \Rightarrow G_M}$$

$$\frac{\Psi \vdash_{\Sigma, S} F_S^- \Rightarrow F_M}{\vdash_{\Sigma} \Box S. F_S^- \Rightarrow \Box S. F_M}$$

$$\boxed{\Psi \vdash_{\Sigma, S}^- F_S^- \Rightarrow F_M}$$

$$\frac{\Psi, x:A_1 \vdash_{\Sigma, S} F_S^- \Rightarrow F_M}{\Psi \vdash_{\Sigma, S}^- \forall x: A_1. F_S^- \Rightarrow \forall x: A. F_M}$$

$$\frac{\Psi, x:DA_1 \vdash_{\Sigma, S} F_S^- \Rightarrow F_M}{\Psi \vdash_{\Sigma, S}^- DA_1 \rightarrow F_S^- \Rightarrow \forall x: A. F_M}$$

$$\frac{\Psi \vdash_{\Sigma, S}^+ F_S^+ \Rightarrow F_M}{\Psi \vdash_{\Sigma, S}^- F_S^+ \Rightarrow F_M}$$

$$\boxed{\Psi \vdash_{\Sigma, S}^+ F_S^+ \Rightarrow F_M}$$

$$\frac{\Psi, x:A_1 \vdash_{\Sigma, S} F_S^+ \Rightarrow F_M}{\Psi \vdash_{\Sigma, S}^+ \exists x:A_1.F_S^+ \Rightarrow \exists x:A.F_M}$$

$$\frac{\Psi \vdash_{\Sigma} DA_1 \text{ type} \quad \Psi, x:A_1 \vdash_{\Sigma, S} F_S^+ \Rightarrow \top}{\Psi \vdash_{\Sigma, S}^+ DA_1 \Rightarrow \exists x:A.\top}$$

### 3.2 Term translation

Note :  $A_0 \sim A$ ,  $A_1 \sim A$ ,  $DA_0 \sim A$ , and  $DA_1 \sim A$ .

$$\boxed{\Psi; \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}} Q_S : G_S \Rightarrow Q_M : G_M}$$

$$\frac{\Psi, \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^- P_S : F_S^- \Rightarrow P_M : F_M}{\Psi; \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}} \text{box } S.P_S : \square S.F_S^- \Rightarrow \text{box } S.P_M : \square S.F_M}$$

$$\boxed{\Psi, \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^- P_S : F_S^- \Rightarrow P_M : F_M}$$

$$\frac{(\mathbf{x} \in F_S^-) \in \Delta_S \quad (\mathbf{x} \in F_M) \in \Delta_M}{\Psi, \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^- \mathbf{x} : F_S^- \Rightarrow \mathbf{x} : F_M}$$

$$\frac{\Psi, \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^+ P_S : F_S^+ \Rightarrow P_M : F_M}{\Psi, \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^- P_S : F_S^+ \Rightarrow P_M : F_M}$$

$$\frac{\Psi, x:A_1; \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^- P_S : F_S^- \Rightarrow P_M : F_M}{\Psi; \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^- \Lambda x:A_1.P_S : \forall x:A_1.F_S^- \Rightarrow \Lambda x:A.P_M : \forall x:A.F_M}$$

$$\frac{\Psi, x:DA_1; \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^- P_S : F_S^- \Rightarrow P_M : F_M}{\Psi, \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^- \Lambda x:DA_1.P_S : DA_1 \rightarrow F_S^- \Rightarrow \Lambda x:A.P_M : \forall x:A.F_M}$$

$$\frac{\Psi \vdash_{\Sigma} M : A_1 \quad \Psi; \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^- P_S : \Pi x:A_1.F_S^- \Rightarrow P_M : \forall x:A.F_M}{\Psi, \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^- P_S M : F_S^- \Rightarrow P_M M : F_M[M/x]}$$

$$\frac{\Psi \vdash_{\Sigma} DM : DA_1 \quad \Psi; \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^- P_S : DA_1 \rightarrow F_S^- \Rightarrow P_M : \forall x:A.F_M}{\Psi, \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^- P_S M : F_S^- \Rightarrow P_M DM : F_M[M/x]}$$

$$\Psi; \Delta_S, \mathbf{x} \in F_S^-; \Delta_M, \mathbf{x} \in F_M \vdash_{\Sigma, \vec{\Gamma}, S}^- P_S : F_S^- \Rightarrow P_M : F_M$$

$$\Psi; \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^- \mu \mathbf{x} \in F_S^- . P_S : F_S^- \Rightarrow \mu \mathbf{x} \in F_M . P_M : F_M$$

$$\frac{(\Omega \in F_S) \in \Delta_S \quad (\Omega, (\Psi' \triangleright \psi \mapsto P) \in F_M) \in \Delta_M}{\Psi, \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^- \Omega : F_S \Rightarrow \Omega, (\Psi' \triangleright \psi \mapsto P) : F_M}$$

$$\boxed{\Psi, \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^+ P_S : {}^+ F_S^+ \rightarrow P_M : F_M}$$

$$\frac{\Psi, x:A_1; \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^+ P_S : F_S^+ \Rightarrow P_M : F_M}{\Psi; \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^+ \langle M, P_S \rangle : \exists x:A_1. F_S^+ \Rightarrow \langle M, P_M \rangle : \exists x:A. F_M}$$

$$\frac{\Psi, x:A_1; \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S} DM : DA_1 \Rightarrow \langle M, \langle \rangle \rangle : \top}{\Psi, \vec{\Delta} \vdash_{\Sigma, \vec{\Gamma}, S}^+ \langle DM \rangle : DA_1 \Rightarrow \langle M, \langle \rangle \rangle : \exists x:A. \top}$$

APPENDIX B

SASyLF judgments required to prove ok-thm.

terminals fn unit

$e ::= x$   
 | "(" ")"  
 | e e  
 | fn  $x : \tau \Rightarrow e[x]$

$\tau ::= \text{unit}$   
 |  $\tau \rightarrow \tau$

$\Gamma ::= *$   
 |  $\Gamma, x:\tau$

judgment has-type:  $\Gamma \vdash e : \tau$   
 assumes  $\Gamma$

----- t-var  
 $\Gamma, x:\tau \vdash x : \tau$

$\Gamma, x1:\tau \vdash e1[x1] : \tau'$   
 ----- t-fn  
 $\Gamma \vdash \text{fn } x1 : \tau \Rightarrow e1[x1] : \tau \rightarrow \tau'$

$\Gamma \vdash e1 : \tau' \rightarrow \tau$   
 $\Gamma \vdash e2 : \tau'$   
 ----- t-app  
 $\Gamma \vdash e1 e2 : \tau$

## BIBLIOGRAPHY

- Aldrich, J., Simmons, R., & Shin, K. (2008). SASyLF: An Educational Proof Assistant for Language Theory. *FDPE '08: Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education* (pp. 31-40). New York, NY: ACM.
- Bertot, Y., & Castéran, P. (2004). *Interactive Theorem Proving and Program Development*. Springer-Verlag.
- Lee, D. K., Crary, K., & Harper, R. (2007). Towards a Mechanized Metatheory of Standard ML. *Symposium on Principles of Programming Languages* .
- Nipkow, T., Paulson, L., & Wenzel, M. (2002). *Isabelle/HOL -- A Proof Assistant for Higher-Order Logic*. Springer.
- Norrish, M. (1998). C Formalised in HOL.
- Pfenning, F., & Elliot, C. (1988). Higher-order Abstract Syntax. *SIGPLAN '88 conference on Programming Language Design and Implementation* , 199-208.
- Pfenning, F., & Schürmann, C. (1999). System Description: Twelf - A Meta-Logical Framework for Deductive Systems. *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)* (pp. 202-206). Springer-Verlag LNAI.
- Schürmann, C. Automating the meta theory of deductive systems. Thesis, Carnegie Mellon University, December 1998.
- Simmons, R. (2005). *Twelf as a Unified Framework for Language Formalization and Implementation*. Princeton University, New Jersey.

## ABSTRACT

SASyLF, an LF-based proof assistant, is designed to be accessible to students learning type theory and semantics. In this paper, we present a core calculus semantically similar to SASyLF and describe its formalization in LF, or more specifically,  $M_2^+$ , a functional programming language with a type system with dependent types. We represent the semantics of SASyLF in  $M_2^+$ . The formalization of SASyLF can serve as a basis for further language development as well as give potential users tools to reason clearly about the language.