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 its semantics can be understood based on induction over canonical forms in LF. 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  CR-number [subcategory]; third-level
General Terms  term1, term2
Keywords  keyword1, keyword2

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 Schürrmann 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 very 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 in most theorem proving systems because it must be encoded in some way, presenting yet another barrier to using proof assistants in coursework.

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 us-
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. By adopting the LF methodology of re-casting induction over the structure of derivations as induction over canonical forms of LF, we can rely on the previous work that forms the core of the Twelf proof assistant (Schürmann 2000).

- We describe 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 describe the SASyLF proof assistant from a user’s point of view. Section 3 discusses the semantics of SASyLF in terms of the logical framework LF. Section 4 discusses our implementation, experience with the tool, and describes the result of a controlled experiment using SASyLF in an educational context. Finally, Section 5 discusses related work.

2. 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.

2.1 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 τ.

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 ∀x.(B ∧ A[x]) represents that the bound variable x is bound in A but not in B.

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

```
judgment value: e value
----------- val-unit
("" ") value

------------- val-fn
fn x1 : τ ⇒ e1[x1] value

judgment step: e → e

----------- c-app-1
e1 ⇒ e1’ e2

----------- c-app-l
e1 value
----------- c-app-r
(e1 e2) → (e1 e2’)

----------- r-app
(fn x : τ ⇒ e[x]) e2 ⇒ e[e2]
```

Whenever there is such a binding form, the variable should be mentioned elsewhere in the production (the x in fn x : τ ⇒ ...), 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 : τ ⇒ e[x] contains a subexpression with a bound expression variable.

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

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

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

```
Γ ::= * | Γ, x : τ

e ::= fn x : τ ⇒ e[x] | x | e e

tau ::= unit | τ → τ

--------- r-app
(fn x : τ ⇒ e[x]) e2 ⇒ e[e2]
```

SASyLF Proof Assistant
judgment has-type: Gamma |- e : tau
assumes Gamma

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

-------------------------------------- t-var
Gamma, x:tau |- x : tau
Gamma, x:tau |- e[x] : 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 ---

2.2 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 z-app also illustrates how substitution is expressed. 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.

--- 2.3 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 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 category τ).

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.

--- 2.4 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: Γ, J ⊢ J. This is the definition of a hypothetical judgment.
- Weakening. If Γ ⊢ J then Γ, J1 ⊢ J. Additional assumptions cannot interfere with an existing derivation.
- Limited exchange. If Γ, J1, J2 ⊢ J and J2 does not use variables bound in J1, then Γ, J2, J1 ⊢ J. That is, ordering of hypotheses is immaterial, as long as variable binding requirements are respected.
- Contraction. If Γ, J1, J2 ⊢ J then Γ, J1 ⊢ J. Since we can use a hypothesis multiple times, it does not need to be repeated.
- Substitution. If Γ, J1 ⊢ J and Γ ⊢ J1 then Γ ⊢ J. Here we take the derivation of J1 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 prop-
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 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 \( \Gamma \) which is an extension of the context \( \Gamma \) 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 \( \Gamma \) could possibly be used in that premise.

### 2.5 Theorems and Proofs

Figure 4 shows the proof of type preservation for the simply-typed \( \lambda \)-calculus in SASyLF (one case is elided). SASyLF supports theorems of the form "for all \( \forall \) list of metavariables and judgments" there exists \( \exists \) 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, \(* \vdash e' : \tau \text{, 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 gener-

```plaintext
theorem preservation: \( \forall \alpha \vdash e : \tau \)
the rule is stated using fresh metavariables that are bound in the body of the case (\( e1, e2, e1' \text{ in the case of } c\text{-app-1} \)). SASyLF matches the conclusion of the rule to the judgment we are case-analyzing, and determines that \( e \) has been substituted with \( e1 \) and \( e2 \) and that \( e' \) has been substituted with \( e1' \) and \( e2' \).

We proceed to further case analyze on the typing derivation \( dt \). Since we know that \( \alpha = e1 \) \( e2 \) 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 \( e1 \rightarrow e2 \) and \( * \vdash e1 : \tau \rightarrow \tau \). 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 \( d6 : * \vdash e1' : \tau \rightarrow \tau \) and in fact what you get from applying the theorem to those

\( \alpha \).

\(^{3}\)Twelf allows multiple judgments in the exists clause, something SASyLF can encode and which we plan to support in the future.
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 \(* \vdash e' : \tau\) (where in this case e' = e1' e2). We get this by applying rule \(-\app\) 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 analyze on the typing rule (which must be \(-\app\) 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 \(-\app\) we know that the function type \(\tau\' \rightarrow \tau\), 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 \(-\fn\) proves that \(\tau'' = \tau'\), since otherwise the rule could not be applied.

Now, with derivation d7 we have learned that \(*, x:\tau' \vdash e1[x] : \tau\), and we know from d5 that \(* \vdash e2 : \tau'\). From the properties of hypothetical judgments we know that if \(\Gamma, J \vdash J\) and \(\Gamma \vdash J\), then \(\Gamma \vdash J\) (Harper 2008). We appeal to this property with the claim that \(* \vdash e2[e2] : \tau\) by substitution on d7, d5 SASyLF checks that d5 matches the judgment represented by the hypothesis \(x : \tau'\), which is given by rule \(-\var\).

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), “solve” (where a prototype solver attempts to find a derivation automatically), and “unproved” (when we want to postpone proving one branch of a derivation but want to verify the rest of the proof anyway).

3. Semantics of SASyLF

The semantics of SASyLF definitions and proofs are based on the dependently typed logical framework LF (Harper et al. 1993). The convenience of representing formal systems in LF stems from the fact that LF’s notion of variable binding can be used to represent both binding and hypothetical judgments in the encoded language. The judgments as types principle allows us to represent derivations of a judgment in the deductive system of interest (often referred to as the “object language”) as canonical LF terms of a particular type. In doing so, we re-cast the problem of induction over the structure of derivations as a problem of induction over canonical forms of LF.

The mechanization of inductive proofs over canonical forms of LF has been studied extensively in the context of the Twelf implementation (Pfenning and Schürrmann 1999), and we are able to capitalize on that intellectual machinery. The current practice of doing language metatheory in Twelf is described in (Harper and Licata 2007), and it is worth comparing with the methodology of using SASyLF on several points:

- The syntax and judgments of the formal system must be represented in LF. In Twelf, all of LF is available to encode systems; in SASyLF, only a fragment of LF is available, but it is a fragment that still allows for the encoding of most of the systems that are interesting from the standpoint of teaching programming language theory.

- It is often necessary to specify a proof of a judgment in a certain hypothetical context—for instance, it is generally necessary to prove progress theorems for typed languages in an empty variable context, but it is crucial to prove strengthening under arbitrary typing assumptions. Twelf gives the mechanism of regular worlds to describe sets of contexts in which a proof resides, whereas SASyLF uses the assumes declaration to explain what contexts are permitted alongside a judgment.

- There must be a mechanism for describing and verifying induction over the canonical forms of LF in order to state theorems of the form “for all derivations \(\text{list of metavariables and judgments}\) there exists \(\text{judgment}\)”. In Twelf, this is done by encoding the theorem as a relation and using Twelf’s totality checking to check the proof. SASyLF allows users to directly write out inductive proofs in a style closer to the functional \(M_+\) meta-logic that is a part of Twelf (Schürrmann 2000).

In this section, we describe how SASyLF syntax and rule definitions are translated into LF definitions, and how proofs are checked following the principle of induction over canonical LF terms. Readers familiar with Twelf will note that the translation of the SASyLF simply typed \(\lambda\)-calculus declarations is virtually identical to how the simply typed \(\lambda\)-calculus is typically encoded in Twelf; we are not attempting to innovate in the area of metatheory.

3.1 Syntax Definitions

For each syntactic class named in the syntax of a SASyLF file, we declare a LF type—for example, for the definitions in Figure 1 we get the types e, \(\tau\), and Gamma.4 In Twelf we would write this as:

\[ e : \text{type}. \]
\[ \tau : \text{type}. \]

We then declare constructor for each significant production in the grammar. If a production has only terminals, the constructor is just an LF constant. For example, the unit constructor for expressions would be written in Twelf as:

\[ \text{cons_unit} : e. \]

Here \(\text{cons_unit}\) would actually be some made-up, internal constant name. Productions that have nonterminals, like e e, take one curried argument for each nonterminal, of that nonterminal’s type, in the order that the nonterminals appear. So in Twelf notation we have:

\[ \text{cons_app} : e \rightarrow e \rightarrow e. \]

The most interesting case is for productions which include variable binding. In these productions, we expect each bound variable to appear as a literal once in the production, which is interpreted as the binding occurrence for that variable. Each such bound variable must then be bound in one

4 The last of these will be unused, as \(\Gamma\) is only used as a context and contexts are treated specially.
or more of the nonterminals \((e[x])\) in the abstraction production in Figure 1). The constructor we develop ignores the binding occurrences of the variable (i.e., it does nothing special for them), but for each of the nonterminals in which the variable is bound, the constructor takes a function from the types of each of the variables to the type of the nonterminal.

For example, the syntactic construct representing functions in the simply typed lambda calculus, \(\text{fn } x : \tau \Rightarrow e \Rightarrow e\), has a subexpression \(\tau\) and an argument \(e[x]\) where the bound variable \(x\) is an expression variable, that is a variable in the syntactic class \(e\), and would be written in Twelf as:

\[
\text{cons_abs : } \tau \rightarrow (e \rightarrow e) \rightarrow e
\]

If we consider type abstraction instead, with a production \(e ::= \text{fn } \tau \Rightarrow e \Rightarrow t\), then we would have an LF constructor of type \((\tau \rightarrow e) \rightarrow e\), assuming that we had a production \(t ::= t\) which instructs SASyLF to treat \(t\) as a type variable.

A syntactic construct may have a nonterminal with multiple variables bound in it as well. For example, one could define fix-points as \(e ::= \text{fix } f x \Rightarrow e[f][x]\), and the resulting constructor, assuming that SASyLF treats both \(f\) and \(x\) as expression variables, would be:

\[
\text{cons_fix : } (e \rightarrow e) \rightarrow e \rightarrow e
\]

Note that although a nonterminal may have multiple variables bound in it, those variables may only have a base type like \(e,\) not a function type like \(e \rightarrow e\). Thus constructors are inherently second-order at most: SASyLF allows constructors with types like \((e \rightarrow e \rightarrow e) \rightarrow e\) but not, for example, a constructor with type \((\tau \rightarrow e) \rightarrow e\) \(\rightarrow e\) which would be allowable in LF/Twelf. This is the sense in which SASyLF supports only Second-order Abstract Syntax—and it means that certain techniques that are possible in Twelf, for example intrinsic encodings which embed a proof into syntax, are not possible in SASyLF. While this is a limitation, we believe that second-order constructors provide most of the benefit of higher-order constructors, and they allow us to use a clear and simple input syntax that is likely to be more understandable to students.

Finally, we do not produce constructors for variable productions like \(e ::= x\) because we use LF variables directly to represent object language variables.

### 3.2 Judgment and Inference Rule Semantics

The interpretation of a SASyLF judgment is a LF type family that relates the different elements of the judgment, which is a familiar reprise of the judgments as types principle that is central to LF (Harper et al. 1993). For example, the single-step evaluation judgment is defined using a dependent type family \(\text{step} : e \rightarrow e \rightarrow \tau\rightarrow type\). The judgment that \(e1\) steps to \(e2\) is represented by an instance of this type family, the type \(\text{step } e1 \ e2\).

Given this, SASyLF inference rules are represented by constants that have a function type, from the types corresponding to the premises of the rule to the type corresponding to the conclusion of the rule. In order to determine these types, we must parse the judgments in the premise and conclusion down to the LF types they represent. We do this using our implementation of GLR parsing.\(^5\) We then build an LF type recursively, starting from the leaves of the parse tree. Variables \(x\) are replaced with equivalent LF variables (in our concrete representation de Bruijn indices are used); all such variables must be bound in the surrounding context. Metavariabes \(e1\) are replaced with free variables of the appropriate LF type \(e\) (in the example). Expressions of the form \(e1[x]\) or \(e1[e2]\) become LF applications. Here \(e1\) is assumed to have type \(e \rightarrow e\) and so the application is well typed.

When a production is used in the parse tree, we generate a term or type using the constructor or type constructor for that production. If the production has no arguments, we are done. Otherwise, we apply the constructor to the LF terms recursively generated by the parse trees in the respective positions. For example, \(e1 e2 \rightarrow e1' e2'\) turns into the LF type \(\text{step } (\text{cons_app } e1 e2) (\text{cons_app } e1' e2')\).

If a production binds one or more variables, we must build potentially dependent type abstractions to bind the corresponding LF variables. For this, we refer back to the syntax definition, and relate the variables bound in each nonterminal to the place where that variable is mentioned syntactically. We look in the parsed term in that place to find the variable name used in this particular term (which, for example, may be \(x1\) not \(x\)) and then we wrap the LF term that corresponds to a nonterminal with bound variables in the syntax with LF abstractions for each bound variable (in order).

For example, consider converting the conclusion of \(\text{val-fn}\) in Figure 2 into an LF type. Looking at the syntax, we see that the variable \(x\) bound in \(e[x]\) is mentioned syntactically right after the \(\text{fn}\) terminal. SASyLF looks at the conclusion of \(\text{val-fn}\) and identifies that the variable there is \(x1\). According to the translation described above, the expression \(e1[x1]\) is translated into the LF term \(e1 x1\) (i.e., the application of \(e1\) to \(x1\)). Since \(x\) is bound in \(t\) at that position in the syntax, we surround this term with an abstraction to get \(\lambda x1.t (e1 x1)\). We then apply the constructor for function expressions to this term and the variable \(\tau\), and then apply the type constructor for the value judgment to that, finally obtaining the LF type:

\[
\text{value } (\text{cons_fn } \tau \lambda x1.e. (e1 x1))
\]

### 3.3 Translating Contexts

When translating typing rules, there is the additional complexity of translating the context \(\Gamma\) into LF. The \text{assumes} declaration tells SASyLF that \(\Gamma\) should be treated as an LF context rather than as syntax. Therefore, the \(\Gamma\) component of the judgment is ignored both in the type family for that judgment and in the expression translation process denoted above. For example, the type family for the typing judgment is \(\text{has-type} : e \rightarrow t\rightarrow \tau \rightarrow \text{type}\).

Instead, we infer the meaning of \(\Gamma\) from the inference rules for variables, and use that semantic meaning to build a representation of the context appropriate for reasoning in LF. As described before, for each recursive production for \(\Gamma\), we identify a rule \(t\text{-var}\) in this case that unrolls the context once and uses the bound variable in the rest of the judgment. We construct an LF type for the conclusion of rule \(t\text{-var}\), obtaining \(\text{has-type}\ \tau \rightarrow \text{type}\). Whenever we see this production used in a rule, then we interpret it to (a) bind \(x\) as

---

\(^5\) Surprisingly, although we found multiple implementations of GLR parser generators which yield generated code, we could not find a library that, given a grammar, yields a function or object that implements GLR parsing for that grammar. Our open-source GLR parser implementation will be released with SASyLF.
a variable of type e, and (b) encode an assumption that there exists some derivation of the property has-type x tau.

Thus, when we encounter an expression like the premise of rule t-fn, which explicitly declares that the context contains x1:tau, we first generate an LF type as described above, then surround this type with type abstractions that bind x and represent the knowledge that x has type tau. In this example, the initial term we get (ignoring the context) is has-type (e x1) tau'. Wrapping this with the type abstractions representing x1:tau, we get:

\[ \Pi x1: e \ . \ \Pi dx: (has-type x tau) \ . \ has-type (e x1) tau' \]

Here, dx is a variable representing a derivation of the fact that x has type tau. It's in fact exactly how the premise of the \(\lambda\)-calculus function typing rule is usually encoded in Twelf, but for readers unfamiliar with Twelf, an intuition can be formed based on substitution. If e[x1] has type tau' assuming that x1 has type tau, we expect that substituting x1 with some expression e', where e' also has type tau, we should get a term e[e'] that is still well-typed and has type tau'. In the LF type, the type abstraction encodes this directly: it says that if we pass an expression of type e for x1 and we pass in a proof that e has type tau then we will get a proof that the substituted expression has type tau'. In fact, in LF we can get this proof simply by applying the expression e' and a proof dte':(has-type e' tau) to the type abstraction above.

### 3.4 Semantics of SASyLF Proofs

As illustrated in Figure 4, a theorem in SASyLF is of the form “For all derivations and terms of some input forms, there exists a derivation of some output form.” There may be free variables in the derivations in the theorem—for example, in the statement of preservation we mention e, tau, and e'. Here we follow the convention of Twelf (and, informally, mathematics as well) that we universally quantify over any variable that appears in a derivation over which we have universally quantified, and variables that appear only in existentially quantified derivations are also existentially quantified. We could also state this explicitly, in which case the preservation theorem would be:

\[ \forall e \forall tau \forall e' \forall ds: e \rightarrow e' \exists x : \exists e' : \exists tau. \]

SASyLF proofs follow the overall contours of Twelf proofs, despite significant syntactic differences. However, the language SASyLF uses to describe inductive proofs is more explicit in two major ways. First, SASyLF divides derivations into atomic steps, essentially introducing a let-binding at each intermediate step in the derivation of a proof. Second, SASyLF requires developers to explicitly state what is concluded from each derivation—essentially requiring each subterm in a Twelf derivation be annotated with an explicit type. While these changes do make proofs more verbose, they allow us to isolate errors very effectively to a primitive step. As a result, we can provide better error messages to student users.

In the following subsections, we describe the checks for each proof step in more detail. As mentioned before, these checks are semantically equivalent to the checks in the \(M^+\) meta-logic on which Twelf is based. However, in order to make the paper self-contained, we describe these checks in terms of first principles.

### Rule Application

To check a derivation by rule, we start by constructing LF types (as described above) for the judgments that the user specifies as the arguments and result of the rule. Because in the process of case analysis (discussed in detail below) we may have accumulated assumptions about how free variables are instantiated in this branch of the proof, we apply a substitution that represents any assumptions in scope.

Likewise, we construct LF types for each of the premises and the conclusion of the rule. In these types, we replace free variables with fresh ones, to ensure we avoid accidental clashes with free variables in the user’s type.

We then check that the arguments provided by the user are instances of the rule premises. This instance check proceeds by attempting to unify the LF types, then verifies that the unification result can be expressed as a substitution whose domain is only the free variables in the rule’s premises. If the types unify but the instance check fails, it means the rule does not necessarily apply—the user has made an unjustified assumption about the form of an input judgment.

We then take the rule’s conclusion, apply the substitution from the premise instance test, and check that it is an instance of the user’s claimed result. Here we cannot allow the user to specify any structure that is not implied by the rule itself, as this would represent an unjustified assumption about the form of the output of the rule.

### Unification

We use the same unification algorithm as Twelf (Schürrmann 2000). Essentially, this algorithm performs decidable higher order pattern unification (Nipkow 1993) but when so-called flex-flex subterms (i.e. terms of the form e1[e2]) are encountered that do not fit the pattern fragment, unification of these subterms is delayed as long as possible. Typical experience in Twelf is that the unification of other subterms eventually brings these flex-flex terms into the pattern fragment, but if this does not occur then unification will fail (as it can potentially in Twelf—though this is rarely encountered in practice).

As a side note, our unification algorithm, like Twelf’s, enforces the invariant that a term substituted for a free variable may not contain bound variables other than the ones explicitly bound in the free variable. For example, if we have a free variable e[x] in which (only) x is bound, we can substitute it with a term like \(\lambda x.c\times\) but not a term like \(\lambda x.c\ y\) (with c a constructor and x,y bound variables). If unification would force substitution with such a term, the unification fails.

### Context Adaptation

The context at which a rule is applied may, in general, be an unrolling of the context at which a rule is defined. For example, the rule t-app may be applied in the context Gamma, x2:tau rather than the context Gamma. In such cases, the LF type for the conclusion of the rule will not match the LF type for the judgment given by the user, because the latter will have an extra surrounding type abstraction (added as specified in the “Translating Contexts” section above).

We resolve this problem by adapting the types generated from the premises and conclusion of the rule to the context used in the user’s claimed result, by wrapping them in additional type abstractions that correspond to however many extra context unrollings are present in the user’s claimed result. But each additional abstraction also affects the type.
each free variable in an LF type may implicitly contain any variables bound in the LF context, and so when we make these variable bindings explicit, we must also modify the free variables in the type to show that the new variable may be bound.

In the example above, the type for the conclusion of rule \texttt{t-app} must be modified to represent the SASyLF term \(\Gamma, x_1: \tau, x_2: \tau' \vdash e [x_1] e_2 [x_2]: \tau\). One caveat demonstrated here is that \(x\) may not be free in \(\tau\), since the \(\tau\) syntax cannot have any expressions in the \(e\) syntax embedded within it. The property of whether one syntactic form can be transitively embedded in another is called subordination (Schürmann 2000), and we compute this relationship between syntactic forms as needed in order to decide where these new variables may be bound.

**Lemma Application.** Constructing a derivation by using a lemma is identical to applying an inference rule.

**Case Analysis.** When the user applies case analysis, we check each of the cases as follows. First, we unify the LF type for the conclusion of the rule (produced as described above) with the type for the judgment we’re analyzing. We use the results of this to instantiate the rule’s premises.

We then compare the instantiation of the rule provided by the user with the one we have just computed. If the premises and conclusion of the computed rule must be an instance of the premises and conclusion provided by the user. This instance check ensures that the user cannot assume anything more about the form of the rule than is justified.

Finally, we analyze the body of the case analysis. In the body, we know that the free variables in the expression we’re analyzing were instantiated as specified in the conclusion provided by the user. This knowledge is captured by unifying the user’s conclusion with the expression being analyzed, and adding the resulting substitution to the assumptions currently in scope.

**Coverage Checking.** In addition to checking each case of a case analysis, we must verify that the user has provided a case for each possible form of the input. To check coverage, we follow Twelf by computing an LF type for each possible case of the input and verifying that the user has provided a case that matches. There are three kinds of cases to cover: ordinary rule application, use of an explicitly bound variable, and use of a variable that is implicitly bound in the context.

For ordinary rule application, we try unifying the conclusion of the rule with the expression being case analyzed. If the unification succeeds, the user must provide a corresponding case; if the unification fails, it is impossible to derive that expression using the rule and so no case is required.

It is also possible that a judgment could be based on a variable assumption rule—for example, using \texttt{t-var} to prove that \(x\) has type \(\tau\). For every variable explicitly bound in the context of the expression being case analyzed, we must consider the corresponding variable assumption rule. For example, if the context of the expression being case analyzed is \(\Gamma, x_1: \tau, x_2: \tau'\) we must consider two variable assumption rules, one for using \(x_1\) and one for using \(x_2\).

Finally, we must consider using a variable in \(\Gamma\) that is not explicitly bound. For example, if the context of the term being case analyzed is \(\Gamma, x_1: \tau, x_2: \tau'\) we must consider whether rule \texttt{t-var} could be applied to some \(x_3\) of some type \(\tau'\). The case is:

\[
\Gamma, x_3: \tau'', x_1: \tau, x_2: \tau' \vdash x_3 : \tau' \text{ with } \Gamma = \Gamma', x_3: \tau''
\]

It is sufficient, as in Twelf, to describe only one such implicit variable case for each context usage rule such as \texttt{t-var}, because that implicit case can be used to represent anything we might have gotten from the context.

**Induction.** Proofs by induction involve a case analysis, and are generally checked the same way. As in Twelf, if induction is used it must be the first step in a proof—this does not restrict the generality of our system as we can do the induction in a lemma if induction is not naturally the first step. The one difference between induction and case analysis is that we can appeal to the induction hypothesis inside each case, and so we need to track which argument of the theorem we are doing induction over, and what expressions are known to be subderivations of that argument. When the user gets a derivation from the induction hypothesis, we check this as if it were a use of a rule or lemma (see above), but we additionally verify that the inductive argument is a subderivation of the input to the current theorem.

The above describes our current design, which only supports induction over the derivation of a single judgment. However, it should be straightforward to extend our tool to the forms of induction supported by Twelf, including mutually inductive theorems and lexicographic induction orderings.

**Weakening.** The user may use weakening as a justification to add a premise to a judgment already known. In this case we simply check that the conclusion the user gives is identical to what was known before, except for the added premise.

**Exchange.** The user may use exchange as a justification to swap the order of assumptions in the context of a judgment. As with weakening, we verify that the conclusion the user gives is identical to what was known before, except for the swapped assumptions. We must also verify that the variable bound in the assumption that was originally first could not have been used in the second, in which case an exchange would be unsound. This check, like Context Adaptation discussed above, uses the subordination property.

**Contraction.** Contraction is checked just like weakening, except that we verify that the contracted hypotheses were identical.

**Completing a proof.** The last derivation in the proof, and the last derivation in each case of a case analysis or induction, must be the thing we are proving.

**Using an assumption or a previous derivation.** The user may cite an assumption of a theorem or a previously derived judgment as justification for a new conclusion. Although the judgment is already known, we require that the last judgment be the one which proves the theorem, and so judgments might have to be reiterated. We check that the new conclusion is the same as the judgment that was cited, after the assumptions known from any surrounding case analysis are applied.

**Substitution.** Assume the user has claimed that a judgment holds by substitution of derivation \(d_5\) into derivation \(d_7\), as

\[
\Gamma, x_3: \tau'', x_1: \tau, x_2: \tau' \vdash x_3 : \tau' \text{ with } \Gamma = \Gamma', x_3: \tau''
\]

we could in principle have more than one such rule if \(\Gamma\) includes different kinds of variables or assumptions
in Figure 4. The resulting judgment should have one fewer variables in its context compared to derivation $d_7$; the missing variable $x$ is the one we substituted for.

We first compute the form that the substituted derivation should take, given the assumption about $x$ in the context of $d_7$. We do this by identifying the corresponding variable use rule—$t$-var in this case—and substituting the actual variable name $x$ and actual free variables ($\text{tau}'$ in this case) for the variable name and free variables in the rule ($x$ and $\text{tau}$, respectively). In this case, we get the conclusion 

\[
\Gamma, x: \text{tau}' \vdash x : \text{tau}'.
\]

We then replace $x$ with a fresh metavariable $e$ representing the expression to be substituted, and we replace $\Gamma, x: \text{tau}'$ with the assumptions in $d_7$ that came before the variable use—i.e., the assumptions on which the derivation of $x: \text{tau}'$ could legitimately depend. In this case, we get the conclusion 

\[
\Gamma \vdash e : \text{tau}'.
\]

We do this by identifying the corresponding variable name and free variables in the rule ($x$ and $\text{tau}$, respectively). In this case, we get the conclusion 

\[
\Gamma, x: \text{tau}' \vdash x : \text{tau}'.
\]

To finish the substitution check, we ensure that the final expression is $d_7$ without the binding of $x$ in the context, and with $e_2$ substituted for $x$.

**Automatic solving.** If the user claims a judgment and requests that SASyLF find an appropriate derivation, we search for such a derivation using depth-first search and a bounded maximum depth. This is similar to Twelf’s “filling” operation (Schürmann 2000). Our current implementation does not search for richer proofs that involve induction or case analysis, although we plan to extend our implementation to search for these.

**Unproved judgments.** If the user claims a judgment and says it is unproved, we simply give an “unproved warning” and proceed. This allows SASyLF to assist with checking partially complete proofs.

### 4. Implementation and Evaluation

#### 4.1 Tool Support

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

http://www.cs.cmu.edu/~aldrich/SASyLF/

We have implemented the most important checks in the system, 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, and a few corner cases elsewhere.

#### 4.2 Case Studies

The distribution includes several SASyLF examples. The first, in lambda.slf, is a formalization of the simply-typed $\lambda$-calculus, with progress and preservation theorems, as presented in the running example for this paper. Although substitution should take, given the assumption 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.

Finally, we 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.

### 4.3 Controlled Experiment

We performed a 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. However, we believe it has the potential to shed light on the value of SASyLF in a course that covers language theory more comprehensively.

**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 postcondition 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.
4.4 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.

4.5 Experimental Outcomes: Quantitative

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. A t-Test gave a p-value of 0.26, so this result, though encouraging, is not statistically significant at the 95% level.

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. This effect was not statistically significant at the 95% level; the p-value was 0.25 (about a 1 in 4 probability that this was due to chance).

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. The t-Test gave a p-value of 0.11, meaning it is not statistically significant, 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, while our quantitative outcomes did not rise to the level of 95% statistical significance, we saw a number of indications that suggest that the tool may be useful in practice.
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 solved 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. 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.** Among 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 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. However, the tool does not support proofs of language properties, which is our primary goal in this work.

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.

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 on 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 Research Fellowship for the second author, and the Department of Defense.

References


