Evolution of SASyLF 2008-2021

John Tang Boyland*
University of Wisconsin-Milwaukee Milwaukee, Wisconsin, USA
boyland@uwm.edu

SASyLF was released in 2008 and used as a proof assistant in courses at several universities. It proved itself useful and has continued to be used, and each iteration of use has encouraged further development: fixing bugs and adding enhancements. This paper describes how SASyLF was developed while keeping true to its purpose. Most notable are making substitutions explicit, support of "and" and "or," support for mutual and lexicographic induction, and IDE support.

1 Introduction

SASyLF (Second-order Abstract Syntax Logical Framework) is a proof assistant based on LF using higher-order abstract syntax (HOAS) [11], but limited to second-order, for notational simplicity. SASyLF was introduced in 2008 [1] by Aldrich, Simmons and Shin. The project has a webpage (`sasylf.org`) hosted at Carnegie Mellow and a page at github (`https://github.com/boyland/sasylf/`).

SASyLF has been used in courses based on Pierce's *Types and Programming Languages* [13] at Carnegie Mellon University, University of California at Los Angeles, University of Wisconsin-Milwaukee, ETH Zurich, and Northeastern University. The initial paper discusses some of the responses collected from students, including that most felt that SASyLF helped them learn how to write natural langauge proofs. The repeated interaction with new users exposes previously unknown defects, underscores known mis-features and leads to enhancement requests. This paper reports on the work completed (as of SASyLF release 1.5.0) and the plans for the future.

The stated purpose of SASyLF was to provide a gentler learning curve to proof mechanization for students within a type theory course, and secondarily for researchers. The design philosophy of SASyLF was (and is) to follow conventions of paper proofs as closely as possible, but also to make all the steps explicit, so that errors could be easily pinpointed and expressed in terms of these high-level concepts.

For example, when describing the structure of the language supporting the type system, SASyLF follows paper conventions by having the user specify a context free language, for example for the simply-typed lambda calculus with a "unit" constant:

$$\begin{array}{rcl} e & ::= & \lambda x\!:\!\tau \cdot e \\ & \mid & x \\ & \mid & e\, e \\ & \mid & () \end{array}$$

```
syntax
  e ::= fn x : tau => e[x]
     | x
     | e e
     | "(" ")"
```

On the left, we give a typical presentation in a paper, and on the right, we give the equivalent in SASyLF (from Figure 1 [1]). In this example, we see that expressions ("`e`") have four forms: lambda expressions, variables, application and the unit constant ("`()`"). The parentheses for the unit constant in SASyLF have to be quoted since parentheses have special meaning. The other symbols (e.g., "`:`") do not need to be quoted, but may be. The square brackets in SASyLF ("`[.]`") used in the definition of lambda

*Work done in part while the author was in residence at Northeastern University, Boston, MA

J. Marcos, W. Neuper and P. Quaresma (Eds.): Theorem Proving Components for Educational Software 2021 (ThEdu'21)
EPTCS 354, 2022, pp. 87–107, doi:10.4204/EPTCS.354.7

© Boyland, John
This work is licensed under the Creative Commons Attribution License.

expressions express that the body of the lambda may have uses of variable "x" in it. SASyLF supports higher-order abstract syntax (HOAS [11]) so that the user doesn't have to define complex name-based substitution rules.

Another way that shows how SASyLF follows paper conventions for specifying type systems is in the inference rules. For example, consider the declaration of the typing judgment and one of its rules:

(typing) $\Gamma \vdash e : \tau$

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\, e_2 : \tau}\ \text{T-APP}$$

```
judgment has-type: Gamma |- e : tau
assumes Gamma

Gamma |- e1 : tau' -> tau
Gamma |- e2 : tau'
------------------------- t-app
Gamma |- e1 e2 : tau
```

The figure on the left is a typical presentation which shows the form of the typing relation and then uses natural-deduction-style inference rules. On the right, we express the same idea in SASyLF (adapted from Figure 2 [1]). The only significant difference is that in SASyLF, the fact that "`Gamma`" includes bindings that may be used in the expression is declared using "assumes."

The Curry-Howard Correspondence highlights the connection between computation and logic. In particular, a proof can be seen as a computational object, and *vice versa*. The correspondence is often used to highlight the logical foundations of programming, especially to mathematicians. In contrast, SASyLF encourages the dual viewpoint by using programming as a metaphor for proof construction. The purpose is to make the less familiar world of logical proofs accessible to competent programmers. For example, if a proof uses a lemma, it will use syntax strongly resembling a function call, and a proof by induction is expressed with (functional) recursion. An example of a proof in SASyLF in given in the next section in Figure 2.

As originally designed, SASyLF incorporated a number of aspects to help it meet its design goals. In particular:

- The syntax of the terms being studied is described with context-free grammars. Syntactic terms and judgments are described syntactically and parsed using GLR parsing [15].
- Name binding is supported using Higher-Order Abstract Syntax (HOAS) within LF [11].
- Derivations are produced one step at a time ("let normal form").
- Incremental proof construction is aided by the ability to leave unspecified part of the proof while continuing elsewhere.

Seeing all the struggles that compilers students have getting a grammar accepted as LALR(1) (or LL(k)), it is refreshing that users in SASyLF simply need to give a context-free grammar. Ambiguity need only be handled where it occurs, primarily using parentheses. So for example, the following term is ambiguous:

```
fn x => x "(" ")"      must be written
fn x => (x "(" ")")    or
(fn x => x) "(" ")"    .
```

The original SASyLF paper mused the addition of precedence declarations to handle ambiguity, but this was never added. For pedagogical purposes, it seems best to require that all ambiguity be explicitly avoided with parentheses; it makes it clear that the various choices are different ASTs.

The use of HOAS has been mostly successful. The alternative, deBruijn notation, is tricky to learn and even in its clearest exposition ("locally nameless" [8]) requires several technical lemmas for every

use of binding. In HOAS, students mainly need only be aware of standard static scoping rules, plus be cognizant of "alpha renaming," the irrelevance of formal parameter names. The hardest parts to explain are the restrictions on the "assumption rule."

The most serious problem with HOAS is that certain proof approaches are simply not possible. For instance, it's not possible to express that all types of variables in a context have a certain property that is not shared by all types. We have started the theoretical work on extending LF with new features to support the necessary changes to the meta-language [4] but that work has stalled.

In sharp contrast to most other proof assistants, SASyLF eschews the use of "tactics," powerful techniques for generating a proof automatically. At the risk of gross generalization, tactics handle all the easy aspects of proofs but leave the harder parts. This is the wrong approach for a novice, who lacks the intuition for the easier parts, let alone the harder parts.

So it seems that SASyLF as designed has the right approach for educational purposes; our approach to changes is to ensure that these goals are kept in view. In the next section, we discuss the changes already made to SASyLF, and in the following section look at changes that are under consideration.

2 Changes in SASyLF

The original paper anticipated further developments. For reference, they are listed here with a brief comment on to what extent the anticipation was realized.

1. "[We] are developing an integrated development environment that will help both novice and expert users write boilerplate code. For example, it will allow students to drag a rule into a case analysis and have the syntactic structure of the case for that rule generated automatically."

   Indeed, an Eclipse plugin for SASyLF was initiated within a short time and this plugin has undergone numerous improvements. There is still no support for "dragging" a rule into a case analysis, but perhaps better, a "Quick Fix" has been provided which will populate a case analysis with all missing cases.

2. "In the future we plan to support namespaces to allow a file to rely on declarations from another file without worrying about name clashes."

   The module system is still basic, but does permit a proof to be used in another proof without name clashes.

3. "We have considered adding 'by solve' which would automatically search for a derivation, but this poses challenges in terms of giving away a derivation to students in an educational setting."

   Indeed "by solve" does introduce the equivalent of a simple tactic into SASyLF. It was added as a project but hasn't been maintained. Fortunately for educational purposes, it can only find simple proofs.

4. "However, there are a few checks that are not yet implemented, including the checks for substitution, weakening, exchange, and contraction."

   Substitution, weakening and exchange were all fully checked soon after Boyland took over maintenance. Contraction remains a reserved word, but nothing can be proved "by contraction" (the justification is not even accepted by the parser). Since contexts in SASyLF are (currently) inextricably bound up with variable introduction, and since it is not legal to bind the same variable twice ($\Gamma, x:T, x:T \vdash R[x]$ is not legal), contraction is not ever applicable. The superficially similar situation ($\Gamma, x:T, y:T \vdash R[x,y]$) can be already be reduced using "substitution" to ($\Gamma, x:T \vdash R[x,x]$).

5. "Although some cleanup work is needed, the implementation could eventually serve another educational role: illustrating, through a translation from paper-proof syntax into the LF type theory, the formal underpinnings of common notation and some of the basic ideas behind LF-based theorem provers."

   We are not aware of any use of SASyLF for this purpose.

6. "We believe Part 2B [of POPLmark] will also be feasible in SASyLF; Parts 1A and 2A will require the implementation of mutual induction (not expected to be hard to add since the type theory has already been worked out in Twelf).

   Indeed mutual induction was added early on. It is not *needed for Part 2A, but is needed for Part 1A with narrowing being proved inductively with transitivity of algorithmic subtyping. Part 2A has been completely mechanized in SASyLF, but Part 1A is currently not mechanizable due to the context manipulation needed by narrowing.*

7. "This comment suggests the need for better visualizations of proofs——at a minimum, syntax highlighting, but perhaps output in LaTeX or a graphical depiction of a derivation tree. Several students felt that a tree-like presentation of a derivation was easier to read than the linear format supported by the tool."

   The IDE includes syntax highlighting which is a big help to avoid some simple syntax errors, but graphical views have not been implemented. Nor has there been any demand for them, since the original paper, to our knowledge.

8. "Most . . . students still did have times when they wished they had better feedback, for example real-time feed-back on syntax errors and incorrect rule applications, rather than continually rerunning the tool, which was 'disruptive and annoying.' IDE integration could help with such issues."

   Indeed, anecdotally, the IDE has been very helpful to students, although some used the command-line version occasionally.

In the remainder of this section we discuss the changes that actually have been put into effect, as of SASyLF 1.5.0, starting with Eclipse plugin.

2.1 Eclipse Plugin

Programmers expect IDE support while coding, and to continue the identification of proving with coding, SASyLF developers soon provided support in an Eclipse plugin. Incrementally, new features have been added. While editing, users will notice

- an outline view,
- syntax highlighting,
- parenthesis matching,
- automated indentation,
- error and warning markers,
- folding and unfolding of proof segments,
- content assist (e.g. lemma name completion).

Some of these features can be seen in Figure 1. The plugin provides a number of capabilities:

- Checking proofs;

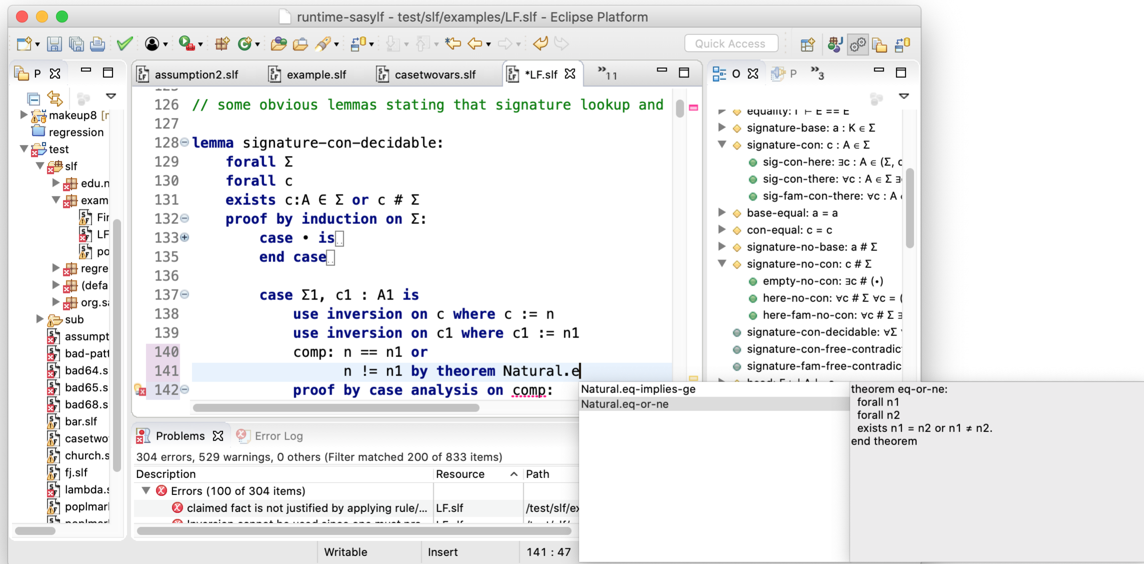

Figure 1: Screenshot of SASyLF 1.5.0 under Eclipse.

- Addition/removal of comments;
- Quick fixes of many kinds of errors;
- Finding the definition of a theorem;
- Changing syntax highlighting colors ("dark" mode available);

Unlike common plugins, such as for Java, the syntax is not checked on every keystroke. The checking process is still non-incremental and so is done only on a save, or on demand (e.g., pressing the green check-mark in the tool bar).

One of the most consequential aspects of the plugin is the "Quick Fix" system. For many errors, the error marker in the left ruler contains a "light bulb" icon (see Figure 1). Clicking on that, or using the "Quick Fix" keyboard short-cut brings up a dialog in which one can select a fix. For missing cases (as this error is), the plugin will add the cases which the proof is currently missing. Indeed this feature is so powerful, that it is recommended to *not* inform students of its existence for the first few weeks, so students will write their own case analyses.

The proof engine of SASyLF (which can be run from the command line) is architecturally separated from the Eclipse plugin architecture. Indeed the "smarts" of "Quick Fix" are available from the command-line, enabling enterprising students to use SASyLF with vim or IntelliJ.

2.2 Syntactic Comparison

SASyLF 1.5.0 accepts almost all legal proofs written in the original syntax, but provides features not originally available. Figure 2 contrasts Figure 4 from the 2008 paper with a proof more idiomatic in current SASyLF. The new proof has italicized comments highlighting the main differences. Another difference is that the names of nonterminals and rules is closer to that used in Pierce's TAPL [13].

The change that makes the new proof *longer* (otherwise, it is noticeably shorter) is the addition of optional "where" clauses [2]. This addition has an important explanatory value: it makes explicit how

```
theorem preservation: forall dt: * |- e : tau
                      forall ds: e -> e'
                      exists * |- e' : tau.
dt': * |- e':tau                   by induction on ds :
  case rule
    d1 : e1 -> e1'
    -------------------- c-app-l
    d2 : e1 e2 -> e1' e2
  is
    dt' : * |- e' : tau        by case analysis on dt :
      case rule
        d3 : * |- e1 : tau' -> tau
        d4 : * |- e2 : tau'
        -------------------------- t-app
        d5 : * |- (e1 e2) : tau
      is
        d6 : * |- e1' : tau' -> tau
                     by induction hypothesis on d3, d1
        dt': * |- e1' e2 : tau by rule t-app on d6, d4
      end case
    end case analysis
  end case

  case rule... // case for rule c-app-r is similar

  case rule
    d1 : e2 value
    ---------------------------------------- r-app
    d2 : (fn x : tau' => e1[x]) e2 -> e1[e2]
  is
    dt' : * |- e' : tau       by case analysis on dt :
      case rule
        d4 : * |- fn x : tau' => e1[x] : tau'' -> tau
        d5 : * |- e2 : tau''
        ----------------------------------------- t-app
        d6 : * |- (fn x : tau' => e1[x]) e2 : tau
      is
        dt' : * |- e' : tau    by case analysis on d4 :
          case rule
            d7: *, x:tau' |- e1[x] : tau
            -------------------------------------- t-fn
            d8: * |- fn x:tau' => e1[x] : tau' -> tau
          is
            d9: * |- e1[e2] : tau by substitution on d7, d5
          end case
        end case analysis
      end case
    end case analysis
  end case
end induction
end theorem
```

```
theorem preservation :
    // Unicode identifiers and operators supported
    forall d: * ⊢ t : T
    forall e: t → t'
    exists * ⊢ t' : T // “.” not required

    // Induction can be declared separately
    use induction on d

    // “proof” serves instead of repeating goal
    proof by case analysis on e:
        case rule
            e1: t1 → t1'
            --------------------- E-App1
            _: (t1 t2) → (t1' t2)
            where t := t1 t2 and t' := t1' t2
            // “where” makes substitution explicit
        is
            // “and” allows proving things together
            d1: * ⊢ t1: T' → T and
            d2: * ⊢ t2: T' by inversion on d
            // “inversion” does a single case analysis

            d1': * ⊢ t1': T' → T
              by induction hypothesis on d1, e1
            proof by rule T-App on d1', d2
        end case

        case rule ... // case E-App2 is similar

        case rule
            v2: t2 value
            -------------------------- E-AppAbs
            _: (λ x:T'. t11[x]) t2 → t11[t2]
            where t := (λ x:T'. t11[x]) t2
             and t' := t11[t2]
            // “.” can be used in syntax, e.g., in lambdas
        is
            d1: * ⊢ λ x:T'.t11[x] : T''→T and
            d2: * ⊢ t2 : T'' by inversion on d
            d11: *, x:T' ⊢ t11[x] : T
                 by inversion on d1
                 where T'' := T'
            // inversions can cause substitutions too

            proof by substitution on d11, d2
        end case
    end case analysis
end theorem
```

Figure 2: Comparing Fig. 4 from [1] (left) to more idiomatic expression in SASyLF 1.5.0 (right).

meta-variables (such as `T''`) are changed by the pattern match. We found that students new to proof mechanization often are unaware of how pattern matching changes the identity of meta-variables. And not only novices. For instance, the answer to exercise 9.3.9 on pages 506 and 507 the first edition of TAPL incorrectly presumed that (the equivalent of) `T'` and `T''` were identical in the E-APPABS case, when that identification only happens later when the typing relation for the lambda expression is inverted.

The other changes here can be seen as conveniences, generalizations and extensions of the original SASyLF features.

2.3 Conjunction and Disjunction

In SASyLF, a theorem or lemma has a single judgment as a result; if two results are desired they must be packaged together in a single judgment. Sometimes a choice result is desired; for example in an introductory type systems course, students prove "progress" with the result that a term is either a value *or* can be evaluated in a further step.

Conjunction and disjunction can be done easily through the introduction of new judgments:

$$\frac{A \quad B}{A \ \texttt{and}\ B} \qquad \frac{A}{A \ \texttt{or}\ B} \qquad \frac{B}{A \ \texttt{or}\ B}$$

Ignoring contexts for now, nothing more at the meta-level is needed for conjunction to work. The system as implemented currently creates these judgments automatically (conjunctions or disjunctions of two or more judgments at a time) on demand. As a special case, the (new) keyword `contradiction` is parsed an empty disjunction.[1]

A technical difficulty with disjunctions arises when one considers that the implicit arguments of a rule includes all free meta-variables of the premises or conclusion. In the case of conjunction, all meta-variables in the conclusion are present in the premises, but in the case of a disjunction, meta-variables free in one disjunct but not in another need to be supplied even when the latter is being used to satisfy the disjunction. For a simple example, consider a typical "progress" theorem, which proves that a term t is a value or that is evaluates $t \to t'$. The question is what should t' be for the disjunction if t is a value?

Since meta-variables in SASyLF are never dependently typed, and they are described with context-free grammars, the types are always inhabited assuming that the non-terminal is *productive*, which is a simple recursive check. Unproductive nonterminals are arguably errors in the specification, and so SASyLF was updated to mandate that all non-terminals be productive. That simple change is enough for the meta-theory to accommodate conjunctions and disjunctions of judgments without contexts.

If a conjunction is proved by a lemma or theorem (or inversion), the syntax requires that the separate judgments be split and named separately, as seen in in Fig. 2 in two cases of inversion. This obviates the need for an explicit elimination rule for conjunctions. Conjunctions are introduced by simply listing the two parts as in "`proof by d1, d2`."

In contrast, using a disjunction needs case analysis, as seen in this excerpt from the "progress" theorem from the mechanization of iso-recursive subtyping from the public repository:

```
ns1: t1 value or t1 -> t1'
    by induction hypothesis on d1
_: t1 t2 -> t' by case analysis on ns1:
    case or e1: t1 -> t1' is
        e: t1 t2 -> t1' t2 by rule E-App1 on e1
```

[1]The other use of the new keyword is the justification "by contradiction on" which refers to an empty case analysis.

```
        end case
        ...
```

The syntax doesn't require that the whole (implicitly created) rule is given, it is necessary only to provide the particular disjunct being handled. Introducing a disjunction is done by proving one of the disjuncts, or a disjunction of a subset of them.

Joining judgments with contexts interacts with the meta-theory because the context is not actually part of the judgment. As a result, a rule declaration can only produce a judgment with an opaque context (e.g., `Gamma`). The premises must use this context, possibly adding extra requirements. For example the typical T-ABS rule has the form

$$\frac{\Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{t}[\mathtt{x}] : \mathtt{T}'}{\Gamma \vdash \lambda \mathtt{x} : \mathtt{T}.\mathtt{t}[\mathtt{x}] : \mathtt{T} \to \mathtt{T}'} \text{ T-ABS}$$

The meta-level expression of this rule is

$$\frac{\Pi x : \mathtt{t}.\mathtt{typing}[x, T] \to \mathtt{typing}[t[x], T']}{\mathtt{typing}[\mathtt{lam}[T, t], \mathtt{arr}[T, T']]} \text{ T-ABS}$$

This form uses `typing` for the LF type of the typing judgment, square brackets $[\ldots]$ for application of arguments to an LF function or dependent type, and $\Pi x : T.T'[x]$ for dependently-typed function types (and traditional function type syntax $T \to T'$ when the result type is independent of the argument type). The constants `lam` and `arr` are the constructors of lambda expressions and arrow types respectively in the syntax being described. Here we distinguish the (LF) type of terms `t` from a meta-variable and implicit argument to the rule t.[2]

So the question is how do we represent the result of a hypothetical canonical forms lemma for arrow types?

$$\mathtt{t} = \lambda \mathtt{x} : \mathtt{T}.\mathtt{t1}[\mathtt{x}] \;\; \mathtt{and} \;\; *, \mathtt{x} : \mathtt{T} \vdash \mathtt{t1}[\mathtt{x}] : \mathtt{T}'$$

In this case, only one judgment uses a context (the equality judgment here only operates on closed terms), and so we can just pull the entire context out and generate the conjoined judgment with the single rule:

$$\frac{\mathtt{t}_1 = \mathtt{t}_2 \qquad \Gamma \vdash \mathtt{t}_3 : \mathtt{T}}{\mathtt{t}_1 = \mathtt{t}_2 \;\; \mathtt{and} \;\; \Gamma \vdash \mathtt{t}_3 : \mathtt{T}} \text{ Eq+T}$$

Then our example can be rendered as follows where the LF context is pulled to the outer level:

$$\Pi x : \mathtt{t}.\mathtt{typing}[x, T] \to \mathtt{Eq{+}T}[t, \mathtt{lam}[T, t_1], t_1[x], T']$$

The excess context will not affect the equality judgment in the premise, since it does not depend on the context.

A similar approach can be used if multiple judgments use the same context and the conjunction of the derivations all use the same derivation, but what if we have a version of the last example where equality needs the context?

$$\Gamma \vdash \mathtt{t} = \lambda \mathtt{x} : \mathtt{T}.\mathtt{t1}[\mathtt{x}] \;\; \mathtt{and} \;\; \Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{t1}[\mathtt{x}] : \mathtt{T}'$$

[2]Some readers may notice that, against representation norms, we don't give t in "eta-expanded" form when we pass it to `lam`. Internally, the eta-expanded form is used.

In this case, pulling out the context to the top would cause the equality judgment to depend on the bindings. The solution is to create a special judgment with a single rule that can add the needed binding just for typing judgment:

$$\frac{\Pi x:\texttt{t}.\texttt{typing}[x,T_1] \rightarrow \texttt{typing}[t_2[x],T_3]}{\texttt{T*V}[T_1,t_2,T_3]}\ \texttt{T*V}$$

Then the desired conjunction can be represented as

$$\texttt{Eq+T*V}[t,\texttt{lam}[T,t_1],T,t_1,T']$$

The same approach can be used to conjoin judgments that use different contexts as long only one them builds on unknown bindings in the LF context. This restriction is moot since SASyLF already bans a theorem (or lemma) or judgment from having two contexts.

Once the meta-theory was handled, what remained was to check the contextual parts that are not handled by the meta-theory (naming of contexts) and to provide implicit coercions to hide the existence of the constructed judgments and rules.

2.4 Syntactic Sugar

It is common in presentations of type systems to give multiple letters for use as meta-variables, for example for types:

$$S,T ::= \texttt{A} \mid S \rightarrow T \mid S+T$$

The grammar for SASyLF had "space" for this extension, so it was added. Around the same time, short-cut definitions were added so that one can write

$$\text{id} := \lambda\texttt{x}.\texttt{x}$$

The operator `:=` is used because we are not defining a new non-terminal, but rather defining a short-hand.

In the same vein, *derived syntax* can be defined. It differs from the latter in having non-terminal parameters:

$$(\texttt{t}_1\texttt{;}\ \texttt{t}_2) := (\lambda\texttt{x}:\texttt{Unit}.\texttt{t}_2)\texttt{t}_1$$

The numbering ensures that the terms are substituted in the correct order. The left-hand side must be parenthesized to avoid parsing ambiguity, and is defensible as one is defining the syntax as a whole.

Derived syntax was easy to add to the system, as it simply added some new productions to the concrete syntax used in the generalized LR parser.

2.5 Induction

The fundamental danger with recursion is non-termination, and the corresponding problem with proofs using induction is circular reasoning. And so induction can only be permitted if the recursion can be shown to always terminate. The standard way to prove termination is with a "measure" that is *reduced* in every call. The measure needs to be a *well-founded* partial order, that is, it has no infinite decreasing chains.

In computer science, *structural* induction is the fundamental induction technique in proofs. It uses the structure of terms: an inductive call with one term is a reduction of the current call if the new term is a sub-term of the current one. Since all terms are (by definition) finite, this measure is guaranteed to be

well-founded. SASyLF uses structural induction with both syntax terms and judgments. A premise used in the rule used to prove judgment instance is a sub-term.

This basic concept is complicated by higher-order terms, from binders and from assumptions. We generalized SASyLF's rules of structure induction on HOAS to follow those of Twelf. The Twelf User Manual [12] describes restrictions sufficient to ensure soundness.

This discussion of structural induction permits us now to discuss the user-visible extensions to SASyLF. One of the earliest additions to SASyLF was mutual induction. Theorems (and lemmas) can be connected into a mutual induction group through the keyword "`and`" and then (forward) references within the group are allowed as long as inductive argument is reduced. And backward references are allowed additionally even if the inductive argument is unchanged.

An immediate issue is that how does one theorem "know" what the induction argument of another theorem is? In the original SASyLF, induction was only indicated in a form of case analysis. But the proof of a mutually inductive theorem may not need any case analysis, and forcing a case analysis would needlessly complicate the proof. And so mutual inductive theorems were assigned their first argument as the inductive argument, by default, if there is no explicit induction declaration.

Some time after mutual induction was added, new syntax was added to permit induction to be declared separate from case analysis:

```
use induction on d
```

For backward compatibility (for the vanishing small SASyLF community), the implicit induction on the first argument was preserved. This implicit induction has been deprecated in SASyLF 1.5.0; all mutually inductive theorems and all theorems using induction (explicit or implicit) must have an explicit declaration of induction to avoid a warning.

The new syntax for declaring induction gave "space" to extend structural to lexicographical induction (as seen in the example proof of "cut-elimination"):

```
use induction on A, d1, d2
```

This powerful induction technique permits induction to seek a reduction in `d1` if `A` stays the same, and further to look for a reduction in `d2` if the first two are unchanged. What makes this reduction measure powerful is that if `A` is reduced, then `d1` and `d2` are unrestricted; they can get arbitrarily bigger.

Another common induction situation is where arguments are swapped in a use of induction. Proofs of transitivity of algorithmic subtyping often have such a situation when handling contravariant type constructors such as in function types. SASyLF was extended to support "unordered" induction (as seen in the example proof of composition of hierarchical substitution in LF):

```
use induction on {α0, α2}, s2
```

This example uses lexicographical induction in which the first argument is an unordered induction of the two "simple types" ($\alpha 0, \alpha 2$), and the second is a substitution judgment (`s2`). Unordered induction requires that under some permutation of the elements, every element reduces or stays the same, with at least one element reducing. The possibility for permutation makes it strictly more powerful than the similar induction schema in Twelf, which is just a special case of lexicographic induction.

The LF hereditary substitution composition example involves mutual induction as well. All theorems in a mutual induction group must declare induction of the same form (same number, groupings and type).

2.6 Special cases

A useful programming idiom (especially in imperative programming) is to handle a special case before a more involved computation, because then the more involved computation does not need to take into the

special case that has already been handled. A similar situation can happen with proofs: there may be a case for one of the inputs to a theorem that if handled ahead of time, the rest of the proof can be easier. These special cases are unremarkable in informal proofs, but don't fit well into a formulation based on functional programming.

Partial case analysis in SASyLF 1.5.0 can be used to handle such situations.

```
do case analysis on ...:
  ...
end case analysis
```

A partial case analysis on a subject has the same kind of cases and the same proof obligation as the surrounding context, but it need only handle a subset of the possible cases. Importantly, the system "remembers" that the cases are already handled and so later in the same context if there is case analysis on the same subject, the cases that are already handled can (and indeed must) be omitted. In particular "by inversion" (only one case) and "by contradiction" (no cases) may be possible when they would not be, previous to the partial case analysis.

Partial case analysis was first introduced as a way to reduce case explosion, but it has served as a useful tool for working around incompleteness in higher-order unification. The mechanization of recent work on iso-recursive subtyping [16] gives an example. The canonical forms lemma for arrow type says that if we have a value of arrow type then the value has the form of a lambda abstraction. The problem is that the typing judgment includes an "unfolding" possibility:

$$\frac{\mathtt{T} = (\mu \mathtt{X}.\mathtt{T}_1[\mathtt{X}]) \qquad \Gamma \vdash \mathtt{t} : \mathtt{T}}{\Gamma \vdash \mathtt{unfold}(\mathtt{T})\,\mathtt{t} : \mathtt{T}_1[\mathtt{T}]}\ \text{T-U}{\small\text{NFOLD}}$$

Recall that the SASyLF term $\mathtt{T}_1[\mathtt{T}]$ is represented with LF term application. The problem is that there is no MGU for the unification problem $T_1[T] \stackrel{?}{=} \mathtt{arr}[T_2, T_3]$. Thus if a SASyLF proof attempts to do a case analysis on a proof of $\Gamma \vdash \mathtt{t} : \mathtt{T}_2 \to \mathtt{T}_3$, the possibility of the T-UNFOLD rule leads to an "incomplete unification" error.

The work-around, is to "disguise" the arrow type in an equality judgment, do a partial case analysis handling the problematic rules (such as T-UNFOLD) and then afterwards invert the equality judgment and proceed as normal.

2.7 Relaxation

The notation that SASyLF supports for using judgments from the LF context only handles the case where the assumption is the last one added, for example:

$$\frac{}{\Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{x} : \mathtt{T}}\ \text{T-V}{\small\text{AR}}$$

Then "weakening" can be used to add more assumptions after the one used. This semantics is supported by the meta-theory when creating a judgment instance.

But pattern matching does not proceed as cleanly. In Twelf, pattern matching cannot reach into the LF context. Instead if one needs to prove a meta-theorem for which a case can be in the LF context, one needs to place an instance of the theorem being proved in the context along with the assumption. This idiom is peculiar to Twelf and not close to how one does the proof on paper.

Instead SASyLF permits case analysis to operate on the context. The problem is that it uses the rule which positions the assumption at the end of the context. For example, if we are writing a substitution

lemma that does case analysis on $\Gamma, \mathtt{x} : \mathtt{T}_2 \vdash \mathtt{t}[\mathtt{x}] : \mathtt{T}_1$, then as well as the case that `t[x]` is `x` (which is handled without needing to reach into Γ), we have the additional case:

$$\frac{}{\Gamma', \mathtt{x}' : \mathtt{T}_1, \mathtt{x} : \mathtt{T}_2 \vdash \mathtt{x}' : \mathtt{T}_1} \text{ T-VAR}$$

The case apparently implies that $\Gamma = \Gamma', \mathtt{x}' : \mathtt{T}_1$, in other words, that the new assumption we found was at the end of the LF context. In the case of the simply-typed lambda calculus, this implication is fine since assumptions can be freely permuted. But in some type systems, the assumptions cannot be permuted, for example in the presence of type variables, as in $F_{\leq}$.

Indeed, one can construct a "proof" of a contradiction in SASyLF if the apparent implication is permitted to have validity (see `bad38.slf` in the public SASyLF repository). Thus case analysis in the context needs proper support in the meta-theory. This support is provided in the concept of "relaxation."

The simple idea at the kernel of relaxation is that rather than saying that the current context is *equal* to the new (smaller) context with the assumption at the end, the current context *includes* the new context with the assumption:

$$\Gamma \geq \Gamma', \mathtt{x}' : \mathtt{T}_1$$

Then any judgment defined in the context on the right can be moved back to Γ through *relaxation*, a form of weakening, although new assumptions are not explicitly added. In the process of relaxation, the variable $\mathtt{x}'$ is disguised by being replaced with $\mathtt{t}[\ldots]$ (the parameter `x` is known to be unused and so can be replaced with anything in scope). Furthermore relaxation gives no status to any intermediate assumptions (the "$\mathtt{x} : \mathtt{T}_2$" in the example). In general there could many assumptions between the one found in the pattern analysis and the known assumptions at the end.

The semantics we give here is a stop-gap until contexts are properly brought into the meta-theory (see the "Future Work" section).

2.8 Making Substitutions Explicit

As mentioned briefly earlier in this work, and as described in much more detail in a separate paper [2], "where" clauses make explicit the implicit substitution of meta-variables. This change is the biggest syntactic change to the original SASyLF but also completely in line with the purposes of the system: making proofs explicit, especially for pedagogical purposes. SASyLF does not require that these "where" clauses are given, unless a certain option is turned on. (In the IDE, the option is on by default.)

As originally implemented, and as described in the paper cited above, the extension had no effect on the operation of the rest of SASyLF; it didn't need to be part of the "trusted core." The substitutions found were strictly *descriptive*.

This aspect is mainly preserved, but in some situations, especially with inversion, the result of the substitution does not have a user-provided presentation, *except* in the "where" clauses. For example (from `good40.slf` in the public repository):

```
theorem gt-anti-refl-use-w2:
    forall n1
    forall d: n1 > n1
    exists contradiction .
    use induction on n1
    use inversion of rule gt-more on d
      where n1 := s n
```

```
        p: n > n by theorem succ-cancels-gt on d
        proof by induction hypothesis on n, p
    end theorem
```

This proof of the anti-reflexivity of gt is an inductive proof, in that it makes a recursive call. Unusually, it doesn't use an explicit case analysis. Instead it uses inversion on the only rule matching the input. In the process, we need a name for the natural number preceding n1. The "where" clause gives the name "n" for this new variable. This means that the theorem input d has type s n > s n. This judgment is passed to the theorem succ-cancels-gt which can strip the s from both sides of the inequality. Then the result can be used in the inductive call (for which n is a sub-term of n1).

2.9 Generalizing Inversion

The example just shown for "where" clauses also demonstrates one of the generalization of inversion added to SASyLF: inversion can be performed solely for the "side-effect" of the unification that it performs. The new syntax "use inversion" is commonly used on equality judgments. Previously, one needed to perform a case analysis which not only was wordy, but it also increased the depth of the proof and its visual complexity. Now with "use inversion," one can simply invert the equality judgment in place and then assume the equality in the remainder of the proof. This approach is much closer to how one would reason informally.

Internally, the "use" syntax is implemented as proving an empty conjunction, a tautology. This prevents a proof (case) from ending with a "use" line. The same technique is used for the "use induction" syntax explained earlier.

As can also be seen, inversions can have associated "where" clauses ("further work" in the "where" clause paper [2]). An interaction with another extension was the ability to invert "or" judgments (when only one possibility remains, usually because a partial case analysis eliminated the other possibilities). The "obvious" syntax for such inversions is the clunky form:

... by inversion of or on *d*

The sequence of three two-letter keywords, each starting with "o" is visually confusing. More on this problem below.

Separately, there was the need to invert syntax: if a non-terminal has only a single (remaining) possibility, inversion is more attractive than case analysis. While it is not common to define non-terminals with only a single possibility, there are good reasons to do so, for example in our mechanization of LF, we define a constructor identifier as a natural number: "c ::= n." More commonly, with partial case analysis, all but one possibility could have already been handled. Again there's a desire for inversion rather than requiring a case analysis.

An inversion on syntax doesn't use rules, and also doesn't yield any judgments, and so the syntax is simply:

use inversion on *nt* where *nt* := *form*

Once this syntax was added, it was a short step further to permit it also for inversion of judgments, making the identification of the rule optional. In particular, inversion of an "or" judgment can simply by done "by inversion on" the judgment instance, which avoids the unpleasing syntax, which is available in SASyLF 1.5.0 but probably won't be used too much.

2.10 Modules

Requiring all the elements of a proof to be in a single file is limiting. Originally, SASyLF even required there to be a single syntax section, although this restriction was lifted. Nonetheless, it's inconvenient to have to (say) include a definition of natural numbers (and especially all operations and theorems about them) in every proof that uses them. It would also be desirable to have a library of theorems about groups in general that could be applied to any operation satisfying the group axioms. Thus it has been a long-term goal to add some form of modules.

A syntax has been defined in which a SASyLF file can include a module header, including a name, a sequences of requirements (syntax, judgments and lemmas) and sequence of provided declarations. The requirements are often "abstract," that is, not defined. When a module is used, the requirements are substituted with new elements. However, the syntax has run ahead of the meta-theory: SASyLF 1.5.0 does not support using modules with requirements. But it does support the use of modules without requirements and a nascent library has been defined with modules for natural numbers and Boolean values.

An important design goal is that a proof should be syntactically independent of the modules it uses: all syntax and judgments defined for the proof need to be defined locally. So when syntax from a module is used, it must be locally defined as well. This gives an option for an alternate syntax to be used, for example:

```
syntax n = Natural.n ::= 0 | 1+n
```

A similar situation applies to judgments:

```
judgment nat-ne = Natural.notequal: n != n
```

Rules and theorems (and lemmas) can be used with name qualification. In case analysis on an imported judgment, the rule names in the cases do not need to be qualified (and doing so will occasion a warning).

SASyLF 1.5.0 uses a package system (whose syntax was defined but ignored in the original SASyLF) and the `Natural` module is actually in the `org.sasylf.util` package, and so the uses described above are not legal unless one includes a local renaming of the module:

```
module Natural = org.sasylf.util.Natural
```

An advantage of defining a shorthand is that the IDE will use it with content assistance: If one types "`by theorem Natural.ne-`" and presses the content-assist key (e.g., control-space), the system will suggest three possible completions with their theorem headers in a helper popup window.

3 Future Work

In this section, we describe some of ideas for further extension. We cannot commit to any of these plans, but all have been discussed to some extent, and seem compatible with the original goals of SASyLF.

3.1 Support for Equality and Reduction in Theorem Output

Currently, if a theorem is able to show that two terms are the same, the only way to communicate that fact to clients is to use an equality judgment. The clients will then immediately invert the equality judgment. This extra step has no counterpart in a textual proof, even one which carefully lists each step. Rather the definition of equality is assumed. For a while, it seemed that the best way forward would be to pre-define equality for user-defined types and add extra support to handle the introduction and elimination. The

handling of contexts complicates the situation, and also the fact that the judgment may "trespass" on syntax needed for other judgments (e.g., an equality judgment used precisely to hide equality from the system temporarily to avoid the incompleteness of higher-order unification).

The syntax of "where" clauses gives a way for equality to be separated from judgments and given it own space. It seems attractive to permit "where" clauses to declared in the obligation of a theorem, as in the following hypothetical example of a uniqueness of addition theorem:

```
theorem plus-unique:
    forall d3: n1 + n2 = n3
    forall d4: n1 + n2 = n4
    exists () where n3 := n4
    ...
end theorem
```

Then one could write:

```
use theorem plus-unique on p, p' where n := n'
```

Or perhaps even

```
n := n' by theorem plus-unique on p, p'
```

The latter syntax would then also be available to inversion as well.

The ability of a theorem to produce equalities would also make it easier for a theorem to produce other non-judgment results as well, such as "reductions," which allow structural induction. For instance

```
theorem gt-reduces:
    forall d: n1 > n2
    exists () where n1 > n2
    ...
end theorem
```

This theorem would prove that `n2` was a sub-term of `n1`.

Adding non-judgment obligations to theorems would affect the rest of the justification system. It must be possible, then, for case analysis (both full and partial) to produce these non-judgment outputs as well. The "`proof`" syntax would need to include these extra obligations as well. And at the end of a sequence of derivations, we must not only check the last derivation, but also check the current substitution to see if the equality (or reduction) is known. One must also consider whether one can use disjunction with reduction ot equality.

3.2 Wildcards in Premises

The simple ability to give "by unproved" as the justification for any step may seem trivial at first, but it is a big help for people writing proofs. Rather than leaving an error in place, the form of the judgment is given. It can be used and checked at later points. Proof "by unproved" permits proofs to be constructed from the end, rather than forwards. First one determines what needs to be proved. Then one determines what rule (or theorem) would produce the desired result and what premises it would need. These can be defined and justified "by unproved" and then the rule application can be checked.

Two students in a type systems class using SASyLF with different levels of experience with provers independently suggested to extend this ability to apply to partial rule applications. Currently, if anything is wrong with a rule application, the whole judgment is flagged as an error, and since unification is used

to check the premises and conclusion together, it is hard to pinpoint an error to a particular spot. The proposal is to permit a premise to be replaced with _ and then if the application is otherwise error-free, only the missing part would be flagged:

```
da: Gamma |- t1 t2 : T' by rule T-App on _, d2
```

The system may even have a suggestion as to the type of the missing premise(s). This idea seems within the capabilities of the existing engine, where each kind of justification would need to decide whether to support such holes. Presumably case analysis and weakening would not need to support holes, but rule/theorem application and conjunction creation would.[3]

One of the students also suggested permitting such wildcards within a term in case one is not quite sure what is proved, as in the following example:

```
da: Gamma |- t1 t2 : _ by rule T-App on d1, d2
```

Again the idea is that an error (or warning) would be generated with information about the missing information. At an extreme level, one could write:

```
da: _ by rule T-App on d1, d2
```

Here the user is not showing any indication of what the result should be at all. From the first public release, SASyLF already in some cases will infer the actual result and print it in an error message, depending on the error encountered. So this ability seems well within the spirit of the tool.

3.3 Generalization of Contexts

The basic idea for handling contexts has not changed from the beginning: a bound variable is associated with a single syntactic production of the context, and this production is associated with an "assumption" rule of a judgment that assumes the context.

The fact that an identifier is known to be a bound variable is due to it occurring inside square brackets in the grammar (only variables may thus appear), and the nonterminal in which the variable name occurs by itself in a production indicates the syntactic restriction on the variable.

The production of the context nonterminal for a variable has a single occurrence of a variable, a single (recursive) instance of the context nonterminal, optionally some other nonterminal occurrences, and any number of "noise" terminals.

The context nonterminal must have a single production with only terminals, and otherwise can only have variable productions, one for each syntactic variable form.

The assumption rule, unlike all other rules anywhere in the system includes a single production of the context nonterminal in the conclusion. This production must be the production for a variable, and must comprise the most unrestricted form of that production. The assumption rule is permitted no premises. The rest of the rule can use no nonterminals not occurring in the context production, and must have at least one use of each nonterminal occurring as well as exactly one use of the variable.

For instance, in the case of the simply-typed lambda-calculus, one can define:

```
syntax
  t ::= lambda x:T . t[x] // x is a variable
      | t t
      | x        // x is of nonterminal t
```

[3] SASyLF 1.5.1 has implemented this requested feature through the "Quick Fix" system.

```
  G ::= *       // terminals only
     | G, x:T // variable production for x

judgment typing: G |- t : T
assumes G

  ------------- T-Var
  G, x:T |- x:T   // verbatim use of variable production

  ⋮
```

This information is used in the conversion of forms into LF. Each explicit binder is converted into two levels of abstraction in LF. So for example the form

```
G, x1: T -> T' |- ...
```

is converted to

$$\Pi x_1 : \texttt{t}.\Pi d_1 : \texttt{typing}[x_1, \texttt{arr}[T, T']]....$$

The second abstraction's formal (here d_1) is never used in the internal form, and so one can use LF's arrow type short-hand:

$$\Pi x_1 : \texttt{t}.\,\texttt{typing}[x_1, \texttt{arr}[T, T']] \to \ldots$$

From this conversion, it can be seen why we need an assumption rule for the variable production: it is needed for representation of the context. Similarly, the fact that the assumption rule cannot mention other nonterminals (not in the variable production) is clear for the same reason of representation, they would be unbound. Premises for the rule would be irrelevant since the assumption is a hypothetical, not a proof. The fact that the context nonterminal needs a terminals-only production reflects the fact that the empty context has no information. Multiple terminals-only productions would be confusing since they would not be distinguished in the meta-theory. But the remaining restrictions in the context and assumption rules are potential points for extension. Lifting these restrictions would somewhat increase the expressiveness of SASyLF without requiring any substantial extensions in the meta-theory. We now turn to a far-reaching extension.

3.4 Contexts as Syntax

One of the restrictions that was discussed above is that a nonterminal must either be identified as a context or as syntax. In the former case, it must not be an object of "assumes" and it can serve as a binder of variables. In the latter case, any judgment that mentions it must have a declaration that it "assumes" the context nonterminal, and aside from highly restricted "assumption" rules, all uses in conclusions of rules must be unrestricted.

This restriction can be lifted aside from two difficulties, one that could be handled withing the current meta-theory but the other causing a major change. The first, minor difficulty is how to handle variable productions, e.g., "`G, x:T`," because if the structure is not a context, then the variable is not bound, and cannot be used. The easy, but unsatisfying answer us that the variable can be simply ignored; it carries no information. For example suppose we define a judgment on types that don't use named variables: `T ground`. Then with the restriction lifted we could defined a judgment that a context uses only ground types:

```
judgment contextground: G ground

  -------- G-Empty
  * ground

  G ground
  T ground
  --------------- G-Var
  (G, x:T) ground
```

The main difficulty concerns when a single nonterminal is treated as a context and as syntax in the same context. Some situations cannot be permitted: since a judgment that "assumes" a context does not actually receive information about the context, it cannot use a judgment that treats it as syntax:

```
G ground  //! cannot be permitted
G, x:T |- t[x] : T'
---------------------------- T-AbsWrong
G |- lam x:T . t[] : T -> T'
```

Indeed such combinations if permitted would render weakening and exchange unsound in general. The other way around is a situation in which a judgment that does *not* assume the context has a premise which does. This situation contains some of the aspects discussed below, but is less interesting because the context cannot be used by any of the other syntactic members of the judgment.

The truly difficult mixing comes in a theorem which has premises which assume the context and those that do not, for instance:

```
theorem ground-typing-implies-something:
    forall d: G |- t : T
    forall g: G ground
    exists ...
end theorem
```

Since the second premise treats `G` syntactically, the theorem cannot assume that it is represented in the LF context of the theorem. Instead, `G` must be an implicit parameter of the theorem. The other premise is represented using the context as an abstraction, or rather as a series of abstractions depending on the size of `G`. LF does not have a way to represent a variable-length series of abstractions. It cannot be represented by a single abstraction with a compound argument because LF provides no way to extract pieces from compound values. Preliminary work to extend LF with variable arity [4] was never completed.

3.5 Parameterized Modules

With SASyLF 1.5.0, we finally have support for modules, albeit without parameters. The syntax and basic idea of module parameters have already been laid out. The module can "require" parameters, and then another proof can instantiate the module with parameters

More details need working out before it makes sense to embark on designing a new set of library modules using parameters: finding the best way to express something, how can the system be made practical, making sure that the result has a sound meta-theory. For all these reasons, development of modules has been slow.

3.6 Subtyping in Syntax

In Pierce's TAPL, values are routinely given as a sub-grammar of terms. SASyLF does not support this idiom; if the examples from TAPL are typed in, they are syntactically legal, but generate ambiguous parses wherever something could be a value or a term. Updating SASyLF to support sub-grammars was identified as a desirable extension in the first partial solution to the original POPLmark challenge [3].

In this case, as opposed to some other desirable extensions, the meta-theory of "refinement types" for LF is already completed [10]. As part of the formalization, it uses intersection types (or rather "intersection sorts"). Indeed, due to pattern matching, a meta-variable may found to be an element of two sub-grammars (sorts) at once. This is rather awkward in SASyLF because the name of a meta-variable is intrinsically bound up in its type: what do we do with (say) $\mathtt{t}'_2$ when it is found to be a value? We would need to update the system to remember what sorts a nonterminal is (currently) known to inhabit. The current sorts would affect which productions are needed for case analysis, perhaps generalizing the information used in partial case analysis. The implementation of the meta-theory would also require a major update to support sorts.

3.7 Co-induction

Proofs in SASyLF assume that syntax and judgments are finite; that structures are defined through a least fixed-point of grammar production and rule application. Generalizing this assumption is useful for the support of coinduction. Kozen and Silva [9] give a good introduction to how coinduction should be supported in informal proofs. This article should be a good guide for extending SASyLF with coinduction, translating the category theory (e.g., the article defined a coinductive datatype as an element of the final coalgebra of a polynomial endofunctor on **Set**) into practical rules.

The biggest obstacle to supporting coinduction over coinductive datatypes is the need to support circular/infinite structures. SASyLF would need syntax to create them, and the internal engine would need a major rewrite to support these structures, in particular defining unification on coinductive structures with variables. Furthermore, since these structures are typed in LF, it is necessary to extend LF to operate on "co-terms." It is not clear whether LF is sound on co-terms.

4 Related Work

In this section, we review some of the other education proof systems particularly interested in helping students construct their own proofs.

Lurch [5] is one example of a system with a similar mission as SASyLF, except in the field of mathematics. It compares most similarly with SASyLF in aiming to check proofs written in the idioms of paper proofs. It works as a "word processor" similar to using SASyLF in an IDE. It provides the standard tools of a rich text editor and indeed recently adds the ability to edit formulas in a simplified text-based equation system (or even in LaTeX). The particular ability of Lurch is that when parts of the text are selected as "meaningful," Lurch will check them. So if a mathematical statement such as "$\forall x.x^2 \geq 0$" is "meaningful", it will expect this statement to be followed by a proof, which is checked. It uses colored icons to indicate which statements are proved, definitely wrong or simply unproved.

AXolotl [6] is a puzzle game for students to learn how to generate proofs using quantified rules. The researcher (Cerna) recognized that students often struggle with unification and substitution in rule application and includes several aids to help students bridge the conceptual gap. Recently the tool has been ported to work as a mobile application [7].

The Students' Proof Assistant [14] takes a different approach. Rather than a stand-alone tool, it is embedded within Isabella/HOL. Unlike the standard Isabelle proof assistant, it keeps the entire tree of the a proof visible to the student working on a proof. This feature makes it comparable to SASyLF, despite the fact that it still supports tactics (being within the Isabelle system). The researchers agree that it's important for students learning to write proofs to see structure of the proof being created.

People experienced in writing proofs can find writing the simple parts of proofs tedious while people just introduced to proving need to work through these details because everything is new. Paradoxically, the expert needs more assistance than the newcomer (albeit only for productivity); in fact the assistance can be a hindrance to the student learning how to prove.

5 Conclusion

In conclusion, SASyLF 1.5.0 carries on the vision of the original SASyLF paper while firming up the foundation, and providing additional convenience. Some of that convenience (e.g. "quick fixes" in the IDE) are best not highlighted at the beginning of a course introducing SASyLF, so as to ensure that important learning steps are not skipped. We anticipate that SASyLF will continue to develop as long as it remains useful pedagogically.

References

[1] Jonathan Aldrich, Robert J. Simmons & Key Shin (2008): *SASyLF: an educational proof assistant for language theory*. In: *Workshop on Functional and Declarative Programming in Education (FDPE 2008)*, ACM Press, New York, pp. 31–40, doi:10.1145/1411260.1411266.

[2] Michael D. Ariotti & John Tang Boyland (2017): *Making Substitutions Explicit in SASyLF*. In: *Logical Frameworks and Meta-Languages: Theory and Practice*. Available at `https://lfmtp.org/workshops/2017/inc/papers/paper_2_ariotti.pdf`.

[3] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie C. Weirich & Stephan A. Zdancewic (2005): *Mechanized metatheory for the masses: The POPLmark challenge*. In: *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005*, *Lecture Notes in Computer Science* 3603, Springer, Berlin, Heidelberg, New York, pp. 50–65, doi:10.1007/11541868_4.

[4] John Boyland & Tian Zhao (2014): *Variable-Arity Functions: Work in Progress*. Presented at LFMTP 2014 (Vienna Summer of Logic). Available as `http://www.cs.uwm.edu/csfac/boyland/papers/lf+variable.pdf`.

[5] Nathan Carter & Kenneth Monks (2013): *Lurch: A word processor that can grade students' proofs*. In: *Work in Progress at CICM*, *CEUR Workshop Proceedings* 1010. Available at `http://ceur-ws.org/Vol-1010/paper-04.pdf`.

[6] David Cerna (2019): *AXolotl: A Self-study Tool for First-order Logic*. Technical Report, JKU RISC.

[7] David M. Cerna, Rafael P.D. Kiesel & Alexandra Dzhiganskaya (2020): *A Mobile Application for Self-Guided Study of Formal Reasoning*. In Pedro Quaresma, Walther Neuper & João Marcos, editors: Proceedings 8th International Workshop on *Theorem Proving Components for Educational Software,* Natal, Brazil, 25th August 2019, *Electronic Proceedings in Theoretical Computer Science* 313, Open Publishing Association, pp. 35–53, doi:10.4204/EPTCS.313.3.

[8] Arthur Charguéraud (2011): *The Locally Nameless Representation*. *Journal of Automated Reasoning*, pp. 1–46, doi:10.1007/s10817-011-9225-2.

[9] Dexter Kozen & Alexandra Silva (2017): *Practical coinduction*. *Mathematical Structures in Computer Science* 27(7), p. 1132–1152, doi:10.1017/S0960129515000493.

[10] William Lovas & Frank Pfenning (2008): *A Bidirectional Refinement Type System for LF*. *Electronic Notes in Theoretical Computer Science* 196, p. 113–128, doi:10.1016/j.entcs.2007.09.021.

[11] Frank Pfenning & Conal Elliott (1988): *Higher-order abstract syntax*. In: *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, ACM Press, New York, pp. 199–208, doi:10.1145/53990.54010.

[12] Frank Pfenning & Carsten Schürmann (2002): *Twelf User's Guide, Version 1.4*. Available at `http://www.cs.cmu.edu/~twelf`.

[13] Benjamin C. Pierce (2002): *Types and Programming Languages*. The MIT Press, Cambridge, Massachussetts, USA and London, England.

[14] Anders Schlichtkrull, Jørgen Villadsen & Andreas Halkjær From (2019): *Students' Proof Assistant (SPA)*. In Pedro Quaresma & Walther Neuper, editors: Proceedings 7th International Workshop on *Theorem proving components for Educational software*, Oxford, United Kingdom, 18 July 2018, *Electronic Proceedings in Theoretical Computer Science* 290, Open Publishing Association, pp. 1–13, doi:10.4204/EPTCS.290.1.

[15] Masaru Tomita, editor (1991): *Generalized LR Parsing*. Springer, US.

[16] Yaoda Zhou, Bruno C. d. S. Oliveira & Jinxu Zhao (2020): *Revisiting Iso-Recursive Subtyping*. *Proceedings of ACM Programming Languages* 4(OOPSLA), doi:10.1145/3428291.