Ott

Effective Tool Support for the Working Semanticist

Peter Sewell* Francesco Zappa Nardelli

with Scott Owens*, Gilles Peskine*, Tom Ridge*, Susmit Sarkar*, Rok Strnisa*.

* University of Cambridge

3rd ParSec Meeting, 13 December 2007

Problem: What is a Programming Language?

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$^\sharp$? Haskell? OCaml?

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

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)

```
\begin{lemma}[Inversion -- expressions]
~\\ \vspace*{-3ex}
\begin{enumerate}[1.]
\item  If $\Actx{\AntConSet_0}{\Gamma}
           \vdash_\updVar \Lrecord{\Llabel{l}_1=e_1,\dots,\Llab
           : \Atype{\Lrecord{\Llabel{l}_1:\tau_1,\dots,\Llabel{
       then
    $\exists \tau_1',\ldots,\tau_n',\AntConSet_1',\dots,\AntCon
    $\forall i \in 1..n$ we have $\Gamma \vdash \tau_i' <: \tau
    $\Actx{\AntConSet_{i-1}'}{\Gamma}
    \vdash_\updVar e_i
    : \Atype{\tau_i'}{\AntConSet_i'}$
\item If $\Actx{\AntConSet_0}{\Gamma} \vdash_\updVar e_1 \; e_2
      then $\exists \tau_2', \AntConSet_1,\AntConSet_2,\hat{\An
      $\Actx{\AntConSet_0}{\Gamma} \vdash_\updVar e_1 : \Atype{
     and
```

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 ($\sim$140 semantic rules) [Rok]; and

(6) a large fragment of OCaml ($\sim$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 λ, in `ott`

$termvar$, x term variable

t	$::=$			term
	$\mid$	x		variable
	$\mid$	$\lambda\, x\, .\, t$	bind x in t	lambda
	$\mid$	$t\ t'$		app
v	$::=$			value
	$\mid$	$\lambda\, x\, .\, t$		lambda

$\boxed{t_1 \longrightarrow t_2}$ t_1 reduces to t_2

$$\frac{}{(\,\lambda\, x\, .\, t_{12}\,)\ v_2 \longrightarrow \{\, v_2\,/\,x\,\}\ t_{12}}\ \textsc{Ax}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t \longrightarrow t_1'\ t}\ \textsc{ctxL}$$

$$\frac{t_1 \longrightarrow t_1'}{v\ t_1 \longrightarrow v\ t_1'}\ \textsc{ctxR}$$

What You Write: Core

```
metavar termvar, x ::=



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

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

subrules v <:: t

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

                                t1 --> t1'                  t1 --> t1'
------------------------ :: ax  -------------- :: ctxL      -------------- :: ctxR
(\x.t1) v2 -->  {v2/x}t1        v t1 --> v t1'              t1 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

```
---------------------------  :: ax_app
(\x.t12) v2 -->  {v2/x}t12
```

- symbolic metavariables (`x`) and nonterminals (`t12`, `v2`), built from roots (`x`, `t`, `v`) and structured suffixes (primes, indices).
- rigid naming convention finds errors and disambiguates
- subtype relation, declared `subrules v <:: t`, and checked, allowing `v2` to appear in a `t` position.
- metasyntax for parentheses and substitution (`M` productions)
- syntax for judgement forms `t1 --> t2` and formulae
- parsing with scannerless memoized CPS'd parser combinators
- find all parses, flag error if ≥ 1

LaTeX eye-candy

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



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

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

subrules v <:: t

terminals :: 'terminals_' ::=
  | \   :: :: lambda {{ tex \lambda }}
  | --> :: :: red {{ tex \longrightarrow }}

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

                                   t1 --> t1'                  t1 --> t1'
------------------------  :: ax    -------------- :: ctxL      -------------- :: ctxR
(\x.t1) v2 -->  {v2/x}t1           v t1 --> v t1'              t1 t --> t1' t
```

LaTeX comments

```
metavar termvar, x ::=   {{ tex \mathit{[[termvar]]} }}  {{ com  term variable }}


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

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

subrules v <:: t

defns Jop ::  ::=
 defn t1 --> t2 :: :: reduction ::  {{ com [[t1]] reduces to [[t2]]}} by

                                       t1 --> t1'                t1 --> t1'
------------------------  :: ax        -------------- :: ctxL    -------------- :: ctxR
(\x.t1) v2 -->  {v2/x}t1               v t1 --> v t1'            t1 t --> t1' t
```

already enough to generate decent typesetting

$termvar, x$	term variable		
t	$::=$		term
	$\mid$	x	variable
	$\mid$	$\lambda\, x\, .\, t$ bind x in t	lambda
	$\mid$	$t\; t'$	app
v	$::=$		value
	$\mid$	$\lambda\, x\, .\, t$	lambda

$\boxed{t_1 \longrightarrow t_2}$ t_1 reduces to t_2

$$\frac{}{(\,\lambda\, x\, .\, t_{12}\,)\, v_2 \;\longrightarrow\; \{\, v_2\, /\, x\,\}\, t_{12}} \quad \text{AX}$$

$$\frac{t_1 \;\longrightarrow\; t_1'}{t_1\; t \;\longrightarrow\; t_1'\; t} \quad \text{CTXL}$$

$$\frac{t_1 \;\longrightarrow\; t_1'}{v\; t_1 \;\longrightarrow\; v\; t_1'} \quad \text{CTXR}$$

and LaTeX filtering, e.g. writing `[[(\x.\x'.t1)t2]]` in LaTeX source

Binding

```
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

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

                                     t1 --> t1'                 t1 --> t1'
 ------------------------ :: ax      -------------- :: ctxL     -------------- :: ctxR
 (\x.t1) v2 -->  {v2/x}t1            v t1 --> v t1'             t1 t --> t1' t
```

Making it mean something: Coq, HOL, and Isabelle

```
metavar termvar, x ::=
    {{ isa string}} {{ coq nat}} {{ hol string}} {{ coq-equality }}

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

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

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

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

                                     t1 --> t1'              t1 --> t1'
------------------------  :: ax      -------------- :: ctxL  -------------- :: ctxR
(\x.t1) v2 -->  {v2/x}t1             v t1 --> v t1'          t1 t --> t1' t
```

Key Idea

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

- metaproductions

```
t :: 't_' ::=
  ...
  | ( 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 $\mathsf{F}_{<:}$ you could annotate:

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

| \ X <: T . t   :: :: TLam
  {{ tex \Lambda [[X]] \mathord{<:} [[T]]. \, [[t]] }}
```

to typeset terms such as `(\X<:T11.\x:X.t12) [T2]` as $(\,\Lambda X{<:}T_{11}.\,\lambda x{:}X.\,t_{12}\,)\ [\,T_2\,]$.

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)

```
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

$$\dfrac{\begin{array}{l} E \vdash e_1 : t_1 \quad \ldots \quad E \vdash e_n : t_n \\ E \vdash \mathit{field\_name}_1 : t \rightarrow t_1 \quad \ldots \quad E \vdash \mathit{field\_name}_n : t \rightarrow t_n \\ t = (\, t'_1 \,,\, \ldots \,,\, t'_l \,)\ \mathit{typeconstr\_name} \\ E \vdash \mathit{typeconstr\_name} \ \triangleright\ \mathit{typeconstr\_name} : \mathit{kind}\ \{\, \mathit{field\_name}'_1 \,;\, \ldots ;\, \mathit{field\_name}'_m \,\} \\ \mathit{field\_name}_1 \ldots \mathit{field\_name}_n\ \mathbf{PERMUTES}\ \mathit{field\_name}'_1 \ldots \mathit{field\_name}'_m \\ \mathbf{length}\,(\, e_1 \,) \ldots (\, e_n \,) \geq 1 \end{array}}{E \vdash \{\, \mathit{field\_name}_1 = e_1 \,;\, \ldots ;\, \mathit{field\_name}_n = e_n \,\} : t}$$

```
E |- e1 : t1 ... E |- en : tn
E |- field_name1 : t->t1 ... E |- field_namen : t->tn
t = (t1', ..., tl') typeconstr_name
E |- typeconstr_name gives typeconstr_name:kind {field_name1'; ...; field_name
field_name1...field_namen PERMUTES field_name1'...field_namem'
length (e1)...(en)>=1
------------------------------------------------------------------------ ::
E |- {field_name1=e1; ...; field_namen=en} : t
```

Lists: symbolic terms involving projection and concatenation

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

$$\frac{}{\{\, l'_1 = v_1 \,,\, ..\,,\, l'_n = v_n \,\} \,.\, l'_j \;\longrightarrow\; v_j}\quad \text{PROJ}$$

$$\frac{t \;\longrightarrow\; t'}{\begin{array}{l}\{\, l_1 = v_1 \,,\, ..\,,\, l_m = v_m \,,\, l = t \,,\, l'_1 = t'_1 \,,\, ..\,,\, l'_n = t'_n \,\} \\ \longrightarrow \{\, l_1 = v_1 \,,\, ..\,,\, l_m = v_m \,,\, l = t' \,,\, l'_1 = t'_1 \,,\, ..\,,\, l'_n = t'_n \,\}\end{array}}\quad \text{REC}$$

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

```
ott -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

```
                                              t1 --> t1'              t1 --> t1'
--------------------------  :: ax           -------------- :: ctxL  -------------- :: ctxR
(\x.t12) v2 -->  {v2/x}t12                  v t1 --> v t1'          t1 t --> t1' t
```

- 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);
- *representation choice*:
  1. generate separate type with injections, or
  2. generate is_v predicates etc.
- so generate the definition of `is_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 add a premise `is_v`.

Subrules (n.s.s.)

Generated 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 `primrec` can't cope with nested patterns. Tuples are nested pairs, so for $\geq$ ternary construtors, need to define extra functions.

2) If `foo<::t`, `baz<::t`, and `t`, `foo`, and `baz` mention each other, need multiple different functions over the same type (`is_foo`, `is_baz : t -> bool`).

Isabelle `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

Coq: `Inductive`, HOL: `Hol_reln`, Isa: `inductive`.

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):

```
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 `t 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.

$$\frac{\Gamma \vdash t_1 : T_1 \quad .. \quad \Gamma \vdash t_n : T_n}{\Gamma \vdash \{\, l_1 = t_1 \,,\, .. ,\, l_n = t_n \,\} : \{\, l_1 : T_1 \,,\, .. ,\, l_n : T_n \,\}} \quad \text{TY\_RCD}$$

```
/\ ( (* Ty_Rcd *)  !(l_t_Typ_list : (label#t#Typ) list) (G:G) .
(((EVERY (\b.b) ((MAP (\(l_,t_,Typ_) . (Ty G t_ Typ_)) l_t_Typ_list)))
 ==>
((Ty G (t_Rec ((MAP (\(l_,t_,Typ_) . (l_,t_)) l_t_Typ_list)))
       (T_Rec ((MAP (\(l_,t_,Typ_) . (l_,Typ_)) l_t_Typ_list))) )))
```

List support in Coq (`ott`-defined lists)

- A grammar definition involving dots (excerpt from $F_{<:}$):

```
T :: 'T_' ::=                                  {{ com type }}
  | { l1 : T1 ,  ... , ln : Tn } :: :: Rec   {{ com record }}
```

- introduces an auxiliary list type:

```
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 (ott-defined lists, ctd.)

- And a type rule involving dots:

```
G |- T1 ... G |- Tn
---------------------- :: Rcd
G |- l1:T1,...,ln:Tn
```

- is defined as:

```
GT : G -> T -> Prop :=
 | GT_Rcd : forall l_T_list G5,
     (GT_list (make_list_G_T
                 (map_list_label_T
                    (fun (l_:label) (T_:T) => (G5,T_))
                    l_T_list))) ->
     GT G5 (T_Rec l_T_list)

with

GT_list : list_G_T -> Prop :=
 | Nil_GT_list : GT_list Nil_list_G_T
 | Cons_GT_list : forall G5 T_ l',
     GT G5 T_ -> GT_list l' ->
     GT_list (Cons_list_G_T G5 T_ l')
```

NB: the type rule requires the defn of another auxiliary list type `GT_list`.

List support in Coq (native lists)

Alternatively,

- rely on native polymorphic lists:

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

- generate the appropriate induction principle:

```
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):

$$
\begin{array}{lcl}
\textit{mse} & ::= & \mathit{metavar} \\
 & | & \mathit{nonterm} \\
 & | & \mathit{auxfn}\,\texttt{(}\mathit{nonterm}\texttt{)} \\
 & | & \mathit{mse}\ \ \texttt{union}\ \ \mathit{mse}'
\end{array}
$$

Bindspec example: patterns

var X :: $termvar$

$$
\begin{array}{lcll}
exp & ::= & X & \\
 & | & \lambda\ X\ .\ exp & \text{bind } X \text{ in } exp \\
 & | & exp\ exp' & \\
 & | & (\ exp\ ,\ exp'\ ) & \\
 & | & \textbf{let}\ pat\ =\ exp\ \textbf{in}\ exp' & \text{bind } b(pat) \text{ in } exp' \\
 & & & \\
pat & ::= & X & b = X \\
 & | & \_ & b = \{\} \\
 & | & (\ pat\ ,\ pat'\ ) & b = b(pat) \cup b(pat')
\end{array}
$$

Example: **let** $(\ x\ ,\ y\ )\ =\ z$ **in** $x\ y$ with its pat subterm $(\ x\ ,\ y\ )$

Bindspec example: `let rec`

var X :: $termvar$

$$
\begin{array}{lcll}
exp & ::= & X & \\
 & | & () & \\
 & | & (\ exp\ ,\ exp'\ ) & \\
 & | & \textbf{let rec}\ X\ =\ exp\ \textbf{in}\ exp' & \text{bind } X \text{ in } exp \\
 & & & \text{bind } X \text{ in } exp'
\end{array}
$$

Here the scope of a binder is two distinct subterms.

Example: **let rec** $X\ =\ (\ X\ ,\ Y\ )$ **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 `t`s, 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 ($\sim$140 semantic rules) [Rok]; and

(6) a large fragment of OCaml ($\sim$265 semantic rules) (*?)

Modular semantics: `ott` source for the TAPL `let` fragment

```
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
```

Does it really work? (ctd.)

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

(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.

(*) Metatheory in progress. Proofs by S. Owens, G. Peskine, T. Ridge, R Strniša.

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
- αProlog (Cheney, Urban), MLSOS (Lakin, Pitts)
- Cαml (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$^{5.92}$ 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!