Production Language Specification

Requirements for Multiple Usages

Peter Sewell

University of Cambridge

With thanks especially to Francesco Zappa Nardelli, Scott Owens, Susmit Sarkar, Stephanie Weirich, Dominic Mulligan, Thomas Tuerk, Kathryn Gray, Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alasdair Reid, Neel Krishnaswami,. . .

RPLS 2025: Real-World Programming Language Specification, Seoul/Remote, 2025-06

Multiple usages? Language specifications should support:

Multiple usages? Language specifications should support:

1. reference documentation in prose

Multiple usages? Language specifications should support:

1. reference documentation in prose and in accessible maths

Multiple usages? Language specifications should support:

1. reference documentation in prose and in accessible maths
2. sanity checking of definition (metalanguage sort-checking and type-checking, ...)

Multiple usages? Language specifications should support:

1. reference documentation in prose and in accessible maths
2. sanity checking of definition (metalanguage sort-checking and type-checking, ...)
3. test oracle, for testing specification/implementation consistent – opsem and alg. type inf
    3.1 decide whether an observed implementation outcome is allowed (maybe with extra instrumentation)

Multiple usages? Language specifications should support:

1. reference documentation in prose and in accessible maths
2. sanity checking of definition (metalanguage sort-checking and type-checking, ...)
3. test oracle, for testing specification/implementation consistent – opsem and alg. type inf
   3.1 decide whether an observed implementation outcome is allowed (maybe with extra instrumentation)
   3.2 exhaustively compute all the allowed outcomes – and web-interface exploration tooling

Multiple usages? Language specifications should support:

1. reference documentation in prose and in accessible maths
2. sanity checking of definition (metalanguage sort-checking and type-checking, ...)
3. test oracle, for testing specification/implementation consistent – opsem and alg. type inf
   - 3.1 decide whether an observed implementation outcome is allowed (maybe with extra instrumentation)
   - 3.2 exhaustively compute all the allowed outcomes – and web-interface exploration tooling
4. prototype implementation – compute one execution, fastish

Multiple usages? Language specifications should support:

1. reference documentation in prose and in accessible maths
2. sanity checking of definition (metalanguage sort-checking and type-checking, ...)
3. test oracle, for testing specification/implementation consistent – opsem and alg. type inf
   3.1 decide whether an observed implementation outcome is allowed (maybe with extra instrumentation)
   3.2 exhaustively compute all the allowed outcomes – and web-interface exploration tooling
4. prototype implementation – compute one execution, fastish
5. specification coverage measurement

Multiple usages? Language specifications should support:

1. reference documentation in prose and in accessible maths
2. sanity checking of definition (metalanguage sort-checking and type-checking, ...)
3. test oracle, for testing specification/implementation consistent – opsem and alg. type inf
   - 3.1 decide whether an observed implementation outcome is allowed (maybe with extra instrumentation)
   - 3.2 exhaustively compute all the allowed outcomes – and web-interface exploration tooling
4. prototype implementation – compute one execution, fastish
5. specification coverage measurement
6. test generation based on the specification

Multiple usages? Language specifications should support:

1. reference documentation in prose and in accessible maths
2. sanity checking of definition (metalanguage sort-checking and type-checking, ...)
3. test oracle, for testing specification/implementation consistent – opsem and alg. type inf
   3.1 decide whether an observed implementation outcome is allowed (maybe with extra instrumentation)
   3.2 exhaustively compute all the allowed outcomes – and web-interface exploration tooling
4. prototype implementation – compute one execution, fastish
5. specification coverage measurement
6. test generation based on the specification
7. generate boilerplate for production implementation (AST type etc)

Multiple usages? Language specifications should support:

1. reference documentation in prose and in accessible maths
2. sanity checking of definition (metalanguage sort-checking and type-checking, ...)
3. test oracle, for testing specification/implementation consistent – opsem and alg. type inf
    3.1 decide whether an observed implementation outcome is allowed (maybe with extra instrumentation)
    3.2 exhaustively compute all the allowed outcomes – and web-interface exploration tooling
4. prototype implementation – compute one execution, fastish
5. specification coverage measurement
6. test generation based on the specification
7. generate boilerplate for production implementation (AST type etc)
8. parse and render small examples (partly symbolic)

Multiple usages? Language specifications should support:

1. reference documentation in prose and in accessible maths
2. sanity checking of definition (metalanguage sort-checking and type-checking, ...)
3. test oracle, for testing specification/implementation consistent – opsem and alg. type inf
   - 3.1 decide whether an observed implementation outcome is allowed (maybe with extra instrumentation)
   - 3.2 exhaustively compute all the allowed outcomes – and web-interface exploration tooling
4. prototype implementation – compute one execution, fastish
5. specification coverage measurement
6. test generation based on the specification
7. generate boilerplate for production implementation (AST type etc)
8. parse and render small examples (partly symbolic)
9. generate parser for prototype or production implementation

Multiple usages? Language specifications should support:

1. reference documentation in prose and in accessible maths
2. sanity checking of definition (metalanguage sort-checking and type-checking, ...)
3. test oracle, for testing specification/implementation consistent – opsem and alg. type inf
   3.1 decide whether an observed implementation outcome is allowed (maybe with extra instrumentation)
   3.2 exhaustively compute all the allowed outcomes – and web-interface exploration tooling
4. prototype implementation – compute one execution, fastish
5. specification coverage measurement
6. test generation based on the specification
7. generate boilerplate for production implementation (AST type etc)
8. parse and render small examples (partly symbolic)
9. generate parser for prototype or production implementation
10. generate pretty printer – and code formatter

Multiple usages? Language specifications should support:

1. reference documentation in prose and in accessible maths
2. sanity checking of definition (metalanguage sort-checking and type-checking, ...)
3. test oracle, for testing specification/implementation consistent – opsem and alg. type inf
    - 3.1 decide whether an observed implementation outcome is allowed (maybe with extra instrumentation)
    - 3.2 exhaustively compute all the allowed outcomes – and web-interface exploration tooling
4. prototype implementation – compute one execution, fastish
5. specification coverage measurement
6. test generation based on the specification
7. generate boilerplate for production implementation (AST type etc)
8. parse and render small examples (partly symbolic)
9. generate parser for prototype or production implementation
10. generate pretty printer – and code formatter
11. theorem-prover definitions for proof

Multiple usages? Language specifications should support:

1. reference documentation in prose and in accessible maths
2. sanity checking of definition (metalanguage sort-checking and type-checking, ...)
3. test oracle, for testing specification/implementation consistent – opsem and alg. type inf
    3.1 decide whether an observed implementation outcome is allowed (maybe with extra instrumentation)
    3.2 exhaustively compute all the allowed outcomes – and web-interface exploration tooling
4. prototype implementation – compute one execution, fastish
5. specification coverage measurement
6. test generation based on the specification
7. generate boilerplate for production implementation (AST type etc)
8. parse and render small examples (partly symbolic)
9. generate parser for prototype or production implementation
10. generate pretty printer – and code formatter
11. theorem-prover definitions for proof
12. generate proof boilerplate (substitution lemmas etc)

Multiple usages? Language specifications should support:

1. reference documentation in prose and in accessible maths
2. sanity checking of definition (metalanguage sort-checking and type-checking, ...)
3. test oracle, for testing specification/implementation consistent – opsem and alg. type inf
   - 3.1 decide whether an observed implementation outcome is allowed (maybe with extra instrumentation)
   - 3.2 exhaustively compute all the allowed outcomes – and web-interface exploration tooling
4. prototype implementation – compute one execution, fastish
5. specification coverage measurement
6. test generation based on the specification
7. generate boilerplate for production implementation (AST type etc)
8. parse and render small examples (partly symbolic)
9. generate parser for prototype or production implementation
10. generate pretty printer – and code formatter
11. theorem-prover definitions for proof
12. generate proof boilerplate (substitution lemmas etc)
13. proof automation for metatheory

Multiple usages? Language specifications should support:

1. reference documentation in prose and in accessible maths
2. sanity checking of definition (metalanguage sort-checking and type-checking, ...)
3. test oracle, for testing specification/implementation consistent – opsem and alg. type inf
    3.1 decide whether an observed implementation outcome is allowed (maybe with extra instrumentation)
    3.2 exhaustively compute all the allowed outcomes – and web-interface exploration tooling
4. prototype implementation – compute one execution, fastish
5. specification coverage measurement
6. test generation based on the specification
7. generate boilerplate for production implementation (AST type etc)
8. parse and render small examples (partly symbolic)
9. generate parser for prototype or production implementation
10. generate pretty printer – and code formatter
11. theorem-prover definitions for proof
12. generate proof boilerplate (substitution lemmas etc)
13. proof automation for metatheory
14. use directly in verification tooling

Multiple usages? Language specifications should support:

1. reference documentation in prose and in accessible maths
2. sanity checking of definition (metalanguage sort-checking and type-checking, ...)
3. test oracle, for testing specification/implementation consistent – opsem and alg. type inf
   - 3.1 decide whether an observed implementation outcome is allowed (maybe with extra instrumentation)
   - 3.2 exhaustively compute all the allowed outcomes – and web-interface exploration tooling
4. prototype implementation – compute one execution, fastish
5. specification coverage measurement
6. test generation based on the specification
7. generate boilerplate for production implementation (AST type etc)
8. parse and render small examples (partly symbolic)
9. generate parser for prototype or production implementation
10. generate pretty printer – and code formatter
11. theorem-prover definitions for proof
12. generate proof boilerplate (substitution lemmas etc)
13. proof automation for metatheory
14. use directly in verification tooling
15. easy reuse and adaption

One metalanguage – and tool suite – to express them all?

One metalanguage – and tool suite – to express them all?

ho ho ho...

(LCF??)

One metalanguage – and tool suite – to express them all?

ho ho ho...

(LCF??)

but nonetheless, a big space between the current miserable state of the art and that

we should see how far we can get – focussing on building things which will actually be useful for some purposes ... and on the fundamental underlying problems

“Production language”?

The problems vary with the technical and social context

...from one researcher exploring a small calculus, to the worldwide C/C++ community

“Production language”?

The problems vary with the technical and social context

...from one researcher exploring a small calculus, to the worldwide C/C++ community

- scale?

“Production language”?

The problems vary with the technical and social context

…from one researcher exploring a small calculus, to the worldwide C/C++ community

- scale?
- specification at design time or post facto?

“Production language”?

The problems vary with the technical and social context

...from one researcher exploring a small calculus, to the worldwide C/C++ community

- scale?
- specification at design time or post facto?
- how closely tied to implementation effort(s)?

“Production language”?

The problems vary with the technical and social context

...from one researcher exploring a small calculus, to the worldwide C/C++ community

- scale?
- specification at design time or post facto?
- how closely tied to implementation effort(s)?
- legacy complexity?

“Production language”?

The problems vary with the technical and social context

...from one researcher exploring a small calculus, to the worldwide C/C++ community

- scale?
- specification at design time or post facto?
- how closely tied to implementation effort(s)?
- legacy complexity?
- community that has to understand the spec?

“Production language”?

The problems vary with the technical and social context

...from one researcher exploring a small calculus, to the worldwide C/C++ community

- scale?
- specification at design time or post facto?
- how closely tied to implementation effort(s)?
- legacy complexity?
- community that has to understand the spec?
- community that controls the spec?

“Production language”?

The problems vary with the technical and social context

...from one researcher exploring a small calculus, to the worldwide C/C++ community

- scale?
- specification at design time or post facto?
- how closely tied to implementation effort(s)?
- legacy complexity?
- community that has to understand the spec?
- community that controls the spec?
- the amount and nature of loose specification?

“Production language”?

The problems vary with the technical and social context

...from one researcher exploring a small calculus, to the worldwide C/C++ community

- scale?
- specification at design time or post facto?
- how closely tied to implementation effort(s)?
- legacy complexity?
- community that has to understand the spec?
- community that controls the spec?
- the amount and nature of loose specification?
- the shape of the semantics – the mathematical structure required?

“Production language”?

The problems vary with the technical and social context

...from one researcher exploring a small calculus, to the worldwide C/C++ community

- scale?
- specification at design time or post facto?
- how closely tied to implementation effort(s)?
- legacy complexity?
- community that has to understand the spec?
- community that controls the spec?
- the amount and nature of loose specification?
- the shape of the semantics – the mathematical structure required?
- how important it is to get it right?

“Production language”?

The problems vary with the technical and social context

…from one researcher exploring a small calculus, to the worldwide C/C++ community

- scale?
- specification at design time or post facto?
- how closely tied to implementation effort(s)?
- legacy complexity?
- community that has to understand the spec?
- community that controls the spec?
- the amount and nature of loose specification?
- the shape of the semantics – the mathematical structure required?
- how important it is to get it right?
- general-purpose vs custom tooling? how opinionated?

“Production language”?

The problems vary with the technical and social context

...from one researcher exploring a small calculus, to the worldwide C/C++ community

- scale?
- specification at design time or post facto?
- how closely tied to implementation effort(s)?
- legacy complexity?
- community that has to understand the spec?
- community that controls the spec?
- the amount and nature of loose specification?
- the shape of the semantics – the mathematical structure required?
- how important it is to get it right?
- general-purpose vs custom tooling? how opinionated?
- commitment to particular language or prover ecosystem?

“Production language”?

The problems vary with the technical and social context

...from one researcher exploring a small calculus, to the worldwide C/C++ community

- scale?
- specification at design time or post facto?
- how closely tied to implementation effort(s)?
- legacy complexity?
- community that has to understand the spec?
- community that controls the spec?
- the amount and nature of loose specification?
- the shape of the semantics – the mathematical structure required?
- how important it is to get it right?
- general-purpose vs custom tooling? how opinionated?
- commitment to particular language or prover ecosystem?
- generation vs abstraction?

The existing tooling?

The existing tooling?

- LaTeX?

The existing tooling?

- LaTeX?
  - Terrible

The existing tooling?

- LaTeX?
  - Terrible
- ACL2/Agda/Idris/HOL4/Idris/Isabelle/Lean/Rocq/Twelf/...

The existing tooling?

- LaTeX?
  - Terrible
- ACL2/Agda/Idris/HOL4/Idris/Isabelle/Lean/Rocq/Twelf/...
  - Terrible

The existing tooling?

- LaTeX?
  - Terrible
- ACL2/Agda/Idris/HOL4/Idris/Isabelle/Lean/Rocq/Twelf/...
  - Terrible
- Ott/Lem/Sail

The existing tooling?

- LaTeX?
  - Terrible
- ACL2/Agda/Idris/HOL4/Idris/Isabelle/Lean/Rocq/Twelf/...
  - Terrible
- Ott/Lem/Sail
  - Terrible

The existing tooling?

- LaTeX?
  - Terrible
- ACL2/Agda/Idris/HOL4/Idris/Isabelle/Lean/Rocq/Twelf/...
  - Terrible
- Ott/Lem/Sail
  - Terrible
- Your favourite research tool

The existing tooling?

- LaTeX?
  - Terrible
- ACL2/Agda/Idris/HOL4/Idris/Isabelle/Lean/Rocq/Twelf/...
  - Terrible
- Ott/Lem/Sail
  - Terrible
- Your favourite research tool
  - Probably terrible?

The existing tooling?

- LaTeX?
  - Terrible
- ACL2/Agda/Idris/HOL4/Idris/Isabelle/Lean/Rocq/Twelf/...
  - Terrible
- Ott/Lem/Sail
  - Terrible
- Your favourite research tool
  - Probably terrible?

...each one great at some aspects of the problem – but as a whole, terrible

Why so terrible?

Why so terrible?

- incentives to build and maintain production-quality tooling

Why so terrible?

- incentives to build and maintain production-quality tooling
- fundamental trade-offs – one can't have all the nice things at once

Why so terrible?

- incentives to build and maintain production-quality tooling
- fundamental trade-offs – one can't have all the nice things at once
- we don't know how

This talk: reflect on experience and highlight some interesting and addressable challenges

Previous language and tool support for (particular kinds of) semantics

Ott

- user-defined concrete and abstract syntax, and inductive relations over it, for PL definitions
- generates LaTeX and HOL4/Isabelle/Rocq definitions

  [Zappa Nardelli, Owens, Sarkar, Sewell, ...] 2005 – now

Lem

- pure functional definitions – roughly core pure OCaml plus libraries for numbers, sets, maps, ...
- generates LaTeX, OCaml, and HOL4/Isabelle/(bad)Rocq definitions

  [Owens, Mulligan, Tuerk, Gray, Bohm, Zappa Nardelli, Sewell, ...] 2010 – now

Sail

- first-order functional/imperative language for instruction-set semantics
- lightweight dependent types for bitvector widths
- generates LaTeX, Asciidoc, C and OCaml emulators, HOL4/Isabelle/Lean/Rocq definitions, SystemVerilog
- Isla symbolic evaluation engine

  [Armstrong, Bauereiss, Campbell, Reid, Gray, Norton, Mundkur, Wassell, French, Pulte, Flur, Stark, Krishnaswami, Sewell] 2013 – now

A little about what each does – and doesn't – do, to sneak up on these questions:

- the foundational question of what formal languages are suitable for this purpose;
- the pragmatic question of what facilities good tool support should provide; and
- the engineering question of actually building these tools to a sufficient quality to be usable by other researchers and by industry staff, not just by their authors.

Ott – for classic PL operational semantics and type systems

Early goal:

> *"to have a source syntax that is as readable as possible, so that one can work for much of the time directly on the source document without constantly re-typesetting"*

(started off as recreational hackery with Francesco Zappa Nardelli, following Acute)

- Acute: high-level programming language design for distributed computation
  [Sewell, Leifer, Wansbrough, Zappa Nardelli, Allen-Williams, Habouzit, Vafeiadis]
  Implemented in FreshOCaml, included some nominal features, opsem and typing specified in munged LaTeX
- Type-Safe Distributed Programming for OCaml [Billings, Sewell, Shinwell, Strniša]

Suppose you have a great idea for a new type discipline

$$
\begin{array}{lcl}
t & ::= & \\
 & | & x \\
 & | & \lambda x.t \\
 & | & t\ t' \\
\\
T & ::= & \\
 & | & X \\
 & | & T \to T' \\
\\
\Gamma & ::= & \\
 & | & \text{empty} \\
 & | & \Gamma, x : T
\end{array}
$$

$$\boxed{\Gamma \vdash t : T}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{GtT\_var}$$

$$\frac{\begin{array}{l}\Gamma \vdash t : T_1 \to T_2 \\ \Gamma \vdash t' : T_1\end{array}}{\Gamma \vdash t\ t' : T_2} \quad \text{GtT\_app}$$

$$\frac{\Gamma, x_1 : T_1 \vdash t : T}{\Gamma \vdash \lambda x_1.t : T_1 \to T} \quad \text{GtT\_lam}$$

Write the LaTeX?

```
%% defn GtT
\newcommand{\mydruleGtTXXvar}[1]{\mydrule[#1]{%
\mypremise{\mymv{x} \mysym{:} \mynt{T} \, \in \, \Gamma}%
}{
\Gamma \vdash \mymv{x} \mysym{:} \mynt{T}}{%
{\mydrulename{GtT\_var}}{}%
}}
\newcommand{\mydruleGtTXXapp}[1]{\mydrule[#1]{%
\mypremise{\Gamma \vdash \mynt{t} \mysym{:} \mynt{T_{{\mathrm{1}}}} \
    rightarrow \mynt{T_{{\mathrm{2}}}}}%
\mypremise{\Gamma \vdash \mynt{t'} \mysym{:} \mynt{T_{{\mathrm{1}}}}}%
}{
\Gamma \vdash \mynt{t} \, \mynt{t'} \mysym{:} \mynt{T_{{\mathrm{2}}}}}{%
{\mydrulename{GtT\_app}}{}%
}}
\newcommand{\mydruleGtTXXlam}[1]{\mydrule[#1]{%
\mypremise{\Gamma \mysym{,} \mymv{x_{{\mathrm{1}}}} \mysym{:} \mynt{T_{{\mathrm
    {1}}}} \vdash \mynt{t} \mysym{:} \mynt{T}}%
}{
\Gamma \vdash \lambda \mymv{x_{{\mathrm{1}}}} \mysym{.} \mynt{t} \mysym{:} \
    mynt{T_{{\mathrm{1}}}} \rightarrow \mynt{T}}{%
{\mydrulename{GtT\_lam}}{}%
}}
\newcommand{\mydefnGtT}[1]{\begin{mydefnblock}[#1]{Γ ⊢ t : T}{}
```

...you just write down your language, and inductive definition rules in it

```
metavar x ::=
metavar X ::=

grammar
  t :: ’t_’ ::=
    | x                 :: :: var
    | \ x . t           :: :: lam
    | t t’              :: :: app

  T :: ’T_’ ::=
    | X                 :: :: var
    | T -> T’           :: :: fn

  G :: ’G_’ ::=
    | empty             :: :: emp
    | G , x : T         :: :: cons

  formula :: ’formula_’ ::=
    | judgement         :: :: judgement
    | x : T in G        :: :: xTG
```

```
defns
  Jtype :: ’’ ::=

defn
  G |- t : T :: :: GtT :: GtT_ by

  x:T in G
  -------- :: var
  G |- x:T


  G |- t : T1->T2
  G |- t’ : T1
  ---------------- :: app
  G |- t t’ : T2


  G,x1:T1 |- t : T
  ------------------ :: lam
  G |- \x1.t : T1->T
```

Ott parses the rules, in that language, and generates the latex you saw. That parsing already does useful sanity checking, helping one keep the thing consistent. No big deal for this tiny lambda calculus, of course – but when things scale, it really helps.

plus a bunch of engineering to make it really usable

- quote parts of a definition in LaTeX: write `\ottdefnsJtype` and get the `Jtype` rules
- embed object language in LaTeX: write `[[\x1.\x2.x1 x2]]` and get $\lambda x_1 . \lambda x_2 . x_1\ x_2$
  (properly parsed and thus sort-checked)
- override default typesetting, per production, nonterminal, or terminal
- support `...` lists, e.g. `G|-t1:T1 ...  G|-tn:Tn`
- add comments to grammars, productions, and rules
- ...

You want prover definitions? Just add defns of the non-free things

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

grammar
  term, t :: 't_' ::=
    | x                 :: :: var
    | \ x . t           :: :: lam
    | t t'              :: :: app

  type, T :: 'T_' ::=
    | X                 :: :: var
    | T -> T'           :: :: fn

  gamma, G :: 'G_' ::=   {{ isa (termvar*type) list }} {{ coq list (termvar*type) }} {{ hol (termvar
    | empty             :: :: emp  {{ isa Nil }} {{ coq G_nil }} {{ hol [] }}
    | G , x : T         :: :: cons {{ isa ([[x]],[[T]])#[[G]] }} {{ coq (cons ([[x]],[[T]]) [[G]]) }} {{

  formula :: 'formula_' ::=
    | judgement         :: :: judgement
    | x : T in G        :: :: xTG
      {{ isa ? G1 G2. [[G]] = G1 @ ([[x]],[[T]])#[[G2]] & [[x]]~:fst ` set G1 }}
```

Generated Rocq (or HOL4, or Isabelle)

```
Inductive type : Set :=
 | T_var (X: typevar)
 | T_fn (T: type) (T': type).

Inductive term : Set :=
 | t_var (x: termvar)
 | t_lam (x: termvar) (t: term)
 | t_app (t: term) (t': term).

Definition gamma : Set := list (termvar*type).

(** definitions *)
(* defns Jtype *)
Inductive GtT : gamma → term → type → Prop :=  (* defn GtT *)
 | GtT_var : forall (G: gamma) (x: termvar) (T: type),
      (bound  x   T   G )  →
     GtT G (t_var x) T
 | GtT_app : forall (G: gamma) (t t': term) (T2 T1: type),
     GtT G t (T_fn T1 T2) →
     GtT G t' T1 →
     GtT G (t_app t t') T2
 | GtT_lam : forall (G: gamma) (x1: termvar) (t: term) (T1 T: type),
     GtT  (cons ( x1 , T1 )  G )  t T →
     GtT G (t_lam x1 t) (T_fn T1 T).
```

Does it work?

This works great as far as it goes.

Modest but very enthusiastic user base, e.g.:

> *I have continued to use Ott and its locally nameless backend for much of my work*
>
> Stephanie Weirich, 2023-08

But...

Challenge 1: first-class grammar constructions

You might think you were defining *a* language, but it ends up a complex family:

- concrete object-language syntax
  - with lexing for concrete variables and constants, and with syntactic sugar
- concrete syntax of symbolic terms used in rules
  - with symbolic metavariables and nonterminals, and with meta-operations, e.g. substitution
- hybrid concrete + symbolic
- abstract syntax that those terms denote
- sub-grammars, e.g. of values as a subgrammar of expressions
- context grammars, e.g. for evaluation contexts, or all contexts
- with source location information
- suppressing some parts for presentation
- and families of related object languages, e.g. with optional type annotation, or extra constructors used only in a dynamic semantics, or parameterised, or for elaborations

Ott supports all that with ad hoc features – but really there's a whole little subject of practical operations on grammar definition fragments that should be set up explicitly

Challenge 2: variable binding

Still a headache

Industrial languages have mostly accreted complex scoping that may need custom scope resolution (or have no scoping, e.g. assembly)

Short of that, one often has complex binding structure, e.g. mutual letrecs, dependent record patterns, etc. Ott lets one express this, e.g.:

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

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

but what does it mean? Really: how can we best compute and reason above it?
Math/tooling options:can get surprisingly far with concrete representation (Ott generates subst fns) and/or manual de Bruijn. Ott locally nameless backend for simple binding, and LNgen tooling [Weirich et al.] Nominal Isabelle [Urban et al.]? MLSOS [Lakin, Pitts]? αProlog [Cheney]? de Bruijn and Autosubst2 tooling [Stark et al.]? [Fiore and Szamozvancev]? [Allais et al.]?

Challenge 3: production parsing, pretty-printing, formatting

Ott supports arbitrary context-free grammars for abstract syntax, with SGLR parser for rules

- expressive, and works well for semantic rules (ambiguity rare, and user can fix if need be)
- but doesn't give a standalone production parser
- and limited fancy lexing

Challenge 3: production parsing, pretty-printing, formatting

Ott supports arbitrary context-free grammars for abstract syntax, with SGLR parser for rules

- expressive, and works well for semantic rules (ambiguity rare, and user can fix if need be)
- but doesn't give a standalone production parser
- and limited fancy lexing

experiment with letting the user specify by quotienting:

- let user specify abstract syntax as *quotient* of concrete grammar definition
- generate menhir parser

Try that for real?

Identify some other sweet spot of grammar expressiveness, or smooth path from initial context-free parsing to production parser with decent errors

(Beware that real languages have accreted syntactic weirdnesses – hard to cover)

Challenge 4: executability!

Ott supports pretty arbitrary inductive relation definitions – nicely expressive.
Suppose you define an operational semantics, type checker, or type inference system.

Can you run it as a (non-optimised) implementation? No.

(unless it happens to be in the scope of Isabelle code generation – and that adds friction, even if so)
(and even when you can, what about error reporting?)
(and what happens e.g. when you want to move from substitution to environments?)

Challenge 4: executability!

Ott supports pretty arbitrary inductive relation definitions – nicely expressive.
Suppose you define an operational semantics, type checker, or type inference system.

Can you run it as a (non-optimised) implementation? No.

(unless it happens to be in the scope of Isabelle code generation – and that adds friction, even if so)

(and even when you can, what about error reporting?)

(and what happens e.g. when you want to move from substitution to environments?)

Can you run it as a test oracle, i.e. given a trace of what a production implementation did, check that? No.

(all Ott can do is generate OCaml datatypes, and substitution, free-variable, and subgrammar functions, to write a conventional implementation above – though even that is surprisingly useful, to keep implementation and semantics in sync)

Challenge 4: executability!

Ott supports pretty arbitrary inductive relation definitions – nicely expressive.
Suppose you define an operational semantics, type checker, or type inference system.

Can you run it as a (non-optimised) implementation? No.

(unless it happens to be in the scope of Isabelle code generation – and that adds friction, even if so)
(and even when you can, what about error reporting?)
(and what happens e.g. when you want to move from substitution to environments?)

Can you run it as a test oracle, i.e. given a trace of what a production implementation did, check that? No.

(all Ott can do is generate OCaml datatypes, and substitution, free-variable, and subgrammar functions, to write a conventional implementation above – though even that is surprisingly useful, to keep implementation and semantics in sync)

Can one identify a sweet spot of inductive-relation definition expressiveness?

Challenge 5a: metalanguage expressiveness: functional specifications

Ott only really supports inductive relation definitions – horribly inexpressive
(some function-definition support, but it's not good)

For quite a bit of classic PL, that's fine, but sometimes one really wants to specify things with pure functional-programming definitions: functions, pattern-matching, some degree of polymorphism, tuples, records, numeric types, lists, sets, finite maps, etc.

Lem: for pure functional specifications

Building substantial specifications, with executability-as-test-oracle a primary concern.

Wanted to avoid theorem-prover lock-in and friction, but also support proof (perhaps later).

Lem: roughly pure core Caml plus typeclasses, and a library of numbers, sets, maps, etc.
Generates OCaml, LaTeX, HOL4, Isabelle, and (bad) Rocq

		raw LoC
C/C++ concurrency	[Batty et al.]	2493
Arm/RISC-V/Power/x86 concurrency	[Flur,Pulte,Sarkar,Simner,Sewell]	19124
Cerberus C semantics	[Memarian]	46868
ELF+DWARF linking and debug info	[Mulligan,Sewell]	28962
SibylFS POSIX filesystems	[Ridge et al.]	7426
CakeML original semantics	[Owens]	5363
OCaml-light (from Ott)	[Owens]	3918
Morello ISA (from Sail)	[Bauereiss,Campbell,Armstrong]	circa 200 000
Armv9-A ISA (from Sail)	[Bauereiss,Campbell,Armstrong]	more
RISC-V ISA (from Sail)	[Bauereiss,Campbell,Armstrong]	less

(the last three use Lem as a convenient intermediate to generate Isabelle and HOL4 from Sail ISA definitions)

Lem source example: DWARF semantics snippet

```
(** evaluation of cfa information from .debug_frame *)
let evaluate_call_frame_instruction (fi: frame_info) (cie: cie) (state: cfa_state)
    (cfi: call_frame_instruction) : cfa_state =
  [...]
  match cfi with
  (* Row Creation Instructions *)
  | DW_CFA_set_loc a                ->
      create_row a
  | DW_CFA_advance_loc d            ->
      create_row (state.cs_current_row.ctr_loc +
        d * cie.cie_code_alignment_factor)
  [...]
  (* CFA Definition Instructions *)
  | DW_CFA_def_cfa r n              ->
      update_cfa (CR_register r (integerFromNatural n))
  | DW_CFA_def_cfa_sf r i           ->
      update_cfa (CR_register r (i * cie.cie_data_alignment_factor))
  | DW_CFA_def_cfa_register r       ->
      match state.cs_current_row.ctr_cfa with
      | CR_register r' i ->
          update_cfa (CR_register r i)
      | CR_undefined ->
          (* FIXME: this is to handle a bug in riscv64-gcc.
          gcc generates "DW_CFA_def_cfa_register: r2 (sp)" as the first instruction.
```

Lem-generated OCaml: DWARF semantics snippet

```
(** evaluation of cfa information from .debug_frame *)
let evaluate_call_frame_instruction (fi: frame_info) (cie1: cie) (state1: cfa_state)
    (cfi: call_frame_instruction) : cfa_state=
  [...]
  (match cfi with
  (* Row Creation Instructions *)
  | DW_CFA_set_loc a              ->
      create_row a
  | DW_CFA_advance_loc d          ->
      create_row ( Nat_big_num.add state1.cs_current_row.ctr_loc
        (Nat_big_num.mul d cie1.cie_code_alignment_factor))
  [...]
  (* CFA Definition Instructions *)
  | DW_CFA_def_cfa( r, n)            ->
      update_cfa (CR_register( r, ( n)))
  | DW_CFA_def_cfa_sf( r, i)         ->
      update_cfa (CR_register( r, ( Nat_big_num.mul i cie1.cie_data_alignment_factor)))
  | DW_CFA_def_cfa_register r     ->
      (match state1.cs_current_row.ctr_cfa with
      | CR_register( r’, i) ->
          update_cfa (CR_register( r, i))
      | CR_undefined ->
          (* FIXME: this is to handle a bug in riscv64-gcc.
          gcc generates "DW_CFA_def_cfa_register: r2 (sp)" as the first instruction.
```

Aside on generating code

Ott and Lem generate code for proof tools – but one should be wary; to be really usable:

- it has to be readable
- idiomatic
- not too noisy
- avoid synthesising unpredictable identifiers
- preserve comments

Limits how fancy a translation one might accept

At scale, the same applies to generating code for execution of semantics – it has to be human-readable for debugging. C.f. Rocq extraction?

(and for generating implementation components, ideally one would support a range of implementation languages)

Levels of executablity:

- concrete execution
  - ...for small-but-tricky examples (e.g. litmus tests)
  - ...for real programs, but slowly
  - ...for real programs, in production (e.g. a type-checker?)
- concrete test-oracle checking
  - ...likewise
- symbolic execution
- inside/outside a prover
- convenient components for typing and execution that can be used in testing setups and model checkers

Probably one wants a stratified specification language with well-defined subsets, of restricted expressivenesses that support additional usages

Lem: Does it work?

Yes and no.

We have been able to build and test those semantics – and we probably wouldn't have if working inside a prover the whole time; it's enabled semantics at scale

People have successfully used multiple prover targets:

- HOL4 proof about the original CakeML semantics [Owens]
- Isabelle proof about the Morello ISA semantics [Bauereiss, T.Sewell]
- HOL4 proof underpinning CakeML with the Sail/Lem/HOL4 Arm-A ISA semantics [Kanabar,Fox,Myreen]

but most of those definitions are really good only for execution and documentation, not proof, and the Lem-generated Rocq definitions are not good

...even though morally these are all just simple functional definitions, that should be expressible in any prover

Lem expressiveness tradeoff

Want enough expressiveness to ease writing specifications

But not so much that it is hard to translate into idiomatic usable definitions for all the targets (OCaml, Rocq, HOL4, Isabelle)

Unclear how well these are reconcilable

Challenge 5b: metalanguage expressiveness: type classes, Lem library, and modularity

How fancy a type system does one want?

OCaml:	ML modules
Rocq:	ML modules, type classes, sections, canonical structures
Isabelle:	type classes, locales, contexts
HOL4:	

Lem: simple type classes – but lack of higher kinds and multiparameter type classes makes working with monads awkward. But those are not supported in all targets.

Library design: reconcile unbounded numbers, sets, etc. with executable representations for finitary data. And the various standard libraries use type classes differently.

And no high-level module system to speak of. Parts of Cerberus now use OCaml modules.

Challenge 5c: metalanguage expressiveness: partiality

Lem permits partial pattern matching, and our models currently do use that – e.g. relying on invariants that the non-dependent Lem type system can't express. (Maybe they shouldn't, but it does reduce noise, from default values or lifted types)

Not a big problem for HOL4 or Isabelle, but problematic for generating decent Rocq.

Termination is mostly "obvious", but not always primitive recursive.

Challenge 5d: metalanguage expressiveness: prop and bool

Lem doesn't distinguish computable and non-computable, but Rocq has to.

Challenge 5e: metalanguage expressiveness: dependent types for bitvectors

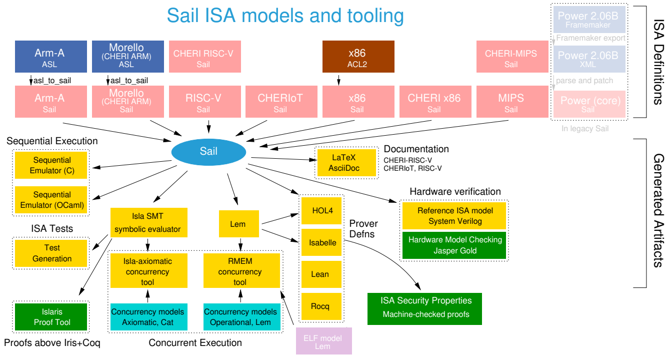
Instruction-set semantics involves many bitvectors, sometimes of dynamically computed sizes, which one really wants to be statically typechecked, so some modest dependent typing is needed – but Lem is non-dependent (following HOL4 and Isabelle)

Sail: for instruction-set architecture (ISA) specifications

more specific goal, so we can do a better job

- Simple first-order imperative language – so can easily generate fast-ish C emulator
- Lightweight dependent types (using Z3) – decidable typechecking, expressive enough for real ISAs
- Uninterpreted register and memory effects – so can glue onto concurrency semantics
- Simple values – so can build symbolic evaluator (Isla) to simplify semantics and run concurrency tests

multiple models and multiple targets (not all plumbed together):



Back to metalanguages for higher-level PL semantics

- What should one use to actually define (say) OCaml, or Haskell, or Rust, or Java?
- What should semantics researchers best use?
- What tooling could I give to undergrad or postgrad PL students, or industry folk?

Have to resolve tension between executability, expressiveness, and suitability for proof

Just Rocq, but with care for extractability?
with enough effort and skill, one can get things done – and c.f. Software Foundations [Pierce et al.] – but still all those limitations

Just Isabelle? Just Lean? (likewise)

Nott?

If one were designing a new metalanguage and tool, one has to think about all those challenges. Some guesses:

0. Keep core user experience of Ott
1. Support families of languages nicely. Surely solvable
2. Variable binding. Is Autosubst2 good enough (for Rocq)? For other provers? More general Nominal?
3. Production parsing and pretty printing. Expressiveness of grammar: reconcile general CFG and production-parsable grammar (by treating abstract as quotient of concrete?). Build parsing and pretty-printing bidirectionally from the outset? Engineer source location info, preservation of whitespace, code formatting.
4. Executability. Expressiveness of inductive rules – identify classes that are enough executable, with/without search, either as implementation or as test-oracle checker.
5. Inductive relations and higher-order functions and quantifiers...
6. Fancy types. Expressiveness of metalanguage type system vs inexpressiveness of some targets. Give up on targetting multiple provers – just Rocq? (or just Isabelle?) (or just Lean?).
   Stratify language, in some checkable way, to give benefits of inexpressiveness where possible. CIC fragments??
   Library and choice of representation – let the user choose e.g. finite sets and bounded numbers for execution, unbounded for proof, case-by-case and use-by-use.
7. Partiality. Machinery to defer totality and termination checking in non-obvious cases, to leave that to prover (and leave unnecessary for execution).
8. Prop vs bool. If targetting Rocq, have to have basically that distinction.
9. Dependent types for bitvectors. Include.

Related work

Lots... shamefully neglected in the talk

- HOL4, Isabelle, Lean, Rocq, etc., and their front-end fancy syntax and extraction mechanisms
- Lem sort-of-executable inductive relations [Williams]
- MLSOS [Lakin, Pitts]
- αProlog [Cheney]
- Spoofax [Visser et al.]
- PLTredex [Flatt et al.]
- Why3 [Filliâtre et al.]
- hs-to-Coq [Weirich et al.]
- K [Rosu et al.]
- SASyLF [Aldrich et al.]
- ClaReT [Boulton]
- Ruler [Dijkstra, Swierstra]
- Jakarta [Barthe et al.]
- MSOS [Mosses]
- ERGO [Lee et al.]
- CENTAUR [Kahn et al.]
- Synthesiser Generator [Reps, Teitelbaum]
- ...

Conclusion

- the foundational question of what formal languages are suitable for this purpose;

  ...don't know – but it seems more within reach

- the pragmatic question of what facilities good tool support should provide;

  ...solid wishlist based on experience

- the engineering question of actually building these tools to a sufficient quality to be usable by other researchers and by industry staff, not just by their authors.

  ...some of those problems could be studied in isolation – but ultimately one has to get over a threshold of usefulness, to tackle the real problem – one has to consider (at least) all these challenges