Ott: Effective Tool Support for the Working Semanticist

Peter Sewell* Francesco Zappa Nardelli† Scott Owens* Gilles Peskine*

Thomas Ridge* Susmit Sarkar* Rok Strniša*

*University of Cambridge †INRIA

http://www.cl.cam.ac.uk/users/pes20/ott

Abstract

It is rare to give a semantic definition of a full-scale programming language, 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, Isabelle, and (in progress) Twelf, 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 (around 306 rules), with machine 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.

Categories and Subject Descriptors D.3.1 [**Programming Languages**]: Formal Definitions and Theory

General Terms Languages, Theory, Verification, Standardization

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, 17 years after publication, the shining example of a language that is defined precisely and is at all widely used (Milner et al. 1990). Even languages such as Haskell (Peyton Jones 2003) and OCaml (Leroy et al. 2005), though designed by PL researchers and in large part based on mathematical papers, rely on prose descriptions.

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, or Twelf (Coq; HOL; Isabelle; Twelf).

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), just keeping a large definition internally syntactically consistent during development is hard, and informal proof becomes quite unreliable. 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 (AFP)), but are specific to a single proof assistant. Both styles of definition make it hard to compose semantics in a modular way, from fragments.

The Dream What, then, is the ideal? We would like to have a metalanguage that is designed for the working semanticist, supporting common notations that have been developed over the years. In an email or working note one might write grammars for languages with complex binding structures, for example

```
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, for example as below.

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

These are intuitively clear, concise, and easy to read and edit. Sadly, they lack both the precision of proof assistant definitions and the production-quality typesetting of LaTeX — but one might hope that only a modicum of information need be added to make them precise, and to automatically *compile* them to both targets.

Contribution We realize this dream: we describe a metalanguage specifically designed for writing semantic definitions, and a tool, `ott`, that sanity-checks such definitions and compiles them into proof assistant code, LaTeX code for production-quality typesetting, and OCaml boilerplate for implementations. The main innovations are:

• ***Metalanguage design to make definitions concise and easy to read and edit (§2).*** The 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. 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 (§3).*** Nontrivial 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 this two 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 what the tool currently implements. Secondly, we define alpha equivalence and capture-avoiding substitution for arbitrary binding specifications, clarifying several issues. Implementing this is future work, but we prove (on paper) that under usable conditions the two notions of substitution coincide.

• ***Compilation to proof assistant code (§4).*** From a single definition in the metalanguage, the `ott` tool can generate proof assistant definitions in Coq, HOL, Isabelle, and (in progress) Twelf. These can then be used as a basis for formal proof and (where the proof assistant permits) code extraction and animation.

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 subgrammar predicates; generation of substitution and free variable functions; (for Isabelle) a tuple encoding for mutually primitive recursive functions, with auxiliary function definitions for nested pattern matching and for nested list types; (for Coq and Twelf) generation of auxiliary list types for the syntax and semantics; (for Coq) generation of useful induction principles when using native lists; (for HOL) a stronger definition library, and (for Twelf) translation of functions into relations.

We aim to generate well-formed and idiomatic definitions, without dangling proof obligations, and in good taste as a basis for user proof development.

• ***Substantial case studies (§5).*** The usefulness of the `ott` metalanguage and tool has been tested in several case studies. We have defined type systems and operational semantics for:

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

grammar
t :: 't_' ::=                                      {{ com term    }}
  | x            ::  :: Var                         {{ com variable}}
  | \ x . t      ::  :: Lam (+ bind x in t +) {{ com lambda  }}
  | t t'         ::  :: App                         {{ com app     }}
  | ( t )        :: M:: Paren                       {{ icho [[t]]  }}
  | { t / x } t' :: M:: Tsub
                        {{ icho (tsubst_t [[t]] [[x]] [[t']])}}

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

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

subrules
  v <:: t

substitutions
  single t x :: tsubst

defns
Jop :: '' ::=

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


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

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

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

Figure 1. A small `ott` source file, for an untyped CBV lambda calculus, with data for Coq, HOL, Isabelle, LaTeX, and OCaml.

(1) small lambda calculi: simply typed (*) and with ML polymorphism, both 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 and Flatt (2006);
(4) a language for rely-guarantee and separation logic Vafeiadis and Parkinson (2007); (*)
(5) formalisation of the core `ott` binding specifications;
(6) Lightweight Java (LJ), a small imperative fragment of Java; (*)
(7) formalisations of Java module system proposals, based on JSR 277/294 (including LJ, around 140 semantic rules); and
(8) a large core of OCaml, including record and datatype definitions (around 306 semantic rules). (*)

For the starred systems soundness results have been proved or are well in progress, in one or more proof assistants. The TAPL and LJ examples show that a very simple form of *modular semantics* provided by `ott` can be effective — those TAPL features are defined in separate files, roughly following the structure of the TinkerType repository used to build the original text (Levin and Pierce 2003). Most of the examples were not done by the tool developers, and (4–8) were primarily driven by other research goals, so we can make a reasonable preliminary assessment of the user experience. So far it is positive. For small calculi it is easy to get started with the tool, and even for large definitions such as (7) and (8) 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. Whilst we provide a common language for semantic definitions, we do not aim to do the same for proofs.

This paper describes the key ideas underlying the `ott` metalanguage; it is not a user guide — but that, along with the tool itself and a number of examples, is available on the web (Sewell and Zappa Nardelli 2007). User feedback is very welcome. There is a long history of related work in this area, discussed in §6, and we conclude in §7.

2. Overview and Metalanguage Design

In this section we give an overview of the metalanguage and tool, including its syntax and type structure.

2.1 A small example

We begin with the example in Fig. 1, which is a complete `ott` source file for an untyped CBV lambda calculus, including the information required to generate proof assistant definitions in Coq, HOL and Isabelle, OCaml boilerplate, and LaTeX. The typeset LaTeX is shown in Fig. 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 nontrivial calculus) which are our main motivation. We comment on those as we go, and invite the reader to imagine the development for their favorite programming language or calculus in parallel.

Core First consider Fig. 1 but ignore the data within `{{ }}` and `(+ +)`, and the `terminals` block. At the top of the figure, the **`metavar`** declaration introduces *metavariables* `termvar` (with synonym `x`), for term variables. The following **`grammar`** introduces grammars for terms, with *nonterminal root* `t`, and for values `v`:

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

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 *terminals* of the grammar (`\ . ( ) { } / -->`) are inferred, as those tokens that cannot be lexed as metavariables or nonterminals, 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 just a single judgement `t1 --> t2` for the reduction relation, defined by three rules. Consider the innocent-looking CBV beta rule:

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

Here the conclusion is a term of the syntactic form of the judgement being defined, `t1 --> t2`. Its two subterms `(\x.t12) v2` and `{v2/x}t12` 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 `t` grammar), just as concrete terms would, but also:

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

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

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

$$\frac{t_1 \ \longrightarrow\ t_1'}{t_1\ t \ \longrightarrow\ t_1'\ t}\ \text{CTX\_APP\_FUN}$$

$$\frac{t_1 \ \longrightarrow\ t_1'}{v\ t_1 \ \longrightarrow\ v\ t_1'}\ \text{CTX\_APP\_ARG}$$

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

- mention symbolic metavariables (`x`) and nonterminals (`t12` and `v2`), built from the metavariable and nonterminal 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`** `v <:: t`, and checked by the tool) to allow `v2` to appear in a position where a term of type `t` is expected; and
- involve syntax for parentheses and substitution. The concrete syntax for these is given by the `Paren` and `Tsub` productions of the `t` grammar, but these are *metaproductions* (flagged **`M`**), for which we do not want abstract syntax constructors.

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, 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. For a very simple example, the Coq compilation for `t` generates a free type with three constructors:

```
Inductive
t : Set :=
   t_Var : termvar -> t
 | t_Lam : termvar -> t -> t
 | t_App : t -> t -> t .
```

The general case, rather more complex than this, is discussed in §4, but for now note that 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

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

```
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
------------------------------------------------------------------------------ :: record_constr
E |- {field_name1=e1; ...; field_namen=en} : t
```

Figure 3. A sample OCaml semantic rule, in LaTeX and `ott` source forms

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

The `tsubst_t` mentioned in the hom for `Tsub` above is a proof assistant identifier for a function that calculates substitution over terms, automatically generated by the **`substitutions`** declaration. We return in §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 nonterminals, 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 Fig. 2, the user can add various data: (1) the `{{ tex \mathit{[[termvar]]} }}` in the **`metavar`** declaration, specifying that `termvars` be typeset in math italic; (2) the `terminals` grammar, overriding the default typesetting for terminals `\` and `-->` by λ and $\longrightarrow$; and (3) `{{ com ...}}` comments, annotating productions and judgements.

One can also write **`tex`** annotations to override the default typesetting at the level of productions, not just tokens. For example, in $\mathrm{F}_{<:}$ one might wish to typeset term abstractions with λ and type abstractions 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]] }}
```

to typeset 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 nonterminal roots, e.g. to typeset a nonterminal 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, 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, and functions for substitution etc. (but not the judgments). 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.

2.2 List forms

For an example that is rather more typical of a large-scale semantics, consider the record typing rule shown in the top half of Fig. 3, taken from our OCaml fragment definition. The first, second, and fourth premises 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.

The bottom half of Fig. 3 shows the source text for that rule — note the close correspondance to the typeset version, making it easy to read and edit. Looking at it more closely, we see *index variables* n, m, and l occuring in suffixes. There are symbolic nonterminals and metavariables indexed in three different ranges: $e_\square$, $t_\square$, and $\mathit{field\_name}_\square$ are indexed from 1 to n, $\mathit{field\_name}'_\square$ is indexed from 1 to m, and $t'_\square$ is indexed 1 to l. To parse list forms involving dots, the tool finds subterms which can be antiunified by abstracting out components of suffixes.

With direct support for lists, we need also direct support for symbolic terms involving list projection and concatenation, e.g. in the rules below (taken from a different case study).

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

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

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.

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

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 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 reasonable 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 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 scannerless memoized CPS'd parser combinators, taking ideas from Johnson (1995), which is simple and sufficiently efficient.

Naming conventions for symbolic nonterminals 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.

2.4 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:

- Both LaTeX and proof assistant files can be *filtered*, replacing delimited `ott`-syntax symbolic terms (or concrete term examples) in documents, e.g. `[[(\x.x x) x' --> t]]`, by their LaTeX or proof assistant rendering. Additionally, LaTeX and proof assistant code can be *embedded* within an `ott` source file (and similarly filtered). Typesetting style is indirected, so that it can be controlled by redefining LaTeX commands.
- The generated LaTeX is factored into LaTeX commands for individual rules, the rules of individual **`defn`**s, etc., up to the complete definition, so that parts or all of the definition can be quoted in other documents.
- The proof assistants each have their own support, more-or-less elaborate, for fancy syntax. For Isabelle the tool can generate these declarations from an `ott` source grammar, so that they can be used in proof scripts and in the displayed goals during interactive proof.
- We support common prefixes for rule names and production names (e.g. the `t_` in Fig. 1), and allow synonyms for nonterminal 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 §2 example:

$$\begin{array}{lllll} t & ::= & & & \\ & | & \lambda\, x\, .\, t & \mathsf{bind}\ x\ \mathsf{in}\ t & \text{lambda} \end{array}$$

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.

3.1 The `ott` binding metalanguage: syntax

We introduce a novel metalanguage for specifying binding structures, expressive enough to cover all the above but remaining simple and intuitive. It comprises two forms of annotation on productions. The first, bind *mse* in *nonterm*, lets one specify that variables bind in nonterminals of the production, as in the lambda production above. Here *mse* is a *metavariable set expression*, e.g. in that lambda production just the singleton metavariable x of the production. A variable can bind in multiple nonterminals, as in the example of a simple recursive `let` below.

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

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'.

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

This is expressed with the second form of annotation: user-defined *auxiliary functions* such as the binders above. This is an auxiliary function defined over the p grammar that identifies a set of variables to be used in the bind annotation on the **let** production.

The syntax of a precise fragment of the binding metalanguage is given in Fig. 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 nonterminals that it is defined over, and that metavariable set expressions are well-sorted.

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

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

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. Fig. 1 contained the block:

```
substitutions
  single t x :: tsubst
```

which causes `ott` to generate proof-assistant functions for single substitution of term variables `x` by terms `t` over all (non-subtype) types of the grammar — here just `t`, so a function named `tsubst_t` is generated. Multiple substitutions can also be generated, and there is similar machinery for free variable functions.

3.2 The `ott` binding metalanguage: semantics

We give meaning to these binding specifications in two ways.

The fully concrete representation The first semantics (and the only one that is currently supported by the tool) 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. In typical call-by-value 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 with concrete variable names, as in the example Coq type `t` of §2.1 (Fig. 5 gives a general grammar of such concrete abstract syntax terms, *casts*). Substitution is defined so as to not substitute for bound variables within their scopes, but without using any renaming. Continuing the §2.1 example, `ott` generates Coq code essentially as below.

```
Fixpoint tsubst_t (t2:t)(x1:termvar)(t2:t){struct t2}:t :=
  match t2 with
  | (t_Var x) =>
      (if eq_termvar x x1 then t2 else (t_Var x))
  | (t_Lam x t1) =>
      t_Lam x (if eq_termvar x1 x) then t1
               else (tsubst_t t2 x1 t1))
  | (t_App t1 t') =>
      t_App (tsubst_t t2 x1 t1) (tsubst_t t2 x1 t')
end.
```

Doing this in the general case highlights a subtlety: when substituting (e.g.) `ts` for `xs`, the only occurrences of `x` that are substitutable are those in instances of productions of the `t` grammar that comprise just a *singleton* `x` (e.g. here the `Var` production), as only there will the result be obviously type correct. Other occurrences, e.g. 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

$$
\begin{array}{lll}
\textbf{metavars} & metavarroot,\ mvr & nontermroot,\ ntr \\
 & terminal,\ t & auxfn,\ f \\
 & prodname,\ pn & variable,\ var \\
\textbf{grammar} & & \\
\quad metavar,\ mv ::= & & \\
\quad\quad |\ metavarroot\ suffix & & \\
\quad nonterm,\ nt ::= & & \\
\quad\quad |\ nontermroot\ suffix & & \\
\quad element,\ e ::= & & \\
\quad\quad |\ terminal & & \\
\quad\quad |\ metavar & & \\
\quad\quad |\ nonterm & & \\
\quad metavar\_set\_expression,\ mse ::= & & \\
\quad\quad |\ metavar & & \\
\quad\quad |\ auxfn\,\texttt{(}nonterm\texttt{)} & & \\
\quad\quad |\ mse\ \texttt{union}\ mse' & & \\
\quad\quad |\ \{\} & & \\
\quad bindspec,\ bs ::= & & \\
\quad\quad |\ \texttt{bind}\ mse\ \texttt{in}\ nonterm & & \\
\quad\quad |\ auxfn\ =\ mse & & \\
\quad prod,\ p ::= & & \\
\quad\quad |\ \texttt{|}\ element_1\ ..\ element_m\ \texttt{::}\ \texttt{::}\ prodname\ \texttt{(+}\ bs_1\ ..\ bs_n\ \texttt{+)} & & \\
\quad rule,\ r ::= & & \\
\quad\quad |\ nontermroot\ \texttt{::}\ \texttt{''}\ \texttt{::=}\ prod_1\ ..\ prod_m & & \\
\quad grammar\_rules,\ g ::= & & \\
\quad\quad |\ \texttt{grammar}\ rule_1\ ..\ rule_m & &
\end{array}
$$

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

$$
\begin{array}{l}
concrete\_abstract\_syntax\_term,\ cast ::= \\
\quad |\ var\ :\ mvr \\
\quad |\ prodname\ (\ cast_1\ ,\ ..,\ cast_m\ )
\end{array}
$$

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

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 Fig. 4, including definitions of substitution and free variables over the general concrete abstract syntax terms of Fig. 5 (Sewell and Zappa Nardelli 2007), but for lack of space we do not include it here.

Alpha equivalence 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 specification. These definitions are again precise, in Ott-Isabelle/HOL, and are available on the web (Sewell and Zappa Nardelli 2007). Here we explain just the key points by two examples.

[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.

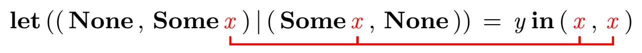
First, consider the OCaml *or-patterns*[2] $p_1 \mid p_2$, e.g. with a pattern grammar

$p ::=$	
$\mid x$	$\mathsf{b} = x$
$\mid (\, p_1 \,,\, p_2 \,)$	$\mathsf{b} = \mathsf{b}(p_1) \cup \mathsf{b}(p_2)$
$\mid p_1 \mid p_2$	$\mathsf{b} = \mathsf{b}(p_1) \cup \mathsf{b}(p_2)$
$\mid$ **None**	$\mathsf{b} = \{\}$
$\mid$ **Some** p	$\mathsf{b} = \mathsf{b}(p)$

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.

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

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 concrete abstract syntax terms *cast*, partial equivalence relations e_{cast} over the *occurrences* of variables within them. In the example it would relate all four occurrences of x to each other, as below, but leave y unrelated.



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

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.

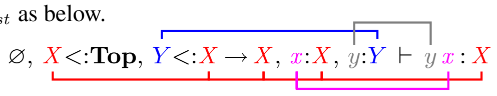
For the second example, consider a system such as $\mathrm{F}_{<:}$ with type environments Γ as below.

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

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

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

it is then easy to see what the binding structure is, and we can depict the e_{cast} as below.

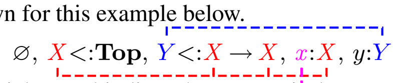


$$\varnothing,\, X{<:}\textbf{Top},\, Y{<:}X \to X,\, x{:}X,\, y{:}Y \;\vdash\; y\, x : X$$

For that type environment in isolation, however, i.e.

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

while in some sense the $X <: \textbf{Top}$ binds in the tail, it must not be alpha-varied — that only becomes possible when it is put in the complete context of a judgement. Our definitions capture this phenomenon by defining for each term *cast* not just an e_{cast} relation for 'closed' binding but also a similar o_{cast} partial equivalence relation for 'open' binding, relating occurrences which potentially may alpha-vary together if this term is placed in a larger, binding, context. The o_{cast} is not directly involved in the definition of alpha equivalence, but is (compositionally) used to calculate the e_{cast}. It is shown for this example below.



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

Nontrivial 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. `t` for `x`), are distinct from the domains and results of auxiliary functions (e.g. `binders`, collecting, from patterns `p`, variables `x`). 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. At present both of these are hand proofs, though above mechanized definitions — we aim to mechanize them in due course.

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 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 $\mathsf{bind}\,mse$ point in the grammar. A 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.

4. Compilation to Proof Assistant Code

Our compilation generates proof-assistant definitions: of types; of functions, for subgrammar predicates, for the binding auxiliaries of §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 mechanized proof. All this is for Coq, HOL, and Isabelle/HOL, and work on compilation to Twelf is in progress.

4.1 Types

Each metavariable declaration gives rise simply to a proof assistant type abbreviation, for example `types termvar = "string"` in the Isabelle generated from Fig. 1. For each nonterminal root of the user's grammar, if (a) it is a maximal element of the subrule order, and (b) no type hom has been specified, then we generate a free type with a constructor for each non-meta production of the grammar (as in the simple §2.1 example of `t`). Nonterminal roots with type homs give rise to type abbreviations. Nonterminal roots that are not maximal, e.g. the `v` of Fig. 1, are represented using the type generated for the (unique) maximal element above them. For these we also generate and use subgrammar predicates that carve out the relevant part of that type, as discussed below.

In general there may be a complex pattern of mutual recursion among these types. Coq, HOL and Isabelle 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.

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

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
t :: E_ ::=
  | { l1 = t1 ,  .. , ln = tn }      :: :: record
```

In HOL and Isabelle we represent these simply with contructors whose argument types involve proof-assistant native list types, e.g.

```
val _ = Hol_datatype ‘
t = E_record of (label#t) list  ‘;
```

For Coq, however, we provide two alternatives: one can either use native lists or lists can be translated away, depending on taste. In the former case we generate an appropriate induction principle using nested fixpoints, as the default principle produced by Coq is too weak to be useful. In the latter case we synthesise an additional type for each type of lists-of-tuples that arises in the grammar. In the example, we need a type of lists of pairs of a `label` and a `t`:

```
Inductive
list_label_t : Set :=
   Nil_list_label_t : list_label_t
 | Cons_list_label_t : label -> t -> list_label_t
     -> list_label_t

with t : Set :=
   E_record : list_label_t -> t .
```

These are included in the topological sort, and utility functions, e.g. to make and unmake lists, are synthesised. A similar translation will 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 we produce `primrecs`. In general we have to deal both with the type dependency (the topologically sorted mutually recursive types described above) and with function dependency — for subgrammar predicates and binding auxiliaries we may have multiple mutually recursive functions over the same type.

Subgrammar Predicates We generate subgrammar predicates to carve out the subsets of each free proof assistant type (from the maximal elements of the subrule order) that represent the non-free rules of the grammar. 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. Note that these can include rules that are maximal elements of the subrule order, e.g. if an expression grammar included a production involving packaged values. The subgrammar 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.

Binding Auxiliaries These functions calculate the intuitive fully concrete interpretations of auxiliary functions defined in bindspecs, as in §3.2, giving proof assistant sets or lists, of metavariables or nonterminals, 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. For each production, for each occurrence of a nonterminal 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 nonterminal 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 nonterminals with type homs, as they may be arbitrarily complex, 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 *nonterminals*, 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.

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, we chose the `primrec` package, to avoid any danger of leaving dangling proof obligations, and because our functions are all intuitively primitive recursive. Unfortunately, in the released (Isabelle2005) version, `primrec` 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, it does not support pattern matching involving nested constructors. We therefore generate auxiliary functions for productions with embedded list types. Isabelle tuples are treated as iterated pairs, so we do the same for productions with tuples of size 3 or more. Isabelle also requires a function definition for each recursive type. In the case where there are multiple uses of the same type (e.g. several uses of `t 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, and it has been suggested that future versions of Isabelle will have a more robust definition package for general functions, which should subsume some of the above.

4.3 Relations

The semantic relations are defined with the proof-assistant inductive relations packages, `Inductive`, `Hol_reln`, and `inductive`, respectively. Each `defns` block gives rise to a potentially mutu-

$$\begin{array}{l}
\mathit{symterm},\ \mathit{st} ::= \\
\quad |\ \mathit{stnb} \\
\quad |\ \mathit{nonterm} \\
\\
\mathit{symterm\_node\_body},\ \mathit{stnb} ::= \\
\quad |\ \mathit{prodname}\,(\,\mathit{ste}_1\,,\,..\,,\,\mathit{ste}_m\,) \\
\\
\mathit{symterm\_element},\ \mathit{ste} ::= \\
\quad |\ \mathit{st} \\
\quad |\ \mathit{metavar} \\
\quad |\ \mathit{var}\ :\ \mathit{mvr}
\end{array}$$

Figure 6. Mini-Ott in Ott: symbolic terms

ally recursive definition of each `defn` inside it (it seems clearer not to do a topological sort here). Definition rules are expressed internally with symbolic terms. We give a simplified grammar thereof in Fig. 6, omitting the symbolic terms for list forms. A symbolic term st for a nonterminal root is either an explicit nonterminal 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.

Further, for each nonterminal of a non-free grammar rule, e.g. a usage of `v'` where `v<::t`, an additional premise invoking the generated subrule predicate for the non-free rule is added, e.g. `is_v 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 nonterminals 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_Typ_list : (label#t#Typ) list`.

```
(* Ty_Rcd *)  !(l_t_Typ_list:(label#t#Typ) list) (G:G) .
(EVERY (\b.b)
  (MAP (\(l_,t_,Typ_). (Ty G t_ Typ_)) l_t_Typ_list))
 ==>
(Ty
  G
  (E_record (MAP (\(l_,t_,Typ_). (l_,t_)) l_t_Typ_list))
  (T_Rec    (MAP (\(l_,t_,Typ_). (l_,Typ_)) l_t_Typ_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 §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#t) list)
       (l_v_list:(label#t) list) (l:label) (t:t) (t':t) .
((EVERY (\(l_,v_). is_v_of_t 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 j'th element:

```
(* Ty_Proj *)  !(l_Typ_list:(label#Typ) list)
       (j:index) (G:G) (t:t) .
((( Ty G t (T_Rec (l_Typ_list)) )))
 ==>
(( Ty
     G
     (t_Proj t  ((\ (l_,Typ_) . l_) (EL j l_Typ_list)))
     ((\ (l_,Typ_) . Typ_) (EL  j l_Typ_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.

As outlined here, the analysis and code generation performed by `ott` is reasonably complex (the tool is around 17 000 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.

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 below.

System	rules	LaTeX	Coq		HOL		Isabelle	
			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	√					√	√[2]
LJ: Lightweight Java	34	√					√	(3)
LJAM: Java Module System	140	√					√	
OCaml fragment	306	√	√		√	√[1]	√	

[1] see below. [2] hand proofs. [3] in progress.

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. For most we use the tool to generate definitions in one or more of Coq, HOL, and Isabelle, indicated by the ticks in the 'defn' columns below, together with the typeset LaTeX. We have tested whether these definitions form a good basis for mechanized proof by machine-checked proofs of metatheoretic results (generally type preservation and progress), indicated by ticks in the 'mt'

```
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
```

Figure 7. An `ott` source file for the `let` fragment of TAPL

columns below. 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 and 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 a very simple form of *modularity* provided by `ott`, allowing clauses of grammars and semantic relations to be split between files. The original TAPL languages were produced using TinkerType (Levin and 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 Fig. 7, to which we add: `bool`, `bool_typing`, `nat`, `nat_typing`, `arrow_typing`, `basety`, `unit`, `seq`, `ascribe`, `product`, `sum`, `fix`, `tuple`, and `variant`, togther with infrastructure `common`, `common_index`, `common_labels`, and `common_typing`.

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

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

RG-Sep language This is a concurrent while language used for ongoing 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 proof that for closed substitutions the two coincide. To date only a hand proof has been completed; we plan to mechanize it in due course.

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 fragment This 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; generative definitions of record and variant types; and generative 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.

Experience Our experience with these examples has been very positive. The tool does make it easy to work with these definitions, allowing one to focus on the content rather than the proof assistant or LaTeX markup. We have not had to hand-edit the Ott output.

For our most substantial example, the OCaml fragment, we have proved type preservation and progress for the expression language, all machine-checked in HOL. The need for alpha-equivalence-aware reasoning arises only for type variables and type schemes. We use 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, we deal directly with the index shifting functions in the Ott source. This proof effort has taken only around 3 man-months, and the preceeding 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. 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.

and the problem of dealing precisely with this notation, with the need for machine support in doing so, has spawned an extensive literature, of which we touch only on the most related points.

The proof assistants that we build on, Coq, HOL, Isabelle, and Twelf, are perhaps the most directly related work (Coq; HOL; Isabelle; Twelf). 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), 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 and 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 workflows 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.

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 (Findler and Matthews 2007), but is by design restricted to animation of reduction semantics. The Ruler system (Dijkstra and Swierstra 2006) provides a language for expressing type rules, generating LaTeX and implementations but not proof assistant definitions, used for a Haskell-like language.

Turning to direct support for binding, Twelf is suited to HOAS representations. FreshML (Shinwell et al. 2003), Alpha Prolog (Cheney and Urban 2004) and MLSOS (Lakin and Pitts 2007) both use nominal logic- and functional programming approaches, the latter two with a view to prototyping of semantics. 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. This seems to us less intuitive than the `ott` binding specifications. 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, including those of Aydemir et al. (2007) and Berghofer and Urban (2006). Owens (1995) discusses pattern binding using locally nameless representations in Isabelle.

The work on concise concrete syntax by Tse and Zdancewic (2006) 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 (yet to appear) 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 modules), so the proof effort is hard to compare, but their semantic rules are further removed from source-language programs.

7. 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 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. First, while the fully concrete representation of binding is surprisingly widely applicable, it is far from expressing all one would like to do. We plan to explore proof assistant representations for arbitrary binding specifications, as outlined in §3.2. Another 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, but the targets are based on different logics, so it is far from trivial.

The Twelf code generation remains to be completed, and a number of other features would be useful: support for function definitions (not just inductive relations); support for contexts, with automatically generated context application and composition functions; support for generating multiple overlapping languages from a single source (e.g. sugared and non-sugared); and generation of production parsers.

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 defintions 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 forward to further experience and user feedback from the tool.

Acknowledgements 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, 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 and EP/C510712, and a Royal Society University Research Fellowship (Sewell).

References

AFP. The archive of formal proofs. `http://afp.sf.net`.

B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory, 2007. `http://www.chargueraud.org/arthur/research/2007/binders/`.

B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark Challenge. In *Proc. TPHOLs, LNCS 3603*, 2005.

S. Berghofer and C. Urban. A head-to-head comparison of de Bruijn indices and names. In *Proc. Int. Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, pages 46–59, 2006.

S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *Int. Symp. on Formal Methods, LNCS 2085*, 2006.

P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Proc. SDE 3*, pages 14–24, 1988.

R. J. Boulton. A tool to support formal reasoning about computer languages. In *Proc. TACAS, LNCS 1217*, pages 81–95, 1997.

A. Charguéraud. Annotated bibliography for formalization of lambda-calculus and type theory. `http://fling-l.seas.upenn.edu/~plclub/cgi-bin/poplmark/index.php?title=Annotated_Bibliography`, July 2006.

J. Cheney and C. Urban. Alpha-Prolog: A logic programming language with names, binding and alpha-equivalence. In *Proc. ICLP, LNCS 3132*, pages 269–283, 2004.

Coq. The Coq proof assistant, v.8.0. `http://coq.inria.fr/`.

A. Dijkstra and S. D. Swierstra. Ruler: Programming type rules. In *Proc. Functional and Logic Programming, LNCS 3945*, 2006.

R. B. Findler and J. Matthews. Revised$^{5.92}$ report on the algorithmic language Scheme, Chapter 10, Formal Semantics, Jan. 2007.

C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proc. CONCUR '96, LNCS 1119*, 1996.

E. Gunter and S. Maharaj. Studying the ML module system in HOL. *The Computer Journal: Special Issue on Theorem Proving in Higher Order Logics*, 1995.

HOL. The HOL 4 system, Kananaskis-3 release. `http://hol.sourceforge.net/`.

Isabelle. Isabelle 2005. `http://isabelle.in.tum.de/`.

M. Johnson. Memoization in top-down parsing. *Comput. Linguist.*, 21(3): 405–417, 1995.

S. Kahrs. Mistakes and ambiguities in the definition of Standard ML. Technical Report ECS-LFCS-93-257, University of Edinburgh, 1993.

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

P. Klint. A meta-environment for generating programming environments. *ACM Trans. on Soft. Eng. and Methodology*, 2(2):176–201, April 1993.

M. R. Lakin and A. M. Pitts. A metalanguage for structural operational semantics. In *Symposium on Trends in Functional Programming*, 2007.

D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of Standard ML. In *Proc. POPL*, January 2007.

P. Lee, F. Pfenning, G. Rollins, and W. Scherlis. The Ergo Support System: An integrated set of tools for prototyping integrated environments. In *Proc. SDE 3*, 1988.

X. Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

X. Leroy et al. The Objective Caml system release 3.09 documentation and user's manual, Oct. 2005.

M. Y. Levin and B. C. Pierce. Tinkertype: A language for playing with formal systems. *Journal of Functional Programming*, 13(2), Mar. 2003.

J. Matthews, R. B. Findler, M. Flatt, and M. Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *Proc. RTA*, 2004.

R. Milner. Implementation and applications of Scott's logic for computable functions. In *Proc. ACM conference on Proving assertions about programs*, pages 1–6, 1972.

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

P. D. Mosses. Pragmatics of Modular SOS. In *Proc. AMAST, LNCS 2442*, pages 21–40, 2002.

M. Norrish. Deterministic expressions in C. In *Proc. 8th ESOP (ETAPS), LNCS 1576*, pages 147–161, 1999.

C. Owens. Coding binding and substitution explicitly in Isabelle. In *Proceedings of the First Isabelle Users Workshop*, pages 36–52, 1995.

S. Owens and M. Flatt. From structures and functors to modules and units. In *Proc. ICFP*, 2006.

S. Peyton Jones, editor. *Haskell 98 Language and Libraries. The Revised Report*. CUP, 2003.

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

F. Pottier. An overview of Cαml. In *ACM Workshop on ML, ENTCS 148(2)*, pages 27–52, Mar. 2006.

T. Reps and T. Teitelbaum. The synthesizer generator. In *Proc. SDE 1*, pages 42–48, 1984.

A. Rossberg. Defects in the revised definition of Standard ML. Technical report, Saarland University, 2001. Updated 2007/01/22.

P. Sewell and F. Zappa Nardelli. Ott, 2007. `http://www.cl.cam.ac.uk/users/pes20/ott/`.

P. Sewell, J. J. Leifer, K. Wansbrough, M. Allen-Williams, F. Zappa Nardelli, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. design rationale and language definition. Technical Report UCAM-CL-TR-605, University of Cambridge Computer Laboratory, Oct. 2004. See also the ICFP'05 paper.

M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Proc. ICFP*, 2003.

C. Strachey. Towards a formal semantics. In *Formal Language Description Languages for Computer Programming*. North Holland, 1966.

R. Strniša, P. Sewell, and M. Parkinson. The Java Module System: core design and semantic definition. In *Proc. OOPSLA*, 2007. To appear.

D. Syme. Reasoning with the formal definition of Standard ML in HOL. In *TPHOLs, LNCS 780*, pages 43–59, 1993.

D. Terrasse. Encoding Natural Semantics in Coq. In *Proc. AMAST, LNCS 936*, pages 230–244, 1995.

S. Tse and S. Zdancewic. Concise concrete syntax, 2006. Submitted. `http://www.cis.upenn.edu/~stse/javac`.

Twelf. Twelf 1.5. `http://www.cs.cmu.edu/~twelf/`.

V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *Proc. CONCUR*, 2007.

M. VanInwegen. *The Machine-Assisted Proof of Programming Language Properties*. PhD thesis, Univ. of Pennsylvania, 1996. Computer and Information Science Tech Report MS-CIS-96-31.