SASyLF: An Educational Proof Assistant for Language Theory

Jonathan Aldrich    Robert J. Simmons    Key Shin
School of Computer Science    Carnegie Mellon University    Microsoft Corporation
Workshop on Mechanizing Metatheory
September 20, 2008
Teaching PL Theory and Proofs is Hard

• Challenges
  • Too easy to get details wrong – never learn concepts
  • Slow feedback loop – wait a week for a corrected homework
  • High time and effort for the payoff

• Tools can potentially help
  • Check the details are correct
  • Provide immediate feedback
  • Raise productivity

• Just like good compilers make programming more accessible
State of the Art

- **Educational Tools** – great for their domain
  - General Mathematics: EPGY theorem proving environment
    - geometry, linear algebra, number theory, etc.
  - Logic: Tarski’s world, Tutch

- **Proof assistants and checkers** – great for experts
  - Isabelle, HOL, PVS, Coq, …
    - Benefit: very expressive, good automation, …
    - Drawback: steep learning curve
      - both for tool and for variable encoding techniques
  - Twelf
    - Higher-order type theory aids in reasoning about programs
      - Avoids variable encodings – potentially more accessible
    - Explicit, machine-checkable proof
      - Small steps students can follow
    - Still a steep learning curve: based on higher-order type theory
      - Challenging to use for undergraduate or intro graduate courses
SASyLF Design Goals

• Easy to learn
  • Familiar syntax – just like we write on paper
  • Little mathematical context – just inference rules

• Matches PL teaching approaches
  • Support for variable binding and α-equivalence

• Explicit notation
  • Every step specified completely

• Rapid feedback
  • Automated checker
  • Support for partial proofs

• Good error messages
  • Local checking and local errors
Outline

• Motivation and Design Goals
• SASyLF intro: addition commutes
• Semantics: the λ-calculus and LF
• POPLmark 2A in SASyLF
• Preliminary teaching experience
• Conclusion
Sum Commutes in SASyLF (Definitions)

Definitions

Natural Numbers

\[ n ::= z \mid s(n) \]

Inference Rules

\[ \frac{z + n = n}{\text{sum-z}} \]

\[ \frac{n_1 + n_2 = n_3}{s(n_1) + n_2 = s(n_3)} \quad \text{sum-s} \]

syntax

\[ n ::= z \]
\[ | \quad s \ n \]

package edu.cmu.cs.sum;

terminals z s

judgment sum: n1 + n2 = n3

---------- sum-z
z + n = n

n1 + n2 = n3
---------- sum-s
s n1 + n2 = s n3
Theorem [sum-z-right]: \( \forall n \ n + z = n \)

\begin{align*}
\text{Proof.} & \quad \text{By induction on } n: \\
\text{Case } z: & \\
& \begin{align*}
z + z &= z & \text{by rule } \text{sum-z} \\
\end{align*} \\
\text{Case } s(n): & \\
& \begin{align*}
n + z &= n & \text{by induction hypothesis} \\
s(n) + z &= s(n) & \text{by rule } \text{sum-s} \\
\end{align*} \\
\text{q.e.d.}
\end{align*}

\begin{align*}
\text{d1: } n + z &= n & \text{by induction on } n: \\
\text{case } z \text{ is} & \\
& \begin{align*}
d2: z + z &= z & \text{by rule } \text{sum-z} \\
\end{align*} \\
\text{end case} \\
\text{case } s(n') \text{ is} & \\
& \begin{align*}
d3: n' + z &= n' & \text{by induction hypothesis on } n' \\
\end{align*} \\
& \begin{align*}
d4: s(n') + z &= s(n') & \text{by rule } \text{sum-s on d3} \\
\end{align*} \\
\text{end case} \\
\text{end induction} \\
\text{end theorem}
\end{align*}
Demonstration: Sum Commutes

See sum.sif in the SASyLF/examples directory.
Outline

- Motivation and Design Goals
- SASyLF intro: addition commutes
- Semantics: the λ-calculus and LF
- POPmark 2A in SASyLF
- Preliminary teaching experience
- Conclusion
Semantics of SASyLF

- Based on LF, like Twelf!
  - The interesting bit is how SASyLF syntax encodes LF

- Twelf
  - Judgments as types
  - Proofs as logic programs
  - Higher-order abstract syntax

- SASyLF
  - Judgments as types
  - Proofs as let-normal functional programs
    - Each intermediate step is named and typed, aiding error messages
  - SASy - Second-order Abstract Syntax
    - Neither syntax nor judgments can include other judgments as parts
    - Same expressiveness as standard paper notation
From SASyLF to LF/Twelf

terminals fn unit

\[ e ::= x \]
\[ \quad | \text{"(" "} \text{")"} \]
\[ \quad | e \ e \]
\[ \quad | \text{fn x : tau} \Rightarrow e[x] \]

\[ \tau ::= \text{unit} \]
\[ \quad | \tau -> \tau \]

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

One LF type per SASyLF nonterminal
From SASyLF to LF/Twelf

terminals fn unit

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

tau ::= unit
   | tau -> tau

e : type.
tau : type.
unit : e.

RHS of SASyLF grammar becomes an LF constructor.
Result is LHS nonterminal
From SASyLF to LF/Twelf

terminals fn unit

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

tau ::= unit
   | tau -> tau

e : type.
tau : type.
unit : e.
app : e -> e -> e.

Nonterminals in RHS become arguments of the nonterminal’s type
From SASyLF to LF/Twelf

terminals fn unit

\[
e ::= x \quad \text{x is a case of e, so variable x has type e.}
\]
\[
e ::= \text{“(” “)”} \quad \text{No constructor created.}
\]
\[
e ::= e \quad \text{e[x] means e is a variable that is bound in e}
\]
\[
e ::= \text{fn x : tau => e[x]} \quad \text{e[x] means x is a variable that is bound in e}
\]
\[
tau ::= \text{unit} \quad \text{Variables appearing in clause ignored when producing constructors}
\]
\[
tau ::= \text{tau -&gt; tau} \quad \text{second argument of LF constructor is a function from type of x to e}
\]

\[
e : \text{type.}
\]
\[
tau : \text{type.}
\]
\[
unit : e.
\]
\[
\text{app: e -&gt; e -&gt; e.}
\]
\[
\text{abs: tau -&gt; (e -&gt; e) -&gt; e.}
\]
From SASyLF to LF/Twelf

terminals fn unit

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

tau ::= unit
  | tau -> tau

e : type.
tau : type.
unit : e.
app: e -> e -> e.
abs: tau -> (e -> e) -> e.
tunit : tau.
arrow : tau -> tau -> tau.

(fn x2 : unit => e1[x2])

(e1 x2)

e1[x2] becomes (e1 x2), where e1 has type e -> e

GLR parser produces a parse tree
From SASyLF to LF/Twelf

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

tau ::= unit
| tau -> tau

The variable declared here is bound here so we identify the variable here and bind it over here
(fn x2 : unit => e1[x2])

GLR parser produces a parse tree

[ x2 ] (e1 x2)
From SASyLF to LF/Twelf

terminals fn unit

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

tau ::= unit
  | tau -> tau

The name x2 is an extension of x (i.e. a number or ’ is added) and x has type e, so x2 has type e

(fn x2 : unit => e1[x2])

[xb:e] (e1 x2)

GLR parser produces a parse tree
From SASyLF to LF/Twelf

terminals fn unit

\[ e ::= x | \text{"("} \text{"\text{"})} | e \ e | \text{fn} \ x : \tau \Rightarrow e[x] \]

tau ::= unit
\[ | \tau \Rightarrow \tau \]

\[(\text{fn} \ x2 : \text{unit} \Rightarrow e1[x2]) \]

\[\text{abs} \ \text{tunit} \ (\text{[x2:e]} \ (e1 \ x2))\]

GLR parser produces a parse tree
From SASyLF to LF/Twelf

Gamma ::= * 
  | Gamma, x:tau

Judgments as types:
A judgment form in SASyLF turns into an LF type family

gamma has-type: Gamma |- e : tau
assumes Gamma

has-type: e -> tau -> type
Gamma is not used in the type family; the assumes clause tells SASyLF it is a stand-in variable for the LF context
From SASyLF to LF/Twelf

Gamma ::= *
    | Gamma, x:tau

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

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

has-type: e -> tau -> type

Variable rules have no LF equivalent; they correspond to using assumptions from the LF context.
From SASyLF to LF/Twelf

Gamma ::= *
    | Gamma, x:tau

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

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

----------------------------- t-fn :
Gamma, x1:tau |- e1[x1] : tau' has-type: (e1 x1) tau'
----------------------------- t-fn ->
Gamma |- fn x1 : tau => e1[x1] : tau -> tau' has-type: (abs tau ([x1:e] (e1 x1))) (arrow tau tau')
From SASyLF to LF/Twelf

Gamma ::= *
    | Gamma, x:tau

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

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

gamma, x1:tau |- e1[x1] : tau'
d------------------------ t-fn
Gamma |- fn x1 : tau => e1[x1] : tau -> tau'

has-type (abs tau ([x1:e] (e1 x1))) (arrow tau tau')
Gamma ::= * 
| \ Gamma, \ x:tau

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

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

------------- t-var
Gamma, x1:tau |- e1[x1] : tau'
t-fn : { x1 : e } { dx1 : has-type x1 tau } 
has-type: (e1 x1) tau'
------------- t-var
Gamma |- fn x1 : tau => e1[x1] : tau -> tau'
------------- t-var
has-type (abs tau ([x1:e] (e1 x1)))
(arrow tau tau')
Case Analysis on Variables

**Lemma** narrow-subtype:

\[\text{forall } dsub : \Gamma, X <: T \dashv\vdash T_1[X] <: T_2[X]\]
\[\text{forall } dsub' : \Gamma \vdash T' <: T\]
\[\text{exists } \Gamma, X <: T' \dashv\vdash T_1[X] <: T_2[X].\]

dres: \(\Gamma, X <: T' \vdash T_1[X] <: T_2[X]\) by **induction on** \(dsub: \)

**Case rule**

\[\text{------------------------ SA-Var} \quad \text{Case: } T_1[X] \text{ is } X\]
\[d1: \Gamma, X <: T \vdash X <: T\]

**is**

\[d2: \Gamma, X <: T' \vdash X <: T' \text{ by rule } \text{SA-Var}\]
\[d3: \Gamma, X <: T' \vdash T' <: T \text{ by weakening on } dsub'\]
\[d4: \Gamma, X <: T' \vdash X <: T \text{ by rule } \text{SA-Trans on } d2, d3\]

**end case**
Case Analysis on Variables

**Lemma** narrow-subtype: 

\[
\forall \text{dsub: } \Gamma, X <: T \quad \vdash T_1[X] <: T_2[X] \\
\forall \text{dsub': } \Gamma \quad \vdash T' <: T \\
\exists \text{Gamma, } X <: T' \quad \vdash T_1[X] <: T_2[X].
\]

dres: Gamma, X <: T' |- T1[X] <: T2[X] **by induction** on dsub:

**Case rule**

\[
\text{\textbf{SA-Var}} \\
\text{d1: Gamma, } X <: T \quad \vdash X <: T
\]

is … end case

**Case rule**

\[
\text{\textbf{SA-Var}} \\
\text{d1: Gamma2, } X' <: T'', X <: T \quad \vdash X' <: T''
\]

is

\[
\text{\textbf{SA-Var}} \\
\text{d2: Gamma2, } X' <: T'' \quad \vdash X' <: T'' \quad \text{by rule \textbf{SA-Var}} \\
\text{d3: Gamma2, } X' <: T'', X <: T' \quad \vdash X' <: T'' \quad \text{by weakening on d2}
\]

end case
Some Checks in SASyLF

- **Case analysis**
  - SASyLF tries unifying each inference rule’s conclusion with the judgment being analyzed
  - Can’t assume anything stronger than case computed by SASyLF (Twelf: input coverage)
  - Can’t miss a case (Twelf: input coverage)

- **Rule/Lemma application**
  - SASyLF unifies rule with sources and conclusion specified by user
  - Can’t assume “extra” structure in or equalities between input and output variables (Twelf: output freeness & output coverage)

- **Induction Hypothesis**
  - Same as rule application, but also the argument we’re inducting over must be a subderivation (Twelf: termination)

- **Substitution**
  - The judgment to be substituted fulfills the other judgment’s hypothesis (Twelf: dependent typechecking)

- **Exchange**
  - The first variable isn’t free in the second hypothesis
Outline

• Motivation and Design Goals
• SASyLF intro: addition commutes
• Semantics: the λ-calculus and LF
• **POPLmark 2A in SASyLF**
• Preliminary teaching experience
• Conclusion
POPLmark 2A in SASyLF

**Syntax**

\[ t ::= \]
- \( x \)
- \( \text{lambda } x: T \Rightarrow t[x] \)
- \( t \ t \)
- \( \text{lambda } X <: T \Rightarrow t[X] \)
- \( t "[" T "]" \)

\[ T ::= \]
- \( X \)
- \( \text{Top} \)
- \( T \rightarrow T \)
- \( \text{all } X <: T \Rightarrow T[X] \)

\[ \Gamma ::= \]
- \( * \)
- \( \Gamma, x : T \)
- \( \Gamma, X <: T \)

**Judgment**

value: \( t \) value

\[ \text{V-Abs} \]
- \( \text{lambda } x : T \Rightarrow t[x] \) value

\[ \text{V-Tabs} \]
- \( \text{lambda } X <: T \Rightarrow t[X] \) value

\[ \text{reduce: } t \rightarrow t \]
- \( t1 \rightarrow t1' \)
- \( \text{E-CtxApp1} \)
- \( t1 \ t2 \rightarrow t1' \ t2 \)
- \( t2 \) value
- \( \text{E-AppAbs} \)
- \( (\text{lambda } x : T11 \Rightarrow t12[x]) \) t2 \( \rightarrow \) t12[t2]
POPLmark 2A in SASyLF

**judgment** has-type: Gamma |- t : T

**assumes** Gamma

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

Gamma, X <: T1 |- t2[X] : T2[X]

----------------------------------- T-Tabs
Gamma |- lambda X <: T1 => t2[X] : all X <: T1 => T2[X]

Gamma |- t1 : all X <: T11 => T12[X]
Gamma |- T2 <: T11

----------------------------------- T-Tapp
Gamma |- t1 "[" T2 "]" : T12[T2]

Gamma |- t : T'
Gamma |- T' <: T

----------------------------------- T-Sub
Gamma |- t : T
POPLmark 2A in SASyLF

**judgment** subtyping: Gamma |- T <: T'

**assumes** Gamma

----------------------- SA-Top
Gamma |- T <: Top

----------------------- SA-Var
Gamma, X <: T |- X <: T

Gamma |- T1 <: T1'
Gamma |- T2' <: T2
----------------------- SA-Arrow
Gamma |- T1' -> T2' <: T1 -> T2

Gamma |- T1 <: T1'
Gamma, X <: T1 |- T2'[X] <: T2[X]
----------------------- SA-All
Gamma |- all X' <: T1' => T2'[X'] <: all X <: T1 => T2[X]

As in the Twelf 2A solution, we use declarative rather than algorithmic subtyping. The algorithmic rules would have a more complicated encoding since the variable use rule is nonstandard.
POPLmark 2A in SASyLF

**judgment** stepsorvalue: t stepsorvalue

- t value
- ----------- stepsorvalue-value
- t stepsorvalue

- t -> t'
- ----------- stepsorvalue-steps
- t stepsorvalue

**judgment** equality: t == t

- ------ equality
- t == t

**Helper judgments**

- stepsorvalue compensates for not having logical “or” built-in (same as Twelf)
- equality isn’t built in either, so we need a judgment
- other similar judgments required
POPLmark 2A in SASyLF

**lemma** arrow-sub-arrow: \( \forall d_{\text{sub}} : * \vdash T \triangleleft T_1 \rightarrow T_2 \) 
\( \exists T = T_1' \rightarrow T_2'' \triangleleft T_1 \rightarrow T_2. \)

**lemma** canonical-form-lambda: \( \forall d_{\text{tv}} : t \text{ value} \) 
\( \forall d_{\text{tt}} : * \vdash t : T_1 \rightarrow T_2 \) 
\( \exists t = \lambda x : T_1' => t'[x]. \)

**lemma** narrow-subtype: \( \forall d_{\text{sub}} : \Gamma, X \triangleleft T \vdash T_1[X] \triangleleft T_2[X] \) 
\( \forall d_{\text{sub}'} : \Gamma \vdash T' \triangleleft T \) 
\( \exists \Gamma, X \triangleleft T' \vdash T_1[X] \triangleleft T_2[X]. \)

**theorem** progress: \( \forall d_{t} : * \vdash t : T \) 
\( \exists t \text{ stepsorvalue}. \)
POPLmark 2A in SASyLF

**lemma** invert-lambda: 

\[ \forall dt:\quad \Gamma \vdash \lambda x:T_1 \Rightarrow t_{12}[x] : T_1 \Rightarrow T \]

\[ \forall dt_2:\quad \Gamma \vdash t_2 : T_2 \]

\[ \forall d_{sub}\quad \Gamma \vdash T_2 <: T_1 \]

\[ \exists \quad \Gamma \vdash t_{12}[t_2] : T. \]

dt_{12}: \quad \Gamma \vdash t_{12}[t_2] : T \quad \text{by induction on } dt:

case rule

\[ d_1:\quad \Gamma, x:T_1 \vdash t_{12}[x] : T \]

\[ \text{T-Abs} \]

\[ d_2:\quad \Gamma \vdash \lambda x:T_1 \Rightarrow t_{12}[x] : T_1 \Rightarrow T \]

is

\[ dt_2':\quad \Gamma \vdash t_2 : T_1 \quad \text{by rule } \text{T-Sub on } dt_2, \quad d_{sub} \]

\[ dt_{12}: \quad \Gamma \vdash t_{12}[t_2] : T \quad \text{by substitution on } dt_2', \quad d_1 \]

day end case
POPLmark 2A in SASyLF

lemma invert-lambda:
  \[\forall \text{dt}: \neg \exists p | \lambda x: T1 \Rightarrow t12[x] : T \Rightarrow T\]
  \[\forall \text{dt2}: \exists t2 : T2\]
  \[\forall \text{dsub}: \neg \exists t2 <: T1\]
  \[\exists \text{exists}: \neg \exists t12[t2] : T.\]

\[\text{dt12}: \neg \exists t12[t2] : T \text{ by induction on } \text{dt:}\]

\textbf{case rule}
\[\text{d1}: \exists p | \lambda x: T1 \Rightarrow t12[x] : T\]
\[\text{d2}: \exists T'' <: T1 \Rightarrow T\]
\[\text{T-Sub}\]
\[\text{d3}: \exists p | \lambda x: T1 \Rightarrow t12[x] : T \Rightarrow T\]

\textbf{is}
\[\text{d4}: \exists T'' = T1' \Rightarrow T' <: T1 \Rightarrow T \text{ by lemma } \text{arrow-sub-arrow on } \text{d2}\]
\[\text{dt12}: \exists t12[t2] : T \text{ by case analysis on } \text{d4:}\]

\textbf{case rule}
\[\text{d5}: \exists T1 <: T1'\]
\[\text{d6}: \exists T' <: T\]
\[\text{arrow-sub}\]
\[\text{d7}: T1' \Rightarrow T' = T1' \Rightarrow T' <: T1 \Rightarrow T\]

\textbf{is}
\[\text{d8}: \exists T2 <: T1' \text{ by rule } \text{SA-Trans on } \text{dsub, d5}\]
\[\text{d9}: \exists t12[t2] : T' \text{ by induction hypothesis on d1, dt2, d8}\]
\[\text{dt12}: \exists t12[t2] : T \text{ by rule } \text{T-Sub on d9, d6}\]

end case
end case analysis
end case
POPLmark 2A in SASyLF

**Theorem** preservation:

forall dt: * |- t : T

forall ds: t -> t'

exists * |- t' : T.
POPLmark 2A in SASyLF

lemma narrow-subtype: forall dsub: Gamma, X <: T |- T1[X] <: T2[X]
  forall dsub': Gamma |- T' <: T
  exists Gamma, X <: T' |- T1[X] <: T2[X].

... d2: Gamma, X <: T, X' <: T2'[X] |- T1''[X][X'] <: T2''[X][X']

... d8: Gamma, X <: T', X' <: T2'[X] |- T1''[X][X'] <: T2''[X][X']
  by induction hypothesis on d2, dsub'
    // does not currently typecheck
    // because X <: T is not rightmost assumption in d2
    // conceptually an easy fix, but haven’t done it yet

We use by unproved instead (representing an unchecked claim).
POPLmark 2A in SASyLF

dt2: * |- t2 : T1
d1: *, x:T1 |- t12[x] : T
dt12: * |- t12[t2] : T by substitution on dt2, d1 // not yet checked (2 others)

dsub': Gamma |- T' <: T
d3: Gamma, X <: T' |- T' <: T by weakening on dsub' // not yet checked (1 other place)
POPLmark 2A Summary

- Readable to non-experts
  - Many definitions just an ascii-ization
- A few simplifications
  - All of which are also in the Twelf solution
- More verbose: 1105 lines long
  - compare: Twelf solution is 745 lines
- SASyLF implementation limitations
  - 5 unchecked substitution or weakening
  - 1 “unproved” due to a spurious warning

- Other challenges?
  - Challenge 1: need mutual induction
  - Challenge 2B: should be doable (with condition on distinct fields)
  - Challenge 3: requires proof search (known how to do in LF)
Outline

• Motivation and Design Goals
• SASyLF intro: addition commutes
• Semantics: the λ-calculus and LF
• POPLmark 2A in SASyLF
• **Preliminary teaching experience**
• Conclusion
Teaching experience: highlights

- SASyLF used in Spring 2008 graduate Analysis course at CMU
  - One assignment, covering small-step semantics
  - No in-class time was spent teaching the tool

- Results of controlled experiment:
  - 11 of 13 tool users: helped find errors in their proofs
  - 12 of 13 tool users: increased confidence that proofs were correct
  - 14 of 16 no-tool users: wished for earlier feedback on mistakes

- Several students dropped out of tool group
  - Usability issues; many have been fixed
  - 7 of 11 tool users would use the tool again (separate survey)
    - 10 of 11 if usability issues addressed

- Quote: “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.”
SASyLF Limitations

- All the same limitations as Twelf
  - e.g. all theorems in $\forall \exists$ form
- Limitations of student-focused design
  - Second-order abstract syntax
    - no intrinsic encodings
  - More readable, but also more verbose
- Some features not yet implemented
  - some checks (substitution, weakening, exchange)
  - mutual induction
- Open questions
  - World subordination and subsumption
  - Advanced LF idioms supported in Twelf

Our goal is to support most or all of what students need to do in an intro course.

SASyLF is (or will be) good enough for some research applications, but don’t expect the full power of tools designed for researchers.
SASyLF: The Gateway Drug for Mechanized Metatheory

- Educational proof assistant for PL theory
  - Familiar syntax, simple semantics
  - Support for variable binding
  - Capable of sophisticated proofs
  - Early feedback with local error messages

- Usable now in teaching
  - Can use without a big learning curve
  - Preliminary evidence of student benefit
  - Adopted in John Boyland’s current type theory course at U Wisconsin-Milwaukee

- IDE and more features on the way

http://www.sasylf.org/