Proof-Directed Programming
Twelf - A Case Study

Carsten Schürmann
IT University of Copenhagen
joint work with Frank Pfenning

December 18, 2006

Motto

First there was the invariant,
Then there was the program.

Our Technical Endeavor

Goal: Implement a theorem prover/proof assistant.

Example:

$$\dfrac{\begin{array}{c}\overline{A\ \mathsf{true}}^{\,u} \\ \vdots \\ p\ \mathsf{true}\end{array}}{\neg A\ \mathsf{true}}\ \mathsf{negI}^{p,u} \qquad \dfrac{A\ \mathsf{true} \qquad \neg A\ \mathsf{true}}{C\ \mathsf{true}}\ \mathsf{negE}$$

Task 1: Representation of deductive systems.

- Judgments and evidence [Martin-Löf '98]
- Logical Framework.

Task 2: Reasoning about deductive systems.

- Unification, normalization, theorem proving.

Sample derivation

Lemma: If A true, then $\neg\neg A$ true.

Proof:

$$\dfrac{\dfrac{\dfrac{}{A \text{ true}}\,u \qquad \dfrac{}{\neg A \text{ true}}\,v}{p \text{ true}}\,\text{negE}}{\neg\neg A \text{ true}}\,\text{negI}^{p,v}$$

Logical Frameworks

- Hereditary Harrop formulas.

  Isabelle, λProlog

- λ^{Π} (LF).

  Automath, LF, Elf, Twelf

- Substructural logical frameworks.

  Forum, LLF, OLF

- Equational logic, rewriting.

  Maude, ELAN

- Constructive type theories.

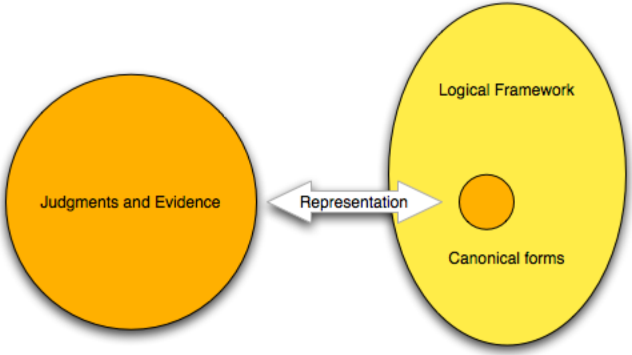
  ALF, Agda, Coq, LEGO, Nuprl

Philosophical Foundation

Judgments-as-types There is a one to one correspondence between constructs of the objects language, and their *canonical* representations in the logical framework. We are only interested in *adequate* representations, where every representation is meaningful.

Judgments-as-propositions There is a *predicate* that states the relevant property that is true about the representation of a construct of the object language. Terms that do not specify the predicate are meaningless.

Representation Methodology



Logical framework LF

- Dependently-typed λ-calculus.
- Function spaces *exclusively* for representation of variables.
- Definitional equality: β, η rules.
- Every term has a canonical (β-normal, η-long) form.
- Therefore: hypothetical judgments.
- Adequacy.
- Judgments encoded by type families *a*.
- Inference rules encoded by object constants *c*.

Logical framework LF (cont'd)

$$\begin{array}{rl} \text{Kinds} & K ::= \mathsf{type} \mid A \rightarrow K \\ \text{Types} & A, B ::= a \mid A \rightarrow B \mid \Pi x : A.\, B \\ \text{Objects} & M, N ::= x \mid c \mid M\ N \mid \lambda x : A.\, M \end{array}$$

Signature

Formulas Encoded as $\mathsf{wff} : \mathsf{type}$ and $\mathsf{neg} : \mathsf{wff} \to \mathsf{wff}$

Judgment $\ulcorner A \text{ true} \urcorner = \mathsf{type}$ and thus $\mathsf{true} : \mathsf{wff} \to \mathsf{type}$

Rules

$$\left\ulcorner \dfrac{\begin{array}{c} \overline{A \text{ true}}^{\,v} \\ \mathcal{D} \\ p \text{ true} \end{array}}{\neg A \text{ true}} \mathsf{negI}^{p,v} \right\urcorner = \mathsf{negI}\ \ulcorner A \urcorner\ (\lambda p : \mathsf{wff}.\ \lambda v : \mathsf{true}\ \ulcorner A \urcorner.\ \ulcorner \mathcal{D} \urcorner)$$

and thus

$$\mathsf{negI} : \Pi A : \mathsf{wff}.\ (\Pi p : \mathsf{wff}.\ \mathsf{true}\ A \to \mathsf{true}\ p) \to \mathsf{true}\ (\mathsf{neg}\ A)$$

Signature (cont'd)

$$\ulcorner \dfrac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ A\ \mathsf{true} & \neg A\ \mathsf{true} \end{array}}{B\ \mathsf{true}}\,\mathsf{negE} \urcorner \quad = \quad \mathsf{negE}\ \ulcorner A \urcorner\ \ulcorner B \urcorner\ \ulcorner \mathcal{D}_1 \urcorner\ \ulcorner \mathcal{D}_2 \urcorner$$

and thus

$$\mathsf{negE} : \Pi A : \mathsf{wff}.\ \mathsf{true}\ A \to \Pi B : \mathsf{wff}.\ \mathsf{true}\ (\mathsf{neg}\ A) \to \mathsf{true}\ B$$

Remark: We can always infer A.

Sample derivation

Lemma: If A true, then $\neg\neg A$ true.

Proof:

$$\dfrac{\dfrac{\dfrac{}{A \text{ true}}\,u \qquad \dfrac{}{\neg A \text{ true}}\,v}{p \text{ true}}\,\text{negE}}{\neg\neg A \text{ true}}\,\text{negI}^{p,v}$$

In LF: The type true $\ulcorner A \urcorner \rightarrow$ true $(\neg\neg\ulcorner A \urcorner)$ is inhabited by the following object:

$$\begin{array}{l}(\lambda u : \text{true}\ulcorner A \urcorner.\ \text{negI}\ \ulcorner A \urcorner \\ \quad (\lambda p : \text{wff}.\ \lambda v : \text{true}\ (\text{neg}\ \ulcorner A \urcorner). \\ \quad\quad \text{negE}\ \ulcorner A \urcorner\ (\text{neg}\ \ulcorner A \urcorner)\ u\ p\ v))\end{array}$$

… and in Twelf

```
wff :  type.
neg :  wff -> wff.

true :  wff -> type.
negI : ({p:wff} true A -> true p) -> true (neg A).
negE : true A -> {B:wff} true (neg A) -> true B.

s :  true A -> true (neg (neg A))
   = [u] negI ([p][v] negE u p v).
```

Algorithms implemented in Twelf

Inference of implicit arguments.
Type checking algorithm.
Type inference algorithm.
Logic programming interpretation.

- Curry Howard isomorphism via proof search.
- Proving a meta theorem = define judgment and rules.

$$\mathcal{D} :: \mathrm{thm}\ (\overset{\mathcal{D}_1}{N\ \mathrm{odd}})\ (\overset{\mathcal{D}_2}{M\ \mathrm{odd}})(\overset{\mathcal{D}_3}{N + M = K})\ (\overset{\mathcal{E}}{K\ \mathrm{even}})$$

Mode checking.
Termination checking.
World checking.
Coverage checking.

Twelf

Implementation LF : One single language for

- representation of deductive systems,
- representation of the meta-theory.

Applications

- Proof-Carrying code. [Necula et al'96, Appel et al'01]
- Proof-Carrying authentication. [Felten et al'00]
- Typed Assembly Language. [Crary'01]
- Logical Relation Proofs. [Sarnat'05]
- Verification of *full* SML's internal language. [Crary et al'07]
- ...

But how did we implement it?

```
% mkdir twelf
% cd twelf
% mkdir src
% cd src
% mkdir lambda
% cd lambda
% xemacs intsyn.sig
```

... and now what?

Remainder of the talk

Proof-directed programming.

Design Decisions.

Case Study: the Twelf implementation.

Anecdotes.

Conclusion.

Proof Directed Programming

Proof Directed Programming

Methodology

Think about the invariants first.

Think about the programs as proof.

Act of Programming

Refine invariants as necessary.

Then refine the code.

Act of Debugging

Don't run a program to understand its behavior.

Don't test!

Think about it! [Harper'99]

Don't run code you haven't verified yourself.

Empirical case study

Twelf is a product of proof directed programming.

Idealized Code Quality Metric

#(all function calls) -
#(recursive calls that correspond to inductive steps) -
#(non-recursive calls that correspond to verified lemmas)

Conjecture: Minimal *idealized code quality metric* implies maximal *code quality*.

- Slow but steady. ("days per line" instead "lines per day")
- Premature optimizations considered harmful.
- Spent as much time on invariants as on code.
- Organized code walks.

Design Decisions

Design decisions

Choice of implementation language.

- Functional programming language.
- Imperative programming language.
- Object-oriented programming language.

Principles

- Respect: Code locality.
- Guidance: Typing system.
- Trust: Your invariants.
- Fear: Destructive update on logical variables.

Design decisions (cont'd)

Choice of variable, constant representation.

- "Named" representation
- de Bruijn encoding [de Bruijn 76]
- Hybrid encoding [Crole et al '02]
- Nominal [Pitts '03]
- Higher-order [Church '40]

Verbosity? Logic variables?

Choice of kinds, types, and expressions.

- Direct.
- Spine calculus. [Cervesato et al. '97]
- Canonical forms. [Watkins '04]
- Explicit substitutions. [Abadi et al '96]
- Pure type systems. [Barendregt '91]

How much information to represent?

What role do normal forms play?

Design decisions (cont'd)

Programming a proof assistant is a constraint satisfaction problem

Closed world assumption

- Code extensions, new features, and new developments invalidate old choices.
- Keep in mind: This is a historical talk about 1997.
- Discard your code often and rewrite!

How to solve this dilemma?

- Experience.
- Learn from the experts.
- Ask an oracle.

Case Study: the Twelf implementation.

Design Decisions

de Bruijn indices:

2 instead of y

Explicit substitutions: (simple types)

$$A, B, C, D \vdash 3.1.\uparrow^4 : B, D$$

instead of

$$a : A, b : B, c : C, d : D \vdash b/x, d/y : x : B, y : D.$$

Dependent types:

$$A, B, C \vdash 2 : B[\uparrow^2]$$

Spine notation:

$$negi\ (\ulcorner A \urcorner; (neg\ \ulcorner A \urcorner); u; p; v; \mathrm{nil})$$

instead of

$$(((((negi\ \ulcorner A \urcorner)\ (neg\ \ulcorner A \urcorner))\ u)\ p)\ v)$$

Syntactic Categories, Internal Syntax

Variable	de Bruin indices k.
Constant	indices into an array c.
Logic variable	$X^{\Gamma,V}$
Head	$H ::= c \mid k$.
Level	$L ::= \text{type} \mid \text{kind}$.
Expression	$U, V, W ::= \lambda V.U \mid \Pi V.W \mid H \cdot S \mid U \cdot S \mid X^{\Gamma,V} \mid L \mid U[\sigma]$
Spine	$S ::= \text{nil} \mid U; S \mid S[\sigma]$
Substitution	$\sigma ::= F.\sigma \mid \uparrow^k$
Front	$F ::= U \mid k$
Gamma	$\Gamma ::= \cdot \mid \Gamma, V$

Typing Judgment: Substitutions and Spines

$\boxed{\Gamma \vdash \sigma : \Gamma'}$

$$\frac{}{\Gamma, V_k \ldots V_1 \vdash \uparrow^k : \Gamma}\, shift \qquad \frac{\Gamma \vdash F : V[\sigma] \quad \Gamma \vdash \sigma : \Gamma'}{\Gamma \vdash F.\sigma : \Gamma', V}\, dot$$

$\boxed{\Gamma \vdash S : V \gg W}$

$$\frac{}{\Gamma \vdash \mathrm{nil} : V \gg V}\, nil \qquad \frac{\Gamma \vdash U : V \quad \Gamma \vdash S : W[U.\uparrow^0] \gg V'}{\Gamma \vdash U; S : \Pi V.\, W \gg V'}\, app$$

$$\frac{\Gamma \vdash \sigma : \Gamma' \quad \Gamma' \vdash S : V \gg W}{\Gamma \vdash S[\sigma] : V[\sigma] \gg W[\sigma]}\, sclo$$

Typing Judgments: Heads and Expressions

$\boxed{\Gamma \vdash H : V}$

$$\frac{}{\Gamma, V_k \ldots V_1 \vdash k : V_k[\uparrow^k]}\, var \qquad \frac{\Sigma(c) = V}{\Gamma \vdash c : V}\, const$$

$\boxed{\Gamma \vdash U : V}$

$$\frac{\Gamma, V \vdash U : W}{\Gamma \vdash \lambda V.\, U : \Pi V.\, W}\, lam \qquad \frac{\Gamma, V \vdash W : U(L)}{\Gamma \vdash \Pi V.\, W : U(L)}\, pi$$

$$\frac{\Gamma \vdash H : V \quad \Gamma \vdash S : V \gg W}{\Gamma \vdash H \cdot S : W}\, root \qquad \frac{\Gamma \vdash U : V \quad \Gamma \vdash S : V \gg W}{\Gamma \vdash U \cdot S : W}\, redex$$

$$\frac{}{\Gamma \vdash \mathrm{type} : \mathrm{kind}}\, type \qquad \frac{}{\Gamma \vdash X^{\Gamma, V} : V}\, evar \qquad \frac{\Gamma \vdash \sigma : \Gamma' \quad \Gamma' \vdash U : V}{\Gamma \vdash U[\sigma] : V[\sigma]}\, eclo$$

Back to the sample derivation

Lemma: If A true, then $\neg\neg A$ true.

Proof:

$$\dfrac{\dfrac{\dfrac{}{A \text{ true}}\,u \qquad \dfrac{}{\neg A \text{ true}}\,v}{p \text{ true}}\,\text{negE}}{\neg\neg A \text{ true}}\,\text{negI}^{p,v}$$

Internal:
$$\begin{aligned}
\#1 &= \text{wff} \\
\#2 &= \text{neg} \\
\#3 &= \text{true} \\
\#4 &= \text{negI} \\
\#5 &= \text{negE}
\end{aligned}$$

$$\begin{aligned}
s \quad &: \quad \Pi\#3 \cdot (A; nil).\ \#3 \cdot (\#2 \cdot (\#2 \cdot (A; nil); nil); nil) \\
&= \quad \lambda\#3 \cdot (A; nil).\ (\#4 \cdot (A; \lambda(\#1\ nil).\ \lambda(\#3 \cdot (\#2 \cdot (A; nil); nil)). \\
&\qquad\qquad \#5 \cdot (A; (\#2 \cdot (A; nil)); (3; nil); (2; nil); (1; nil); nil)))
\end{aligned}$$

Derived rules of inference

Substitution expansion

$$\begin{array}{l}\text{If } \Gamma \vdash \sigma : \Gamma' \\ \text{then } \Gamma, V[\sigma] \vdash 1.\sigma \circ \uparrow : \Gamma', V\end{array} \quad \text{(dot1)}$$

```
fun dot1 (s as Shift (0)) = s
  | dot1 s = Dot (Idx(1), comp(s, shift))
```

Admissible rules of inference

Subsitution composition

If $\Gamma \vdash \sigma : \Gamma'$
and $\Gamma' \vdash \sigma' : \Gamma''$
then $\Gamma \vdash \sigma' \circ \sigma : \Gamma''$. (comp)

```
fun comp (Shift (0), s) = s
  | comp (s, Shift (0)) = s
  | comp (Shift (n), Dot (Ft, s)) = comp (Shift (n-1), s)
  | comp (Shift (n), Shift (m)) = Shift (n+m)
  | comp (Dot (F, s), s') = Dot (fSub(F, s'), comp (s, s'))
```

Example: Type inference

Invariant `inferExp` $(\Gamma, U) \hookrightarrow V'$

If U is in whnf

and $\Gamma \vdash U : V$

then $\Gamma \vdash V \equiv V'$

otherwise exception `Error` is raised.

Unfortunately

One cannot prove it directly.

Therefore

Generalize invariant!

Example: Type inference (cont'd)

Generalization to accommodate explicit substitutions

Invariant

`inferExp` $(\Gamma, (U, \sigma)) \hookrightarrow (V', \sigma')$

If	U is in whnf
and	U contains no logical variables
and	$\Gamma \vdash \sigma : \Gamma_1$
and	σ contains no logical variables
and	$\Gamma_1 \vdash U : V$
then	there exists a substitution σ'
	and $\Gamma \vdash \sigma' : \Gamma'$
	and $\Gamma' \vdash V' : L$
	such that $\Gamma \vdash V[\sigma] \equiv V[\sigma'] : L$

otherwise exception `Error` is raised.

Example: Type inference (cont'd)

```
fun inferExpW (G, (Uni (L), _)) =
    (Uni (inferUni L), id)
  | inferExpW (G, (Pi ((D, _) , V), s)) =
    (checkDec (G, (D, s));
    inferExp (Decl (G, decSub (D, s)), (V, dot1 s)))
  | inferExpW (G, (Root (C, S), s)) =
    inferSpine (G, (S, s), whnf (inferCon (G, C), id))
  | inferExpW (G, (Lam (D, U), s)) =
    (checkDec (G, (D, s));
    (Pi ((decSub (D, s), Maybe),
      EClo (inferExp (Decl (G, decSub (D, s)),
        (U, dot1 s)))), id))
```

Comments

- Explicit substitutions and spines pervasively used in Twelf implementation.
- Pleasant organizing force.
- We'll justify some of choices through anecdotal evidence.

Anecdotes

Anecdote 1: Unification

Head clash $(c \cdot S)[\sigma] \approx (c \cdot S')[\tau]$ if and only if $S[\sigma] \approx S'[\tau]$.

Spine calculus exposes head!

Higher-order unification problems

$$(\lambda V_1.\, U_1)[\sigma] \approx (\lambda V_2.\, U_2)[\tau]$$
$$\text{if and only if}$$
$$V_1[\sigma] \approx V_2[\tau] \text{ and } U_1[1.\sigma\circ \uparrow] \approx U_2[1.\tau\circ \uparrow]$$

Eta expansion invariant!

Closures $(U[\sigma])[\sigma'] \approx U'[\tau]$ if and only if $U[\sigma \circ \sigma'] \approx U'[\tau]$.

Explicit substitutions.

Anecdote 1: Unification (cont'd)

Logic Variables $X^{\Gamma,V}[\sigma] \approx U[\tau]$ iff $X^{\Gamma,V} := U[\sigma \circ \tau^{-1}]$

Problem: τ^{-1} doesn't always exists.

Consider pattern substitutions. [Miller '91]

Postpone none-pattern equations as constraints.

Observation: We can make τ^{-1} always exists that cannot always be applied:

Example:

$$(3.5.1.\uparrow^5)^{-1} = 3.\_.1.\_.5\uparrow^3$$

[unpublished]

Front $F ::= U \mid k \mid \_$

$$\frac{}{\Gamma \vdash \_ : V}\ \mathit{undef}$$

Observation: Failure of inversion can be pushed into substitutions.

Anecdote 2: Type Variables

Type reconstruction: Turn `[u] negI ([p][v] negE u p v)` into

$$(\lambda u : \mathsf{true}\ulcorner A \urcorner.\ \mathsf{negI}\ \ulcorner A \urcorner\ (\lambda p : \mathsf{wff}.\ \lambda v : \mathsf{true}\ (\mathsf{neg}\ \ulcorner A \urcorner).\ \mathsf{negE}\ \ulcorner A \urcorner\ (\mathsf{neg}\ \ulcorner A \urcorner)\ u\ p\ v))$$

Unification invariant The terms are fully η-expanded.

But Unknown types of omitted arguments.

Thus No type level logic variables.

Solution Two phase algorithm. [Harper et al'02]

1. Approximate types.
2. Reconstruct erased indices.

Anecdote 3: Logic Variables

Observation Type inference total on canonical forms.

Idea Let $X^{\Gamma, V}$ logic variable. Γ can always be derived.

And thus
```
datatype Exp =
   ...
   | EVar of (Exp option ref * Exp)
   | ...
```

But... Let's look at the abstraction algorithm.

Anecdote 3: Logic Variables (cont'd)

Abstraction Pi-closure of free variables in declarations.

Example Reconstruction of leading omitted `{A:wff}` for

```
negE : true A -> {B:wff} true (neg A) -> true B.
```

Observation In general, we need to access Γ.

$$\cdot \vdash A : \mathit{type} \quad \text{under free logic variables } K, X^{\Gamma, V}$$

Abstraction Algorithm

1. First, we form a type B, by replacing all $X := \lambda\Gamma.\, 1 \cdot (n; n-1; \ldots 1; \mathrm{nil})$.
2. Second by induction hypothesis on K and $\Pi(\Pi\Gamma.\, V).\, B : \mathrm{type}$, compute the closed pi-closure.

Anecdote 3: Logic Variables (cont'd)

But Explicit substitutions and dependencies showstopper!

Recall Invariant of type inference.

$$\Gamma' \vdash \sigma : \Gamma \quad \text{and} \quad \Gamma \vdash X : V$$

Problem Given σ and non-empty Γ.

- $\sigma = F.\sigma'$ by assumption.
- $\Gamma \vdash F : V[\sigma']$ by inversion.
- $\Gamma \vdash F : W$ by type inference.
- $V = W[\sigma'^{-}1]$ only if σ invertible.

Thus First version of Twelf was incomplete.

Moral We spent too much time on doing the wrong thing.

Anecdote 4: On Explicit Substitutions

Explicit substitutions: [Dowek,Hardin,Kirchner,Pfenning'96]

$$\sigma, \tau \mid F \cdot \sigma \mid \sigma \circ \tau \mid id \mid \uparrow$$

Question: declared connectives vs. defined connectives.
Twelf implementation:

$$\text{Normal form:} \sigma ::= F \cdot \sigma \mid \uparrow^n$$

Weakening substitutions.

$$\omega ::= 1.\omega \circ \uparrow \mid \omega \circ \uparrow \mid id$$

Compact normal forms.
Which connectives to take primitive? [CS'01]

Anecdote 5: Defined Object Constants

Problem When to expand notational definitions?

Crucial Equality algorithms, e.g. unification.

Definition $d = U$ is semantically transparent iff

$$d \cdot S \equiv d \cdot S' \quad \text{if and only if} \quad S \equiv S'$$

[Pfenning, CS'98]

Requirement All arguments d must occur in rigid positions.

Example $d = \lambda c : \mathrm{wff} \rightarrow \mathrm{wff}.\, \lambda p : \mathrm{wff}.\, c \cdot (p; \mathit{nil})$ is not valid.

Example $\mathit{neg} = \lambda p : \mathrm{wff}.\, \mathrm{imp} \cdot (p; \mathrm{false}; \mathrm{nil})$; is valid.

Quiz:

- How many bugs did we have in the initial compilation of Twelf?
- What kind of bugs here they?
- Where there any soundness bugs?

Conclusion

- Proof directed implementation worked well.
- Required: a few dry runs.
- You need to want to strive for beauty.
- Settle foundations, the rest will fall in place.
- Spines + explicit substitutions are organizing the code.
- What I said here worked also for
  - world checking,
  - termination checking,
  - mode checking,
  - or coverage checking.
- We still need to do a codewalk for the next release.