Effective Tool Support for the Working Semanticist

Peter Sewell* Francesco Zappa Nardelli


* University of Cambridge

3rd ParSec Meeting, 13 December 2007
A programming language is a rather large body of new and somewhat arbitrary mathematical notation introduced in the hope of making the problem of controlling computing machines somewhat simpler.

Strachey, *Formal Language Description Languages*, 1964 (1st IFIP Working Conference)
Problem

Semantic definitions of programming languages are:

(a) vitally necessary

(b) (i) rather easy, for a little calculus (e.g. untyped λ)
(ii) for a full-scale programming language, rather tricky.
   Almost *no* full-scale languages have complete definitions, still!

Standard ML (1990)
Java? C♯? Haskell? OCaml?

Why? Partly because the *metalanguages* we have for writing semantics make it much harder than necessary.
Available metalanguages

Option 1: \texttt{\LaTeX}, the (usual) metalanguage of choice for informal maths

\checkmark \hspace{1cm} \text{production-quality typesetting}

but, at this scale (100s of pages or 10 000 lines of definition):

\xmark \hspace{1cm} \text{syntactic overhead of \LaTeX\ markup — prohibitive obfuscation}

\xmark \hspace{1cm} \text{no automatic checking of sanity properties}

\xmark \hspace{1cm} \text{no support for conformance checking}

\xmark \hspace{1cm} \text{informal proof is unreliable and hard to maintain}

(we tried this for Acute — an 80 page defn — with rudimentary tool support)
\begin{lemma}[Inversion -- expressions]
\vspace*{-3ex}
\begin{enumerate}[1.]
\item If $\Actx{\AntConSet_0}{\Gamma} \vdash_{\updVar} \Lrecord{l_1=e_1, \ldots, l_n=e_n} : \Atype{l_1: \tau_1, \ldots, l_n: \tau_n}{\AntConSet_n}$
then
$\exists \tau_1', \ldots, \tau_n', \AntConSet_1', \ldots, \AntConSet_{n-1}', \AntConSet_n' \supseteq \AntConSet_n$
such that
$\forall i \in 1..n$ we have $\Gamma \vdash \tau_i' <: \tau_i$
and
$\Actx{\AntConSet_{i-1}'}{\Gamma} \vdash_{\updVar} e_i : \Atype{\tau_i'}{\AntConSet_i'}$
\item If $\Actx{\AntConSet_0}{\Gamma} \vdash_{\updVar} e_1 \; e_2 : \Atype{\tau_2}{\AntConSet}$
then $\exists \tau_2', \AntConSet_1, \AntConSet_2, \hat{\AntConSet}'$
such that
$\Actx{\AntConSet_0}{\Gamma} \vdash_{\updVar} e_1 : \Atype{\tau_1 \Auarrow{\hat{\updVar}}{\hat{\AntConSet}} \tau_2}{\AntConSet_1}$
and
$\Actx{\AntConSet_1}{\Gamma} \vdash_{\updVar} e_2 : \Atype{\tau_1}{\AntConSet_2}$
and $\AntConSet' \subseteq \AntConSet_2$ and $(\hat{\updVar} = \Aupd \Rightarrow (\updVar = \Aupd \land \AntConSet' \subseteq \hat{\AntConSet}')$.
\end{enumerate}
\end{lemma}
Available metalanguages

Option 1: \LaTeX, the (usual) metalanguage of choice for informal maths

✓ production-quality typesetting

but, at this scale (100s of pages or 10 000 lines of definition):

× syntactic overhead of \LaTeX markup — prohibitive obfuscation
× no automatic checking of sanity properties
× no support for conformance checking
× informal proof is unreliable and hard to maintain

(we tried this for Acute — an 80 page defn — with rudimentary tool support)
Available metalanguages

Option 2: Coq/HOL/Isabelle/Twelf, for formal mathematics.

- automatic checking of sanity properties, and of proofs

but

- limited typesetting (maintain both in sync?)
- syntactic noise (hard to read & edit the sources)
- nontrivial encodings (subgrammars, binding, limited inductive defns,...)
- steep learning curves
- community partition

Both make it hard to re-use definitions, or compose from fragments.
Our Solution

A metalanguage specifically designed for writing semantic definitions, and a tool, **ott** that:

- takes an ASCII definition of a language syntax and semantics, supporting good concise notation — close to what we would write in informal mathematics;
- compiles it into *\LaTeX*, Coq, HOL, and Isabelle versions of the definition, and OCaml boilerplate code (a Twelf backend is under development);
- builds a parser and pretty-printer (into L/C/H/I) for symbolic and concrete terms of the defined language.
- supports an expressive (but still simple and intuitive) language for specifying binding structures.
Tested With Substantial Case Studies

(1) small lambda calculi: untyped, simply typed (*), and with ML polymorphism, all CBV;

(2) TAPL systems — booleans, naturals, functions, base types, units, seq, ascription, lets, fix, products, sums, tuples, records, variants; (*)

(3) Leroy’s JFP module system, with a term language and operational semantics [Scott];

(4) combination of separation logic with rely-guarantee reasoning [Matt];

(5) Lightweight Java (*), and Java module system proposals based on JSR 277 and JSR 294 (∼140 semantic rules) [Rok]; and

(6) a large fragment of OCaml (∼265 semantic rules) (*)
Part 1: Introduction: The Dream

Part 2: Metalanguage overview

Part 3: Compilation to proof assistant code

Part 4: Binding specifications

Part 5: Case studies

Part 6: Conclusion
Example: Untyped CBV $\lambda$, in $\texttt{ott}$

$$
t \ ::= \quad \text{term}
| \ x \quad \text{variable}
| \ \lambda x . t \quad \text{bind } x \text{ in } t \quad \text{lambda}
| \ t \ t' \quad \text{app}

v \ ::= \quad \text{value}
| \ \lambda x . t \quad \text{lambda}

\hline
\begin{align*}
t_1 & \rightarrow t_2 \\
(\lambda x . t_{12}) v_2 & \rightarrow \{v_2 / x\} t_{12} & \text{AX} \\
t_1 & \rightarrow t'_1 \\
t_1 t & \rightarrow t'_1 t & \text{CTXL}
\end{align*}

\hline
\begin{align*}
t_1 & \rightarrow t'_1 \\
v t_1 & \rightarrow v t'_1 & \text{CTXR}
\end{align*}

$$
metavar termvar, x ::= 

grammar 
t ::= \t_\ ::= 
\mid x ::= :: Var
\mid \ x . t ::= :: Lam
\mid t t' ::= :: App
\mid ( t ) ::= M ::= Paren
\mid \{ t / x \} t' ::= M ::= Tsub

v ::= \v_ ::= 
\mid \ x . t ::= :: Lam

subrules v <:: t

defns Jop ::= ::= 
defn t1 --> t2 ::= :: reduction ::= by

-------------------- :: ax -------------------- :: ctxL -------------------- :: ctxR
(\x.t1) v2 --> \{v2/x\}t1
v t1 --> v t1'
t1 t --t t1' t
What’s going on?

That can already be parsed, sanity-checked, and generate default typesetting.

No built-in assumptions on the form of the syntax or semantics — the tool simply lets you define syntax and relations over it.

Basic entities: metavariables, nonterminals, terminals (implicit), and judgements

The grammar specifies

- the concrete syntax of object-language terms
- the abstract syntax for proof assistant representations
- the *symbolic terms* used in semantic rules

Context-free grammar.
What’s going on?

Consider that innocent-looking CBV rule

\[
\text{ax_app} (\lambda x. t_{12}) v_2 \rightarrow \{v_2/x\} t_{12}
\]

- symbolic metavariables (\(x\)) and nonterminals (\(t_{12}, v_2\)), built from roots (\(x, t, v\)) and structured suffixes (primes, indices).
- rigid naming convention finds errors and disambiguates
- subtype relation, declared \(v :<: t\), and checked, allowing \(v_2\) to appear in a \(t\) position.
- metasyntax for parentheses and substitution (\(M\) productions)
- syntax for judgement forms \(t_1 \rightarrow t_2\) and formulae
- parsing with scannerless memoized CPS’d parser combinators
- find all parses, flag error if \(\geq 1\)
**LaTeX eye-candy**

metavar termvar, x ::= \{ tex \textit{[[termvar]]} \}

**grammar**

\( t :: 't_' ::= \)

\( | \ x :: : : \text{Var} \)

\( | \ \backslash \ \ x \ . \ t :: : : \text{Lam} \)

\( | \ t \ t' :: : : \text{App} \)

\( | \ ( \ t ) :: \text{M} :: \text{Paren} \)

\( | \ { \ t / x \ } \ t' :: \text{M} :: \text{Tsub} \)

\( v :: 'v_-' ::= \)

\( | \ \backslash \ \ x \ . \ t :: : : \text{Lam} \)

\( \text{terminals} :: : '\text{terminals_}' ::= \)

\( | \ \backslash :: : \lambda \ \{ \ \text{tex} \ \lambda \ \} \)

\( | \ --\rightarrow :: : \text{red} \ \{ \ \text{tex} \ \text{longrightarrow} \ \} \)

**subrules** v <:: t

**defs** Jop :: ::= 

\( \text{defn} \ \ t1 \ \rightarrow \ \ t2 :: : : \text{reduction} :: \)

\( \begin{align*}
\longrightarrow & :: \text{ax} \\
\longrightarrow & :: \text{ctxL} \\
\longrightarrow & :: \text{ctxR}
\end{align*} \)

\( (\backslash \ x . \ t1) v2 \rightarrow \{v2/x\}t1 \)

\( v \ t1 \rightarrow v \ t1' \)

\( t1 \ t \rightarrow t1' \ t \)
metavar termvar, x ::= \{\{ tex \texttt{mathit{[[termvar]]}} \} \} \{\{ com term variable \}\}

grammar

\[ t ::= 't_' ::= \{\{ com term \}\} \]
\[ | x ::= :: \text{Var} \{\{ com variable\}\} \]
\[ | \backslash x . t ::= :: \text{Lam} \{\{ com lambda \}\} \]
\[ | t \ t' ::= :: \text{App} \{\{ com app \}\} \]
\[ | ( t ) ::= M :: \text{Paren} \{\{ com app \}\} \]
\[ | \{ t / x \} t' ::= M :: \text{Tsub} \{\{ com lambda \}\} \]

v ::= 'v_' ::= \{\{ com value \}\}
\[ | \backslash x . t ::= :: \text{Lam} \{\{ com lambda \}\} \]

subrules v <:: t

defs Jop :: ::=
\[ \text{defn } t1 \rightarrow t2 :: :: \text{reduction} :: \{\{ com [[t1]] reduces to [[t2]]\}\} \text{ by} \]

\[ \begin{align*}
\text{------------------------} & : \text{ctxL} \\
(\backslash x . t1) v2 \rightarrow & \{ v2 / x \} t1 \\
\text{--------------} & : \text{ctxR} \\
t1 \rightarrow t1' & : \text{ax} \\
t1 \rightarrow t1' & : \text{ctxL} \\
v t1 \rightarrow v t1' & : \text{ax} \\
t1 \ t \rightarrow t1' \ t & : \text{ctxR}
\end{align*} \]
already enough to generate decent typesetting

\begin{align*}
t & ::= \quad \text{term} \\
& \mid x \quad \text{variable} \\
& \mid \lambda x . t \quad \text{bind } x \text{ in } t \quad \text{lambda} \\
& \mid t \ t' \quad \text{app} \\

v & ::= \quad \text{value} \\
& \mid \lambda x . t \quad \text{lambda} \\

\frac{t_1 \rightarrow t_2}{t_1 \text{ reduces to } t_2} \\

\frac{(\lambda x . t_{12}) v_2 \rightarrow \{ v_2 / x \} t_{12}}{\text{AX}} \\

\frac{t_1 \rightarrow t'_1}{t_1 \ t \rightarrow t'_1 \ t} \quad \text{CTXL} \\

\frac{t_1 \rightarrow t'_1}{v \ t_1 \rightarrow v \ t'_1} \quad \text{CTXR}
\end{align*}

and \LaTeX filtering, e.g. writing \[
[(\x.\x'.t1)t2] \]
in \LaTeX source
metavar termvar, x ::= 

grammar

\[ t :: 't_' ::= \]
\[ | x :: Var \]
\[ | \ x . t :: Lam (+ bind x in t +) \]
\[ | t t' :: App \]
\[ | ( t ) :: M :: Paren \]
\[ | \{ t / x \} t' :: M :: Tsub \]

v :: 'v_' ::= 
\[ | \ x . t :: Lam \]

subrules v <:: t

defs Jop ::= ::= 
\[ \text{defn } t1 \rightarrow t2 :: :: \text{reduction} :: \]
\[ \text{by} \]
\[ \text{--------------------------} :: ax \]
\[ \text{--------------------------} :: ctxL \]
\[ \text{--------------------------} :: ctxR \]
\[ (\ x . t1 \) v2 \rightarrow \{v2/x\}t1 \]
\[ v t1 \rightarrow v t1' \]
\[ t1 t \rightarrow t1' t \]
metavar termvar, x ::= 
  {{ isa string}} {{ coq nat}} {{ hol string}} {{ coq-equality }}

grammar

\[
\begin{align*}
  t & ::= \text{'}t_{-}\text{'} ::= \\
  & | x :: Var \\
  & | \ \ x . t ::= \text{Lam (+ bind } x \text{ in } t +) \\
  & | t \ t' ::= \text{App} \\
  & | ( t ) ::= M ::= \text{Paren} \quad \{\text{ ich } [[t]] \} \\
  & | \{ t / x \} t' ::= M ::= \text{Tsub} \quad \{\text{ ich (tsubst}_t [[t]] [[x]] [[t']])\} \\

  v & ::= \text{'v_{-}' ::=} \\
  & | \ \ x . t ::= \text{Lam} \\

\end{align*}
\]

subrules v <:: t 
  substitutions single t x :: tsubst

defns Jop : :: = 
  defn t1 --> t2 :: :: reduction :: by 

  \[
  \begin{align*}
  & \text{------------------------} \quad \text{:: ax} \quad \text{--------------------} \quad \text{:: ctxL} \quad \text{--------------------} \quad \text{:: ctxR} \\
  & (\x.t1) v2 --> {v2/x}t1 \\
  & v t1 --> v t1' \\
  & t1 t --> t1' t
  \end{align*}
  \]
**Key Idea**

Users define not just the concrete/abstract syntax of the object language, but syntax used in semantic rules, including

- **metaproductions**

  \[
  t ::= 't' ::= \\
  \ldots \\
  | ( t ) :: M :: Paren \{ ich [[[t]]] \} \\
  | \{ t / x \} t' :: M :: Tsub \{ ich (tsubst_t [[[t]]] [[[x]]] [[[t']]])\}
  \]

- and productions of a *formula grammar* (by default, just the judgement forms).

The *homs* say what these mean — clauses of primitive recursive functions from symbolic terms to the character-strings of generated \LaTeX/Coq/HOL/Isabelle code.

This (and subrules) means the metalanguage can be relatively simple.
Similarly for productions, and \LaTeX. For example, for $F_\prec$: you could annotate:

\[
| x : T . t :: :: \text{Lam} \\
\{\{ \text{tex} \ \lambda[[x]] \ \text{mathord}{::} [[[T]]. \ \mathbin{\,,} [[[t]]] }\}\}
\]

\[
| X \prec: T . t :: :: T\text{Lam} \\
\{\{ \text{tex} \ \Lambda[[X]] \ \text{mathord}{\prec:} [[[T]]. \ \mathbin{\,,} [[[t]]] }\}\}
\]

to typeset terms such as $(\langle X\prec:T_{11} . x:X . t_{12}\rangle) [T_2]$ as $(\Lambda X \prec:T_{11} . \lambda x:X . t_{12}) [ T_2 ]$.

Similarly for type homs to use non-free proof assistant types.
What you get: code for our favourite theorem prover

Definition termvar_t := nat.
Lemma eq_termvar_t: forall (x y : termvar_t), {x = y} + {x <> y}. Proof. decide equality. Defined.

Inductive t_t : Set :=
  t_Var : termvar_t -> t_t
| t_Lam : termvar_t -> t_t -> t_t
| t_App : t_t -> t_t -> t_t.

Definition is_v (t0:t_t) : Prop :=
  match t0 with
  | (t_Var x) => False
  | (t_Lam x t) => (True)
  | (t_App t t') => False end.

Fixpoint tsubst_t (t0:t_t) (termvar0:termvar_t) (t1:t_t) {struct t1} : t_t :=
  match t1 with
  | (t_Var x) => if eq_termvar_t x termvar0 then t0 else (t_Var x)
  | (t_Lam x t) => t_Lam x (if list_mem eq_termvar_t termvar0 (cons x nil) then t else (tsubst_t t0 termvar0 t))
  | (t_App t t') => t_App (tsubst_t t0 termvar0 t) (tsubst_t t0 termvar0 t’) end.

Inductive E : t_t -> t_t -> Prop :=
  ax : forall v2 x t12, is_v v2 -> E (t_App T v2) ( tsubst_t v2 x t12 )
| ctxL : forall t1 t1’ t, E t1 t1’ -> E (t_App t1 t) (t_App t1’ t)
| ctxR : forall t1 v t1’, is_v v -> E t1 t1’ -> E (t_App v t1) (t_App v t1’).
What you get: code for another theorem prover (Isa)

theory out = Main:
types termvar = "string"

datatype t =
    t_Var "termvar"
  | t_Lam "termvar" "t"
  | t_App "t" "t"

consts is_v :: "t => bool"
primrec
  "is_v ((t_Var x)) = False"
  "is_v ((t_Lam x t)) = (True)"
  "is_v ((t_App t t')) = False"

consts tsubst_t :: "t => termvar => t => t"
primrec
  "tsubst_t t0 termvar0 (t_Var x) = (if x=termvar0 then t0 else (t_Var x))"
  "tsubst_t t0 termvar0 (t_Lam x t) = (t_Lam x (if termvar0 mem [x] then t else (tsubst_t t0 termvar0 t)))"
  "tsubst_t t0 termvar0 (t_App t t') = (t_App (tsubst_t t0 termvar0 t) (tsubst_t t0 termvar0 t'))"

consts E :: "(t*t) set" inductive E
intros
axI: "[|is_v v2|] ==> ((t_App T v2), (tsubst_t v2 x t12)) : E"
ctxLI: "[| (t1, t1') : E|] ==> ((t_App t1 t), (t_App t1' t)) : E"
ctxRI: "[|is_v v ; (t1, t1') : E|] ==> ((t_App v t1), (t_App v t1')) : E"
end
What you get: code for yet another theorem prover (HOL)

```plaintext
val _ = new_theory "out";

val _ = type_abbrev("termvar", '"string"');
val _ = Hol_datatype '  
t = t_Var of termvar
  | t_Lam of termvar => t
  | t_App of t => t ';

val _ = ottDefine "is_v_of_t" '  
  ( is_v_of_t (t_Var x) = F)
  /
  ( is_v_of_t (t_Lam x t) = (T))
  /
  ( is_v_of_t (t_App t t') = F) ';

val _ = ottDefine "tsubst_t" '  
  ( tsubst_t t5 x5 (t_Var x) = (if x=x5 then t5 else (t_Var x)))
  /
  ( tsubst_t t5 x5 (t_Lam x t) = t_Lam x (if MEM x5 [x] then t else (tsubst_t t5 x5 t)))
  /
  ( tsubst_t t5 x5 (t_App t t') = t_App (tsubst_t t5 x5 t) (tsubst_t t5 x5 t')) ';

val (Jop_rules, Jop_ind, Jop_cases) = Hol_reln '  
  ( (* ax *) !(x:termvar) (t12:t) (v2:t) .
    ((is_v_of_t v2))
    ==>
    ( ( reduce (t_App (t_Lam x t12) v2) ( tsubst_t v2 x t12 )))
  )
  /
  ...
val _ = export_theory ();
```

...etc
Lists: a more typical example rule, from Not-so-mini-Caml

\[ E \vdash e_1 : t_1 \ldots E \vdash e_n : t_n \]
\[ E \vdash \text{field\_name}_1 : t \rightarrow t_1 \ldots E \vdash \text{field\_name}_n : t \rightarrow t_n \]
\[ t = (t'_1, \ldots, t'_l) \text{typeconstr\_name}_n \]
\[ E \vdash \text{typeconstr\_name} \triangleright \text{typeconstr\_name} : \text{kind} \{ \text{field\_name}_1'; \ldots; \text{field\_name}_m' \} \]
\[ \text{field\_name}_1 \ldots \text{field\_name}_n \textbf{PERMUTES} \text{field\_name}_1' \ldots \text{field\_name}_m' \]
\[ \text{length}(e_1) \ldots (e_n) \geq 1 \]

\[ E \vdash \{ \text{field\_name}_1 = e_1; \ldots; \text{field\_name}_n = e_n \} : t \]
Lists: symbolic terms involving projection and concatenation

\[ t :: 't_ ::= \{ \{ \text{com term } \} \} \]
\[ | \{ l_1 = t_1 , \ldots , l_n = t_n \} :: :: \text{Rec} \{ \{ \text{com record } \} \} \]

\[ \frac{ \{ l'_1 = v_1 , \ldots , l'_n = v_n \} \cdot l'_j \rightarrow v_j }{ t \rightarrow t' } \quad \text{PROJ} \]
\[ \frac{ t \rightarrow t' }{ \{ l_1 = v_1 , \ldots , l_m = v_m , l = t , l'_1 = t'_1 , \ldots , l'_n = t'_n \} \rightarrow \{ l_1 = v_1 , \ldots , l_m = v_m , l = t' , l'_1 = t'_1 , \ldots , l'_n = t'_n \} } \quad \text{REC} \]

Comprehensions — with explicit index \( i \) and bounds \( 0 \) to \( n - 1 \)

\[ \frac{ \Gamma \vdash t : \{ \overline{l_i : T_i} \}_{i \in 0..n-1} }{ \Gamma \vdash t . l_j : T_j } \quad \text{PROJ} \]

or with \( 1..n \), unspecified, or upper-only bounds)
Part 1: Introduction: The Dream
Part 2: Metalanguage overview
Part 3: Compilation to proof assistant code
Part 4: Binding specifications
Part 5: Case studies
Part 6: Conclusion
So what does that mean?
the language definition in your favorite prover

ottoml -isabelle test10.thy -coq test10.v -hol test10Script.sml test10.ott

Compile to proof assistant definitions of:

• Types
• Subrule predicates
• Auxiliary functions (from bindspecs)
• Free variable functions
• Substitutions
• Inductive definitions of judgements

Aim to be: (a) well-formed, without dangling proof obligations, (b) idiomatic — a good basis for later proof. Don’t want to have to edit the generated code!
Types

- **type abbrevs** for metavars and non-free rules (with type homs);

  ```
  types termvar = "string"
  ```

  (not so simple — Coq equality)

- **free types** for other rules;

  ```
  datatype
t =
    t_Var "termvar"
  | t_Lam "termvar" "t"
  | t_App "t" "t"
  ```

  (n.s.s. — only for the top elements of the subrule order)

  (n.s.s. — take care of p.a. lexical conventions and reserved words (e.g. ‘t’))

- **dependency analysis** and topological sort (trivial for this example)

  (n.s.s. — watch out for rules with type homs)
Subrules

\[
\begin{align*}
\text{t1} & \rightarrow \text{t1}' \\
\text{v} & \rightarrow \text{v} \text{t1}' \\
\text{t} & \rightarrow \text{t1}'
\end{align*}
\]

- Declare subrule relationship \( v <::: t \) in source;
- Ott checks that it holds (for all productions of rules on left of \(<::: \), exists a production of upper bound which — up to \(<::: \) — is a superproduction);
- \textit{representation choice}:
  1. generate separate type with injections, or
  2. generate \textit{is} \textit{v} predicates etc.
- so generate the definition of \textit{is} \textit{v}
  - In general more complex — eg if expr isn’t free.
- then, anyplace where we use a nonterm of a sub-type in a rule, we \textbf{add a premise} \textit{is} \textit{v}.
Subrules (n.s.s.)

Generated Isabelle:

```isabelle
consts
  is_v_of_t :: "t => bool"

primrec
"is_v_of_t (t_Var x) = (False)"
"is_v_of_t (t_Lam x t) = ((True))"
"is_v_of_t (t_App t t') = (False)"
```

- Coq Definition/Fixpoint; HOL ottDefine; Isabelle primrec
- for each non-free type, and following dependencies
- define by pattern matching (pp of canonical symterm of production)
- each clause is a disjunction across subproductions, of a conjunction across subterms
Subrules (really n.s.s.)

1) Isabelle \texttt{primrec} can’t cope with nested patterns. Tuples are nested pairs, so for $\geq$ ternary constructors, need to define extra functions.

2) If $\texttt{foo} <: t$, $\texttt{baz} <: t$, and $t$, $\texttt{foo}$, and $\texttt{baz}$ mention each other, need multiple different functions over the same type ($\texttt{is\_foo, is\_baz : t -> bool}$).

Isabelle \texttt{primrec} can’t do that, so must do tuple encoding, define intended functions by projections, generate statements of characterisation lemmas, and generate (trivial) proof scripts.
Inductive definition rules


Use explicit mutual recursion structure from the user.

Each definition rule gives rise to an implication, that the premises (Ott formulas) imply the conclusion (an Ott symbolic term of the judgement being defined).

Inductive reduce : t -> t -> Prop :=
  | ax_app : forall (x:termvar) (t12:t) (v:t),
    is_v_of_t v ->
    reduce (t_App (t_Lam x t12) v) (tsubst_t v x t12)
  | ctx_app_fun : forall (t1:t) (t_5:t) (t1’:t),
    reduce t1 t1’ ->
    reduce (t_App t1 t_5) (t_App t1’ t_5)
  | ctx_app_arg : forall (v:t) (t1:t) (t1’:t),
    is_v_of_t v ->
    reduce t1 t1’ ->
    reduce (t_App v t1) (t_App v t1’).
List support

t :: 't_' ::= {{ com term }}
| { l1 = t1 , ... , ln = tn } :: :: Rec {{ com record }}

We’d like to use nested list types, e.g. as in the HOL (nice and easy):

```latex
val _ = Hol_datatype ' 
Typ =
   T_Var of typevar 
| T_Top
| T_Fun of Typ => Typ
| T_Forall of typevar => Typ => Typ
| T_Rec of (label#Typ) list
';
```
List support (n.s.s.)

But then:

HOL: need smarter `ottDefine` to define functions

Isabelle: for `primrec`, need to define additional functions for list cases, can’t nest. If multiple occurrences of `list`, need multiple copies of additional functions.

Coq: choice

- native lists
  - Coq can’t decide termination of functions — Soln: local fixpoints, obfusc
  - Coq-synthesised `ind. prin.` for inductive defns too weak — Soln: synthesise a better one

- build list types (also needed for Twelf)
  - need to build lots (list types arising from rules)
  - maybe awkward for proofs
List support — rules

The set of symbolic terms of the definition rule are analysed together to find list forms with the same bounds. A single proof assistant variable is introduced for each such, with appropriate projections and list maps/foralls at the usage points.

\[
\Gamma \vdash t_1 : T_1 \quad \ldots \quad \Gamma \vdash t_n : T_n \\
\Gamma \vdash \{ l_1 = t_1, \ldots, l_n = t_n \} : \{ l_1 : T_1, \ldots, l_n : T_n \}
\]

\( \text{TY}_{RCD} \)

\( \text{TY}_{RCD} \)
List support in Coq (\texttt{ott}-defined lists)

- A grammar definition involving dots (excerpt from $F_\prec$):

  \[
  T :: 'T' ::= \quad \text{\{ com type \}}
  \ |
  \{ l1 : T1 , \ldots , ln : Tn \} :::: \text{Rec} \quad \text{\{ com record \}}
  \]

- introduces an auxiliary list type:

  ```coq
  Inductive list_label_T : Set :=
  Nil_list_label_T : list_label_T
  | Cons_list_label_T : label -> T -> list_label_T
  -> list_label_T
  
  with
  
  T : Set :=
  | T_Rec : list_label_T -> T.
  ```
List support in Coq (\texttt{ottt}-defined lists, ctd.)

- And a type rule involving dots:

\[
\begin{align*}
G &\vdash T_1 \ldots G \vdash T_n \\
\hline
& : \text{Rcd} \\
G &\vdash l_1:T_1, \ldots, l_n:T_n
\end{align*}
\]

- is defined as:

\[
\begin{align*}
\text{GT} &: G \to T \to \text{Prop} := \\
& | \text{GT}_{\text{Rcd}} : \forall l_1:T_1\text{list} G_5, \\
& \quad \text{GT\_list} (\text{make\_list\_G\_T} \\
& \quad \quad (\text{map\_list\_label\_T} \\
& \quad \quad \quad (\text{fun} (l_1: \text{label}) (T_1:T) \Rightarrow (G_5,T_1)) \\
& \quad \quad l_1:T\text{list}))) \to \\
& \quad \text{GT} G_5 (T_{\text{Rec}} l_1:T\text{list})
\end{align*}
\]

with

\[
\begin{align*}
\text{GT\_list} &: \text{list\_G\_T} \to \text{Prop} := \\
& | \text{Nil\_GT\_list} : \text{GT\_list} \text{Nil\_list\_G\_T} \\
& | \text{Cons\_GT\_list} : \forall G_5 T_1 \text{list} \text{G\_T}, \\
& \quad \text{GT} G_5 T_1 \to \text{GT\_list} \text{G\_T} \\
& \quad \text{GT\_list} (\text{Cons\_list\_G\_T} G_5 T_1 \text{G\_T} T_1) \\
\end{align*}
\]

NB: the type rule requires the defn of another auxiliary list type \text{GT\_list}. 
List support in Coq (native lists)

Alternatively,

- rely on native polymorphic lists:

  ```coq
  Inductive T : Set :=
  | TyRecord : list (label*T) -> T.
  ```

- generate the appropriate induction principle:

  ```coq
  Section T_rect.
  Variable P : T -> Type.
  Variable Q : list (label*T) -> Type.
  Hypotheses
  (f3 : forall l : list (label * T), Q l -> P (TyRecord l)).

  Fixpoint T_rect' (T0 : T) : P T0 :=
  match T0 return (P T0) with
  | TyRecord l =>
    f3 ((fix phi (l:list (label*T)) {struct l} : Q l :=
        match l return Q l with
        | nil => g
        | ((l0,T0)::l') =>
          (h l0 (T_rect' T0:P T0) (phi l':Q l')
           : Q ((l0,T0)::l'))
        end) l)
  end.
  End T_rect.
  ```

- careful use of local fixpoints to convince Coq that the functions terminate
Part 1: Introduction: The Dream
Part 2: Metalanguage overview
Part 3: Compilation to proof assistant code
Part 4: Binding specifications
Part 5: Case studies
Part 6: Conclusion
...and Binding

- For now, generating a **fully-concrete representation** for variables. Fine for semantics of whole programs (really, those that don’t shadow std library), as we only do subst of (almost) closed values.

- say in source what binds in what via the **bindspec language**;

- generated substitution functions;
  - don’t subst for $x$ under a binder for $x$!

- can bind metavars or (!) nonterms

- substitution functions have to match the inductive structure of syntax.
  - and *deal with coq/isa pickiness*. 
Bindspec language

Snippets inside (+ and +) belong to the bindspec language. Two kind of defns:

- **bind mse in nonterm**
  declares the metavariables and nonterminals denoted by mse are binding in nonterm

- **auxfn = mse**
  lets the user define *auxiliary functions* to collect selected mvrs and ntrs of subterms

A subset of the mse grammar (also support list forms here):

\[
\text{mse} ::= \text{metavar} \\
| \text{nonterm} \\
| \text{auxfn(nonterm)} \\
| \text{mse union mse'}
\]
**Bindspec example: patterns**

```plaintext
var X :: termvar

exp ::= X
    | λ X . exp  bind X in exp
    | exp exp'    bind b(pat) in exp'
    | ( exp , exp' )
    | let pat = exp in exp' bind b(pat) in exp'

pat ::= X b = X
    | _ b = {}
    | ( pat , pat' ) b = b(pat) ∪ b(pat')
```

Example: let ( x , y ) = z in x y with its pat subterm ( x , y )
Bindspec example: let rec

\[
\begin{align*}
\text{var } X &:: \text{ termvar} \\
\text{exp} &::= X \\
& \quad | \quad () \\
& \quad | \quad ( \text{exp} , \text{exp}' ) \\
& \quad | \quad \text{let rec } X = \text{exp in } \text{exp}' \\
& \quad | \quad \text{bind } X \text{ in } \text{exp} \\
& \quad | \quad \text{bind } X \text{ in } \text{exp}'
\end{align*}
\]

Here the scope of a binder is two distinct subterms.

Example: \textbf{let rec } X = ( X , Y ) \textbf{ in } ( X , Y )
Binding examples

In practice, the bindspec language is fairly expressive.

Among the examples we built:

- multiple `let rec` with multiple clauses and argument patterns;
- or patterns;
- join calculus definitions.
Binding — what does it mean?
Substitutions...

Often need to use (single/multiple) substitutions (not always, e.g. LJ)

- Say in source what substitution functions we need

```
substitutions
  single t x :: tsubst
  single T X :: Tsubst
  multiple t x :: m_t_subst
```

Here `single t x` builds substitution functions that replace `singleton productions x in the t grammar` by ts, and recursively where dependency analysis requires, over each syntactic type.

```
consts
tsubst_t :: "t => termvar => t => t"
primrec
  "tsubst_t t5 x5 (t_Var x) = ((if x=x5 then t5 else (t_Var x)))"
  "tsubst_t t5 x5 (t_Lam x t) = (t_Lam x (if x5 mem [x] then t else (tsubst_t t5 x5 t)))"
  "tsubst_t t5 x5 (t_App t t') = (t_App (tsubst_t t5 x5 t) (tsubst_t t5 x5 t'))"
```

- users define whatever syntax they want, then refer to the generated functions in isa/coq/hol homs using `meta productions`:

```
  | { t / x } t' :: M :: Tsub  {{ icho (tsubst_t [[t]] [[x]] [[t']])}}
```
Substitutions...

n.s.s.: have to deal with similar function definition issues as before
Part 1: Introduction: The Dream
Part 2: Metalanguage overview
Part 3: Compilation to proof assistant code
Part 4: Binding specifications
Part 5: Case studies
Part 6: Conclusion
Does it really work?

(1) small lambda calculi: untyped, simply typed (*), and with ML polymorphism, all CBV;

(2) TAPL systems — booleans, naturals, functions, base types, units, seq, ascription, lets, fix, products, sums, tuples, records, variants; (*)

(3) Leroy’s JFP module system, with a term language and operational semantics [Scott];

(4) combination of separation logic with rely-guarantee reasoning [Matt];

(5) Lightweight Java, and Java module system proposals based on JSR 277 and JSR 294 (∼140 semantic rules) [Rok]; and

(6) a large fragment of OCaml (∼265 semantic rules) (*)
Modular semantics: \texttt{ott} source for the TAPL \texttt{let} fragment

\begin{verbatim}
grammar
  t :: Tm ::=          {{ com terms: }}
  | let x = t in t' :: Let (+ bind x in t' +)  {{ com let binding }}

defns
  Jop ::= '': ::= 

  defn
  t --> t' ::= :: red :: E. {{ com Evaluation }} by

  ------------------------------- :: LetV
  let x=v1 in t2 --> [x|->v1]t2

  t1 --> t1'

  ------------------------------- :: Let
  let x=t1 in t2 --> let x=t1' in t2

defns
  Jtype ::= '': ::= 

  defn
  G |- t : T ::= :: typing :: T. {{ com Typing }} by

  G |- t1:T1
  G,x:T1 |- t2:T2

  ------------------------------- :: Let
  G |- let x=t1 in t2 : T2
\end{verbatim}
## Does it really work? (ctd.)

<table>
<thead>
<tr>
<th>file(s)</th>
<th>system</th>
<th>(\LaTeX)</th>
<th>Isabelle</th>
<th>Coq</th>
<th>HOL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>defns</td>
<td>mt</td>
<td>defns</td>
<td>mt</td>
<td></td>
</tr>
<tr>
<td>test10.ott</td>
<td>untyped CBV lambda</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test10st.ott</td>
<td>simply typed CBV lambda</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test8.ott</td>
<td>ML polymorphism</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>sys-bool</td>
<td>TAPL - boolean values</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sys-arith</td>
<td>TAPL - integers</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sys-puresimple</td>
<td>TAPL - simply typed lambda</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sys-tybool</td>
<td>TAPL - stl + bool</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sys-sortoffullsimple</td>
<td>TAPL - ... lots ...</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sys-tuple</td>
<td>TAPL - tuples</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sys-puresub</td>
<td>TAPL - subtypes</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sys-purercdsusb</td>
<td>TAPL - record subtypes</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sys-roughlyfullsimple</td>
<td>TAPL - most but subtyping</td>
<td>√</td>
<td>√</td>
<td>√*</td>
<td></td>
</tr>
<tr>
<td>test7a.ott</td>
<td>POPLmark (F_{&lt;}):</td>
<td>√</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>test7.ott</td>
<td>POPLmark (F_{&lt;}): with records</td>
<td>√</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>LJ</td>
<td>Lightweight Java</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LJm</td>
<td>Lightweight Java/JSR 277</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LJp</td>
<td>Lightweight Java/JSR 294</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>leroy-jfp96</td>
<td>Modules (Leroy '96)</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>not-so-mini-caml</td>
<td>OCaml fragment</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(1) The generated proof assistant \(F_{<}\): definitions are well-formed but the concrete variable representation in the generated code is not satisfactory here — this version of the type system disallows all shadowing, so typing is not preserved by reduction.

Part 1: Introduction: The Dream
Part 2: Metalanguage overview
Part 3: Compilation to proof assistant code
Part 4: Binding specifications
Part 5: Case studies
Part 6: Conclusion
Related work — tool support

- The Munger (Wansbrough), bla (Fairbairn), TTP (Peskine)
- Coq/HOL/Isa/Twf: various fancy syntax and typesetting support
- $\alpha$Prolog (Cheney, Urban), MLSOS (Lakin, Pitts)
- C$\alpha$ml (Pottier, 05): generate OCaml code for ASTs with binding
- Sugar (Tse, Zdancewic, 05): generate parser and ASTs for concrete terms
- PLT Redex (Matthews, Findler, Flatt, Felleisen, 04)
- TinkerType (Pierce, Levin, 00)
- CLaReT (Boulton, 98): parser, pp, and HOL defns from den.sem.
- ASF+SDF
- Animator Generator (Berry, 91)
- Centaur (Kahn et al, 88)
- Synthesizer Generator (Reps, Teitelbaum, 83)
Related work — a sample of large-scale language definitions

- Revised\textsuperscript{5.92} Scheme (PLT Redex) (07)
- an ML internal language (Twelf) (Lee, Crary, Harper, 07)
- a Java-like language (Isabelle) (Klein, Nipkow, 06)
- Haskell98-ish static semantics (\LaTeX) (Faxén, 02)
- C expressions (HOL) (Norrish, 98)
- various TALs
- ...

Ott development

Next release:

- a shiny *fast* GLR parser (by Scott Owens) (testing)
- grammars of term contexts, generation of context application (done)
- Isabelle2007 target (ongoing with isa07 team)

Very soon:

- compiling to *locally nameless* quotiented binding representation
- function definitions, not just relations (presently embed)

Work in progress:

- compiling to *nominal logic* in Isabelle
- multiple overlapping languages, and syntactic sugar
- distinctness predicates
Conclusion — The Hope

We hope this will enable a phase change, making it feasible, without heroic effort, to routinely work with rigorous semantic definitions of realistic languages (and interesting calculi).

Further, we hope it will make it easy to

- build semantics modularly,
- exchange definitions of calculi and languages, and fragments thereof, across the community,
- move smoothly from informal maths to rigorous definitions,
- investigate more uniform proof assistant support for proof.

The tool, documentation, and examples are available at 
http://moscova.inria.fr/~zappa/software/ott
(ocaml code, BSD-style licence). Try it!