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 $\quad n \quad ::= \quad z \mid s(n)$

Inference Rules

$$\frac{}{z + n = n}\ \textit{sum-z}$$

$$\frac{n_1 + n_2 = n_3}{s(n_1) + n_2 = s(n_3)}\ \textit{sum-s}$$

```
package edu.cmu.cs.sum;

terminals z s

syntax

n ::= z
  |   s n

judgment sum: n1 + n2 = n3

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

n1 + n2 = n3
------------------------- sum-s
s n1 + n2 = s n3
```

Sum Commutes in SASyLF (Lemma)

Theorem [sum-z-right]: $\forall n \quad n + z = n$

Proof. By induction on n:

Case z**:**

$z + z = z$ *by rule sum-z*

Case $s(n)$**:**

$n + z = n$ *by induction hypothesis*

$s(n) + z = s(n)$ *by rule sum-s*

q.e.d.

```
theorem sum-z-right: forall n
                     exists n + z = n.

d1: n + z = n by induction on n:

case z is
    d2: z + z = z           by rule sum-z
end case

case s n' is
    d3: n' + z = n'         by induction
                            hypothesis on n'

    d4: s n' + z = s n'     by rule sum-s on d3
end case

end induction
end theorem
```

Demonstration: Sum Commutes

See sum.slf in the SASyLF/examples directory

Outline

- Motivation and Design Goals
- SASyLF intro: addition commutes
- **Semantics: the λ-calculus and LF**
- POPLmark 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 - <u>S</u>econd-order <u>A</u>bstract <u>Sy</u>ntax
    - 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
    | "(" ")"
    | e e
    | fn x : tau => e[x]

tau ::= unit
    | tau -> tau
```

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

tau ::= unit
      | tau -> tau
```

x is a case of e,
so variable x has type e.
No constructor created.

e[x] means
x is a variable
that is bound in e

Variables appearing in
clause ignored when
producing constructors

```
e : type.
tau : type.

unit : e.
app: e -> e -> e.
abs: tau -> (e -> e) -> e.
```

second argument of LF constructor
is a function from type of x to e

From SASyLF to LF/Twelf

terminals fn unit	e : type. tau : type.

e ::= x
    | “(” “)”
    | e e
    | fn x : tau => e[x]
	unit : e.
	app: e -> e -> e.
	abs: tau -> (e -> e) -> e.

tau ::= unit | tau -> tau	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
```

```
e : type.
tau : type.

unit : e.
app: e -> e -> e.
abs: tau -> (e -> e) -> e.

tunit : tau.
arrow : tau -> 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])   [x2   ] (e1 x2)

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
```

```
e : type.
tau : type.

unit : e.
app: e -> e -> e.
abs: tau -> (e -> e) -> e.

tunit : tau.
arrow : tau -> tau -> tau.
```

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

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

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
```

```
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])
```

```
abs tunit ([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

```
judgment 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

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

                                           t-fn :{ x1 : e }
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

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

                                             t-fn :{ x1 : e } { dx1 : has-type x1 tau }
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')
```

Case Analysis on Variables

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].

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

case rule
----------------------- SA-Var
d1: Gamma, X <: T |- X <: T

Case: T1[X] is X

is
d2: Gamma, X <: T' |- X <: T' **by rule** SA-Var
d3: Gamma, X <: T' |- T' <: T **by weakening on** dsub'
d4: Gamma, X <: T' |- X <: T **by rule** SA-Trans **on** d2, d3
end case

Case Analysis on Variables

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].

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

case rule
---------------------- SA-Var
d1: Gamma, X <: T |- X <: T
is … end case

Case: T1[X] is X

case rule
--- SA-Var
d1: Gamma2, X' <: T'', X <: T |- X' <: T''
is
d2: Gamma2, X' <: T'' |- X' <: T'' **by rule** SA-Var
d3: Gamma2, X' <: T'', X <: T' |- X' <: T'' **by weakening on** d2
end case

Case: T1[X] is an arbitrary other variable X' from Gamma

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
   |    lambda x:T => t[x]
   |    t t
   |    lambda X <: T => t[X]
   |    t "[" T "]"

T ::=   X
   |    Top
   |    T -> T
   |    all X <: T => T[X]

Gamma ::= *
    | Gamma, x : T
    | Gamma, X <: T
```

```
judgment value: t value

------------------------------ V-Abs
lambda x:T => t[x] value

---------------------------------- V-Tabs
lambda X <: T => t[X] value
```

Limitation: value is a judgment, not a subset of syntax (like Twelf solution)

```
judgment reduce: t -> t

t1 -> t1'
------------------ E-CtxApp1
t1 t2 -> t1' t2

t2 value
----------------------------------------------- E-AppAbs
(lambda x:T11 => t12[x]) t2 -> t12[t2]
```

Limitation: reduction is done with context rules, not explicit evaluation contexts (like Twelf solution)

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
```

Right out of the POPLmark specification

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.

```
--------------------- SA-Refl
Gamma |- T <: T

Gamma |- T1 <: T2
Gamma |- T2 <: T3
------------------------ SA-Trans
Gamma |- T1 <: T3
```

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 dsub: * |- T <: T1 -> T2
      (similar for Tarrow)        exists       T == T1' -> T2'' <: T1 -> T2.

lemma canonical-form-lambda :     forall dtv :  t value
      (similar for type           forall dtt :  * |- t : T1 -> T2
      lambda)                     exists       t == lambda x : T1' => t'[x].

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].

theorem progress :                forall dt :  * |- t : T
                                  exists       t stepsorvalue.
```

POPLmark 2A in SASyLF

```
lemma invert-lambda:    forall dt:   * |- lambda x:T11 => t12[x] : T1 -> T
                        forall dt2:  * |- t2 : T2
                        forall dsub: * |- T2 <: T1
                        exists       * |- t12[t2] : T.

dt12: * |- t12[t2] : T by induction on dt:

case rule
    d1: *, x:T1 |- t12[x] : T
    ---------------------------------------------- T-Abs
    d2: * |- lambda x:T1 => t12[x] : T1 -> T
is
    dt2‘ : * |- t2 : T1 by rule T-Sub on dt2, dsub
    dt12: * |- t12[t2] : T by substitution on dt2’, d1
end case
```

POPLmark 2A in SASyLF

```
lemma invert-lambda:          forall dt:          * |- lambda x:T11 => t12[x] : T1 -> T
                              forall dt2:         * |- t2 : T2
                              forall dsub:        * |- T2 <: T1
                              exists              * |- t12[t2] : T.
dt12: * |- t12[t2] : T by induction on dt:
case rule
    d1: * |- lambda x:T11 => t12[x] : T''
    d2: * |- T'' <: T1 -> T
    --------------------------------------------------- T-Sub
    d3: * |- lambda x:T11 => t12[x] : T1 -> T
is
    d4:   T'' == T1' -> T' <: T1 -> T by lemma arrow-sub-arrow on d2
    dt12:* |- t12[t2] : T             by case analysis on d4:
    case rule
        d5: * |- T1 <: T1'
        d6: * |- T' <: T
        -------------------------------- arrow-sub
        d7: T1' -> T' == T1' -> T' <: T1 -> T
    is
        d8:   * |- T2 <: T1'          by rule SA-Trans on dsub, d5
        d9:   * |- t12[t2] : T'       by induction hypothesis on d1, dt2, d8
        dt12: * |- t12[t2] : T        by rule 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/