JFP **20** (1): 71–122, 2010.
doi:10.1017/S0956796809990293

Ott: Effective tool support for the working semanticist

PETER SEWELL[1], FRANCESCO ZAPPA NARDELLI[2], SCOTT OWENS[1], GILLES PESKINE[1], THOMAS RIDGE[1], SUSMIT SARKAR[1] and ROK STRNIŠA[1]

[1]*University of Cambridge, Computer Laboratory, William Gates Building, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom and* [2]*INRIA Paris-Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France*
(*e-mail:* Peter.Sewell@cl.cam.ac.uk)

Abstract

Semantic definitions of full-scale programming languages are rarely given, despite the many potential benefits. Partly this is because the available *metalanguages* for expressing semantics – usually either LaTeX for informal mathematics or the formal mathematics of a proof assistant – make it much harder than necessary to work with large definitions. We present a metalanguage specifically designed for this problem, and a tool, Ott, that sanity-checks such definitions and compiles them into proof assistant code for Coq, HOL, and Isabelle/HOL, together with LaTeX code for production-quality typesetting, and OCaml boilerplate. The main innovations are (1) metalanguage design to make definitions concise, and easy to read and edit; (2) an expressive but intuitive metalanguage for specifying binding structures; and (3) compilation to proof assistant code. This has been tested in substantial case studies, including modular specifications of calculi from the TAPL text, a Lightweight Java with Java JSR 277/294 module system proposals, and a large fragment of OCaml (OCaml_{light}, 310 rules), with mechanised proofs of various soundness results. Our aim with this work is to enable a phase change: making it feasible to work routinely, without heroic effort, with rigorous semantic definitions of realistic languages.

1 Introduction

Problem Writing a precise semantic definition of a full-scale programming language is a challenging task that has been done only rarely, despite the many potential benefits. Indeed, Standard ML remains, 20 years after publication, the shining example of a language that is defined precisely and is at all widely used (Milner *et al.* 1990). The recent R[6]RS Scheme standard (Sperber *et al.* 2007) contains a (non-normative) operational semantics for a large part of the language, but even languages such as Haskell (Peyton Jones 2003) and OCaml (Leroy *et al.* 2005), though designed by programming language researchers and in large part based on mathematical papers, rely on prose descriptions.

This article is an expansion of a paper presented at ICFP 2007 in Freiburg.

Precise semantic definitions are rare for several reasons, but one important reason is that the *metalanguages* that are available for expressing semantic definitions are not designed for this application, making it much harder than necessary to work with large definitions. There are two main choices for a metalanguage:

1. Informal mathematics, expressed in LaTeX (by far the most common option).
2. Formalised mathematics, in the language of a proof assistant such as Coq, HOL, Isabelle/HOL, or Twelf (Coq 2008; HOL 2007; Isabelle 2008; Twelf 2005).

For a small calculus either can be used without much difficulty. A full language definition, however, might easily be 100 pages or 10,000 lines. At this scale the syntactic overhead of LaTeX markup becomes very significant, getting in the way of simply reading and writing the definition source. The absence of automatic checking of sanity properties becomes a severe problem – in our experience with the Acute language (Sewell *et al.* 2004; Sewell *et al.* 2007a), just keeping a large definition internally syntactically consistent during development is hard, and informal proof becomes quite unreliable, as highlighted by the POPLmark challenge (Aydemir *et al.* 2005). Further, there is no support for relating the definition to an implementation, either for generating parts of an implementation or for testing conformance. Accidental errors are almost inescapable (Kahrs 1993; Rossberg 2001).

Proof assistants help with automatic checking, but come with their own problems. The sources of definitions are still cluttered with syntactic noise, non-trivial encodings are often needed (e.g. to deal with subgrammars and binding, and to work around limitations of the available polymorphism and inductive definition support), and facilities for parsing and pretty printing terms of the source language are limited. Typesetting of definitions is supported only partially and only in some proof assistants, so one may have the problem of maintaining machine-readable and human-readable versions of the specification, and keeping them in sync. Moreover, each proof assistant has its own (steep) learning curve, the community is partitioned into schools (few people are fluent in more than one), and one has to commit to a particular proof assistant from the outset of a project.

A more subtle consequence of the limitations of the available metalanguages is that they obstruct re-use of definitions across the community, even of small calculi. Research groups each have their own private LaTeX macros and idioms – to build on a published calculus, one would typically re-typeset it (possibly introducing minor hopefully inessential changes in the process). Proof assistant definitions are more often made available (e.g. in the Archive of Formal Proofs, Klein *et al.* 2009), but are specific to a single proof assistant. Both styles of definition make it hard to compose semantics in a modular way, from fragments.

Contribution We describe a metalanguage specifically designed for writing semantic definitions and a tool, Ott, that sanity-checks such definitions and compiles them: into LaTeX code; proof assistant code in Coq, HOL or Isabelle/HOL; and OCaml boilerplate.

This metalanguage is designed to make it easy to express the syntax and semantics of an object language, with as little meta-syntactic noise as possible, by directly

supporting some of the informal-mathematics notation that is in common use. For example, in an email or working note one might write grammars for object languages with complex binding structures:

```
t ::=
  | let p = t in t'          bind binders(p) in t'
p ::=
  | x                        binders = x
  | { l1=p1,...,ln=pn }      binders = binders(p1 ... pn)
```

and informal semantic rules:

```
G |- t1:T1  ... G |- tn:Tn
-----------------------------------------
G |- {l1=t1,...,ln=tn} : {l1:T1,...,ln:Tn}
```

These are intuitively clear, concise and (especially when compared with LaTeX or prover source) easy to read and edit. Sadly, they lack both the precision of proof assistant definitions and the production-quality typesetting of LaTeX – but it turns out that only a modicum of information need be added to make them precise. An Ott source file includes that information. It can be used in various ways: simply for lightweight error checking and production-quality typesetting of a definition, and of informal proof; or as a front end to a proof assistant, generating proof assistant definitions as a basis for formal proof. Ott is not itself a proof tool: our focus on engineering language definitions complements the ongoing work by many groups on engineering language metatheory proof.

In more detail, the main innovations are

• ***Metalanguage design to make definitions concise and easy to read and edit (Section 2).*** The Ott metalanguage lets one specify the syntax of an object language, together with rules defining inductive relations, for semantic judgements. Making these easy to express demands rather different syntactic choices to those of typical programming languages: we allow arbitrary context-free grammars of symbolic terms, including direct support for subgrammars, lists and context grammars, and enforce rigid metavariable naming conventions to reduce ambiguity. The tool builds parsers and pretty printers for symbolic and concrete terms of the object language.

• ***An expressive metalanguage (but one that remains simple and intuitive) for specifying binding (Section 3).*** Non-trivial object languages often involve complex forms of binding: not just the single binders of lambda terms, which have received much attention, but also structured patterns, multiple mutually recursive `let` definitions, or-patterns, dependent record patterns, etc. We introduce a metalanguage that can express all these but that remains close to informal practice. We give it three interpretations. Firstly, we define substitution and free variable functions for a "fully concrete" representation, not quotiented by alpha equivalence. This is not appropriate for all examples, but suffices for surprisingly many cases (including those below), and is implemented. Secondly, we define alpha equivalence and capture-avoiding substitution for arbitrary binding specifications, clarifying several issues. We present this as a standard of comparison for future work: implementing the

general case would be a substantial challenge. We prove (on paper) that under usable conditions the two notions of substitution coincide. Thirdly, we describe an implementation of the "locally nameless" representation (Pollack 2006) for a restricted class of binding specifications, which does give canonical representatives for alpha equivalence classes, and comment on what would be involved in implementing support for the Nominal Isabelle package (Urban 2008).

• ***Compilation to proof assistant code (Section 4).*** From a single definition in the metalanguage, the Ott tool can generate proof assistant definitions in Coq, HOL and Isabelle/HOL. These can then be used as a basis for formal proof and (where the proof assistant permits) code extraction and animation. We aim to generate well-formed and idiomatic definitions, without dangling proof obligations, and in good taste as a basis for user proof development.

This compilation deals with the syntactic idiosyncrasies of the different targets and, more fundamentally, encodes features that are not directly translatable into each target. The main issues are dependency analysis; support for common list idioms; generation and use of subrule predicates and context grammar application functions; generation of substitution and free variable functions; (for Isabelle/HOL) a tuple encoding for mutually primitive recursive functions, with auxiliary function definitions for nested pattern matching and for nested list types; (for Coq) generation of auxiliary list types for the syntax and semantics; (for Coq) generation of useful induction principles when using native lists; and (for HOL) a stronger definition library.

• ***Substantial case studies (Section 5).*** The usefulness of the Ott metalanguage and tool has been tested in a number of case studies. Firstly, we have some modest test cases:

1. small lambda calculi: (a) untyped, (b) simply typed and (c) with ML polymorphism, all call-by-value (CBV);
2. systems from TAPL (Pierce 2002) including booleans, naturals, functions, base types, units, seq, ascription, lets, fix, products, sums, tuples, records and variants;
3. the path-based module system of Leroy (1996), with a term language and operational semantics based on Owens & Flatt (2006); and
4. formalisation of the core Ott binding specifications.

Secondly, we have some more substantial developments:

5. Lightweight Java (LJ), a small imperative fragment of Java;
6. LJAM formalisations of Java module system proposals, based on JSR 277/294 (including LJ, 163 semantic rules) (Strniša *et al.* 2007); and
7. a large core of OCaml, including record and datatype definitions (OCaml_{light}, 310 semantic rules) (Owens 2008).

There have also been several developments primarily by other users, including:

8. a language for rely-guarantee and separation logic (Vafeiadis & Parkinson 2007);

9. an object calculus with nominal inheritance and both untyped and typed classes (Gray 2008);
10. a formalisation of the semantics of value commitment and its implementation using audit logs (Fournet *et al.* 2008);
11. Scalina, an object calculus with type-level abstraction (Moors *et al.* 2008);
12. a formalisation of C++ Concepts (Zalewski 2008; Zalewski & Schupp, 2009);
13. LFJ (Delaware *et al.* 2009), an extension of LJ with Features;
14. work relating two kinds of dependent-contract language (Greenberg *et al.*, 2009); and
15. the semantics of a CBV dependently typed language with a parameterised definition of program equivalence (Jia *et al.*, 2009).

Almost all of these involve Ott definitions of type systems and operational semantics, and (at least) (1b), (2), (5), (6), (7), (8), (14) and (15) have machine-checked proofs of metatheory based on Ott-generated proof assistant code.

We discuss the experience of using Ott in Section 6. There is a long history of related work in this area, discussed in Section 7. We conclude in Section 8.

This paper is a revised and extended version of (Sewell *et al.* 2007b). The examples have been revised to reflect the current release of the tool (version 0.10.17, released 2009.07.19), the case study descriptions have been updated, and there are new subsections describing the support for contexts, discussing parsing issues, formally defining alpha equivalence for Ott binding specifications, and discussing support for the locally nameless representation. A user guide is available on the web, along with the tool itself (under a BSD-style licence), a number of examples, and a mailing list discussion forum (Sewell & Zappa Nardelli 2007). User feedback is very welcome.

2 Overview and metalanguage design

In this section we give an overview of the metalanguage. The basic idea is to let the user specify the concrete and abstract syntax of their object language, including whatever meta-notation is needed to express its semantics, together with translations from clauses of that syntax to proof assistant and LaTeX code. Definitions of inductive relations can then be given using that syntax and translated to the various targets. This core functionality is extended with support for various pervasive semantic idioms, including subgrammars, list forms, context grammars, and binding specifications, to make a pragmatically useful tool.

2.1 A small example

We begin with the example in Figure 1, which is a complete Ott source file for an untyped CBV lambda calculus, including the information required to generate LaTeX, OCaml boilerplate and proof assistant definitions in Coq, HOL and Isabelle/HOL.

```
metavar var, x ::=   {{ com  term variable }}
{{ isa string}} {{ coq nat}} {{ hol string}} {{ coq-equality }}
{{ ocaml int}} {{ lex alphanum}} {{ tex \mathit{[[var]]} }}

grammar
term, t :: 't_' ::=                              {{ com term    }}
  | x             ::   :: var                      {{ com variable}}
  | \ x . t       ::   :: lam (+ bind x in t +) {{ com lambda  }}
  | t t'          ::   :: app                      {{ com app     }}
  | ( t )         :: S :: paren                    {{ icho [[t]]  }}
  | { t / x } t' :: M :: sub
                         {{ icho (tsubst_term [[t]] [[x]] [[t']])}}

val,v :: 'v_' ::=                                {{ com value   }}
  | \ x . t       ::   :: lam                      {{ com lambda  }}

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

subrules
  val <:: term

substitutions
  single term var :: tsubst

defns
Jop :: '' ::=

 defn
 t1 --> t2 :: ::reduce::'' {{ com [[t1]] reduces to [[t2]]}} by


    -------------------------  :: ax_app
    (\x.t1) v2 -->  {v2/x}t1

    t1 --> t1'
    ------------- :: ctx_app_fun
    t1 t --> t1' t

    t1 --> t1'
    ------------- :: ctx_app_arg
    v t1 --> v t1'
```

Fig. 1. A small Ott source file, for an untyped CBV lambda calculus, with data for Coq, HOL, Isabelle/HOL, LaTeX and OCaml.

The typeset LaTeX is shown in Figure 2. This is a very small example, sufficing to illustrate some of the issues but not the key problems of dealing with the scale and complexity of a full language (or even a non-trivial calculus) which are our main motivation. We comment on those as we go, and invite the reader to imagine the development for their favourite programming language or calculus in parallel.

Core First consider Figure 1 but ignore the data within `{{ }}` and `(+ +)`, and the **`terminals`** block. At the top of the figure, the **`metavar`** declaration introduces *metavariables* `var` (with synonym `x`), for term variables. The following **`grammar`**

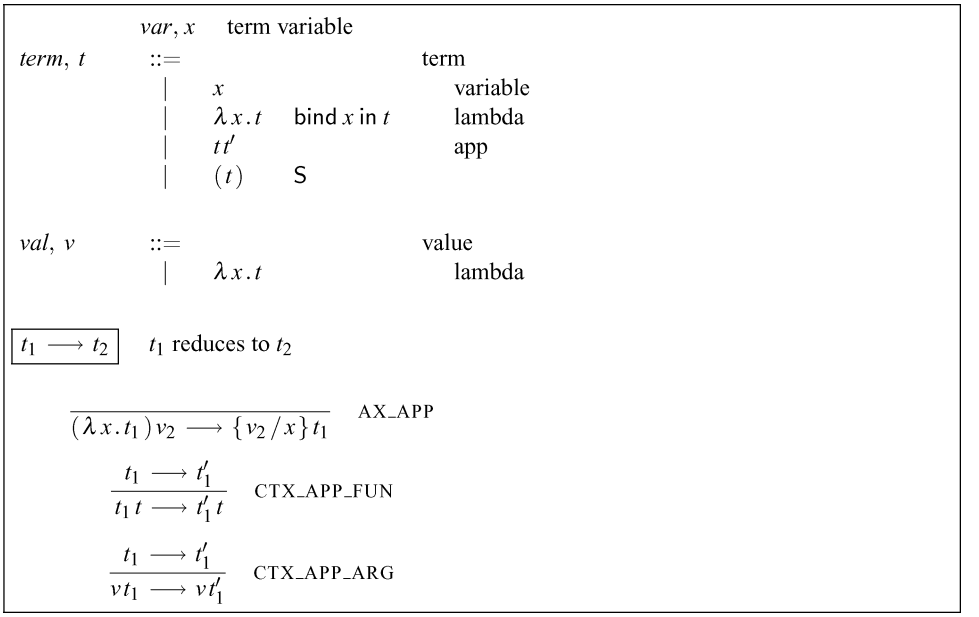


Fig. 2. LaTeX output generated from the Figure 1 source file.

introduces grammars for terms, with *non-terminal root* `term` (with synonym `t`), and for values, with non-terminal root `val` (and synonym `v`):

```
term, t :: 't_' ::=
  | x              ::   :: var
  | \ x . t        ::   :: lam
  | t t'           ::   :: app
  | ( t )          :: S :: paren
  | { t / x } t'   :: M :: sub

val, v :: 'v_' ::=
  | \ x . t        ::   :: lam
```

This specifies the concrete syntax of object-language terms, the abstract syntax representations for proof-assistant mathematics, and the syntax of symbolic terms to be used in semantic rules. The `paren` and `sub` productions are *metaproductions* (flagged **`S`** or **`M`**), introducing syntax which is not part of the object language but is needed when writing semantic rules. Metaproductions flagged **`S`** are syntactic sugar, differing from those flagged **`M`** only in that they are permitted when parsing concrete terms. The productions are named `t_var`, `t_lam`, etc., taking the common prefixes `t_` and `v_` as specified on the first line of each part of the grammar. The *terminals* of the grammar (`\ . ( ) { } / -->`) are inferred, as those tokens that cannot be lexed as metavariables or non-terminals, avoiding the need to specify them explicitly.

Turn now to the **`defns`** block at the bottom of the figure. This introduces a mutually recursive collection of judgments, here a single judgement `t1 --> t2` for

the reduction relation, defined by three rules. Consider the innocent-looking CBV beta rule:

```
------------------------  :: ax_app
(\x.t1) v2 -->  {v2/x}t1
```

The conclusion is a term of the syntactic form of the judgement being defined, here `t1 --> t2`. Its two subterms `(\x.t1) v2` and `{v2/x}t1` are *symbolic* terms for the `t` grammar, not concrete terms of the object language. They involve some object-language constructors (instances of the `lam` and `app` productions of the `term` grammar), just as concrete terms would, but also:

- mention symbolic metavariables (`x`) and non-terminals (`t1` and `v2`), built from metavariable and non-terminal roots (`x`, `t`, and `v`) by appending structured suffixes – here just numbers;
- depend on a subtype relationship between `v` and `t` (declared by the **`subrules`** `val <:: term`, and checked by the tool) to allow `v2` to appear in a position where a term of type `term` is expected; and
- involve syntax for parentheses and substitution, as specified by the `paren` and `sub` metaproductions.

The `ax_app` rule does not have any premises, but the other two rules do, e.g.

```
t1 --> t1'
-------------- :: ctx_app_arg
v t1 --> v t1'
```

Here the premises are instances of the judgement being defined, but in general they may be symbolic terms of a `formula` grammar that includes all judgement forms by default, but can also contain arbitrary user-defined formula productions, for side conditions.

This core information is already a well-formed Ott source file that can be processed by the tool, sanity-checking the definitions, and default typeset output can be generated.

Proof assistant code To generate proof assistant code we first need to specify the proof assistant representations ranged over by metavariables: the **`isa`**, **`coq`** and **`hol`** annotations of the **`metavar`** block specify that the Isabelle/HOL, Coq and HOL `string`, `nat` and `string` types be used. For Coq the **`coq-equality`** generates an equality decidability lemma and proof script for the type.

The proof assistant representation of abstract syntax is then generated from the grammar, as we describe in detail in Section 4. A grammar such as that for `term` above will give rise to a proof assistant type with a constructor corresponding to each of its (non-meta) productions. The metaproductions do not give rise to proof assistant constructors. Instead, the user can specify an arbitrary translation for each. These translations ('*homs*') give clauses of functions from symbolic terms to the character string of generated proof-assistant code. In this example, the `{{` **`icho`** `[[t]] }}` hom for the `paren` production says that `(t)` should be translated into just the translation of `t`, whereas the `{{` **`icho`** `(tsubst_term [[t]] [[x]]`

`[[t']])}}` hom for `sub` says that `{t/x}t'` should be translated into the proof-assistant application of `tsubst_term` to the translations of `t`, `x`, and `t'`. Here the '**`icho`**' specifies that these translations should be done uniformly for Isabelle/HOL, Coq, HOL and OCaml output; one can also specify different translations for each.

The example `val` grammar for values is declared to be a subgrammar of that for `term`, and so will not be represented with a new proof assistant type. Instead, Ott will generate a proof assistant predicate `is_val_of_term` to pick out the relevant part of the representation type for `term`.

The `tsubst_term` mentioned in the hom for `sub` above is a proof assistant identifier for a function that calculates substitution over terms, automatically generated by the **`substitutions`** declaration. We return in Section 3 to what this does, and to the meaning of the binding specification `(+ bind x in t +)` in the `lam` production.

Homs can also be used to specify proof assistant *types* for non-terminals, in cases where one wants a specific proof assistant type expression rather than a type freely generated from the syntax.

Tuned typesetting To fine-tune the generated LaTeX, to produce the output of Figure 2, the user can add some of the remaining data in Figure 1: the `{{ tex \mathit{[[var]]} }}` in the **`metavar`** declaration, specifying that `var`s be typeset in math italic; the **`terminals`** grammar, overriding the default typesetting for terminals `\` and `-->` by λ and $\longrightarrow$; and `{{ com ...}}` comments, annotating productions and judgements. An option controls whether non-sugar metaproductions are typeset.

One can also write **`tex`** annotations to override the default typesetting at the level of productions, not just tokens. For example, in System $F_{<:}$ (Curien & Ghelli 1991; Cardelli *et al.* 1994), where one has both term and type abstractions, one might wish to typeset the former with λ and the latter with Λ, and fine-tune the spacing. Writing productions

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

will typeset $F_{<:}$ lambda terms such as `(\X<:T11.\x:X.t12) [T2]` as $(\Lambda X <: T_{11}.\, \lambda x : X \,.\, t_{12})\ [T_2]$. These annotations define clauses of functions from symbolic terms to the character string of generated LaTeX, overriding the built-in default clause. Similarly, one can control typesetting of symbolic metavariable and non-terminal roots, e.g. to typeset a non-terminal root `G` as Γ.

Concrete terms To fully specify the concrete syntax of the object language one need only add definitions for the lexical form of variables, i.e. the concrete instances of metavariables, with the `{{ lex alphanum }}` hom in the **`metavar`** block. Here **`alphanum`** is a built-in regular expression. Concrete examples can then be parsed by the tool and pretty printed into LaTeX or proof assistant code.

OCaml boilerplate The tool can also generate OCaml boilerplate: type definitions for the abstract syntax, functions for substitution, etc., to use as a starting point

```
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_namem'}
field_name1...field_namen PERMUTES field_name1'...field_namem'
length (e1)...(en)>=1
------------------------------------------------------------------------ :: rc
E |- {field_name1=e1; ...; field_namen=en} : t
```

$$\frac{\begin{array}{l} E \vdash e_1 : t_1 \quad ... \quad E \vdash e_n : t_n \\ E \vdash \mathit{field\_name}_1 : t \to t_1 \quad ... \quad E \vdash \mathit{field\_name}_n : t \to t_n \\ t = (t'_1, ..., t'_l)\, \mathit{typeconstr\_name} \\ E \vdash \mathit{typeconstr\_name} \rhd \mathit{typeconstr\_name} : \mathit{kind}\{\mathit{field\_name}'_1; ...; \mathit{field\_name}'_m\} \\ \mathit{field\_name}_1 ... \mathit{field\_name}_n \ \mathbf{PERMUTES} \ \mathit{field\_name}'_1 ... \mathit{field\_name}'_m \\ \mathbf{length}(e_1)...(e_n) \geq 1 \end{array}}{E \vdash \{\mathit{field\_name}_1 = e_1; ...; \mathit{field\_name}_n = e_n\} : t} \ \text{RC}$$

Fig. 3. A sample OCaml$_{light}$ semantic rule, in Ott source and LaTeX forms.

for implementations. To do this one need specify only the OCaml representation of metavariables, by the **ocaml** hom in the **metavar** block, and OCaml homs for metaproductions, here already included in the uniform **icho** homs. Ott does not generate OCaml code for the definitions of judgments. If the judgement definitions are in a suitable form for automatic generation, then the various proof assistant support for code extraction or generation may be used from the Ott-generated prover code.

2.2 List forms

For an example that is rather more typical of a large-scale semantics, consider the record typing rule shown Figure 3, taken from our OCaml$_{light}$ definition. The top half of Figure 3 shows the source text for that rule, and the bottom half the automatically typeset version – note the close correspondence between the two, making it easy to read and edit the source.

The first, second and fourth premises of the rule are uses of judgement forms; the other premises are uses of `formula` productions with meanings defined by homs. The rule also involves several *list forms*, indicated with dots '...', as is common in informal mathematics. Lists are ubiquitous in programming language syntax, and this informal notation is widely used for good reasons, being concise and clear. We therefore support it directly in the metalanguage, making it precise so that we can generate proof assistant definition clauses, together with the LaTeX shown. Looking at the list forms more closely, we see *index variables* n, m and l occurring in suffixes. There are symbolic non-terminals and metavariables indexed in three different ranges: $e_\Box$, $t_\Box$ and $\mathit{field\_name}_\Box$ are indexed from 1 to n, $\mathit{field\_name}'_\Box$ is indexed from 1 to m, and $t'_\Box$ is indexed 1 to l. To parse list forms involving dots, the tool finds subterms which can be anti-unified by abstracting out components of suffixes. For example, the input

```
E |- e1 : t1 ... E |- en : tn
```

is parsed using a production `formula ::= formula1 .. formulam` which allows a list of formulae to be regarded as a single formula. The subterms `E |- e1 : t1` and `E |- en : tn` can each be parsed as a `formula`. Abstracting corresponding occurrences of the suffices `1` and `n` gives us an anti-unifier `E |- e□ : t□`, which can be mapped back onto the original terms with $\square \mapsto$ `1` and $\square \mapsto$ `n` respectively – and so we have found a correct list form.

With direct support for lists, we need also direct support for symbolic terms involving list projection and concatenation, e.g. in the record rules below:

$$\frac{}{\{\, l'_1 = v_1 \,,\, .. \,,\, l'_n = v_n \,\}\,.\, l'_j \longrightarrow v_j} \quad \textsc{Proj}$$

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

Lastly, one sometimes wants to write list *comprehensions* rather than dots, for compactness or as a matter of general style. We support comprehensions of several forms, e.g. with explicit index i and bounds 0 to $n-1$, as below, and with unspecified or upper-only bounds.

$$\frac{\Gamma \vdash t \;:\; \{\, \overline{l_i \,:\, T_i}^{\,i \in 0..n-1} \,\}}{\Gamma \vdash t\,.\, l_j \;:\; T_j} \quad \textsc{Proj}$$

Other types commonly used in semantics, e.g. finite maps or sets, can often be described with this list syntax in conjunction with type and metaproduction homs to specify the proof assistant representation.

2.3 Context rules

Term contexts of various kinds are also very common in language semantics, e.g. for evaluation contexts, or to express congruence closure. Ott supports the definition of single-hole contexts. For example, suppose one has a term grammar as below:

```
term, t :: 't_' ::=                                {{ com term    }}
  | x                  ::   :: var                 {{ com variable}}
  | \ x . t             ::  :: lam (+ bind x in t +) {{ com lambda  }}
  | t t'               ::   :: app                 {{ com app     }}
  | ( t1 , .... , tn ) ::   :: tuple               {{ com tuple   }}
  | ( t )              :: S :: paren               {{ icho [[t]]  }}
  | { t / x } t'       :: M :: sub
                            {{ icho (tsubst_term [[t]] [[x]] [[t']])}}
  | E . t              :: M :: ctx
                            {{ icho (appctx_E_term [[E]] [[t]])}}
                            {{ tex [[E]] \cdot [[t]] }}
```

A context grammar is declared as a normal grammar but with a single occurrence of the terminal `__` in each production, e.g. as in the grammar for `E` below:

```
E :: 'E_' ::=
  | __ t                              ::  :: appL
  | v __                              ::  :: appR
```

```
| \ x . __                                  ::  :: lam
| ( t1 ( __  t2 ) )                        ::  :: strangeNestedApp
| ( v1 , .. , vm , __ , t1 , .. , tn ) ::  :: tuple
```

Given this, a **contextrules** declaration:

```
contextrules
  E _:: term :: term
```

causes Ott to (a) check that each production of the `E` grammar is indeed a context for the `term` grammar, and (b) to generate proof assistant functions, e.g. here a function that will be called `appctx_E_term`, to apply a context to a term. As the `strangeNestedApp` production shows, context productions can involve nested term structure.

In general, context rule declarations have the form

```
contextrules
  ntE _:: nt1 :: nt2
```

where `ntE`, `nt1` and `nt2` are non-terminal roots. This declares contexts `ntE` for the `nt1` grammar, with holes in `nt2` positions.

The proof assistant representation for context grammars is just like that for other grammars, with new proof assistant types and/or predicates as appropriate. Just as before, context grammars may be non-free, mentioning non-terminals of subgrammars. In the example above, the `E` grammar is non-free, as it mentions the subgrammar non-terminal `v`, so the tool generates a proof assistant type with constructors `E_appL`, `E_appR`, etc., together with a predicate `is_E_of_E` (using the generated `is_val_of_term` predicate) which picks out the elements of that type that actually represent contexts.

Just as for substitutions, the context application function is typically used by adding a metaproduction to the term grammar. Above we added a metaproduction `E.t` to the `t` grammar with an `icho` hom that uses `appctx_E_term`. That can then be used in relations:

```
t --> t'
------------ :: ctx
E.t --> E.t'
```

2.4 Syntactic design

Some interlinked design choices keep the metalanguage general but syntactically lightweight. Issues of concrete syntax are often best avoided in semantic research, tending to lead to heated and unproductive debate. In designing a usable metalanguage, however, providing a lightweight syntax is important, just as it is in designing a usable programming language. We aim to let the working semanticist focus on the content of their definitions without being blinded by markup, inferring data that can reasonably be inferred while retaining enough redundancy that the tool can do useful error checking of the definitions. Further, the community has developed a variety of well-chosen concise notations; we support some (though not all) of these.

The tradeoffs are rather different from those for conventional programming language syntax. There, the grammar is usually fixed, but programs may be large. One often restricts to LALR(1) grammars for fast parsing, using a parser generator, and engineers the grammar to remove ambiguity, at the cost of some complexity. Here, it is essential to support user-defined notation and standard informal-mathematics idioms. Semantic definitions are generally small compared to programs (large definitions might be 10,000 lines, whereas large programs are several orders of magnitude bigger). The user-defined grammar may be ambiguous, but the symbolic expressions that appear in semantic rules rarely are, and are usually small, so we can ask users to explicitly disambiguate where necessary.

There are no built-in assumptions on the structure of the mathematical definitions (e.g. we do not assume that object languages have a syntactic category of expressions, or a small-step reduction relation). Instead, the tool supports definitions of arbitrary syntax and of inductive relations over it. Syntax definitions include the full syntax of the symbolic terms used in rules (e.g. with metaproductions for whatever syntax is desired for substitution). Judgements can likewise have arbitrary syntax, as can formulae.

The tool accepts arbitrary context-free grammars, so the user need not go through the contortions required to make a non-ambiguous grammar (e.g. for `yacc`). Abstract syntax grammars, considered concretely, are often ambiguous, but the symbolic terms used in rules are generally rather small, so this ambiguity rarely arises in practice. Where it does, we let the user resolve it with production-name annotations in terms. The tool finds all parses of symbolic terms, flagging errors where there are multiple possibilities. It uses a scannerless generalised LR (SGLR) parsing approach, taking ideas from Rekers (1992), Visser (1997) and McPeak & Necula (2004), which is simple and sufficiently efficient.

Naming conventions for symbolic non-terminals and metavariables are rigidly enforced – they must be composed of one of their roots and a suffix. This makes many minor errors detectable, makes it possible to lex the suffixes, and makes parsing much less ambiguous.

Highly ambiguous list forms are among the most difficult kinds of inputs to parse correctly. For example, consider the following grammar of a typing contexts Γ:

$$\begin{array}{rcl} \Gamma & ::= & x : t \\ & | & \Gamma_1, \ldots, \Gamma_n \end{array}$$

along with an example typing context:

```
G',x1:t1,..,xn:tn,x':t',x1:t1,..,xn:tn,x'':t''
```

The example should be parsed as a flat list of five elements: a symbolic non-terminal `G'`, a literal list form (`x1:t1,..,xn:tn`), a single list element (`x':t'`), a second literal list form, and a second single list element. It should not be parsed, for example, as a list of two elements, themselves containing two and three elements, although a naive interpretation of the grammar would allow this.

SGLR parsing does not natively support our list notation, so we first convert list-containing productions to plain context-free grammars before building the SGLR

parser. The Γ grammar becomes

$$\begin{array}{lcl} \Gamma & ::= & x{:}t \\ & | & \epsilon \\ & | & \Gamma' \\ \Gamma' & ::= & \Gamma'' \\ & | & \Gamma'', \Gamma' \\ \Gamma'' & ::= & \Gamma \\ & | & \Gamma''' \\ \Gamma''' & ::= & \textit{literal list forms over } \Gamma \end{array}$$

Reject productions and priority restrictions (Visser 1997) are generated to ensure that terms can be unambiguously parsed against this highly ambiguous grammar. The usual efficient, parser-table-construction-time interpretation of priority restrictions is insufficient here due to the possibility of parses with long chains of trivial injections (e.g. Γ derives Γ′ derives Γ″ derives Γ). Thus, the tool filters the parse graph after creation to reject parses with priority violations spanning such chains.

2.5 Workflow

To make the Ott tool more usable in realistic workflows, we have had to attend to some conceptually straightforward but pragmatically important engineering issues. We mention a few to give the flavour:

Modular definitions The tool supports a simple but very useful form of modular semantics. By default, if it is given multiple Ott source files, then these are effectively concatenated. If the `-merge true` option is given, however, then identically-named grammars from different source files are merged into one, and similarly for identically named relation definitions, so different aspects of a language definition can be defined in different files.

Filtering Both LaTeX and proof assistant files can be *filtered*, replacing delimited Ott-syntax symbolic terms (or concrete term examples) in documents. For example, given the Ott definition in Figure 1, one can write `[[ (\x.x x) x']]` in a LaTeX file. Ott can then act as a preprocessor for LaTeX, replacing that by LaTeX source that renders as $(\lambda x.x\,x)\,x'$. This also provides a useful sanity check, e.g. in informal proofs, simply by parsing the symbolic terms used. Filtering for the other targets is similar. For example, in a filtered HOL file the `[[ (\x.x x) x']]` would be replaced by the HOL term `(t_app (t_app (t_lam x (t_var x)) (t_var x)) (t_var x'))`.

Additionally, LaTeX and proof assistant code can be *embedded* within an Ott source file (and similarly filtered).

Using the generated LaTeX Ott can produce either a standalone LaTeX file (with a default preamble) or a file that can be included in other documents. The generated LaTeX is factored into LaTeX commands for individual rules, the rules of individual **defn**s, and so on, up to the complete definition, so that parts or all of the definition can be quoted in other documents in any order, without any resort to cut and paste.

The typesetting style is indirected, so that it can be controlled by redefining LaTeX commands (and those redefinitions can be embedded in an Ott source file).

Fancy syntax The proof assistants each have their own support, more-or-less elaborate, for fancy syntax. For Isabelle/HOL the Ott user can specify the data for syntax declarations with additional homs, or the tool can generate them from an Ott source grammar. They can then be used in proof scripts and in the displayed goals during interactive proof.

Naming We support common prefixes for rule names and production names (e.g. the `t_` in Figure 1), and allow synonyms for non-terminal and metavariable roots (e.g. if one wanted S, T and U to range over a grammar of types).

3 Binding specifications and substitution

How to deal with *binding*, and the accompanying notions of substitution and free variables, is a key question in formalised programming language semantics. It involves two issues: one needs to fix on a class of binding structures being dealt with, and one needs proof-assistant representations for them.

The latter has been the subject of considerable attention, with representation techniques based on names, De Bruijn indices, higher order abstract syntax (HOAS), locally nameless terms, nominal sets and so forth, in various proof assistants. The annotated bibliography by Charguéraud (2006) collects around 40 papers on this, and it was a central focus of the POPLmark challenge (Aydemir *et al.* 2005).

Almost all of this work, however, deals only with the simplest class of binding structures, the *single binders* we saw in the lambda abstraction production of the Section 2 example:

$$\begin{array}{lll} \mathit{term},\ t & ::= & \\ & \mid\ \lambda\, x\, .\, t \quad \mathsf{bind}\ x\ \mathsf{in}\ t & \text{lambda} \end{array}$$

in which a single variable binds in a single subterm. Realistic programming languages often have much more complex binding structures, e.g. structured patterns, multiple mutually recursive `let` definitions, comprehensions, or-patterns and dependent record patterns. We therefore turn our attention to the potential range of binding structures. In Section 3.1 we introduce a novel metalanguage for specifying binding structures, expressive enough to cover all the above but remaining simple and intuitive. We describe two semantics for the binding metalanguage: a fully concrete semantics, in Section 3.2, which is implemented in the tool, and a reference definition of alpha equivalence, introduced in Section 3.3 and formalised in Section 3.4. Finally, we discuss support for locally nameless and nominal alpha-respecting representations, in Section 3.5; we have implemented the former for a restricted class of binding specifications.

3.1 The Ott binding metalanguage: syntax

The binding metalanguage comprises two forms of annotation on productions. The first, bind *mse* in *nonterm*, is used in the lambda production above. That production

has a metavariable x and a non-terminal t, and the binding annotation expresses that, in any concrete term of this production, the variable in the x position binds in the subterm in the t position. A variable can bind in multiple subterms, as in the example of a simple recursive **let** below:

$$
\begin{array}{lll}
t ::= & & \\
\quad \mid \ \mathbf{let\,rec}\ x\ =\ t\ \mathbf{in}\ t' & & \mathsf{bind}\ x\ \mathsf{in}\ t \\
& & \mathsf{bind}\ x\ \mathsf{in}\ t'
\end{array}
$$

In general a production may require more than just a single variable to bind, and so in the general case *mse* ranges over *metavariable set expressions*, which can include the empty set, singleton metavariables (e.g. the x above, implicitly coerced to a singleton set) and unions.

More complex examples require one to collect together sets of variables. For example, the grammar below has structured patterns, with a $\mathbf{let}\, p\ =\ t\ \mathbf{in}\ t'$ production in which all the binders of the pattern p bind in the continuation t'.

$$
\begin{array}{lll}
t ::= & & \\
\quad \mid \ x & & \\
\quad \mid \ (\, t_1\, ,\, t_2\,) & & \\
\quad \mid \ \mathbf{let}\, p\ =\ t\ \mathbf{in}\, t' & & \mathsf{bind}\ \mathrm{binders}(p)\ \mathsf{in}\ t' \\
& & \\
p ::= & & \\
\quad \mid \ \_ & & \mathrm{binders} = \{\} \\
\quad \mid \ x & & \mathrm{binders} = x \\
\quad \mid \ (\, p_1\, ,\, p_2\,) & & \mathrm{binders} = \mathrm{binders}(p_1) \cup \mathrm{binders}(p_2)
\end{array}
$$

Here the bind clause binds all of the variables collected as binders(p). We see a user-defined *auxiliary function* called binders, which is defined by structural induction over patterns p to build the set of variables mentioned in a pattern. The clauses that define the binders auxiliary are the second form of binding annotation. For example, binders(x) is the singleton set $\{\mathrm{x}\}$, while binders(((x, x), y)) is the set $\{\mathrm{x}, \mathrm{y}\}$. A definition may involve many different auxiliary functions; 'binders' is a user identifier, not a keyword.

The syntax of a precise fragment of the binding metalanguage is given in Figure 4, where we have used Ott to define part of the Ott metalanguage. A simple type system (not shown) enforces sanity properties, e.g. that each auxiliary function is only applied to non-terminals that it is defined over, and that metavariable set expressions are well-sorted, not mixing distinct classes of variables.

Further to that fragment, the tool supports binding for the list forms of Section 2.2. Metavariable set expressions can include lists of metavariables and auxiliary functions applied to lists of non-terminals, e.g. as in the record patterns below:

$$
\begin{array}{lll}
p ::= & & \\
\quad \mid \ x & & \mathrm{b} = x \\
\quad \mid \ \{\, l_1\ =\ p_1\, ,\, ..\, ,\, l_n\ =\ p_n\, \} & & \mathrm{b} = \mathrm{b}(p_1..p_n)
\end{array}
$$

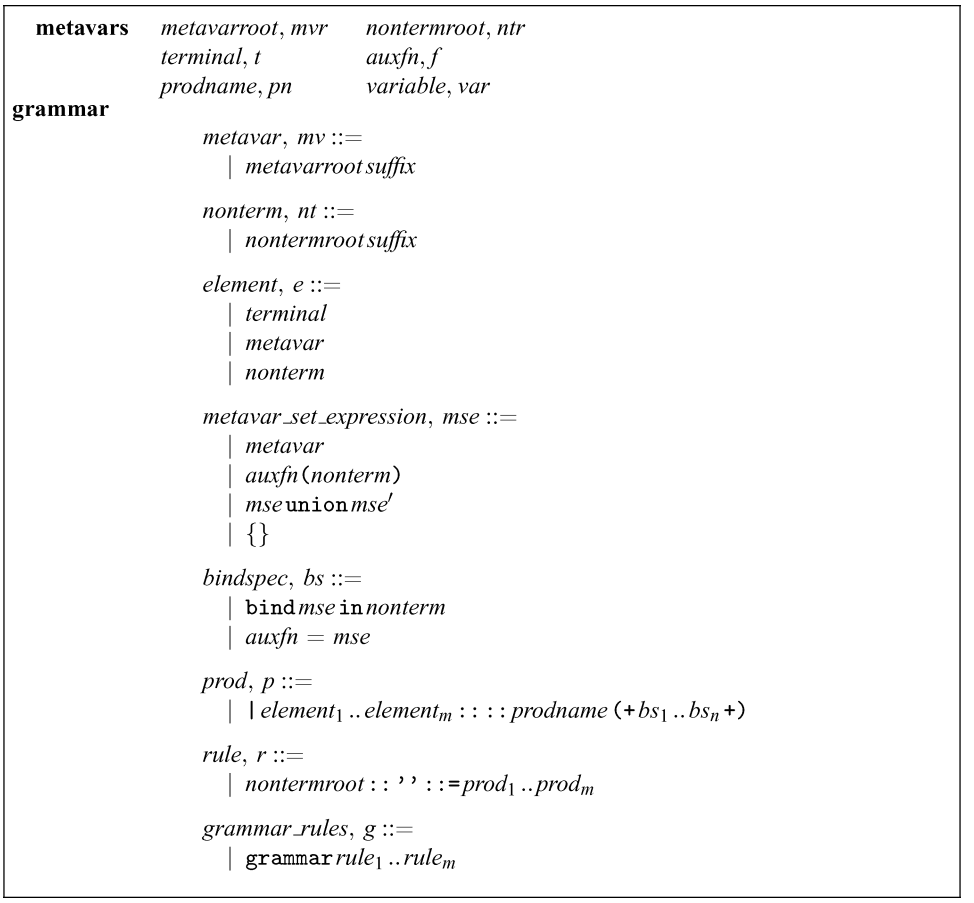


Fig. 4. Mini-Ott in Ott: the binding specification metalanguage.

This suffices to express the binding structure of almost all the natural examples we have come across, including definitions of mutually recursive functions with multiple clauses for each, join-calculus definitions (Fournet *et al.* 1996), dependent record patterns, and many others.

Given a binding specification, the tool can generate substitution functions automatically. Figure 1 contained the block:

```
substitutions
  single term var :: tsubst
```

which causes Ott to generate proof-assistant functions for single substitution of term variables by terms over all (non-subgrammar) types of the grammar – here that is just `term`, and a substitution function named `tsubst_term` is generated. Multiple substitutions can also be generated, and there is similar machinery for free variable functions.

$$
\begin{array}{l}
\mathit{concrete\_abstract\_syntax\_term},\ \mathit{cast} ::= \\
\quad \mid \mathit{var} : \mathit{mvr} \\
\quad \mid \mathit{prodname}\,(\mathit{cast}_1, .., \mathit{cast}_m)
\end{array}
$$

Fig. 5. Mini-Ott in Ott: concrete abstract syntax terms.

3.2 The Ott binding metalanguage: the fully concrete semantics

We give meaning to these binding specifications in two ways. The first semantics is what we term a *fully concrete* representation. Perhaps surprisingly, a reasonably wide range of programming language definitions can be expressed satisfactorily without introducing alpha equivalence (we discuss what can and cannot be expressed in Section 6). In typical CBV or call-by-name languages, there is no reduction under term variable binders. The substitutions that arise therefore only substitute *closed* terms, so there is no danger of capture. The fully concrete representation uses abstract syntax terms containing concrete variable names (Figure 5 gives a general grammar of such concrete abstract syntax terms, *cast*s). Substitution is defined so as to not substitute for bound variables within their scopes, but without using any renaming. Section 4.2 shows an example of the generated code.

Doing this in the general case highlights a subtlety: when substituting, e.g. `ts` for `xs` in the Figure 1 language, the only occurrences of `x` that are substitutable are those in instances of productions of the `term` grammar that comprise just a *singleton* `x` (just the `var` production), as only there will the result be obviously type correct. Other occurrences (the `x` in the `lam` production, or the x in the pattern grammars above), are not substitutable, and, correspondingly, should not appear in the results of free variable functions. In natural examples one might expect all such occurrences to be bound at some point in the grammar.

A precise definition of this fully concrete representation is available for the Mini-Ott of Figure 4, including definitions of substitution and free variables over the general concrete abstract syntax terms of Figure 5 (Peskine *et al.* 2007). Given the preceding remarks it is essentially straightforward.

3.3 The ott binding metalanguage: the alpha-equivalence semantics, informally

The fully concrete representation suffices for the case studies we describe here (notably including the OCaml fragment), but sometimes alpha equivalence really is needed – e.g. where there is substitution under binders, for dependent type environments,[1] or for compositional reasoning about terms. We have therefore defined notions of alpha equivalence and capture-avoiding substitution over concrete abstract syntax terms, again for an arbitrary Mini-Ott object language and binding

[1] The POPLmark $F_{<:}$ example is nicely expressible in Ott as far as LaTeX output goes, but its dependent type environments would require explicit alpha conversion in the rules to capture the intended semantics using the fully concrete representation. In single-binder versions of $F_{<:}$ the Ott support for locally nameless representations can be used (see Section 3.5).

specification. We first explain the key points with two examples, and in the following subsection give the main part of the formal definition.

First, consider the OCaml *or-patterns*[2] $p_1 \mid p_2$, e.g. with a pattern grammar

$$
\begin{array}{ll}
p ::= & \\
\quad \mid\ x & \mathrm{b} = x \\
\quad \mid\ (\, p_1 \,,\, p_2 \,) & \mathrm{b} = \mathrm{b}(p_1) \cup \mathrm{b}(p_2) \\
\quad \mid\ p_1 \mid p_2 & \mathrm{b} = \mathrm{b}(p_1) \cup \mathrm{b}(p_2) \\
\quad \mid\ \mathbf{None} & \mathrm{b} = \{\} \\
\quad \mid\ \mathbf{Some}\, p & \mathrm{b} = \mathrm{b}(p) \\
\quad \mid\ (\, p \,) & \mathsf{S}
\end{array}
$$

This would be subject to the conditions (captured in type rules) that for a pair pattern $(\, p_1 \,,\, p_2 \,)$ the two subpatterns have disjoint domain, whereas for an or-pattern $p_1 \mid p_2$ they have equal domain and types. One can then write example terms such as that below:

$$\mathbf{let}\,(\,(\,\mathbf{None}\,,\,\mathbf{Some}\,x\,)\mid(\,\mathbf{Some}\,x\,,\,\mathbf{None}\,)\,)\;=\;y\;\mathbf{in}\,(\,x\,,\,x\,)$$

Here there is no simple notion of 'binding occurrence'. Instead, one should think of the two occurrences of x in the pattern, and the two occurrences of x in the continuation, as all liable to alpha-vary together. This can be captured by defining, inductively on a concrete abstract syntax term *cast*, a partial equivalence relation *closed_reln* over the *occurrences* of variables within it. In the example it would relate all four occurrences of x to each other, as below, but leave y unrelated.

$$\mathbf{let}\,(\,(\,\mathbf{None}\,,\,\mathbf{Some}\,\underbrace{x}\,)\mid(\,\mathbf{Some}\,\underbrace{x}\,,\,\mathbf{None}\,)\,)\;=\;y\;\mathbf{in}\,(\,\underbrace{x}\,,\,\underbrace{x}\,)$$

Given this, one can define two terms to be alpha equivalent if their equivalence classes of occurrences can be freshly renamed to make them identical.

For the second example, consider a system such as $\mathrm{F}_{<:}$ with type environments Γ as below:

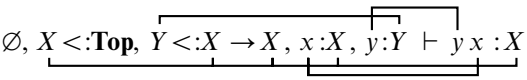
$$
\begin{array}{ll}
\Gamma ::= & \\
\quad \mid\ \varnothing & \\
\quad \mid\ \Gamma,\, X <: T & \\
\quad \mid\ \Gamma,\, x : T &
\end{array}
$$

In setting up such a system, it is common to treat the terms and types up to alpha equivalence. There is then a technical choice about whether the judgements are also taken up to alpha equivalence: in typing judgements $\Gamma \vdash t : T$, one can either treat Γ concretely or declare the domain of Γ to bind in t and in T. Suppose one takes the second approach, and further has each element of Γ ($X <: T$ or $x : T$) binding (X or x) in the succeeding elements. (All these options can be expressed in the Ott bindspec metalanguage.) For a complete judgement such as

$$\varnothing,\, X <: \mathbf{Top},\, Y <: X \to X\,,\, x : X\,,\, y : Y \;\vdash\; y\, x : X$$

[2] Similar binding occurs in the join-calculus, where a join definition may mention the 'bound' names arbitrarily often on the left.

it is then easy to see what the binding structure is, and we can depict the *closed_reln* as below:



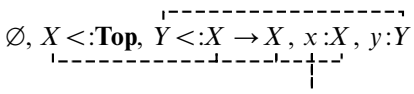
Now consider that type environment in isolation, however,

$$\emptyset,\ X <: \mathbf{Top},\ Y <: X \to X,\ x : X,\ y : Y$$

Here, while in some sense the $X <: \mathbf{Top}$ binds in the succeeding part of the type environment, it must not be alpha-varied, e.g. to

$$\emptyset,\ W <: \mathbf{Top},\ Y <: W \to W,\ x : W,\ y : Y$$

as that would give a different type environment (which would type different terms). Alpha conversion of the X becomes possible only when the type environment is put in the complete context of a judgement. Our definitions capture this phenomenon by defining for each term *cast* not just a *closed_reln* relation for 'closed' binding but also a similar *open_reln* partial equivalence relation for 'open' binding, relating occurrences which potentially may alpha-vary together if this term is placed in a larger, binding, context. In the example that larger context would be an instance of $[\cdot] \vdash t : T$, from the production for the judgement $\Gamma \vdash t : T$. The *open_reln* is not directly involved in the definition of alpha equivalence, but is (compositionally) used to calculate the *closed_reln*. It is shown for this example below[3]:



Non-trivial open binding also occurs in languages with dependent patterns, e.g. those with pattern matching for existential types.

We increase confidence in these definitions by proving a theorem that, under reasonable conditions, substitution of closed terms in the fully concrete representation coincides with capture-avoiding substitution for our notion of alpha equivalence for arbitrary binding specifications. The conditions involve the types of the desired substitution and the auxiliary functions present – to a first approximation, that the types of substitutions (e.g. in the pair pattern example above, terms of non-terminal t for variables x), are distinct from the domains and results of auxiliary functions (e.g. the binders above collects variables x from patterns p). In the absence of a widely accepted alternative class of binding specifications, there is no way to even formulate 'correctness' of that notion in general, but for specific examples one can show that it coincides with a standard representation. We did that (a routine exercise), for the untyped lambda calculus. Both of these are hand proofs, though above mechanised definitions.

Generating proof assistant code that respects this notion of alpha equivalence, for arbitrary binding specifications, is a substantial question for future work. It could be addressed directly, in which case one has to understand how to generalise the existing

[3] Here the x is in a singleton equivalence class by itself (indicated by a short vertical dashed line), whereas, because the grammar was set up to extend to the right, with a production $\Gamma, x : T$, the final variable y is not in the open binding relation at all.

proof assistant representations, and what kind of induction schemes to produce, or via a uniform translation into single binders – perhaps introducing proof-assistant binders at each bind *mse* point in the grammar. A perhaps more tractable (but still rather expressive) subclass of binding specifications can be obtained by simple static conditions that guarantee that there is no 'open' binding.

3.4 The Ott binding metalanguage: alpha equivalence, formally

In this subsection we describe our general definition of alpha equivalence for arbitrary Ott binding specifications in the Mini-Ott-in-Ott language, as introduced by example in the previous subsection.

For brevity we present only an extract with the key parts of the definition. The full definition is itself expressed in Ott, from which well-formed Isabelle/HOL code is generated. These definitions are available on the web (Peskine *et al.* 2007). The definition is somewhat involved, and implementing it in a proof assistant would be challenging. But it does deal with the full generality of Ott binding specifications, and therefore may be useful as a standard of comparison for more restricted proposals. Readers not concerned with the technicalities of complex binding specifications may like to skip to the next subsection.

To simplify the notation, we suppose that productions contain no terminals, so *element* ranges only over *metavar* and *non-term*. The definition is phrased in terms of occurrences *oc* within concrete abstract syntax terms *cast*: lists of natural number indices describing paths from the root of a term, with the empty list indicating the root itself. For example, the concrete abstract syntax term representing $\lambda a.a\,b$ of the Figure 1 grammar is `t_lam(a:var, t_app(a:var, b:var))`; prefixing subterms with their occurrences we have

$$^{[]}\texttt{t\_lam(}^{[0]}\texttt{a:var, }\ ^{[1]}\texttt{t\_app(}^{[1,0]}\texttt{a:var, }\ ^{[1,1]}\texttt{b:var))}$$

We also use partial equivalence relations (PERs) over sets of occurrences, for convenience represented not as binary relations but as sets of pairwise-disjoint sets of occurrences. The definitions are parameterised by a grammar g, as in Figure 4, consisting of rules for each non-terminal root, each of which comprises several named productions, each of which has a list of elements and binding specifications.

We begin with several auxiliary notions:

- $p \in g\,(ntr)$: g has production p for *ntr*
- $g \vdash f\,(pn) = mse$: g defines auxiliary function f at production name *pn* to be *mse*
- $cast\,@oc = cast'$: the subterm of *cast* at *oc* is $cast'$
- **head** $oc = i$: *oc* starts with a branch i
- **closure** *oc_reln*: the finest PER that is coarser than the set of sets of occurrences *oc_reln*
- $\bigcup oc\_reln$: the support of the PER *oc_reln*
- $i::oc\_set$: the lifting of occurrences *oc_set* from the ith subterm of a *cast*, i.e. $\{i::oc \mid oc \in oc\_set\}$
- **vars_of** *oc_set* **from** *cast*: the set of variables at the occurrences *oc_set* in *cast*

Given a bind clause `bind` *mse* `in` *nt* attached to a production, we can now define the interpretation of the metavariable expression *mse* on a term *cast* generated by that production. This (routine) interpretation, written $[\![mse]\!]_g(cast)$, picks out the occurrences of variables within *cast* that are referred to by *mse*. It is defined as follows:

$$\frac{}{[\![\{\}\ ]\!]_g(cast) = \{\}} \quad \text{FUNSPEC\_INTERP\_MSE\_EMPTY}$$

$$\frac{1:\ [\![mse]\!]_g(cast) = oc\_set \qquad 2:\ [\![mse']\!]_g(cast) = oc\_set'}{[\![mse\ \cup\ mse']\!]_g(cast) = oc\_set\ \cup\ oc\_set'} \quad \text{FUNSPEC\_INTERP\_MSE\_UNION}$$

$$\frac{\begin{array}{l}1:\ |\ e_1\,..\,e_n\ ::::\ pn\ (+\,bs_1\,..\,bs_m\,+)\ \in\ g\,(\,ntr\,)\\ 2:\ e_l\ =\ mv\\ 3:\ cast_l\ =\ v'\end{array}}{[\![mv]\!]_g(pn\,(\,cast_1\,,\,..\,,\,cast_q\,)) = \{\,l\,::\,[\,]\,\}} \quad \text{FUNSPEC\_INTERP\_MSE\_MV}$$

$$\frac{\begin{array}{l}1:\ |\ e_1\,..\,e_q\ ::::\ pn\ (+\,bs_1\,..\,bs_m\,+)\ \in\ g\,(\,ntr\,)\\ 2:\ e_l\ =\ nt\\ 3:\ cast_l\ =\ pn'\,(\,cast'_1\,,\,..\,,\,cast'_{q'}\,)\\ 4:\ g\ \vdash\ f\,(\,pn'\,)\ =\ mse'\\ 5:\ [\![mse']\!]_g(cast_l) = oc\_set\end{array}}{[\![f\,(nt\,)]\!]_g(pn\,(\,cast_1\,,\,..\,,\,cast_q\,)) = l\,::\,oc\_set} \quad \text{FUNSPEC\_INTERP\_MSE\_F}$$

The FUNSPEC_INTERP_MSE_MV rule finds the relevant (*pn*) production for the term, and finds in that production that the lth component is the metavariable *mv* in question. The corresponding variable v' in the term is thus at occurrence $l::[]$. The FUNSPEC_INTERP_MSE_F rule, for an auxiliary function f of a non-terminal *nt*, again finds the relevant production, where that non-terminal is the lth component. The lth subterm $cast_l$ is an instance of production name pn', and the definition of f for that production name is mse'. We thus calculate the interpretation of mse' on the subterm $cast_l$ and lift those occurrences to occurrences in the whole term. For example, in the `lam` production of the Figure 1 grammar,

```
| \ x . t      ::   :: lam (+ bind x in t +)
```

we have a bindspec **`bind x in t`**. The FUNSPEC_INTERP_MSE_MV rule picks out the 'binder' $^{[0]}$`a:var` within the term $\lambda a.a\,b$:

$$[\![\mathtt{x}]\!]_g(^{[]}\mathtt{t\_lam}(^{[0]}\mathtt{a:var},\ ^{[1]}\mathtt{t\_app}(^{[1,0]}\mathtt{a:var},\ ^{[1,1]}\mathtt{b:var}))) = \{[0]\}$$

We can now define the interpretation of an auxfn f on *cast* in grammar g, written $[\![f\,]\!]_g\,(\,cast\,)$:

$$\frac{\begin{array}{l}1:\ cast\ =\ pn\,(\,cast_1\,,\,..\,,\,cast_q\,)\\ 2:\ g\ \vdash\ f\,(\,pn\,)\ =\ mse\\ 3:\ [\![mse]\!]_g(cast) = oc\_set\end{array}}{[\![f\,]\!]_g(cast) = oc\_set} \quad \text{FUNSPEC\_INTERP\_AUXFN\_DEF}$$

We say an occurrence *oc* **in** *cast* **is revealable** if there exists some auxiliary function f in the grammar such that $oc \in [\![f]\!]_g\,(cast)$. It is these occurrences which may (if they are not closed-bound deeper in the term) lead to non-trivial 'open' binding.

We now define, inductively over the structure of a *cast*, two partial equivalence relations over the occurrences of variables within it: one (*closed_reln*) for the 'closed' binding and one (*open_reln*) for the 'open' binding. These are partial equivalence relations with disjoint support, that is, no occurrence is related (to anything) by both relations. There may be variable occurrences within *cast* that are not related at all (so-called 'free variables'). In the base case, for variables, both relations are empty. The inductive step proceeds as follows.

Suppose $cast = pn\,(\overline{cast_i}^{\,i})$ and the production for pn is of the form

$$\mid \overline{e_i}^{\,i} :: :: pn\ (+\ \overline{bs_j}^{\,j}\ +)$$

Also suppose that, for each i, we have recursively calculated PERs $closed\_reln_i$ and $open\_reln_i$ on the subterms $cast_i$. (In the $\lambda a.a\,b$ example both of these are empty.) We prefix each occurrence in those PERs by the corresponding i, to lift them into occurrences of *cast*, and take their unions. Here the union remains a PER because the supports of the individual sets are disjoint.

$$\begin{array}{lcl} sub\_closed\_reln & = & \bigcup_i \{i :: oc\_set \mid oc\_set \in closed\_reln_i\} \\ sub\_open\_reln & = & \bigcup_i \{i :: oc\_set \mid oc\_set \in open\_reln_i\} \end{array}$$

We look up the binding specifications attached to the production *pn*. Recall that each of these is of the form '`bind` mse_j `in` nt_j'. For such a specification, we interpret mse_j over *cast*, calculating the set $[\![mse_j]\!]_g(cast)$ of all variable occurrences that that mse_j may refer to.

We now consider each concrete syntactic variable v appearing at any occurrence in $[\![mse_j]\!]_g(cast)$. In the $\lambda a.a\,b$ example, this is just the `a` appearing at [0], while for the previous or-pattern example, it would be the two occurrences of x in the pattern. For each such variable, all binding occurrences of v and all bound occurrences of v should be related to each other. The potential binding occurrences are those that lie within the interpretation of mse_j. In the $\lambda a.a\,b$ example this is the [0] we found above.

The potential bound occurrences are those that lie within the term lying at position i, where i is the position of nt_j in the production. In the $\lambda a.a\,b$ example, the `t` in the bindspec **`bind x in t`** refers to position 1 of the production `\ x . t`. Looking at the [1,...] subterm of our cast $^{[]}$`t_lam(`$^{[0]}$`a:var,` $^{[1]}$`t_app(`$^{[1,0]}$`a:var,` $^{[1,1]}$`b:var))` we have $^{[1]}$`t_app(`$^{[1,0]}$`a:var,` $^{[1,1]}$`b:var)`. The only occurrence of `a` within this is [1,0].

However, we should be careful to remove any occurrences that are already closed-bound within subterms. Thus, this step defines a $new\_binding\_reln_1$ as follows:

$$
\begin{array}{l}
new\_binding\_reln_1 = \\
\quad \textbf{closure}\ \{\ (\ (\{\, oc \in [\![mse_j]\!]_g(cast) \mid cast@oc = v \,\} \cup \\
\qquad\qquad \{\, oc \mid \exists i.(\, nt_j = e_i \wedge \textbf{head}\, oc = i \wedge cast@oc = v \,)\,\}) \\
\qquad\quad -\bigcup sub\_closed\_reln\,) \\
\qquad | \, v \in \textbf{vars\_of}\ [\![mse_j]\!]_g(cast)\ \textbf{from}\ cast \wedge \\
\qquad\ bs_j = \texttt{bind}\ mse_j\ \texttt{in}\ nt_j \\
\qquad \}
\end{array}
$$

In the simple example, this just relates [0] and [1, 0], as would be expected.

Note that in general the sets of occurrences for each choice of v and j are not necessarily disjoint, since the same variable may bind in more than one subterm, and, conversely, a single variable may be bound from more than one *mse*. Hence we take the closure of the resulting set of sets. For an example, consider the recursive `let` production

$$
\begin{array}{lll}
t ::= & & \\
\quad \mid\ \textbf{let rec}\ x = t\ \textbf{in}\ t' & & \mathsf{bind}\ x\ \mathsf{in}\ t \\
& & \mathsf{bind}\ x\ \mathsf{in}\ t'
\end{array}
$$

where the new equivalence relationships for the two bindspecs need to be coalesced into one.

We then combine the above with the open binding relation from the subterms:

$$new\_binding\_reln_2 = \textbf{closure}(new\_binding\_reln_1 \cup sub\_open\_reln)$$

A **closure** operator is again required. This deals with the case where one has a non-trivial *sub_open_reln* capturing the internal open binding within subterms, and the current production binds all the variables of the open binding in some other term; the subsets arising from the two sources need to be coalesced. For example, in the type environment example from the previous section, at the point when a typing context is used in a judgement, such as going from

$$\emptyset,\ X <: \textbf{Top},\ Y <: X \to X,\ x : X,\ y : Y$$

to

$$\emptyset,\ X <: \textbf{Top},\ Y <: X \to X,\ x : X,\ y : Y \ \vdash\ y\ x : X$$

the open binding relation in the typing context (e.g. for variable X above) must be closed with the new occurrences in the right-hand side of the judgement.

Finally, the PER *open_reln* for this term is obtained by filtering the $new\_binding\_reln_2$ above to retain only the equivalence classes containing revealable occurrences, while the PER *closed_reln* is the relation with all equivalence classes consisting entirely of non-revealable occurrences, combined with the *sub_closed_reln*

from subterms.

$$\begin{array}{lll} \mathit{open\_reln} & = & \{(\mathit{oc\_set}' \in \mathit{new\_binding\_reln2}) \\ & & \quad \mid (\exists \mathit{oc}.((\mathit{oc}\ \mathbf{in}\ \mathit{cast}\ \mathbf{is\ revealable}) \wedge (\mathit{oc} \in \mathit{oc\_set}'))) \} \\ \mathit{closed\_reln} & = & \mathit{sub\_closed\_reln} \cup (\mathit{new\_binding\_reln2} - \mathit{open\_reln}) \end{array}$$

We now turn to the definition of alpha-equivalence on concrete terms. Given terms cast_1 and cast_2, with associated PERs $\mathit{closed\_reln}_1$, $\mathit{open\_reln}_1$, $\mathit{closed\_reln}_2$ and $\mathit{open\_reln}_2$, we say they are alpha-equivalent if they have the same closed binding sets, and, for each occurrence not in the closed binding set, the occurrence is defined for one term if it is for the other, and the terms have the same label (variable, at the leaves, or production-name, at interior nodes) at that occurrence. In other words, they are alpha-equivalent iff they are identical except for the choice of variable names at closed occurrences, and their closed binding PER is the same.

To take an example, the following two terms:

$$X = (\lambda \overbrace{x}.(x\ y))(\lambda \underbrace{x}.(x\ y)) \qquad \text{and} \qquad Y = (\lambda \overbrace{z}.(z\ y))(\lambda \underbrace{w}.(w\ y))$$

are alpha-equivalent since their closed binding sets of occurrences are identical, and all occurrences not in the closed binding sets are defined for one term if and only if it is defined for the other, and further, such occurrences have nodes with the same label.

However, the term

$$(\lambda \overbrace{x.(x\ x)})(\lambda \underbrace{x}.(x\ y))$$

is not alpha-equivalent to X (or Y), since the closed binding sets are different. Finally, concrete names that are open-bound are significant: the following are not alpha-equivalent, as the occurrences of variable X on the left have different labels from the occurrences of variable W on the right, though the closed and open binding sets of the two terms are the same.

$$\overbrace{\underbrace{\varnothing, X <: \mathbf{Top}}, \underbrace{Y <: X \to X, x : X, y : Y}}^{\text{closed}} \qquad \overbrace{\underbrace{\varnothing, W <: \mathbf{Top}}, \underbrace{Y <: W \to W, x : W, y : Y}}^{\text{closed}}$$

3.5 Implementing simple binding specifications with a locally nameless representation

The locally nameless representation for terms up to alpha equivalence (Pollack 2006) is an hybrid representation that uses De Bruijn indexes for bound variables, and a concrete representation for free-variables. This representation seems to have several advantages over pure De Bruijn representations, but requires a non-trivial encoding to convert a language definition from the usual notation to the theorem prover one.

We have implemented support for the locally nameless representation of languages defined using single binders, i.e. Ott definitions with bindpsec annotations of the form `bind` *mv* `in` *nt*. At present this is for the generated Coq code only, not for HOL or Isabelle/HOL. The user can specify which metavariables are to be represented in locally nameless style using a `{{ repr-locally-nameless }}` hom, e.g.

```
metavar var, x ::=
 {{ repr-locally-nameless }}
 {{ tex \mathit{ [[var]] } }} {{ com  term variable  }}
```

They can then write syntax definitions as usual, as in the following syntax for the lambda calculus:

```
grammar
 term, t :: 'term_' ::=                          {{ com term }}
  | x              ::   :: var                       {{ com variable }}
  | \ x . t        ::   :: lam   (+ bind x in t +) {{ com abstraction }}
  | t1 t2          ::   :: app                       {{ com application }}
  | ( t )          :: S :: paren {{ coq [[t]] }}
  | { t2 / x } t1  :: M :: tsub  {{ coq (open_term_wrt_term[[x t1]] [[t2]])}}
```

The only change from the Figure 1 example is in the Coq hom for `tsub` production, which makes use of an `open_term_wrt_term` function; this (and some other locally nameless infrastructure) is automatically generated. No other change to the Ott source is required. In particular, any definitions of judgements, e.g. the definition of the reduction relation of Figure 1, are automatically compiled to use the locally nameless representation. This compilation is described in Section 4.4. It uses cofinite quantification, as advocated by Aydemir *et al.* (2008).

The LNgen tool of Aydemir and Weirich (2009) takes a grammar specified in a proper subset of the Ott input language and generates a locally nameless representation. This is complementary to our work: LNgen produces not just free variable and substitution functions (similar to those that Ott generates), but also theorems about those functions, tactics for choosing fresh names, and a recursion scheme for the definition of functions. However, it does not currently deal with definitions of semantic judgements.

A different representation approach has been followed in Nominal Isabelle (Urban 2008), which provides an Isabelle/HOL package for defining and reasoning about datatypes with binding. Currently, this supports only single binders, and it would be straightforward to compile this subset of the Ott metalanguage to Nominal Isabelle (the biggest difficulty is to rearrange symbolic terms so that binding metavariables appear close to the non-terminals they bind in).

4 Compilation to proof assistant code

Our compilation generates proof-assistant definitions: of types; of functions, for subrule predicates, for the binding auxiliaries of Section 3, for single and multiple substitution and for free variables; and of relations, for the semantic judgements. We generate *well-formed* proof assistant code, without dangling proof obligations, and try also to make it *idiomatic* for each proof assistant, to provide a good basis for mechanised proof. All this is for Coq, HOL and Isabelle/HOL. In simple cases the three are very similar, modulo the details of prover syntax. The complete generated code for the Figure 1 example is shown in Figures 9, 10 and 11, in Section 6 for Coq, HOL and Isabelle/HOL respectively. They are presented exactly as generated except for some whitespace.

4.1 Types

Each metavariable declaration gives rise simply to a proof assistant type abbreviation, for example `Definition var := nat` in the Coq generated from Figure 1. These abbreviations can be suppressed by adding the declaration {{ **phantom** }}, which is useful to avoid duplicate definitions of types already defined in an imported library. For Coq the **coq-equality** generates an equality decidability lemma and proof script for the type:

```
Lemma eq_var: forall (x y : var), x = y + x <> y.
Proof.
  decide equality; auto with ott_coq_equality arith.
Defined.
Hint Resolve eq_termvar : ott_coq_equality.
```

Non-terminal roots with type homs give rise to type abbreviations, as for metavariables. For other non-terminals, in simple cases each non-terminal root of the user's grammar is compiled to a free proof assistant type. For example, the Coq compilation for `term` of Figure 1 generates a free type with three constructors, corresponding to the three non-meta productions of the `term` grammar:

```
(** syntax *)
Inductive term : Set :=
 | t_var : var -> term
 | t_lam : var -> term -> term
 | t_app : term -> term -> term.
```

Non-terminal roots that are not maximal in the subrule order, e.g. the values `val` of Figure 1, are represented using the type generated for the (unique) maximal non-terminal root above them. For these we also generate subrule predicates that carve out the relevant part of that type, e.g. the following Coq definition that picks out the elements of type `term` that represent abstract syntax terms of the `val` grammar:

```
(** subrules *)
  Definition is_val_of_term (t5:term) : Prop :=
    match t5 with
    | (t_var x) => False
    | (t_lam x t) => (True)
    | (t_app t t') => False
  end.
```

The general case is more complex, as the grammars for the maximal non-terminal roots may themselves mention other non-maximal non-terminals (e.g. if the `term` grammar had occurrences of `val` in its definition). In such cases we generate both a type and a predicate. In more detail, the non-free grammar rules are the least subset of the rules that either (1) occur on the left of a subrule (`<::`) declaration, or (2) have a non-meta production that mentions a non-free rule. The subrule predicate for a type is defined by pattern matching over constructors of the maximal type above it – for each non-meta production of the maximal type it calculates a disjunction over all the productions of the lower type that are subproductions of it, invoking other subrule predicates as appropriate.

In general there may also be a complex pattern of mutual recursion among these types. Coq, HOL and Isabelle/HOL all support mutually recursive type definitions (with `Inductive`, `Hol_datatype` and `datatype` respectively), but it is desirable to make each mutually recursive block as small as possible, to simplify the resulting induction principle. Accordingly, we topologically sort the rules according to a dependency order, generating mutually recursive blocks for each connected component and inserting any (singleton) type abbreviations where they fit.

We also have to choose a representation for productions involving list forms. For example, for a language with records one might write

```
metavar label, l ::= {{ hol string }} {{ coq nat }}
indexvar index, n ::=  {{ hol num }} {{ coq nat }}
grammar
term, t :: 't_' ::=
  | { l1 = t1 ,  .. , ln = tn }      :: :: record
```

These records can be represented simply with constructors whose argument types involve proof-assistant native list types, e.g. in HOL:

```
val _ = Hol_datatype `
  term = t_record of (label#term) list  `;
```

For Coq there are two standard choices, native lists or encoding; we support both. The Ott default is to generate native lists, but here the induction principle inferred by Coq is too weak to be useful, so we also generate an appropriate induction principle using nested fixpoints. Alternatively, Ott can translate away the list types, synthesising an additional type for each type of lists-of-tuples that arises in the grammar. Coq is then able to generate useful induction principles. In the example, we need a type of lists of pairs of a `label` and a `term`:

```
Inductive list_label_term : Set :=
 | Nil_list_label_term : list_label_term
 | Cons_list_label_term :
     label -> term -> list_label_term -> list_label_term
with term : Set :=
 | t_record : list_label_term -> term.
```

These are included in the topological sort, and utility functions, e.g. to make and unmake lists, are synthesised. (A similar translation would be needed for Twelf, as it has no polymorphic list type.) We also generate, on request, default Coq proofs that there is a decidable equality on various types.

4.2 Functions

The generated functions are defined by pattern-matching and recursion. The patterns are generated by building canonical symbolic terms from the productions of the grammar. The recursion is essentially primitive recursion: for Coq we produce `Fixpoints` or `Definitions` as appropriate; for HOL we use an `ottDefine` variant of the `Define` package; and for Isabelle/HOL we produce `primrecs` or (on request) `funs`. We have to deal both with the type dependency (the topologically sorted

mutually recursive types described above) and with function dependency – for subrule predicates and binding auxiliaries we may have multiple mutually recursive functions over the same type.

Binding Auxiliaries These functions calculate the intuitive fully concrete interpretations of auxiliary functions defined in bindspecs, as in Section 3.2, giving proof assistant sets or lists, of metavariables or non-terminals, over each type for which the auxiliary is defined.

Substitutions and free variables The generated substitution functions also walk over the structure of the free proof assistant types. Continuing the Figure 1 example, given the declaration

```
substitutions
  single term var :: tsubst
```

we generate Coq code for substituting `t`'s for `x`'s in each generated type as below (The choice of synthesised fresh names could be improved here.):

```
(** substitutions *)
Fixpoint tsubst_term (t5:term) (x5:var) (t_6:term) struct t_6 : term :=
  match t_6 with
  | (t_var x) => (if eq_var x x5 then t5 else (t_var x))
  | (t_lam x t) => t_lam x (if list_mem eq_var x5 (cons x nil) then t
                            else (tsubst_term t5 x5 t))
  | (t_app t t') => t_app (tsubst_term t5 x5 t) (tsubst_term t5 x5 t')
end.
```

For each production, for each occurrence of a non-terminal *nt* within it, we first calculate the things (of whatever type is in question) binding in that *nt*, i.e. those that should be removed from the domain of any substitution pushed down into it. In simple cases these are just the interpretation of the *mse′* (of the right type) from any bind *mse′* in *nt* of the production. The substitution function clause for a production is then of one of two forms: either (1) the production comprises a single element, of the metavariable that we are substituting for, and this is within the rule of the non-terminal that it is being replaced by or (2) all other cases. For (1) the element is compared with the domain of the substitution, and replaced by the corresponding value from the range if it is found. For (2) the substitution functions are mapped over the subelements, having first removed any bound things from the domain of the substitution. (Substitution does not descend through non-terminals with type homs, as they may involve arbitrarily complex user-defined proof assistant types for which it would be unclear what to do, so these should generally only be used at upper levels of a syntax, e.g. to use finite maps for type environments.) The fully concrete interpretation also lets us define substitution for *non-terminals*, e.g. to substitute for compound identifiers such as a dot-form `M.x`. This is all done similarly, but with differences in detail, for single and for multiple substitutions, and for the corresponding free variable functions. For all these we simplify the generated functions by using the dependency analysis of the syntax, only propagating recursive calls where needed.

Context application For context grammars we generate functions that apply contexts to terms. In the Section 2.3 example, the declaration

```
contextrules
  E _:: term :: term
```

makes the tool check that the `E` grammar is a context grammar for `term` with `term` holes. We therefore generate a function taking an `E` and a `term` and returning a `term`, by pattern matching on the (non-meta) `E` productions. The generated Coq version is below:

```
(** context application *)
Definition appctx_E_term (E5:E) (term5:term) : term :=
  match E5 with
  | (E_appL t) => (t_app term5 t)
  | (E_appR v) => (t_app v term5)
  | (E_lam x) => (t_lam x term5)
  | (E_nested t1 t2) => (t_app t1 (t_app term5 t2))
  | (E_tuple v_list t_list) =>
       (t_tuple ((app v_list (app (cons term5 nil) (app t_list nil)))))
end.
```

The right-hand sides of each clause are produced essentially by parsing the corresponding production of the `E` grammar as if it were a symbolic `term`, with the `term` argument (here `t6`) in place of the hole.

Dealing with the proof assistants Each proof assistant introduced its own further difficulties. Leaving aside the purely syntactic idiosyncrasies, which are far from trivial, but not very interesting:

For Coq, when translating lists away, generation of functions over productions that involve list types must respect that translation. We therefore generate auxiliary functions that recurse over those list types. Coq also needs an exact dependency analysis.

For HOL, the standard `Define` package tries an automatic termination proof. This does not suffice for all cases of our generated functions involving list types, so we developed an `ottDefine` variant, with stronger support for proving termination of definitions involving list operators.

For Isabelle/HOL, we chose the `primrec` package, to avoid any danger of leaving dangling proof obligations, and because our functions are all intuitively primitive recursive. Unfortunately, `primrec` (in the then-current Isabelle 2005 or more recent Isabelle 2008 versions) does not support definitions involving several mutually recursive functions over the same type. For these we generate single functions calculating tuples of results, define the intended functions as projections of these, and generate lemmas (and simple proof scripts) characterising them in terms of the intended definitions. Further, `primrec` does not support pattern matching involving nested constructors. We therefore generate auxiliary functions for productions with embedded list types. Isabelle/HOL tuples are treated as nested pairs, so we do the same for productions with tuples of size 3 or more. Isabelle/HOL also requires a function definition for each recursive type. In the case where there are multiple uses

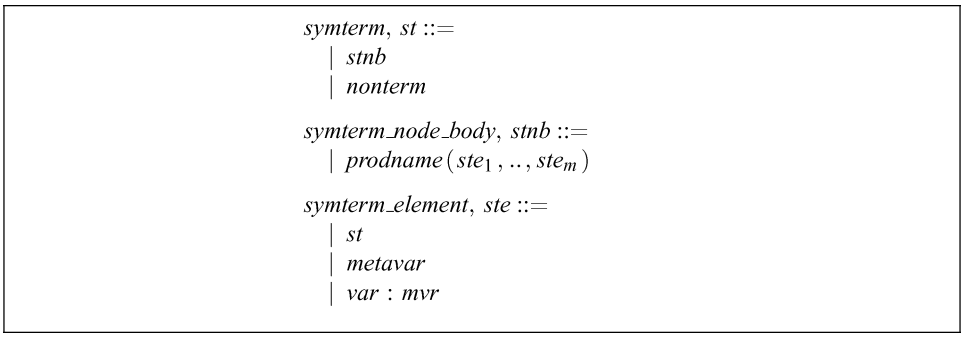


Fig. 6. Mini-Ott in Ott: symbolic terms.

of the same type (e.g. several uses of `term list` in different productions) all the functions we wish to generate need identical auxiliaries, so identical copies must be generated. In retrospect, the choice to use `primrec` is debatable. The recent Isabelle 2008 has a more robust definition package for general functions, called `fun`, which should subsume some of the above. Ott has an option to generate `fun` functions but experiments suggest that the package cannot automatically prove termination for all the generated functions.

4.3 Relations

The semantic relations are defined with the proof-assistant inductive relations packages, `Inductive`, `Hol_reln` and `inductive` (or, on request, `inductive_set`), respectively. For the Figure 1 example, the generated Coq code is as follows:

```
(** definitions *)
(* defns Jop *)
Inductive reduce : term -> term -> Prop :=    (* defn reduce *)
 | ax_app : forall (x:var) (t1 v2:term),
     is_val_of_term v2 ->
     reduce (t_app (t_lam x t1) v2) (tsubst_term v2 x t1)
 | ctx_app_fun : forall (t1 t t1':term),
     reduce t1 t1' ->
     reduce (t_app t1 t) (t_app t1' t)
 | ctx_app_arg : forall (v t1 t1':term),
     is_val_of_term v ->
     reduce t1 t1' ->
     reduce (t_app v t1) (t_app v t1').
```

Note that the tool has added `is_val_of_term` predicates as appropriate, where the user's rules mentioned non-terminals such as `v2` of non-free types, and that the homomorphisms for metaproductions have been in-lined, e.g. the `(tsubst_term v2 x t1)`.

In general, each `defns` block gives rise to a potentially mutually recursive definition of each `defn` inside it. (In contrast to the recursive types representing grammars, where the tool calculates a topological sort, for `defns` it seems clearer for the user to specify the recursive structure.) Definition rules are expressed internally with symbolic terms. We give a simplified grammar thereof in Figure 6, omitting the

symbolic terms for list forms. A symbolic term *st* for a non-terminal root is either an explicit non-terminal or a node, the latter labelled with a production name and containing a list of *symterm_element*s, which in turn are either symbolic terms, metavariables, or variables. Each definition rule gives rise to an implicational clause, essentially that the premises (Ott symbolic terms of the `formula` grammar) imply the conclusion (an Ott symbolic term of whichever judgement is being defined). Symbolic terms are compiled in several different ways:

- Nodes of non-meta productions are output as applications of the appropriate proof-assistant constructor (and, for a subrule, promoted to the corresponding constructor of a maximal rule).
- Nodes of meta productions are transformed with the user-specified homomorphism.
- Nodes of judgement forms are represented as applications of the defined relation in Coq and HOL, and as set-membership assertions in Isabelle/HOL.

Further, for each non-terminal of a non-free grammar rule, e.g. a usage of `v'` where `val<::term`, an additional premise invoking the generated subrule predicate for the non-free rule is added, e.g. `is_val_of_term v'`. For Coq and HOL, explicit quantifiers are introduced for all variables mentioned in the rule.

Supporting list forms requires some additional analysis. For example, consider the record typing rule below:

$$\frac{\Gamma \vdash t_0 : T_0 \quad .. \quad \Gamma \vdash t_{n-1} : T_{n-1}}{\Gamma \vdash \{ l_0 = t_0 , .. , l_{n-1} = t_{n-1} \} : \{ l_0 : T_0 , .. , l_{n-1} : T_{n-1} \}} \quad \textsc{Ty\_Rcd}$$

We analyse the symbolic terms in the premises and conclusion to identify lists of non-terminals and metavariables with the same bounds – here $t_0..t_{n-1}$, $T_0..T_{n-1}$ and $l_0..l_{n-1}$ all have bounds $0..n-1$. To make the fact that they have the same length immediate in the generated code, we introduce a single proof assistant variable for each such collection, with appropriate projections and list maps/foralls at the usage points. For example, the HOL for the above is essentially as follows, with an `l_t_T_list : (label#term#typ) list`.

```
(* Ty_Rcd *)  !(l_t_T_list:(label#term#typ) list) (G:G) .
(EVERY (\b.b)
  (MAP (\(l_,t_,T_). (Ty G t_ T_)) l_t_T_list))
 ==>
(Ty
  G
  (E_record (MAP (\(l_,t_,T_). (l_,t_)) l_t_T_list))
  (T_Rec    (MAP (\(l_,t_,T_). (l_,T_)) l_t_T_list)))
```

This seems to be a better idiom for later proof development than the alternative of three different list variables coupled with assertions that they have the same length. The HOL code for the Rec rules we saw in Section 2.2 is below – note the list-lifted usage of the `is_v_of_t` predicate, and the list appends (`++`) in the conclusion.

```
(* reduce_Rec *)  !(l'_t'_list:(label#term) list)
       (l_v_list:(label#t) list) (l:label) (t:t) (t':t) .
((EVERY (\(l_,v_). is_val_of_term v_) l_v_list) /\
```

```
(( reduce t t' )))
 ==>
(( reduce (t_Rec (l_v_list ++ [(l,t)] ++ l'_t'_list))
          (t_Rec (l_v_list ++ [(l,t')] ++ l'_t'_list))))
```

For the PROJ typing rule we need a specific projection (the HOL EL) to pick out the jth element:

```
(* Ty_Proj *) !(l_T_list:(label#typ) list) (j:index) (G:G) (t:t) .
((( Ty G t (T_Rec (l_T_list)) )))
 ==>
(( Ty
     G
     (t_Proj t  ((\ (l_,T_) . l_) (EL j l_T_list)))
     ((\ (l_,T_) . T_) (EL  j l_T_list))))
```

For Coq, when translating away lists, we have to introduce yet more list types for these proof assistant variables, in addition to the obvious translation of symbolic terms, and, more substantially, to introduce additional inductive relation definitions to induct over them.

4.4 Locally nameless representation

For generation of Coq code using the locally nameless representation, consider again the lambda calculus example from Section 3.5, in which the user specified that term variables should be represented in locally nameless style:

```
metavar var, x ::= {{ repr-locally-nameless }} {{ com  term variable  }}
```

Ott will then generate the datatype below to represent the Section 3.5 term grammar:

```
Inductive term : Set :=
 | t_var_b : nat -> term
 | t_var_f : var -> term
 | t_lam : term -> term
 | t_app : term -> term -> term.
```

Productions containing metavariables susceptible to binding (e.g. `t_var`) give rise to two distinct constructors, one (`t_var_b`) for De Bruijn indices to be used when the metavariable is bound, and one (`t_var_f`) for 'free' variables. The type `var`, together with several useful lemmas and functions, is defined in a `Metatheory` libary, distributed with Ott. Binder metavariables are erased from productions (here the `t_lam` production does not carry a `nat` or `var`), as in De Bruijn representations.

Two groups of support functions are automatically generated: `open` functions, to perform substitutions on De Bruijn indexes, and `lc` predicates, to test whether terms are locally closed. The other support functions, for free variables and free-variable substitutions, are generated if the user declares appropriate **`substitutions`** and **`freevars`** sections.

Ott automatically compiles the symbolic terms that appear in rule definitions into the appropriate terms in locally nameless style. For instance, the typing rule for the simply typed lambda-calculus:

```
E,x1: S |- t : T
------------------ :: lambda
E |- \x1.t : S->T
```

is compiled into its locally nameless representation:

```
Inductive typing : env -> term -> type -> Prop := (* defn typing *)
 | ...
 | typing_lambda : forall (L:vars) (E:env) (t:term) (S T:type),
    (forall x, x \notin L ->
      typing (E & x ~ S) (open_term_wrt_term t (t_var_f x)) T) ->
    typing E (t_lam t) (type_arrow S T).
```

To do so, Ott follows the algorithm below. For each rule,

1. for each non-terminal that appears in the rule, compute the maximal set of binders under which it appears. For example, in the rule above, the maximal set of binders for the non-terminal `t` is the singleton $\{x\}$, and it is the empty set for all the other non-terminals;
2. for each pair of a non-terminal and maximal-binder-set collected in Phase 1, go over all the occurrences of the non-terminal in the rule and open them with respect to all the variables in the maximal binding set except those under which this particular occurrence is bound. In the example, this amounts to opening the occurrence of `t` in the premise with respect to the metavariable `x`;
3. quantify, using cofinite quantification, each metavariable that has been used to open a non-terminal; and
4. add hypotheses about local-closure to guarantee the invariant that if a derivation holds, then the top-level terms involved are locally closed.

This algorithm works well in practice, but in some cases the user may want finer control on which non-terminals are opened, and with respect to which metavariables. Consider for instance the CBV beta-reduction rule:

```
-------------------------  :: ax_app
(\x1.t1) v2 -->  {v2/x1}t1
```

A naive application of the algorithm described above would open the right-hand side occurrence of `t1` with respect to a cofinitely quantified `x`. Substitution would then be used to replace the occurrences of `x` with `v2`, resulting in the awkward term

```
reduce
 (t_app (t_lam t1) v2)
 (tsubst_term v2 x (open_term_wrt_term t1 (t_var_f x)))
```

An idiomatic locally nameless translation of the CBV beta-reduction rule would instead directly rely on the `open` function to substitute `v2` for the bound occurrences of `x` in `t1`, as in:

```
reduce
 (t_app (t_lam t1) v2)
 (open_term_wrt_term t1 v2)
```

To let the user specify this translation behaviour, we introduced special production homomorphisms. In the Section 3.5 production for substitutions,

System	rules	LaTeX	Coq		HOL		Isabelle/HOL	
			defn	mt	defn	mt	defn	mt
untyped CBV lambda (Fig. 1)	3	√	√		√		√	
simply typed CBV lambda	6	√	√	√	√	√	√	√
ML polymorphism	22	√	√		√		√	
TAPL full simple	63	√	√	√	√	√	√	√
POPLmark $F_{<:}$ with records	48	√						
Leroy JFP96 module system	67	√			√			
RG-Sep language	22	√	√	√				
Mini-Ott-in-Ott	55	√					√	hand proofs
LJ: Lightweight Java	85	√					√	√
LJAM: Java Module System	163	√					√	√
OCaml_{light}	310	√	√		√	√	√	

Fig. 7. Case studies.

```
term, t :: 't_' ::= ...
 | { t2 / x } t1 :: M :: tsub  {{ coq (open_term_wrt_term[[x t1]] [[t2]])}}
```

the homomorphism refers to the non-terminal `t1` as `[[x t1]]` instead of the usual `[[t1]]`. The prefixed `x` specifies that occurrences of `t1` should not be opened with respect to the metavariable `x`. The Ott algorithm to compile symbolic terms then translates the `ax_app` rule into the idiomatic Coq shown above.

5 Case studies

Our primary goal is to provide effective tool support for the working semanticist. Assessing whether this has been achieved needs substantial case studies. Accordingly, we have specified various languages in Ott, defining their type systems and operational semantics, as in Figure 7.

These range in scale from toy calculi to a large fragment of OCaml. They also vary in kind: some are post-facto formalizations of existing systems, and some use Ott as a tool in the service of other research goals. Some use it purely for sanity checking and typesetting, whereas others use it also to produce definitions for mechanised proof, in one or more of Coq, HOL and Isabelle/HOL, indicated by the ticks in the 'defn' columns. We have tested whether these definitions form a good basis for such proof by machine-checked proofs of metatheoretic results (generally type preservation and progress), indicated by ticks in the 'mt' columns. We did not aim to prove results in all provers for all examples, but rather all provers for some

```
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
```

Fig. 8. An Ott source file for the `let` fragment of TAPL.

examples, and some substantial examples for each prover: 'mt' cells without a tick indicate that we did not attempt that case. The 'rules' column gives the number of semantic rules in each system, as a crude measure of its complexity. The sources, generated code and proof scripts for most of these systems are available (Sewell & Zappa Nardelli 2007).

TAPL full simple This covers most of the simple features, up to variants, from TAPL (Pierce 2002). It demonstrates the utility of the simple form of modularity provided by Ott. The original TAPL languages were produced using TinkerType (Levin & Pierce 2003) to compose features and check for conflicts. Here we build a system, similar to the TinkerType `sys-fullsimple`, from Ott source files that correspond roughly to the various TinkerType components, each with syntax and semantic rules for a single feature. The Ott source for `let` is shown in Figure 8, to which we add: `bool`, `bool_typing`, `nat`, `nat_typing`, `arrow_typing`, `basety`, `unit`, `seq`, `ascribe`, `product`, `sum`, `fix`, `tuple` and `variant`, together with infrastructure `common`, `common_index`, `common_labels` and `common_typing`.

It also proved easy to largely reproduce the TAPL visual style, and to add subtyping (though we did no metatheory for subtyping).

Leroy JFP96 module system This formalizes the path-based type system of Leroy (1996) (Section 4), extended with a term language and an operational semantics.

RG-Sep language This is a concurrent while language used for work combining Rely-Guarantee reasoning with Separation Logic, defined and proved sound by Vafeiadis and Parkinson (2007).

Mini-Ott-in-Ott This precisely defines the Ott binding specifications (without list forms) with their fully concrete representation and alpha equivalence. The metatheory here is a hand proof that for closed substitutions the two coincide.

LJ and LJAM LJ, by Strniša and Parkinson, is an imperative fragment of Java. LJAM extends that (again using Ott modularity) with a formalization of the core part of JSR-277 and a proposal for JSR-294, which together form a proposal for a Java module system (Strniša *et al.* 2007).

***OCaml*$_{\mathbf{light}}$** OCaml$_{light}$ (Owens 2008) covers a substantial core of OCaml – to a first approximation, all except subtyping, objects, and modules. Notable features that are handled are ML-style polymorphism; pattern matching; mutable references; finiteness of the integer type; definitions of type aliases (added after the conference paper Owens 2008); definitions of record and variant data types; and exception definitions. It does not cover much of the standard library, mutable records, arrays, pattern matching guards, labels, polymorphic variants, objects or modules.

We have tried to make our definition mirror the behaviour of the OCaml system rather closely. The OCaml manual (Leroy *et al.* 2005) defines the syntax with a BNF; our syntax is based on that. It describes the semantics in prose; our semantics is based on a combination of that and our experience with the language.

This proof effort took only around 7–8 man-months, and the preceding definition effort was only another few man-weeks. Compared with our previous experiences this is remarkably lightweight: it has been possible to develop this as an example, rather than requiring a major research project in its own right. Apart from Ott, the work has been aided by HOL's powerful first-order reasoning automation and its inductive definition package, and by the use of the concrete representation.

6 Experience

In this section we assess to what extent Ott succeeds in our goal of providing effective tool support for semantics, and whether Ott language definitions are indeed 'intuitively clear, concise, and easy to read and edit'. On the whole our experience has been positive, and we describe some good points (and some less good points) here. For small calculi it is easy to get started with the tool, and even for large definitions such as (6) and (7) one can focus on the semantic content rather than the LaTeX or proof assistant markup. The proof assistant representations we generate are reasonably uniform, which should enable the development of reusable proof tactics, libraries and idioms, specific to each proof assistant.

This is necessarily only anecdotal evidence, absent the possibility of controlled experiments on a statistically significant sample of language designers, but it is based

on some non-trivial use of the tool, much of which was by people who were not the main Ott designers and implementers.

Informal use We first consider use of the tool for informal semantics, not mechanised in a proof assistant. Here the main alternative would be to use LaTeX directly, and escaping the syntactic overhead of reading and writing LaTeX source is a big win, as can be seen even in the small examples of Figures 1 and 3. The homs allow a very modular approach for specifying the LaTeX output: a change in a hom applies automatically everywhere it is used, which could only be achieved in LaTeX with considerable coding.

The lightweight error checking that one gets from parsing symbolic terms in rules, and by enforcing naming conventions, is also a big win. In our previous work on the Acute language (Sewell *et al.* 2004; Sewell *et al.* 2007a) we wrote a complete language definition, of around 80 dense pages. That was typeset using an early predecessor of Ott, typesetting ASCII syntax with a tool that essentially lexed the source and translated it token by token. Keeping the definitions self-consistent during development was a major problem. In contrast, as Ott understands the grammar of symbolic terms (including the grammar of judgements), and parses the input, it detects many simple errors very quickly. This quick feedback was also reported as something the students liked when Ott was used for teaching (Vitek, written personal communication, January 2009).

The quality of the typeset output is on the whole good, as shown in Figures 2 and 3, and the published papers on LJAM, OCaml$_{light}$, and the other usages. The output is fairly customisable, but not arbitrarily so, so one has to give up some control. However, we have only very rarely been moved to paste and hand-edit the generated LaTeX, which is key for ongoing use.

As for negative points, the user does sometimes need to write some Ott boilerplate, e.g. a formula for checking equality for each non-terminal where that is used, and grammar rules for instances of option types. Much of this would be eliminated if Ott allowed parameterised non-terminals. The Ott error reporting could also be improved. There is also only limited control of the typesetting layout, e.g. for line breaks or tabbed alignment within typeset symbolic terms.

Generation of proof assistant definitions Here the basic point is that the generated definitions are accepted by the provers and are usable as a basis for mechanised proof, again without hand-editing. This is indicated by the presence of ticks in the 'mt' column of Figure 7 for all three target provers, including type preservation and progress proofs for the most substantial examples (LJAM and OCaml$_{light}$, machine-checked in Isabelle/HOL and HOL respectively).

Generation of proof assistant definitions for multiple provers from the same Ott source file also works. We did this for the modest examples of the simply typed CBV lambda calculus and the TAPL 'full simple' system, proving type soundness results in each prover. We are told also that Delaware *et al.* (2009) took the Ott sources for LJ, which had data only for Isabelle/HOL output, and found it quick and painless to add homs to generate a Coq definition. They then hand-edited that to add the extensions they were interested in, though it appears that at least some of this hand-editing was to use Coq native lists, which Ott does support.

One should ask whether the generated prover code is really idiomatic. For the Figure 1 example, given the choice of the fully concrete representation for binding, we think the generated prover code is essentially identical to what one might write by hand, except perhaps for some extra parentheses and the names of some variables. The complete generated code for the Figure 1 example is shown in Figures 9, 10 and 11, for Coq, HOL and Isabelle/HOL respectively. Some users might make greater use of the prover fancy syntax support, especially for Isabelle/HOL using the Proof General interface (Aspinall 2000). In bigger examples there are some cases where Ott is not expressive enough. For example, in LJAM there are various lookup functions, e.g. to find a class definition. As Ott does not currently support user-defined functions, these were written as inductive relation definitions, but then proved equivalent to functional versions defined directly in Isabelle/HOL (using filtering to translate into the representation type of abstract syntax terms).

Is it easier to read and write Ott source than prover code? For the Figure 1 example, one can contrast the Ott source with the prover code in Figures 9, 10 and 11. One can puzzle out the latter without much effort, but we do find the former more transparent. The gain in readability becomes more significant for larger examples, where one might have a hundred times as many rules and a great deal of boilerplate substitution code. For OCaml$_{light}$ the Ott source, the typeset specification, and the generated HOL were all useful, and the experience with LJAM is 'definitely'. The direct support for user syntax, lists, subgrammars and context grammars is all useful. In general, if one is working with semantics expressed as rules over the abstract syntax, we expect the answer will be 'often, yes'. But an expert in a particular prover can certainly get things done directly. Moreover, if one is mostly working with sets or some other prover library types and operations, not mostly with an abstract syntax, or if one is making non-trivial use of prover type classes or dependent types, then the answer would be 'no'.

The typesetting support of all three provers is limited. Isabelle has perhaps the most sophisticated system, but it seems to be token-based, not grammar-based, so there is no analogue of our `tex` homs that would allow easy control of the typesetting of each syntactic form without editing the main source. It is also not convenient to quote parts of a development (including type definitions) out of order in another document. For HOL, we have used Wansbrough's `HolDoc` tool, which does support quoting but is also token-based (albeit with prefix operators) and is less robust than one might hope. For Coq, `coqdoc` supports production of HTML for documenting libraries, but work such as the Clight of Blazy and Leroy (2009) is typeset by manual transliteration of Coq specifications; as they note, this can introduce or (worse) mask errors.

Binding specification and representation The tool currently implements the fully concrete representation, for arbitrary binding specifications, and the locally nameless representation, for single-binder specifications only.

In cases where the fully concrete representation suffices, it is very easy to work with, and the expressiveness of arbitrary Ott binding specifications is useful. This includes a surprising range of languages, including our LJAM and OCaml$_{light}$ examples and their progress and type preservation proofs. The former does not depend on alpha

```
(* generated by Ott 0.10.17 from: ../tests/non_super_tabular.ott ../tests/test10.ott *)

Require Import Arith.
Require Import Bool.
Require Import List.

(** syntax *)
Definition var := nat.
Lemma eq_var: forall (x y : var), {x = y} + {x <> y}.
Proof.
  decide equality; auto with ott_coq_equality arith.
Defined.
Hint Resolve eq_var : ott_coq_equality.


Inductive term : Set :=
 | t_var : var -> term
 | t_lam : var -> term -> term
 | t_app : term -> term -> term.


(** library functions *)
Fixpoint list_mem (A:Set) (eq:forall a b:A,{a=b}+{a<>b}) (x:A) (l:list A) {struct l} : bool :=
  match l with
  | nil => false
  | cons h t => if eq h x then true else list_mem A eq x t
end.
Implicit Arguments list_mem.


(** subrules *)
Definition is_val_of_term (t5:term) : Prop :=
  match t5 with
  | (t_var x) => False
  | (t_lam x t) => (True)
  | (t_app t t') => False
end.


(** substitutions *)
Fixpoint tsubst_term (t5:term) (x5:var) (t_6:term) {struct t_6} : term :=
  match t_6 with
  | (t_var x) => (if eq_var x x5 then t5 else (t_var x))
  | (t_lam x t) => t_lam x (if list_mem eq_var x5 (cons x nil) then t else(tsubst_term t5 x5 t))
  | (t_app t t') => t_app (tsubst_term t5 x5 t) (tsubst_term t5 x5 t')
end.


(** definitions *)

(* defns Jop *)
Inductive reduce : term -> term -> Prop :=    (* defn reduce *)
 | ax_app : forall (x:var) (t1 v2:term),
     is_val_of_term v2 ->
     reduce (t_app  (t_lam x t1)  v2)  (tsubst_term  v2   x   t1 )
 | ctx_app_fun : forall (t1 t t1':term),
     reduce t1 t1' ->
     reduce (t_app t1 t) (t_app t1' t)
 | ctx_app_arg : forall (v t1 t1':term),
     is_val_of_term v ->
     reduce t1 t1' ->
     reduce (t_app v t1) (t_app v t1').
```

Fig. 9. Generated Coq from Figure 1.

```
(* generated by Ott 0.10.17 from: ../tests/non_super_tabular.ott ../tests/test10.ott *)
theory test10
imports Main Multiset
begin

(** syntax *)
types "var" = "string"

datatype "term" =
   t_var "var"
 | t_lam "var" "term"
 | t_app "term" "term"



(** subrules *)
consts
is_val_of_term :: "term => bool"
primrec
"is_val_of_term (t_var x) = (False)"
"is_val_of_term (t_lam x t) = ((True))"
"is_val_of_term (t_app t t') = (False)"


(** substitutions *)
consts
tsubst_term :: "term => var => term => term"
primrec
"tsubst_term t5 x5 (t_var x) = ((if x=x5 then t5 else (t_var x)))"
"tsubst_term t5 x5 (t_lam x t) = (t_lam x (if x5 mem [x] then t else (tsubst_term t5 x5 t)))"
"tsubst_term t5 x5 (t_app t t') = (t_app (tsubst_term t5 x5 t) (tsubst_term t5 x5 t'))"


(** definitions *)
(*defns Jop *)
inductive reduce :: "term \<Rightarrow> term \<Rightarrow> bool"
where
(* defn reduce *)

ax_appI: "\<lbrakk>is_val_of_term v2\<rbrakk> \<Longrightarrow>
reduce ((t_app  (t_lam x t1)  v2)) ( (tsubst_term  v2   x   t1 ) )"

| ctx_app_funI: "\<lbrakk>reduce (t1) (t1')\<rbrakk> \<Longrightarrow>
reduce ((t_app t1 t)) ((t_app t1' t))"

| ctx_app_argI: "\<lbrakk>is_val_of_term v ;
reduce (t1) (t1')\<rbrakk> \<Longrightarrow>
reduce ((t_app v t1)) ((t_app v t1'))"



end
```

Fig. 10. Generated Isabelle/HOL from Figure 1.

equivalence at all. In the latter, the need for alpha-equivalence-aware reasoning arises only for type variables and type schemes. We used a De Bruijn encoding of type variables to support the formal proof effort. Since Ott does not currently support the automatic generation of such representations for HOL, we dealt directly with the index shifting functions in the Ott source, which was relatively straightforward.

Of course, there are many important cases where the fully concrete representation would *not* suffice, and where a manual encoding of terms up to alpha equivalence would be heavy. These include many dependently typed systems, systems where one needs to reduce under binders, and work considering the contextual semantics of arbitrary subterms under binders (instead of solely the behaviour of whole

```
(* generated by Ott 0.10.17 from: ../tests/non_super_tabular.ott ../tests/test10.ott *)
(* to compile: Holmake test10Theory.uo   *)
(* for interactive use:
  app load ["pred_setTheory","finite_mapTheory","stringTheory","containerTheory","ottLib"];
*)

open HolKernel boolLib Parse bossLib ottLib;
infix THEN THENC |-> ## ;
local open arithmeticTheory stringTheory containerTheory pred_setTheory listTheory
  finite_mapTheory in end;

val _ = new_theory "test10";

(** syntax *)
val _ = type_abbrev("var", ``:string``);
val _ = Hol_datatype `
term =
   t_var of var
 | t_lam of var => term
 | t_app of term => term
`;


(** subrules *)
val _ = ottDefine "is_val_of_term" `
    ( is_val_of_term (t_var x) = F)
/\  ( is_val_of_term (t_lam x t) = (T))
/\  ( is_val_of_term (t_app t t') = F)
`;

(** substitutions *)
val _ = ottDefine "tsubst_term" `
    ( tsubst_term t5 x5 (t_var x) = (if x=x5 then t5 else (t_var x)))
/\  ( tsubst_term t5 x5 (t_lam x t) = t_lam x (if MEM x5 [x] then t else (tsubst_term t5 x5 t)))
/\  ( tsubst_term t5 x5 (t_app t t') = t_app (tsubst_term t5 x5 t) (tsubst_term t5 x5 t'))
`;

(** definitions *)
(* defns Jop *)

val (Jop_rules, Jop_ind, Jop_cases) = Hol_reln `
(* defn reduce *)

( (* ax_app *)  !(x:var) (t1:term) (v2:term) . (clause_name "ax_app") /\
((is_val_of_term v2))
 ==>
( ( reduce (t_app  (t_lam x t1)  v2)  (tsubst_term  v2   x   t1 )  )))

/\ ( (* ctx_app_fun *)  !(t1:term) (t:term) (t1':term) . (clause_name "ctx_app_fun") /\
(( ( reduce t1 t1' )))
 ==>
( ( reduce (t_app t1 t) (t_app t1' t) )))

/\ ( (* ctx_app_arg *)  !(v:term) (t1:term) (t1':term) . (clause_name "ctx_app_arg") /\
((is_val_of_term v) /\
( ( reduce t1 t1' )))
 ==>
( ( reduce (t_app v t1) (t_app v t1') )))

`;



val _ = export_theory ();
```

Fig. 11. Generated HOL from Figure 1.

programs). When working with small calculi rather than full languages, one often does not need complex binding specifications (single binders are enough) and there the locally nameless representation should suffice. Our preliminary experiments formalising the simply typed lambda calculus, $F_{<:}$, and (following Benton & Koutavas 2007) the nu-calculus of Pitts (1993), suggest that the generated Coq is usable in this case.

Type system If one considers the type system of the Ott metalanguage, ignoring syntactic issues, it essentially supports inductively defined relations over a term language of mutually recursive labelled sums of products, with subtyping (from subrule declarations) and hard-coded support for lists of products, over uninterpreted proof-assistant type expressions. This is combined with support for context grammars and for simple mixin-like modules.

The TAPL and LJ examples show that these simple modules can be effective: the TAPL features are defined in separate files, roughly following the structure of the TinkerType repository used to build the original text (Levin & Pierce 2003), and the LJAM definitions reuse most of LJ.

There is no support for parameterised grammars or for polymorphic term constructors, both of which would be very useful, e.g. for a polymorphic user-defined option type, or for a polymorphic equality formula, or for conditionals at arbitrary types. There is also no support for user-defined (total) recursive functions over the term language. This would also be very useful, and we have experimented with some implementation, but it really requires polymorphism to make the right-hand sides of typical functions easy to express.

General Ott was developed pragmatically to provide useful tool support. This has good and bad effects: when working within its scope, it does (in our experience) make it remarkably easy to write and work with semantic definitions, but there is an accumulation of features. There is no doubt scope for some re-engineering.

The tool provides a relatively smooth path from informal to formal semantics: one can quickly get production-quality definitions for typesetting (arguably with a gentler learning curve than that of the provers) and then shift to generating prover code. Of course, some rearrangement of the definitions may be needed at that point, but we expect this will often be minor. One can retarget Ott definitions between provers, but at present there is no automatic way to port prover definitions up to Ott.

Of course, the tool also has the usual disadvantages of a pre-processor: one has a somewhat more complex build process, with error messages at different points. One could imagine a much tighter integration of the tool (or of specific features) with the provers, e.g. to get user syntax appearing in goals.

As outlined here, the analysis and code generation performed by Ott is reasonably complex (the tool is around 24,000 lines of OCaml). It is therefore quite possible that the generated code is not what is intended, either because of soundness bugs in the tool (though none such are known at present) or through misunderstanding of its semantics, and one should not treat the tool as part of a trusted chain – it is necessary in principle to look over the generated definitions. In any proof effort,

however, one will have to become intimately familiar with those definitions in any case, so we do not regard this as a problem.

7 Related work

As Strachey (1966) writes in the Proceedings of the first IFIP Working Conference, *Formal Language Description Languages*:

> 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.

The problem of dealing precisely with this notation, with the need for machine support in doing so, has spawned an extensive literature. We discuss only the most related previous work.

The proof assistants that we build on, Coq, HOL, and Isabelle/HOL, together with Twelf, are perhaps the most directly related work (Coq 2008; HOL 2007; Isabelle 2008; Twelf 2005). Ever since original LCF (Milner 1972), one of the main intended applications of these and related systems has been reasoning about programs and programming languages, and they have been vastly improved over the years to make this possible. Recently they have been used for a variety of substantial languages, including for example the verifying compiler work of Blazy *et al.* (2006) (Coq), a C expression semantics by Norrish (1999) (HOL), work on Java by Klein and Nipkow (2006) (Isabelle/HOL) and an internal language for SML by Lee *et al.* (2007) (Twelf). They are, however, all more-or-less general-purpose tools – by adding front-end support that is specific to the problem of defining programming language syntax and semantics, we believe Ott can significantly ease the problems of working with large language definitions.

Several projects have aimed at automatically generating *programming environments* and/or *compilers* from language descriptions, including early work on the Synthesiser Generator (Reps & Teitelbaum 1984). Kahn's CENTAUR system (Borras *et al.* 1988) supported natural-semantics descriptions in the TYPOL language, compiling them to Prolog for execution, together with a rich user interface including an editor, and a language METAL to define abstract and concrete syntax (Terrasse 1995 also considered compilation of TYPOL to Coq). Related work by Klint (1993) and colleagues produced the ASF+SDF Meta-environment. Here SDF provides rich support for defining syntax, while ASF allows for definitions in an algebraic specification style. Again it is a programming environment, with a generic syntax-directed editor. The ERGO Support System (Lee *et al.* 1988) also had a strong user-interface component, but targeted (among others) ADT-OBJ and λProlog. Mosses's work on Action Semantics and Modular SOS (Mosses 2002) has been supported by various tools, but makes strong assumptions on the form of the semantic relations being defined. Moving closer in goals to Ott, ClaReT (Boulton 1997) took a sophisticated description of syntax and pretty printing, and a denotational semantics, and generated HOL definitions.

In contrast to the programming environments above, Ott is a more lightweight stand-alone tool for definitions, designed to fit in with existing editing, LaTeX and proof-assistant work flows and requiring less initial investment and commitment to use. (Its support for production parsing and pretty printing is less developed than several of the above, however.) Moreover, in contrast to CENTAUR and to research on automatic compiler generation, Ott is not focussed on producing *executable* definitions – one can define arbitrary semantic relations which may or may not be algorithmic. The generality of these arbitrary inductive relation definitions means that Ott should be well-suited to much present-day semantics work, for type systems and/or operational semantics.

The Jakarta toolset, by Barthe *et al.* (2001), shares many high-level goals with Ott. It aims to combine the advantages of a semantics prototyping environment, with support for readable specifications and animation, with those of a prover. Its JSL specification language can be compiled into (Coq) prover code; JSL specifications are 'easy to read, extend, and manipulate'. The details are very different: JSL types are first-order polymorphic types, and it supports functions defined by conditional rewrite rules. The system generates proof support, including inversion principles for such functions.

The SL system of Xiao *et al.* (2000) has a metalanguage for specifying semantics with conditional rewrite rules and evaluation contexts. Dynamic constraints can be declared as disjunctions of patterns, e.g. for specifying object-language values, and contexts and hole-filling are typed. The emphasis is on compilation of such definitions to interpreters. Later work considered automation of unique decomposition proofs (Xiao *et al.* 2001). PLTredex (Matthews *et al.* 2004) is a domain-specific language for expressing reduction relation definitions and animating them. It is currently being used on a 'full-language' scale, for an R6RS Scheme definition (Sperber *et al.* 2007), but is by design restricted to animation of reduction semantics. The Ruler system (Dijkstra & Swierstra 2006) provides a language for expressing type rules, generating LaTeX and implementations but not proof assistant definitions, used for a Haskell-like language. The SASyLF system of Aldrich *et al.* (2008) has a simple Ott-like language for user-defined syntax and semantic rules, but with an integrated proof language. It is intended for educational use, not for large-scale semantics.

Turning to direct support for binding, Twelf is suited to HOAS representations. FreshML (Shinwell *et al.* 2003), Alpha Prolog (Cheney & Urban 2004) and MLSOS (Lakin & Pitts 2007) use nominal logic-programming and functional-programming approaches, the latter two with a view to prototyping of semantics. The Nominal Isabelle package (Urban 2008) integrates support for datatype definitions with (at present) single binders into Isabelle/HOL. Cαml (Pottier 2006) is the most substantial other work we are aware of that introduces a large and precisely defined class of binding specifications, from which it generates OCaml code for type definitions and substitutions. Types can be annotated with sets of atom sorts, with occurrences of atoms of those sorts treated as binding within them. inner and outer annotations let one specify that subterms are either inside or outside an enclosing binder. For example, a lambda calculus with single binders might be expressed as

follows (this is an extract of an example from the Cαml distribution, omitting the patterns that it includes):

```
type expression =
  | EVar of atom var
  | ELambda of < lambda >
  | EApp of expression * expression

type lambda binds var =
    atom var * inner expression
```

This seems to us somewhat less intuitive than the Ott binding specifications, especially in more complex examples. We conjecture that the two have mutually incomparable expressiveness.

Representing binding within proof assistants was a key aspect of the POPLmark challenge (Aydemir *et al.* 2005), and several comparisons have been produced (Berghofer & Urban 2006; Aydemir *et al.* 2008). Owens (1995) discusses pattern binding using locally nameless representations in Isabelle/HOL. We mentioned the LNgen tool of Aydemir and Weirich (2009) in Section 3.5.

The work on concise concrete syntax by Tse and Zdancewic (2008) has similar lightweight syntax definition goals to Ott, taking a concise description of a grammar but producing the conventional object-language parsing and pretty printing tools.

It is interesting to contrast our OCaml fragment example with attempts to verify aspects of the SML Definition. Early attempts, by Syme (1993), VanInwegen (1996) and Gunter and Maharaj (1995), faced severe difficulties, both from the mathematical style of the Definition and the limitations of HOL at the time whereas, using Ott and HOL 4, we have found our example reasonably straightforward. Lee *et al.* (2007) take a rather different approach. They factor their (Twelf) definition into an internal language, and a substantial elaboration from a source language to that. They thus deal with a much more sophisticated type theory (aimed at supporting source features that we do not cover, including SML modules), so the proof effort is hard to compare, but their semantic rules are further removed from source-language programs.

8 Conclusion

Summary We have introduced the Ott metalanguage and tool for expressing semantics, incorporating metalanguage design to make definitions easy to read and edit, a novel and expressive metalanguage for expressing binding, and compilation to multiple proof assistants.

We hope that this work will enable a phase change: from the current state, in which working with fully rigorous definitions of real programming languages requires heroic effort, to a world in which that is routine.

The Ott tool can be used in several different ways. Most simply, it can aid informal LaTeX mathematics, permitting definitions, and terms in proofs and exposition, to be written without syntactic noise. By parsing (and so sort-checking) this input it quickly catches a range of simple errors, e.g. inconsistent use of judgement forms.

There is then a smooth path to fully rigorous proof assistant definitions: those Ott definitions can be annotated with the additional information required to generate proof assistant code. In general one may also want to restructure the definitions to suit the formalization. Our experience so far suggests that this is not a major issue, and hence that one can avoid early commitment to a particular proof assistant. The tool can be used at different scales: it aims to be sufficiently lightweight to be used for small calculi, but it is also designed and engineered with the pragmatics of working with full-scale programming languages in mind. Our case studies suggest that it achieves both goals. Furthermore, we hope it will make it easy to re-use definitions of calculi and languages, and also fragments thereof, across the community. Widely accepted de facto standard definitions would make it possible to discuss proposed changes to existing languages in terms of changes to those definitions, rather than solely in terms of toy calculi.

Future work There are many interesting directions for future work. Ott is intended as a pragmatic and useful tool, and several extensions would be highly desirable for many applications:

1. While the fully concrete representation of binding is surprisingly widely applicable, it is far from being able to express all one would like to do. The tool should also be able to generate proof assistant definitions using up-to-alpha representations, e.g. in locally nameless or nominal styles. We have done the former for the single-binder case, but how to do this for arbitrary Ott binding specifications, which are very expressive, is an open problem, and one might well need to initially restrict to some better-behaved class.
2. The tool should directly support user-defined functions, in addition to (but along the same lines as) the current support for relations.
3. Improved support for multiple overlapping languages is needed, e.g. for sugared and non-sugared languages. At present only very simple sugared forms, that can be translated away with a context-free proof assistant hom, are supported. This might be coupled with better support for modular definitions.
4. The Coq, HOL and Isabelle/HOL output stages are all similar, in that usages of homs and auxiliary functions can all be expressed simply with proof assistant functions. A Twelf output stage would also be desirable, but needs a translation into a relational style.

Closer integration of Ott (or of some of its features) with the proof assistants would also be desirable.

A more mathematical question is to consider in what sense the definitions Ott generates for the different target proof assistants have the same meaning. This is intuitively plausible when one considers the generated definitions, but the targets are based on different logics, so it is far from trivial.

With more experience using the tool, we aim also to polish the generated proof-assistant definitions and improve the available proof automation – for example, to make proof scripts less dependent on the precise structure and ordering of the definitions.

Being able to easily generate definitions for multiple proof assistants also opens up new possibilities for (semi-)automatically *testing* conformance between semantic definitions and production implementations, above the various proof assistant support for proof search, tactic-based symbolic evaluation, code extraction from proofs, and code generation from definitions.

Finally, we look forwards to further experience and user feedback from the tool.

Acknowledgments

We thank the other members of the POPLmark team, especially Benjamin Pierce, Stephanie Weirich and Steve Zdancewic, for interesting discussions on this work, James Leifer for comments on a draft; Arthur Charguéraud for his work on the `Metatheory` library for Coq; our early adopters for user feedback; and Keith Wansbrough, Matthew Fairbairn and Tom Wilkie for their work on various Ott predecessors. We acknowledge the support of EPSRC grants GR/T11715, EP/C510712 and EP/F036345, and a Royal Society University Research Fellowship (Sewell).

References

Aldrich, J., Simmons, R. J. & Shin, K. (2008) SASyLF: An Educational Proof Assistant for Language Theory. In *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education (FDPE '08)*. ACM, Victoria, BC, pp. 31–40.

Aspinall, D. (2000) Proof general: A generic tool for proof development. In *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Proceedings*, Graf, S. & Schwartzbach, M. I. (eds), Lecture Notes in Computer Science, vol. 1785. Springer-Verlag, Berlin, pp. 38–42.

Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N., Pierce, B. C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S. & Zdancewic, S. (2005) Mechanized metatheory for the masses: The POPLmark challenge. In *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Proceedings*, Hurd, J. & Melham, T. F. (eds), Lecture Notes in Computer Science, vol. 3603, Springer, Oxford, pp. 50–65.

Aydemir, B., Charguéraud, A., Pierce, B. C., Pollack, R. & Weirich, S. (2008) Engineering Formal Metatheory. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, San Francisco, pp. 3–15.

Aydemir, B. & Weirich, S. (2009) LNgen: Tool support for locally nameless representation. Available at: `http://www.cis.upenn.edu/~baydemir/papers/lngen/` Accessed 8 January 2010.

Barthe, G., Dufay, G., Huisman, M. & de Sousa, S. M. (2001) Jakarta: A toolset to reason about the javacard platform. In *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Proceedings*, Attali, I. & Jensen, T. (eds), Lecture Notes in Computer Science, vol. 2140. Springer-Verlag, Cannes, pp. 2–18.

Benton, N. & Koutavas, V. (2007) A mechanized bisimulation for the nu-calculus. Available at: `http://research.microsoft.com/en-us/um/people/nick/nubisim.pdf` Accessed 8 January 2010.

Berghofer, S. & Urban, C. (2006) A head-to-head comparison of de Bruijn indices and names. In *Proceedings of International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP), ENTCS 174(5)*, pp. 53–67.

Blazy, S. & Leroy, X. (2009) Mechanized semantics for the Clight subset of the C language, *J. Autom. Reasoning*, 43 (3): 263–288.

Blazy, S., Dargaye, Z. & Leroy, X. (2006) Formal Verification of a C compiler front-end. In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, Misra, J., Nipkow, T. & Sekerinski, E. (eds), Lecture Notes in Computer Science, vol. 4085. Springer-Verlag, Hamilton, Canada, pp. 460–475.

Borras, P., Clément, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B. & Pascual, V. (1988) CENTAUR: The system. In *Proceedings of the third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 3)*. ACM, pp. 14–24.

Boulton, R. J. (1997) A tool to support formal reasoning about computer languages. In *Tools and Algorithms for the Construction and Analysis of Systems, Third International Workshop, TACAS '97, Proceedings*, Brinksma, E. (ed), Lecture Notes in Computer Science, vol. 1217. Springer, Enschede, The Netherlands, pp. 81–95.

Cardelli, L., Martini, S., Mitchell, J. C. & Scedrov, A. (1994) An extension of system F with subtyping, *Inf. Comput.*, 109 (1/2): 4–56.

Charguéraud, A. (2006) Annotated bibliography for formalization of lambda-calculus and type theory. Available at: `http://arthur.chargueraud.org/projects/binders/biblio.php` Accessed 8 January 2010.

Cheney, J. & Urban, C. (2004) Alpha-Prolog: A logic programming language with names, binding and alpha-equivalence. In *Logic Programming, 20th International Conference, ICLP 2004, Proceedings*, Demoen, B. & Lifschitz, V. (eds), Lecture Notes in Computer Science, no. 3132. Springer-Verlag, Saint-Malo, France, pp. 269–283.

Coq. (2008). The Coq proof assistant, v.8.1. Available at: `http://coq.inria.fr/` Accessed 8 January 2010.

Curien, P.-L. & Ghelli, G. (1991) Subtyping + Extensionality: Confluence of beta-eta-top reduction in $F_{<=}$. In *Theoretical Aspects of Computer Software, International Conference, TACS '91, Proceedings*, Ito, T. & Meyer, A. R. (eds), Lecture Notes in Computer Science, vol. 526. Springer, Sendai, Japan, pp. 731–749.

Delaware, B., Cook, W. & Batory, D. (2009) A machine-checked model of safe composition. In *Proceedings of the 8th Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, Charlottesville, Virginia, ACM pp. 31–35.

Dijkstra, A. & Swierstra, S. D. (2006) Ruler: Programming type rules. In *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Proceedings*, Hagiya, M. & Wadler, P. (eds), Lecture Notes in Computer Science, vol. 3945. Springer, Fuji-Susono, Japan, pp. 30–46.

Fournet, C., Gonthier, G., Lévy, J.-J., Maranget, L. & Rémy, D. (1996) A calculus of mobile agents. In *CONCUR '96, Concurrency Theory, 7th International Conference, Proceedings*, Montanari, U. & Sassone, V. (eds), Lecture Notes in Computer Science, vol. 1119. Springer, Pisa, pp. 406–421.

Fournet, C., Guts, N. & Zappa Nardelli, F. (2008) A formal implementation of value commitment. In *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Proceedings*, Drossopoulou, S. (ed.), Lecture Notes in Computer Science, vol. 4960. Springer, Budapest, pp. 383–397.

Gray, K. E. (2008) Safe cross-language inheritance. In *ECOOP 2008 – Object-Oriented Programming, 22nd European Conference, Proceedings*, Vitek, J. (ed.), Lecture Notes in Computer Science, vol. 5142. Springer, Paphos, Cyprus, pp. 52–75.

Greenberg, M., Pierce, B. & Weirich, S. Contracts made manifest. In *Proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, Madrid. 2009.

Gunter, E. & Maharaj, S. (1995) Studying the ML module system in HOL. *The Computer Journal: Special Issue on Theorem Proving in Higher Order Logics*, 38 (2): 142–151.

HOL. (2007) The HOL 4 system, Kananaskis-4 release. Available at: `http://hol.sourceforge.net/` Accessed 8 January 2010.

Isabelle. (2008) Isabelle 2008. Available at: `http://isabelle.in.tum.de/` Accessed 8 January 2010.

Jia, L., Zhao, J., Sjöberg, V. & Weirich, S. Dependent types and program equivalence. In *Proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, Madrid. 2009.

Kahrs, S. (1993) *Mistakes and Ambiguities in the Definition of Standard ML*. Tech. Rep., ECS-LFCS-93-257. University of Edinburgh.

Klein, G. & Nipkow, T. (2006) A machine-checked model for a Java-like language, virtual machine, and compiler, *ACM Trans. Program. Lang. Syst.*, 28 (4): 619–695.

Klein, G., Nipkow, T. & Paulson, L. (eds) (2009) The archive of formal proofs. Available at: `http://afp.sf.net` Accessed 8 January 2010.

Klint, P. (1993) A meta-environment for generating programming environments, *ACM Trans. Softw. Eng. Method.*, 2 (2): 176–201.

Lakin, M. R. & Pitts, A. M. (2007) A Metalanguage for Structural Operational Semantics. In Trends in Functional Programming, *Eighth Symposium on Trends in Functional Programming (TFP 2007)*. Draft proceedings, Morazán, M. T. (ed.), vol. 8, Intellect, New York, pp. 19–35.

Lee, D. K., Crary, K. & Harper, R. (2007) Towards a Mechanized Metatheory of Standard ML. In *Proceedings 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, Nice, pp. 173–184.

Lee, P., Pfenning, F., Rollins, G. & Scherlis, W. (1988) The Ergo Support System: An integrated set of tools for prototyping integrated environments. In *Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM, Boston, pp. 25–34.

Leroy, X., Doligez, D., Garrigue, J., Rémy, D. & Vouillon, J. (2005) *The Objective Caml System Release 3.09 Documentation and User's Manual*. URL `http://caml.inria.fr/pub/docs/manual-ocaml/`(3.11 from 2008) Accessed 8 January 2010.

Leroy, X. (1996) A syntactic theory of type generativity and sharing, *J. Funct. Program.*, 6 (5): 667–698.

Levin, M. Y. & Pierce, B. C. (2003) Tinkertype: A language for playing with formal systems, *J. Funct. Program.*, 13 (2). 295–316.

Matthews, J., Findler, R. B., Flatt, M. & Felleisen, M. (2004) A visual environment for developing Context-sensitive term rewriting systems. In *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Proceedings*, van Oostrom, V. (ed.), Lecture Notes in Computer Science, vol. 3091. Springer, Aachen, Germany, pp. 301–311.

McPeak, S. & Necula, G. C. (2004) Elkhound: A fast, practical GLR parser generator. In *Compiler Construction, 13th International Conference, CC 2004, Proceedings*, Duesterwald, E. (ed.), Lecture Notes in Computer Science, vol. 2985. Springer, Barcelona, pp. 73–88.

Milner, R. (1972) Implementation and applications of Scott's logic for computable functions. In *Proceedings ACM Conference on Proving Assertions About Programs*, ACM, Las Cruces, New Mexico, pp. 1–6.

Milner, R., Tofte, M. & Harper, R. (1990) *The Definition of Standard ML*. MIT Press.

Moors, A., Piessens, F. & Odersky, M. (2008) Safe type-level abstraction in Scala. International Workshop on Foundations of Object-Oriented Languages. *FOOL workshop*, San Francisco. `http://fool08.kuis.kyoto-u.ac.jp/program.html` Accessed 8 January 2010.

Mosses, P. D. (2002) Pragmatics of modular SOS. In *Proceedings of 9th International Conference on Algebraic Methodology and Software Technology (AMAST '02), LNCS 2442*,

Kirchner, H. & Ringeissen, C. (eds), Lecture Notes in Computer Science, vol. 2422. Springer, Saint-Gilles-les-Bains, pp. 21–40.

Norrish, M. (1999) Deterministic expressions in C. In *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, Proceedings*, Swierstra, S. D. (ed.), Lecture Notes in Computer Science, vol. 1576. Springer, Amsterdam, pp. 147–161.

Owens, C. (1995) Coding binding and substitution explicitly in Isabelle. In *Proceedings of the First Isabelle Users Workshop*, Cambridge, pp. 36–52. Available at: `http://www.cl.cam.ac.uk/~lp15/papers/Workshop/` Accessed 8 January 2010.

Owens, S. (2008) A sound semantics for OCaml$_{light}$. In *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Proceedings*, Drossopoulou, S. (ed.), Lecture Notes in Computer Science, vol. 4960. Springer, Budapest, pp. 1–15.

Owens, S. & Flatt, M. (2006) From structures and functors to modules and units. In *Proceedings of 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006)*. ACM, Portland, Oregon, pp. 87–98.

Peskine, G., Sarkar, S., Sewell, P. & Zappa Nardelli, F. (2007) Binding and substitution (note). Available at: `http://www.cl.cam.ac.uk/users/pes20/ott/` Accessed 8 January 2010.

Peyton Jones, S. (ed) (2003) *Haskell 98 Language and Libraries. The Revised Report*. Cambridge University Press.

Pierce, B. C. (2002) *Types and Programming Languages*. MIT Press.

Pitts, A. M. & Stark, I. D. B. (1993) Observable properties of higher order functions that dynamically create local names, or: What's new? In *Mathematical Foundations of Computer Science, 18th International Symposium, MFCS'93, Proceedings*, Borzyszkowski, A. M., & Sokolowski, S. (eds), Lecture Notes in Computer Science, vol. 711. Springer-Verlag, Gdansk, Poland, pp. 122–141.

Pollack, R. (2006) Reasoning about languages with binding. Available at: `http://homepages.inf.ed.ac.uk/rpollack/export/bindingChallenge_slides.pdf`. Slides. Accessed 8 January 2010.

Pottier, F. (2006) An overview of Cαml. In *ACM Workshop on ML, ENTCS*, vol. 148, no. 2, pp. 27–52.

Rekers, J. (1992) *Parser Generation for Interactive Environments*. Ph.D. Thesis, University of Amsterdam.

Reps, T. & Teitelbaum, T. (1984) The synthesizer generator. In *Proceedings of the first ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 1)*, Pittsburgh, pp. 42–48. `http://portal.acm.org/citation.cfm?id=390010.808247` Accessed 8 January 2010.

Rossberg, A. (2001) *Defects in the Revised Definition of Standard ML*. Tech. Rep., Saarland University. Updated 2007/01/22.

Sewell, P., Leifer, J. J., Wansbrough, K., Allen-Williams, M., Zappa Nardelli, F., Habouzit, P. & Vafeiadis, V. (2004) *Acute: High-Level Programming Language Design for Distributed Computation. Design Rationale and Language Definition*. Tech. Rep., UCAM-CL-TR-605. University of Cambridge Computer Laboratory.

Sewell, P., Leifer, J. J., Wansbrough, K., Zappa Nardelli, F., Allen-Williams, M., Habouzit, P. & Vafeiadis, V. (2007a) Acute: High-level programming language design for distributed computation, *J. Funct. Program.*, 17 (4–5): 547–612. Invited submission for an ICFP 2005 special issue.

Sewell, P. & Zappa Nardelli, F. (2007) Ott, Freiburg. Available at: `http://www.cl.cam.ac.uk/users/pes20/ott/` Accessed 8 January 2010.

Sewell, P., Zappa Nardelli, F., Owens, S., Peskine, G., Ridge, T., Sarkar, S. & Strniša, R. (2007b) Ott: Effective tool support for the working semanticist. In *Proceedings of the 12th*

ACM SIGPLAN International Conference on Functional Programming (ICFP 2007). ACM, pp. 1–12.

Shinwell, M. R., Pitts, A. M. & Gabbay, M. J. (2003) FreshML: Programming with binders made simple. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP 2003)*. ACM, Uppsala, pp. 263–274.

Sperber, M., Dybvig, R. K., Flatt, M. (eds), Anton Van Straaten, K., Richard, C., William, J. R. (eds), Revised[5] Report on the Algorithmic Language Scheme, Findler, R. B. & Jacob M. (Authors, formal semantics). (2007) Revised[6] report on the algorithmic language Scheme. Available at: `http://www.r6rs.org/` Accessed 8 January 2010.

Strachey, C. (1966) Towards a formal semantics. In *Formal Language Description Languages for Computer Programming*. North Holland, pp. 198–220.

Strniša, R., Sewell, P. & Parkinson, M. (2007) The Java module system: Core design and semantic definition. In *Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2007)*. ACM, Montreal, pp. 499–514.

Syme, D. (1993) Reasoning with the formal definition of standard ML in HOL. In *Higher Order Logic Theorem Proving and its Applications, 6th International Workshop, HUG '93, Proceedings*, Joyce, J. J. & Seger, C.-J. H. (eds), Lecture Notes in Computer Science, vol. 780. Springer-Verlag, Vancouver, pp. 43–59.

Terrasse, D. (1995) Encoding natural semantics in Coq. In *Algebraic Methodology and Software Technology, 4th International Conference, AMAST '95, Proceedings*, Alagar, V. S. & Nivat, M. (eds), Lecture Notes in Computer Science, vol. 936. Springer, Montreal, pp. 230–244.

Tse, S. & Zdancewic, S.. (2008) *Concise Concrete Syntax*. Tech. Rep., MS-CIS-08-11. University of Pennsylvania. Available at: `http://www.cis.upenn.edu/~stevez/papers/TZ08tr.pdf` Accessed 8 January 2010.

Twelf. (2005). Twelf 1.5. Available at: `http://www.cs.cmu.edu/~twelf/` Accessed 8 January 2010.

Urban, C. (2008) Nominal techniques in Isabelle/HOL, *J. Autom. Reasoning*, 40 (4): 327–356.

Vafeiadis, V. & Parkinson, M. (2007) A marriage of rely/guarantee and separation logic. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, Proceedings*, Caires, L. & Vasconcelos, V. T. (eds), Lecture Notes in Computer Science, vol. 4703. Springer, Lisbon, pp. 256–271.

VanInwegen, M. (1996) *The Machine-Assisted Proof of Programming Language Properties*. Ph.D. Thesis, University of Pennsylvania. Computer and Information Science Tech Report MS-CIS-96-31.

Visser, E. (1997) *Syntax Definition for Language Prototyping*. Ph.D. Thesis, University of Amsterdam.

Xiao, Y., Ariola, Z & Mauny, M. (2000) From syntactic theories to interpreters: A specification language and its compilation. In *First International Workshop on Rule-Based Programming (RULE 2000)*, Derschowitz, N. & Kirchner, C. (eds). Available at: `http://arxiv.org/abs/cs.PL/0009030` Accessed 8 January 2010.

Xiao, Y., Sabry, A. & Ariola, Z. M. (2001) From syntactic theories to interpreters: Automating the proof of unique decomposition, *Higher Order Symbol. Comput.*, 14 (4): 387–409.

Zalewski, M. (2008) *A Semantic Definition of Separate Type Checking in C++ with Concepts—Abstract Syntax and Complete Semantic Definition*. Tech. Rep., 2008:12. Department of Computer Science and Engineering, Chalmers University.

Zalewski, M. & Schupp, S. (2009) A semantic definition of Separate type checking in C++ with concepts. *J. Object Technol.* 8 (5): 105–132.