SASyLF: An Educational Proof Assistant for Language Theory

Jonathan Aldrich Robert J. Simmons
School of Computer Science,
Carnegie Mellon University, Pittsburgh, PA, USA
{aldrich, rjsimmon}@cs.cmu.edu

Key Shin
Microsoft Corporation,
Redmond, WA, USA
kshin@microsoft.com

Abstract

Teaching and learning formal programming language theory is hard, in part because it's easy to make mistakes and hard to find them. Proof assistants can help check proofs, but their learning curve is too steep to use in most classes, and is a barrier to researchers too.

In this paper we present SASyLF, an LF-based proof assistant specialized to checking theorems about programming languages and logics. SASyLF has a simple design philosophy: language and logic syntax, semantics, and meta-theory should be written as closely as possible to the way it is done on paper. We describe how we designed the SASyLF syntax to be accessible to students learning type theory, and how students can understand its semantics directly in terms of the theory they are taught in class. SASyLF can express proofs typical of an introductory graduate type theory course. SASyLF proofs are generally very explicit, but its built-in support for variable binding provides substitution properties for free and avoids awkward variable encodings. We describe preliminary experience teaching with SASyLF.

Categories and Subject Descriptors F.4.1 [*Mathematical Logic*]: Mechanical theorem proving

General Terms Design, Documentation, Human Factors, Languages, Theory, Verification

1. Introduction

Teaching and doing research in formal language theory is hard. By formal language theory, in this context, we mean formalizing a language through operational semantics and typing rules, or formalizing a logic as a set of inference rules, and then proving meta-theorems such as type soundness (for languages) or admissibility of cut (for logics).

The difficulties with teaching and learning formal language theory are closely related to difficulties inherent in doing research in the field: it can be extremely tedious to state things formally and precisely without making small syntactic errors, but these minor errors can conceal significant flaws. There has recently been a great deal of energy directed towards the promotion of computer-verified proof as a means to cope with the minutiae inherent in much of formal language theory, for instance (Aydemir et al. 2005, 2008).

We believe a similar effort is due in the practice of teaching formal language theory, which raises an additional set of challenges. To learn effectively, students must focus on higher-level concepts like induction, yet many mistakes are made at a much lower level, e.g. skipping a step in a proof or applying an inference rule when the facts used do not match the rule's premises. Students may not even recognize they have made a mistake, and so do not seek out help. They only learn of their mistake a week or two later, when the TA hands back a paper splashed with red ink. At that point, the student may have forgotten why the mistake was made, and the learning opportunity is lost. A tool that could provide *immediate feedback* would help students get it right in the first place, and also help them to realize when they need to ask an instructor for help with the more challenging concepts.

Proof assistants like Isabelle/HOL (Nipkow et al. 2002), Coq (Bertot and Castran 2004), and Twelf (Pfenning and Schürmann 1999) have been used to formalize language semantics and prove meta-theorems. However, even in the research community, mechanically checked proofs are the exception rather than the rule. This may be partly a productivity issue, but the steep learning curve of these tools, and the non-trivial techniques for encoding program semantics in them, likely play a role. The Ott tool (Sewell et al. 2007) allows users to write down language syntax and semantics in a convenient paper-like notation, but does not support expressing or proving theorems—this must be done in a separate tool, and users must pay the cost of learning it. Unfortunately, due to learning curve issues, the use of these assistants in teaching formal language theory is rare, despite the help they could in theory be to students.

In this paper, we present the SASyLF ("Sassy Elf") theorem proving assistant. SASyLF has a simple design philosophy: language and logic syntax, semantics, and meta-theory should be written as closely as possible to the way it is done on paper. Proofs are very explicit, for the benefit of teaching. Error messages are given in terms of the source proof, not in terms of the assistant's underlying theory. Finally, SASyLF is specialized for reasoning about languages, programs, and logics—more generally, any system with variable binding. Variable binding is a persistent source of complication for proving language meta-theorems in most theorem proving systems because it must be encoded in some way, presenting yet another barrier to using proof assistants in coursework.

FDPE'08, September 21, 2008, Victoria, BC, Canada.

The logical framework LF (Harper et al. 1993) avoids the need to encode variable binding by building variables into the formal system, allowing languages to be formalized using higher-order abstract syntax. SASyLF is based on LF type theory and also builds in the notion of variable binding, but uses a familiar surface syntax and (partly as a consequence) is restricted to second-order abstract syntax for usability. The name reflects that it is a Second-order Abstract Syntax Logical Framework. As a result of SASyLF's design, proofs are clean, simple, and look almost exactly as they do on paper.

Contributions. The contributions of this paper include:

- We describe a concrete notation for expressing language syntax, semantics, and meta-theory. Our notation for syntax and semantics, though developed independently, is close to that of Ott, presumably because both systems mimic paper definitions. However, we provide a more direct notation for the scope of variable binding and avoid potentially confusing artifacts related to theorem provers. Our meta-theory notation is new but is based closely on standard paper notations, such as are used in courses at our university and others.
- We describe how our proof assistant can be used to express and verify proofs by induction over the structure of derivations. We describe the semantics of SASyLF constructs from the perspective of a student user of the system; internally, SASyLF's semantics rely on previous foundational work in the Twelf proof assistant (Schürmann 2000).
- We describe preliminary experience suggesting that the tool is usable and useful in educational practice, and can express proofs typical of an introductory graduate type theory course.

Outline. In the next section, we explain the rationale behind the design of SASyLF. Section 3 describes the SASyLF proof assistant from a user's point of view. Section 4 discusses our implementation, and section 5 covers our experience with the tool and describes the result of a limited controlled experiment using SASyLF in an educational context. Finally, Section 6 discusses related work.

2. Design Goals

In order to be most useful in an educational context, SASyLF was designed to meet the following goals and design characteristics:

Low Adoption Barrier. Few instructors are willing to spend much time in class introducing a tool, even a very useful one. Thus, to be adopted, SASyLF must be designed so that students can pick it up on their own, or with at most a recitation section spent on the tool.

Familiar Syntax. To facilitate a low adoption barrier, SASyLF's syntax is designed to be as close as possible to the notation often used in programming language theory. The Ott tool took the same approach for describing syntax and inference rules, but perhaps even more important in our context is that proofs are done in the same consistent syntax. While Ott provides significant value, students who will be writing metatheoretic proofs will still have to pay the cost of learning the syntax of another tool.

Familiar Mathematical Context. SASyLF's design is intended to minimize dependency on and mathematical context other than that covered in a programming language course. SASyLF, for example, builds in no set theory; its entire formalism is based on inference rules, induction over these, and variable binding, all of which are taught in programming language theory courses. While set theory is often covered anyway in these courses, by not relying on it we retain a much simpler syntax and tool interface.

Support for variable binding. In most proof assistants, variable binding is supported through one of two encodings that substantially complicate the metatheory compared to what is done on paper. One option is encoding variables using de Bruijn indices—natural numbers that indicate how many binding sites lie between the variable's definition and the current use point—but this encoding is unintuitive and requires cumbersome numerical manipulations. Another option is using atomic names for variables, which is more natural, but which introduces the need to test alpha-equivalence all over the place—an idea which is built into paper proofs.

In contrast, like Twelf, SASyLF builds in the concept of variable binding, avoiding awkward encodings, and ensuring that alpha-equivalent terms are identical. Instructors who want to study encodings for pedagogical purposes can still do so in SASyLF, simply by not using the built-in bindings, but after this study is complete students can move forward with the more natural variable binding support.

Simple, Explicit Tool Interface. Many proof assistants support sophisticated and customizable proof tactics that provide expert users enormous leverage in writing proofs—but this design comes at the cost of forcing every user of the tool to learn the unfamiliar tactic concepts and how to use them.

In contrast, SASyLF has a very simple tool interface—run the tool on a file and get an "OK" or a list of errors. There are no tactics, only low-level steps like "apply a rule" or "perform case analysis" that are already taught in programming language theory courses. The proof notation is therefore very detailed and explicit, which increases the overhead of the tool, but also contributes to to student learning by ensuring they think about and write down each step of the proof.

Simplicity vs. productivity. The potential cost of a simple interface is tedious detail for advanced users. SASyLF is focused on novice users, so this is less of a drawback than it might be, but it is still an issue for students proving more complicated theorems. We could add simple tactics like automated proof search, but we hesitate to do so because novices could use it as a shortcut and thereby avoid learning the basic details of proofs. Instead, 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.

Incremental proof development. In order to achieve rapid feedback, students need to check their work after writing a small portion of a proof or making small changes. SASyLF allows students to omit any part of a proof by justifying a fact as "unproved." The system will check the rest of the proof as if that fact had been proved. A warning is given to ensure that students don't forget to complete the proof later.

```
terminals fn unit value

syntax

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

tau ::= unit
    |   tau -> tau

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

Figure 1. Syntactic definitions for the λ-calculus in SASyLF

Local checking and localized errors. In Twelf, a derivation is built up functionally by applying one rule directly to the result of another. SASyLF essentially converts a Twelf-style proof into let-normal form, explicitly binding each intermediate fact established by the student. This results in a more verbose proofs, but also makes it very easy to localize error messages. Instead of a unification failure that could be due to a mistake anywhere in a derivation tree, SASyLF can localize the error message to an individual rule application, and can then potentially give very good error messages.

3. The SASyLF Proof Assistant

In this section we describe the syntax and informal semantics of the SASyLF meta-logic, using the simply typed λ-calculus as an illustrative example.

3.1 Syntax

Figure 1 shows the header and syntax declarations for the simply-typed lambda calculus in SASyLF. The **`terminals`** declaration declares identifiers that are used as terminals in the target language grammar. This information is not strictly necessary, as we could infer terminals, but declaring them explicitly helps us detect errors like misspellings better for students.

The **`syntax`** block declares a grammar for all the syntactic constructs of the language and accompanying theoretical constructs like typechecking contexts Γ. The grammar is given in a conventional BNF form. On the left hand side of each production is the "name" that will identify that syntactic form. In Figure 1, for example, expressions are given the identifying name `e` and types are given the identifying name `tau`. We use the notation `e[x]` to denote that `x` is a variable that is bound in `e`. This is a notation that may be more familiar from logic, where the formula $\forall x.(B \wedge A[x])$ represents that the bound variable x is bound in A but not in B. Whenever there is such a binding form, the variable should be mentioned elsewhere in the production (the `x` in `fn x : tau ...`), and that is inferred to be the binding occurrence of the variable. SASyLF observes that `x` is one of the cases in the grammar for `e` and thus concludes that `x` is an expression variable, that is to say a variable in the syntactic class `e`, and therefore that the construct `fn x : tau => e[x]` contains a subexpression with a bound expression variable.

Parentheses are special in SASyLF—they are used to disambiguate the way that expressions should be parsed—so to use the ML notation for the unit expression in the target language we must quote the parentheses in SASyLF, indicating that they should be treated as terminals. Other symbols, like `:`, `=`, and `>` are automatically assumed to be terminals when they occur.

We define types `tau` and contexts `Gamma` exactly as one might do it in a paper. We will see later that the form of `Gamma` is specially chosen to enable it to be treated as the LF context in the underlying theory of the tool.

```
judgment value: e value

------------- val-unit
"(" ")" value

------------------------- val-fn
fn x1 : tau => e1[x1] value


judgment step: e -> e

e1 -> e1'
--------------- c-app-l
e1 e2 -> e1' e2

e1 value
e2 -> e2'
------------------ c-app-r
(e1 e2) -> (e1 e2')

e2 value
------------------------------- r-app
(fn x : tau => e[x]) e2 -> e[e2]
```

Figure 2. Operational semantics for the λ-calculus in SASyLF

3.2 Operational Semantics

The operational semantics of the λ-calculus are shown in Figure 2. We first define a judgment for values, giving the judgment a name (`value`) followed by a syntactic form (`e value`). Then inference rules defining the judgment are given. Each rule is a series of premises, one per line, followed by a horizontal line, the name of the rule, and the conclusion. SASyLF checks that each of the premises and conclusion can parse as one of the judgment forms in the system (perhaps one that is yet to be defined—judgments may be recursive). We want SASyLF to deal with grammars that may be ambiguous, since we don't want parsing knowledge to be a prerequisite for students to use SASyLF. Therefore we use a GLR parsing algorithm (Tomita 1987) that can parse strings against an ambiguous grammar as long as that particular string's parse tree is unambiguous.[1]

Rules are interpreted schematically. In the rules in Figure 2, the instances of `e1`, `e2`, `e1'`, or `e2'` are treated as schematic variables (or metavariables) that can stand for any expression, that is to say any inhabitant of the syntactic class `e`.

The rules for single-step evaluation in Figure 2 show how premises are declared, as well as how parentheses may be used to clarify how a string should be parsed. The beta reduction rule `r-app` also illustrates how substitution is ex-

[1] Precedence declarations could be supported in future work, although they may be confusing to students who have not taken a course that covers parsing.

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

---------------------- t-unit
Gamma |- "("")" : unit

----------------------- t-var
Gamma, x:tau |- x : tau

Gamma, x1:tau |- e[x1] : tau'
---------------------------------------- t-fn
Gamma |- fn x : tau => e[x] : tau -> tau'

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

Figure 3. Typing rules for the λ-calculus in SASyLF

pressed. In one part of the rule, we see that `x` is bound in `e`. On the right hand side of the reduction, we substitute `e2` for all occurrences of `x` in `e`. This corresponds again using the convention for substitution in logic.[2] Since variable binding is built into SASyLF, the semantics of substitution are capture-avoiding by definition.

3.3 Typing Rules

The typing rules for the simply-typed λ-calculus are shown in Figure 3. The `has-type` judgment is declared just like previous judgments, except that the **`assumes`** declaration tells SASyLF that `Gamma` is not merely a syntactic form, but is intended to represent a context with typing assumptions. As discussed later, the tool checks that `Gamma` is indeed used as a proper typing context, and in consequence, provides basic properties like substitution, exchange, and weakening for free.

Rules `t-unit` and `t-var` are standard. Rule `t-fn` shows that bound variable names like `x` are not significant; we have renamed `x` as `x1` in the premise for illustrative purposes. However, the name of a variable or metavariable determines what syntactic category that variable or metavariable belongs to. A metavariable consists of the identifier for its syntactic form (in the case of expressions, this identifier is "`e`") with an optional suffix consisting of any number followed by zero or more primes. In the typing rules in Figure 3, we have have `e` and `e1` as metavariables for expressions (members of the syntactic class `e`), while the metavariables `tau` and `tau'` are type metavariables (members of the syntactic class `tau`). In the same way, variables like `x` must be based on variable names used in the syntax, and they are given types that reflect the syntactic categories they are a part of (e.g. `x` is an expression variable). In the future we plan to support namespaces to allow a file to rely on declarations from another file without worrying about name clashes.

[2] We could use the substitution syntax typical of programming languages: $\{e2/x\}e$, but it is slightly more syntactically cumbersome and the logic syntax has the advantage of making binding explicit as well as substitution. The representation `e [e2]` also has the benefit of being closer to the underlying LF theory that performs substitution as application.

3.4 Rules for Variable Binding and Contexts

Like Twelf, SASyLF builds in the concept of variable binding and hypothetical judgments. If constructs such as variable binding (e.g. `e[x]`) and hypothetical judgments (e.g. **`assumes`** `Gamma` above) are used, they must follow well-formedness rules (discussed below). Following these rules gives us the following standard properties of hypothetical judgments, taken from (Harper 2008):

- **Reflexivity.** Every judgment is a consequence of itself: $\Gamma, J \vdash J$. This is the definition of a hypothetical judgment.
- **Weakening.** If $\Gamma \vdash J$ then $\Gamma, J_1 \vdash J$. Additional assumptions cannot interfere with an existing derivation.
- **Limited exchange.** If $\Gamma, J_1, J_2 \vdash J$ and J_2 does not use variables bound in J_1, then $\Gamma, J_2, J_1 \vdash J$. That is, ordering of hypotheses is immaterial, as long as variable binding requirements are respected.
- **Contraction.** If $\Gamma, J_1, J_1 \vdash J$ then $\Gamma, J_1 \vdash J$. Since we can use a hypothesis multiple times, it does not need to be repeated.
- **Substitution.** If $\Gamma, J_1 \vdash J$ and $\Gamma \vdash J_1$ then $\Gamma \vdash J$. Here we take the derivation of J_1 and substitute it in for any uses of the assumption in the derivation of J.

Properties like weakening and substitution are very common lemmas in programming language proofs, and so it is nice to get them for free. As with Twelf, support for these properties biases the proof assistant towards encoding logics and languages that admit these properties—encoding systems like linear logic that do not have weakening is still possible but may not benefit as much from SASyLF's built-in binding support.

To make our reasoning sound, however, the rules of the system must be structured in a way that justifies these properties. We enforce a standard set of rules for well-formed hypothetical judgments, also taken from (Harper 2008):

- All variables used in a judgment must be bound, either in a surrounding binding form within that judgment, or else within some hypothesis in the context Γ.
- A context must represent a list of judgments. We enforce this by requiring the form of any context declared in an **`assumes`** clause (typically named `Gamma` but the name is not significant) to have a base case that defines neither a variable nor a judgment, and a set of recursive cases that use `Gamma` in exactly one location and bind exactly one variable. There may be more than one such case, for example in the polymorphic lambda calculus `Gamma` typically includes cases `Gamma, x:tau` and `Gamma, t:type`.

  For each recursive case, there must be a rule similar to `t-var` above that shows what judgment the case corresponds to. These rules must have Gamma unrolled once (with the relevant recursive case), and must use the variable bound in the recursive case in the rest of the judgment. Semantically, the `t-var` rule allows us to interpret that `x:tau` in `Gamma` *has the semantic meaning* that whatever expression `e` is later bound to the variable `x`, there must be a proof of `Gamma |- e : tau`. This will justify that our `has-type` judgment is preserved when substituting `e` for `x`.
- A hypothetical judgment in the premise of an inference rule may only use a context Γ' which is an extension of the

```
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
```

Figure 4. Preservation proof for the λ-calculus in SASyLF

context Γ in the conclusion (i.e. it must have all the same assumptions but may have some more as well). As an exception, we allow premises that have no context in cases where an argument based on subordination (discussed later) can show that no variables in Γ could possibly be used in that premise.

3.5 Theorems and Proofs

Figure 4 shows the proof of type preservation for the simply-typed λ-calculus in SASyLF (one case is elided). SASyLF supports theorems of the form "for all ≪*list of metavariables and judgments*≫ there exists ≪*judgment*≫." SASyLF shares this limitation with Twelf,[3] and while it limits what SASyLF can prove, experience with Twelf shows that this form of theorem is still useful for a lot of programming language theory, and it corresponds to proofs that are naturally expressed by induction and case analysis on derivations.

Syntactically, one must give a name for the theorem and for the derivation of each input judgment. The derivation names (`dt` and `ds`) are used to refer to judgments within the proof.

A proof is a list of judgments, each with a justification. The preservation theorem uses induction, induction hypothesis, case analysis, application of inference rules, and substitution as justifications.

The preservation proof begins by stating the judgment we want to prove, `* |- e' : tau`, giving it the name `dt'`, and stating that it is justified by induction over the derivation of the evaluation judgment `ds`.

We immediately do a case analysis on the rules used to derive this judgment. Each case in the case analysis is introduced with one of the rules that could be used to generate `ds`. The rule is stated using fresh metavariables that are bound in the body of the case (`e1`, `e2`, and `e1'` in the case of `c-app-l`). SASyLF matches the conclusion of the rule to the judgment we are case-analyzing, and determines that `e` has been substituted with `e1 e2` and that `e'` has been substituted with `e1' e2`.

We proceed to further case analyze on the typing derivation `dt`. Since we know that `e=e1 e2` there is only one possible case, rule `t-app`. SASyLF will try all the other rules but will discover that their conclusions don't match the form `e1 e2`; if any of them matched, SASyLF would report an error stating which rule needs to be added to the case analysis.

In these two nested cases, we have learned that `e1 -> e1'` and `* |- e1 : tau' -> tau`. We therefore can apply the induction hypothesis, naming the two facts just mentioned with their names `d1` and `d3`. SASyLF checks that the derivations used to instantiate the "forall" clauses of the theorem in fact match those clauses, and checks that the resulting derivation `d6 : * |- e1' : tau' -> tau` is in fact what you get from applying the theorem to those inputs. SASyLF also verifies that the derivation passed in for the argument of the theorem we are doing induction over, `e1 -> e1'`, is a subderivation of the thing we analyzed by induction, `e -> e'`.

Finally, SASyLF checks that the last step in the proof of each case (and of the main theorem) is a statement of the thing we are trying to prove, namely `* |- e' : tau` (where in this case `e'=e1' e2`). We get this by applying rule `t-app` to the derivations we got from the second case analysis and the induction hypothesis. SASyLF performs checks similar to those for the induction hypothesis, except of course that the subderivation check is not relevant.

The case for evaluating the argument of an application is similar, so it isn't shown in Figure 4. However, the beta rule case is interesting because it involves an application of substitution. After case analyzing on the application rule, we further case analyse on the typing rule (which must be `t-app` again, though with the arguments in a different form) and then on the typing rule for the function.

An interesting point about how SASyLF checks the proof is illustrated by the fact that when case analyzing rule `t-app` we know that the function type is `tau' -> tau`, but we do not know until the later case analysis that the argument `x` of the function has type `tau'`. All we know is that it is some type `tau''`. The case analysis of the function typing with rule `t-fn` proves that `tau''=tau'`, since otherwise the rule could not be applied.

[3] Twelf allows multiple judgments in the exists clause, something SASyLF can encode and which we plan to support in the future.

Now, with derivation `d7` we have learned that `*, x:tau' |- e1[x] : tau`, and we know from `d5` that `* |- e2 : tau'`. From the properties of hypothetical judgments we know that if $\Gamma, J_1 \vdash J$ and $\Gamma \vdash J_1$ then $\Gamma \vdash J$ (Harper 2008). We appeal to this property with the claim that

`* |- e1[e2] : tau` **`by substitution on`** `d7, d5`

SASyLF checks that `d5` matches the judgment represented by the hypothesis `x:tau'`, which is given by rule `t-var`.

In addition to the proof justifications shown here, the tool supports others including use of a lemma, weakening, exchange, contraction, assumption/previous (when we just need to cite an input or previous derivation), and "unproved" (when we want to postpone proving one branch of a derivation but want to verify the rest of the proof anyway). 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.

4. Implementation

An open-source implementation of the SASyLF proof assistant is available at:

`http://www.sasylf.org/`

The core of the implementation is complete, including the checks for rule application, case analysis, and case coverage. The system passes all the examples in this paper, plus the examples in the distribution (discussed below). However, there are a few checks that are not yet implemented, including the checks for substitution, weakening, exchange, and contraction.

We initially began implementing SASyLF in Standard ML, because Twelf is written in Standard ML and we hoped to translate SASyLF input into Twelf and use the existing proof checker. However, while fleshing out this strategy we discovered that using Twelf directly would make it difficult to achieve our goal of giving good, localized error messages to the student user. SASyLF proofs are written in a functional programming style, but Twelf proofs are in a logic programming style. SASyLF can give good error messages in part because proofs are in a kind of let-normal form, with each intermediate step in a derivation named explicitly and separately checkable for errors. Twelf, as a logic programming language, has no "let" construct, and would force us to compile away these lets, making it difficult to reconstruct which step in the proof led to a Twelf error. Furthermore, SASyLF supports nested, one-level-at-a-time case analysis, while Twelf supports case analysis many levels deep but only at the top level of a logic definition, meaning that significant transformation would be necessary and error messages related to missing cases would be difficult to patch back into SASyLF.

To get good error messages, we therefore decided to reimplement the Twelf checks—which while technically complex, are not large in size due to the few constructs in the LF theory. We chose the Java programming language because of the good libraries and frameworks available, most notably for IDEs (Eclipse in this case) and parser generators (javacc), which are unavailable or of lower quality in Standard ML. The choice of Java made certain aspects of the implementation painful, especially the unification algorithm, which suffers greatly from the lack of pattern matching in Java. However, unification is only 1700 of 13000 lines of code in the system, and there were some advantages to Java even beyond the good libraries and the availability of Java programmers to contribute to the project. For example, elements of the AST could implement multiple interfaces simultaneously because of Java's subtyping support, which helped avoid duplicated code in many cases. Overall, our reimplementation certainly came at a cost, but it also avoided significant challenges in translating error messages between two systems, saved substantial parsing effort due to use of a good parser generator, and will make integration with an eventual IDE much easier.

Despite this reimplementation, our checker translates SASyLF input into LF type theory, and then proceeds essentially as Twelf does, recasting induction over the structure of deriviations as induction over canonical forms of LF. The internal format of theorems in the SASyLF prover represents syntax and judgments in LF just as Twelf does. Proofs are represented differently, however, essentially as functions in let-normal form that accept input derivations, perform case analysis as necessary, and produce output derivations, possibly by making recursive calls (uses of the induction hypothesis). Our internal representation is very close to the functional M_2^+ meta-logic described in Schürmann's thesis on Twelf (Schürmann 2000), except that his representation lacks a let form while our representation is in let-normal form. We thus follow M_2^+ closely in our checker, benefiting from the existing proof of soundness for M_2^+.

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.

5. Evaluation

5.1 Case Studies

The distribution includes several SASyLF examples. The first, in lambda.slf, is a formalization of the simply-typed λ-calculus, with progress and preservation theorems, as presented in the running example for this paper. Although substitution is built into SASyLF, we also prove an explicit substitution theorem in the normal inductive way, to show how it can be done. Substitution is also interesting because we are verifying a property in a non-empty context, unlike progress and preservation.

Two files provide formalizations of Hoare's WHILE language using a semantics with an explicit environment (which actually does not take advantage of SASyLF's variable binding support). The file while1.slf contains a simple derivation showing how a WHILE program executes in a big-step semantics, while while2.slf proves a Factorial function correct with respect to the semantics. In both cases, we assume an oracle for the arithmetic, using "unproved" for all mathematical judgments.

The file lambda-loc.slf contains the untyped lambda calculus with locations added and a store well-formedness rule. This example is interesting because preservation of store well-formedness requires an explicit substitution lemma, as well as a strengthening lemma that shows that expressions in the store cannot depend on variables bound in the context.

We also include sum.slf, a axiomatization of addition and a proof that addition is commutative. All these proofs check without errors, and without warnings except for the warnings about the unformalized arithmetic in the WHILE proofs.

Finally, we have developed a candidate solution to Part 2A of the POPLmark challenge, suggesting that SASyLF can

prove many theorems covered even in more advanced graduate courses. We believe Part 2B 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). Part 3 would require automated search for derivations, which is potentially in conflict with our educational goals of explicit proofs without tactics, but may be useful educationally in that it allows students to easily experiment with their operational semantics.

5.2 Controlled Experiment

We performed a limited, controlled experiment to determine whether SASyLF can aid students in learning about formal modeling and proofs about programs. The setting of the study was the first author's Analysis of Software Artifacts class, which teaches both formal (e.g. proofs) and informal (e.g. testing) approaches to analyzing software. This setting was not an ideal test of SASyLF, because formal language theory makes up only a small part of the course (one assignment), and because one of the most important features of SASyLF, variable binding, was not exercised in any significant way. The experiment was optional in the course because the tool was immature, meaning we had fewer participants than would be ideal. Finally, we were only able to compare SASyLF against a control of no tool use; since this is the most common case today, we decided this was a more appropriate control than comparing to other tools. Despite these limitations, we believe the study has the potential to shed light on the value of SASyLF.

Methodology. We recruited 34 volunteers for the study from among class members. Of these, 17 were assigned to use the tool for their assignment, and 17 were assigned to do the assignment on paper (acting as a control group). The assignment had three parts. First, students were to show a derivation of how simple programs in Hoare's WHILE language execute using big-step semantics. Second, students were to prove inductively, based on the same semantics, that a program that multiplies through repeated addition is correct. Third, students were to use Hoare Logic to prove that the same multiplication program meets its pre- and post-condition specification. While these examples illustrated the ability of SASyLF to check students' work, they made little use of variable binding, a concept that was not a focus in this course (the Hoare Logic example does use variable binding but it is treated as a black box).

Students in both cases did the same problems, and had the same access to examples and expert help. The tool group received a starting file with the proper judgments formalized and some worked examples; the tool was able to verify the correctness of their derivations (but used "unproved" for arithmetic judgments). The control group received PDF and LaTeX sources for the same judgments and worked examples. The tool group received no specific training in the tool other than annotated example files, although short demonstrations were requested and given to many students at office hours.

The outcomes we measured included performance on related midterm exam questions; subjective confidence in formal ability before and after the assignment; and qualitative impressions about the usability and benefits of the tool.

5.3 Experimental Outcomes: Qualitative

We experienced a relatively high dropout rate for students in the tool group. Of the 17 students in this group, 9 used the tool all the way through the study, and 3 additional students turned in at least part of the assignment in the tool notation. The other 5 students dropped use of the tool entirely. Three students gave a reason for dropping use of the tool: two said it took too much time, the third said it was too difficult to gradually draft a proof using the tool and paper made this process easier. Other general issues with the tool raised included usability problems, challenges getting program literals to parse (perhaps an issue with our GLR parsing strategy), and a high learning curve for the tool.

Of the 12 tool users who completed their post surveys, 7 would like to use the tool again on a similar assignment, 1 did not answer, and 4 would prefer not to use the tool on a future assignment. Of the 4 who did not want to use the tool again, 3 would consider it if the usability problems they saw were fixed.

The above results suggest that there are significant usability issues with the tool, many of which involved the challenge of learning the tool's syntax. Despite this, a majority of participants were successful at using the tool, and most students either preferred using the tool for assignments or would consider it if these issues were resolved.

We asked members of the control group whether they encountered situations where they were unsure whether their proof was correct, and if there were situations where they wished they had earlier feedback on errors in their proof. 14 of the 16 control group members who submitted post surveys cited specific examples of each of these situations. A number of students mentioned small mistakes or typographical errors as issues they were worried about. One student also said that "errors...discovered after completing [one draft of the assignment] caused heavy rework."

Although not all of these situations would necessarily be remedied by a tool, these responses do suggest that there is substantial room to help students by confirming the correctness of a proof or providing earlier feedback on errors.

In the tool post-survey, we asked if students felt the tool increased or decreased their confidence that their proofs were correct; 12 of 13 students said it was increased, the other student said there was no effect. We asked if the tool helped or hindered in finding errors in proofs; 11 of 13 students said the tool helped, and 2 said there was no effect. We asked if the tool directly helped students to learn concepts in the course, or if it was a barrier; 6 students said it helped, 5 students said there was no effect, and 2 students said the tool was a barrier.

Overall, our qualitative results suggest that, despite significant usability problems with the beta version of the tool, a number of students found the tool helpful and would use it again.

5.4 Student Experiences.

Paper and Tool Use. Many students used a hybrid strategy where they would prototype their proofs on paper and then formalize them in the tool. One student said "On trickier proofs I had to go to paper to clearly state what I was trying to prove.... [When using the checker] instead of thinking about how to make a beautiful statement that reveals something true, I was just concerned with jamming the nuts and bolts together until the checker was happy." These comments suggest there is a need for a more natural proof interface, and perhaps the ability to do proofs at a higher level of ab-

straction. Some students also did not understand how to use "unproved" to check partial proofs, and therefore felt their proofs had to be perfect before running the tool—pointing to the need for at least minimal training or suggestions for using the tool. At the same time, other users found that the tool added significant value. One said, "I actually did the entire assignment on paper first and then moved over to using the tool. I found the paper approach really easy. But once I started using the tool I started understanding the concepts better."

Readability of proofs in the tool was an issue. One student said "The tool found problems with my proof. However, the proof in the tool was less readable, so I am less confident that my manual inspection found bugs the checker missed." (our prototype checker still had a few missing checks) 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.

One student noted a trade-off between the confidence provided by the tool and the additional time it took to use the tool: "I am more confident that I applied the concepts correctly. However, I think using the tool was more time consuming. I think I might have learned more if given more complex examples to work out on paper." Providing a graphical interface for manipulating proofs was suggested by several students, and such an interface might reduce the time overhead of using the tool.

Tool Feedback. Several students felt they benefited from earlier feedback from the tool: "The tool provided earlier feedback so I did not face situation where I wished I had feedback earlier." Most tool students still did have times when they wished they had better feedback, for example real-time feedback 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.

Most students also still had some questions that were unanswered by the tool, because the tool messages indicate that there is a problem but are not always able to point to why the problem exists. One student said, "It helped in finding errors, but often the errors it found were because of typos and cut-and-paste mistakes, which I wouldn't have if I had done it on paper." On the other hand, other students were satisfied with the tool, one saying "Most questions I had could be solve by the output from the tool."

One area for improvement would be positive feedback from the tool that judgments are correct. One student said "A graphical interface that showed which judgments were definitely OK would assure me."

Effect on Student Thoughts. Use of the tool did seem to have some effect on how students thought about problems, helping them to make important distinctions in the proof. One student said, "I ran into a lot of situations where I was thinking, 'gee, now do I need to prove 1+1=2, or rather than i+1=2 when i=1?"' While this student was expressing some frustration, the statement shows the student was thinking about a distinction that many students simply gloss over on paper, sometimes resulting in incorrect proofs. Other students felt that the tool gave them the ability to think at a higher level: "With more confidence in the logical flow of my proof, I could spend more time on the larger picture."

Summary. Overall, the comments provided by students provide insight into a number of ways in which the tool contributed to their learning process. We also received a number of concrete suggestions for improving the tool, which we intend to implement in future work.

5.5 Experimental Outcomes: Quantitative

We collected several quantitative measures in the experiment. The effect of the tool, if present, was too small to be statistically significant with our sample size. Without claiming it is conclusive, we nevertheless report the data we found, recognizing that at best it suggests hypotheses to be verified in more comprehensive, future studies.

The midterm exam had a question asking students to produce a derivation of execution in the WHILE language, and a question asking students to derive intermediate assertions in a Hoare Logic proof of the correctness of a WHILE program. Both tasks were similar to what students did on the homework. The 9 students who had successfully completed the entire assignment with the tool did an average of one point better on these questions, than the 17 students in the control group (statistically insignificant, p-value 0.26).

It is possible that only the best students completed the assignment with the tool. However, the 1 point difference persists even if we compare a subset of the control group that is matched in terms of academic program and grade in a previous formal course, to the students in the tool group.

We also measured confidence with formal mathematics and proofs, on a scale of 1-5 from very unconfident to very confident. We measured the difference in confidence before and after the case study, comparing all students in the tool group (including those who dropped out, as long as they turned in their post-survey form) with students in the control group. We found that on average the tool group's confidence increased by an average of 0.08 points, while the control group's confidence decreased by an average of 0.21 points (statistically insignificant, p-value 0.25).

The 5 students who completed the assignment entirely using the tool and reported their time (many students did not report time, although we asked them to) spent about 2 hours longer than the 14 control group students who reported their time (statistically insignificant, p-value 0.11). This effect could easily be due to chance, but it is possible that the tool encouraged or required students to spend more time on the assignment, which may have in turn affected the other outcomes of the study.

In summary, our quantitative outcomes were not statistically significant, but they do suggest areas for investigation in future studies.

6. Related Work

Educational Tools for Mathematics. Barland et al. argue that the increasing demand for reliable software systems creates a pressing need for a stronger focus on logic in computer science education, and suggest an integrated, tool-supported educational approach (Barland et al. 2000). They argue that tool support is key to reaching students with a broader range of learning styles (Tobias 1992).

Many tools have been developed for teaching mathematics. One example, similar to our work in intent but for different areas of math, is the EPGY Theorem Proving Environment, which was used to help students learn to do proofs in geometry, linear algebra, number theory, and other topics (Sommer and Nuckols 2004). This tool is aimed at helping

students write theorems according to standard mathematical practice, verifying student reasoning and automatically proving side conditions that are routinely omitted in standard practice. The tool provides a graphical user interface with menu-based interaction.

Educational Tools for Logic. Along similar lines, a number of tools have been developed to aid in the teaching of logical reasoning. Generally, they allow the structured application of inference rules by a student in order to complete a proof. Many also provide a graphical user interface displaying proof trees, inference rules, and giving help as needed (see, for example, (Scheines and Sieg 1994), or (Goldson and Reeves 1993) for a survey). A fairly rich collection of tools is Hyperproof, Tarski's World, and Turing's World which integrates deductive and semantic reasoning (Barwise and Etchemendy 1998). In particular, it recognizes the importance of counterexamples and the connection between domains and the logical formalisms describing them.

Previous work, however, pays little attention to the computational interpretation of logic or applications to programming languages, which is central to our work.

Educational Tools for Program Semantics. The Tutch tool for constructive logic allows students to write down a proof very similarly to the way they write down a program and then have it analyzed by a checker separately for correctness very similarly to the the way a program is compiled (Abel et al. 2001). This analogy to programming proved to be very helpful, and anecdotal evidence and course evaluations strongly suggest the important role this tool has played in the students learning experience. It has been used in a multidisciplinary junior-level logic class at Carnegie Mellon (computer science, mathematics, and philosophy) since 2000.

Matthews et al have developed a tool for specifying and experimenting with operational semantics using contextual rewriting rules (Matthews et al. 2004). This has been beneficial in the presentation of the Scheme language, especially as it forces one to be formal about evaluation strategies and their interaction with effects. The visual interface allows students to animate the operational semantics and experience the reduction rules in action, making an abstract formalism tangible. The DrScheme tool supports proofs of language properties indirectly through an interface to ACL2 (Eastlund et al. 2007).

Tools have been developed to support compiler or interpreter development using ideas from operational semantics, including ASF+SDF (van den Brand et al. 2002) and the Relational Meta Language (RML) (Pettersson 1995), and others. However, we are not aware of studies applying these in an educational setting, nor do these tools explicitly support proofs about programs written in the logic.

Logical Frameworks and Twelf. Our work is generally based on the principles of a logical framework (Pfenning 2001), a meta-language in which one can specify and the reason with deductive systems. More specifically, we build on the approach embodied in Twelf (Pfenning and Schürmann 1999). Twelf has been successful as a research tool, and has also been utilized as a teaching tool. Frank Pfenning has taught several senior undergraduate and graduate courses in the theory of programming languages and written a textbook (Pfenning), currently undergoing revision, that utilize Twelf in an educational setting.

However, Twelf is not quite ready for use in lower-level undergraduate courses. Despite (and in some cases because of) its extremely uniform and minimalistic style, teaching the tool to students occupies several valuable weeks of course time. On the other hand, the concepts supported by logical frameworks in general and Twelf in particular are important recurring concepts that students must learn in any case, such as variable binding, formal inference, or reasoning under hypotheses. A significant portion of the present work can therefore be seen as a way to harness research tools such as Twelf for use in undergraduate education.

Other Proof Assistants. Researchers have had considerable success using proof assistants for general mathematics, such as Isabelle/HOL (Nipkow et al. 2002) and Coq (Bertot and Castran 2004), to formalize programming language metatheory. These tools have advantages over Twelf and SASyLF, including support for a broader range of theorems (e.g. logical relations, which are not supported in Twelf) and support for high-level proof tactics that raise the level of abstraction of proofs. On the other hand, these tools have a fairly steep learning curve and have not been widely integrated into undergraduate programming language curricula. One particular barrier to teaching language metatheory is that variable binding must be encoded, and while recent work has explored more convenient interfaces to variable binding (Aydemir et al. 2008), these encodings take time to teach and learn, and also distract from the main purpose of course assignments.

One intermediate point in the design space is concurrent work on Abella (Gacek 2008), which builds in variable binding like Twelf and SASyLF, but supports interactive theorem proving with tactics like Isabelle/HOL and Coq. While Abella shares some of SASyLF's educational advantages, and is more powerful and likely more productive for expert users, it does not fulfil our goals in terms of familiar syntax, simplicity of the tool interface, or the explicit nature of proofs.

Ott. The Ott system, like our work, allows users to write down the syntax and semantics of a programming language in much the same notation that would be used on paper (Sewell et al. 2007). However, Ott does not directly support language metatheory; instead, the tool generates definitions capturing the language semantics in the input format of another tool, and users must learn to use that tool in order to prove metatheorems. Ott supports richer syntactic binding forms than our system, but at a cost of a more complex notation for binding and substitution. Furthermore, the Ott tool is limited by a lack of support for capture-avoiding substitution, limiting the metatheory that can be done based on the tool's definitions.

Acknowledgments

We would like to thank Frank Pfenning for his early suggestions on the overall design of the tool. Matthew Rodriguez implemented our GLR parser and the derivation search algorithm. We also appreciate feedback from William Lovas and other members of the LF and Plaid groups at CMU. This work was supported in part by NSF CAREER award CCF-0546550, DARPA grant HR00110710019, a NSF Graduate Resource Fellowship for the second author, and the Department of Defense.

References

Andreas Abel, Bor-Yuh Evan Chang, and Frank Pfenning. Human-Readable, Machine-Verifiable Proofs for Teaching Constructive

Logic. In *Workshop on Proof Transformations, Proof Presentations and Complexity of Proofs (PTP'01)*, June 2001.

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering Formal Metatheory. In *Symposium on Principles of Programming Languages*, 2008.

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *Theorem Proving in Higher Order Logics*, pages 50–65, 2005.

Ian Barland, Matthias Felleisen, Kathi Fisler, Phokion Kolaitis, and Moshe Y. Vardi. Integrating Logic into the Computer Science Curriculum. http://www.cs.utexas.edu/~csed/FM/docs/iticse-fislervardi.pdf, 2000.

Jon Barwise and John Etchemendy. Computers, Visualization, and the Nature of Reasoning. In *The Digital Phoenix: How Computers are Changing Philosophy*, pages 93–116. Blackwell, 1998.

Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.

Carl Eastlund, Dale Vaillancourt, and Matthias Felleisen. ACL2 for Freshmen: First Experiences. In *ACL2 Workshop*, 2007.

Andrew Gacek. The Abella Interactive Theorem Prover (system description). In *IJCAR*, 2008.

D. Goldson and S. Reeves. Using Programs to Teach Logic to Computer Scientists. *Notices of the American Mathematical Society*, 40: 143–148, 1993.

Robert Harper. *Practical Foundations for Programming Languages*. Available at http://www.cs.cmu.edu/~rwh/plbook/book.pdf, 2008.

Robert Harper, Furio Honsell, and Gordon D. Plotkin. A Framework for Defining Logics. *J. ACM*, 40(1):143–184, 1993.

Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A Visual Environment for Developing Context-Sensitive Term Rewriting Systems. In *International Conference on Rewriting Techniques and Applications*. Springer Verlag LNCS 3091, 2004.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle/HOL: A Proof Assistant For Higher-Order Logic. *Lecture Notes in Computer Science*, 2283, 2002.

Mikael Pettersson. *Compiling Natural Semantics*. PhD thesis, Linkoping University, 1995.

Frank Pfenning. Logical Frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 17, pages 1063–1147. Elsevier Science and MIT Press, 2001.

Frank Pfenning. *Computation and Deduction*. Cambridge University Press. In preparation. Draft from April 1997 available electronically at http://www.cs.cmu.edu/~twelf/notes/cd.ps.

Frank Pfenning and Carsten Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *International Conference on Automated Deduction*, pages 202–206, July 1999.

Richard Scheines and Wilfried Sieg. Computer Environments for Proof Construction. *Interactive Learning Environments*, 4:159–169, 1994.

Carsten Schürmann. *Automating the Meta-Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, 2000.

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Tom Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective Tool Support for the Working Semanticist. In *International Conference on Functional Programming*, pages 1–12, 2007.

Richard Sommer and Gregory Nuckols. A Proof Environment for Teaching Mathematics. *J. Autom. Reason.*, 32(3):227–258, 2004.

S. Tobias. Revitalizing Undergraduate Science: Why some things work and most don't. Research Corporation, 1992.

Masaru Tomita. An Efficient Augmented-Context-Free Parsing Algorithm. *Computational Linguistics*, 13(1-2):31–46, 1987.

Mark van den Brand, Jan Heering, Paul Klint, and Pieter Olivier. Compiling Rewrite Systems: The ASF+SDF Compiler. *Transactions on Software Engineering and Methodology*, 24:334–368, 2002.