Formal specification and big-step operational semantics

Lecture 2

MPhil ACS & Part III course, Functional Programming: Implementation, Specification and Verification

Magnus Myreen
Michaelmas term, 2013

Formal methods

Formal specification

Formal methods

Formal specification

What makes it 'formal'?

Formal methods

Formal specification

What makes it ‘formal’?

Answer: ‘formal’ as in “formalised in a logic or calculus”

- has precise meaning
- e.g. lambda calculus, first-order logic or even a programming logic

First-order logic (FOL)

FOL syntax:

terms:

- variables: x, y, z
- constants and functions: f, g, h

formulas:

- predicates: P, Q, R
- connectives: $\wedge, \vee, \implies, \neg$
- quantifiers: $\forall, \exists$

Rules of inference are used to derive theorems:

$$\vdash \quad (\exists x.\ \forall y.\ P\ (x, y)) \implies (\forall y.\ \exists x.\ P\ (x, y))$$

Higher-order logic (HOL)

This course uses ‘formal’ to mean “can be formalised in higher-order logic” (Church’s simple theory of types).

Higher-order logic (HOL)

This course uses 'formal' to mean "can be formalised in higher-order logic" (Church's simple theory of types).

HOL is widely used since:

- more expressive than FOL
- specifications are often more natural, shorter
- originally developed as a foundation for mathematics

Higher-order logic (HOL)

This course uses ‘formal’ to mean “can be formalised in higher-order logic” (Church’s simple theory of types).

HOL is widely used since:

- more expressive than FOL
- specifications are often more natural, shorter
- originally developed as a foundation for mathematics

Main difference between FOL and HOL:

- no distinction between terms and formulas
- functions and predicates are treated as first-class values

Higher-order logic (HOL)

This course uses 'formal' to mean "can be formalised in higher-order logic" (Church's simple theory of types).

HOL is widely used since:

- more expressive than FOL
- specifications are often more natural, shorter
- originally developed as a foundation for mathematics

Main difference between FOL and HOL:

- no distinction between terms and formulas
- functions and predicates are treated as first-class values

$$\vdash \ \forall P.\ \ (\exists x.\ \forall y.\ P\ (x, y)) \implies (\forall y.\ \exists x.\ P\ (x, y))$$

Higher-order logic (HOL)

This course uses 'formal' to mean "can be formalised in higher-order logic" (Church's simple theory of types).

HOL is widely used since:

- more expressive than FOL
- specifications are often more natural, shorter
- originally developed as a foundation for mathematics

Main difference between FOL and HOL:

- no distinction between terms and formulas
- functions and predicates are treated as first-class values

$$\vdash\ \forall P.\ \ (\exists x.\ \forall y.\ P\ (x,y)) \implies (\forall y.\ \exists x.\ P\ (x,y))$$

quantification over a predicate

Familiar notation

Term	*Meaning*
$P(x)$	x **has property** P
$\neg t$	**not** t
$t_1 \ \wedge \ t_2$	t_1 **and** t_2
$t_1 \ \vee \ t_2$	t_1 **or** t_2
$t_1 \Rightarrow t_2$	t_1 **implies** t_2
$\forall x.\ t[x]$	**for all** x **it is the case that** $t[x]$
$\exists x.\ t[x]$	**for some** x **it is the case that** $t[x]$

HOL syntax

HOL $\approx$ (lambda calculus + constants) with ML-like types

HOL syntax

HOL $\approx$ (lambda calculus + constants) with ML-like types

HOL terms:

- variables: x, y, P, R
- constants (abbreviate fixed closed values, e.g. 0)
- function application: $t_1 \ t_2$
- lambda-terms: $\lambda x.\ t$ where x is a variable and t a term

HOL syntax

HOL $\approx$ (lambda calculus + constants) with ML-like types

HOL terms:

- variables: x, y, P, R
- constants (abbreviate fixed closed values, e.g. 0)
- function application: $t_1 \ t_2$
- lambda-terms: $\lambda x.\ t$ where x is a variable and t a term

HOL types:

- atomic types: type constants (e.g. num), type variables
- compound types: built using type operators (e.g. $t \rightarrow t'$)

HOL syntax

HOL $\approx$ (lambda calculus + constants) with ML-like types

HOL terms:

- variables: x, y, P, R
- constants (abbreviate fixed closed values, e.g. 0)
- function application: $t_1 \;\; t_2$
- lambda-terms: $\lambda x.\ t$ where x is a variable and t a term

HOL types:

- atomic types: type constants (e.g. num), type variables
- compound types: built using type operators (e.g. $t \rightarrow t'$)

Formulas: formulas are terms of type bool, i.e. can have value either true or false.

HOL in more detail

Only binding mechanism is λ-abstraction.

HOL in more detail

Only binding mechanism is λ-abstraction.

- quantifiers $\forall$ and $\exists$ are constants
- syntax $\forall x.\, t$ and $\exists x.\, t$ abbreviates $\forall(\lambda x.\, t)$ and $\exists(\lambda x.\, t)$

HOL in more detail

Only binding mechanism is λ-abstraction.

- quantifiers $\forall$ and $\exists$ are constants
- syntax $\forall x.\, t$ and $\exists x.\, t$ abbreviates $\forall(\lambda x.\, t)$ and $\exists(\lambda x.\, t)$

Example:

$\forall n.\ P(n) \Rightarrow P(n+1)$ abbreviates $\forall(\lambda n.\ \Rightarrow(P(n))(P(+\ n\ 1)))$

HOL in more detail

Only binding mechanism is λ-abstraction.

- quantifiers $\forall$ and $\exists$ are constants
- syntax $\forall x.\,t$ and $\exists x.\,t$ abbreviates $\forall(\lambda x.\,t)$ and $\exists(\lambda x.\,t)$

Example:

$\forall n.\ P(n) \Rightarrow P(n+1)$ abbreviates $\forall(\lambda n.\ \Rightarrow(P(n))(P(+\ n\ 1)))$

Only three primitive constants: $=,\ \Longrightarrow,\varepsilon$ (Hilbert's choice)

The rest are defined, e.g.

$$\text{true} \equiv ((\lambda x.\ x) = (\lambda x.\ x))$$

$$\forall \equiv \lambda P.\ (P = \lambda x.\ \text{true})$$

HOL examples

Induction over the natural numbers:

$$\forall P.\ P(0) \wedge (\forall n.\ P(n) \Rightarrow P(n+1)) \Rightarrow \forall n.\ P(n)$$

HOL examples

Induction over the natural numbers:

$$\forall P.\ P(0) \wedge (\forall n.\ P(n) \Rightarrow P(n+1)) \Rightarrow \forall n.\ P(n)$$

Legitimacy of simple recursive function definition:

$$\forall n_0.\ \forall f.\ \exists s.\ (s(0) = n_0) \wedge (\forall n.\ s(n+1) = f(s(n)))$$

Primitive inferences

Eight primitive rules of inference:

$$\frac{}{t \vdash t} \qquad \frac{}{\vdash t = t} \qquad \frac{\Gamma \vdash t_2}{\Gamma - \{t_1\} \vdash t_1 \Rightarrow t_2} \qquad \frac{\Gamma_1 \vdash t_1 \Rightarrow t_2 \qquad\qquad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$$

$$\frac{}{\vdash (\lambda x.\ t_1)t_2 = t_1[t_2/x]} \qquad \frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.\ t_1) = (\lambda x.\ t_2)}$$

$$\frac{\Gamma_1 \vdash t_1 = t_1' \qquad \cdots \qquad \Gamma_n \vdash t_n = t_n' \qquad\qquad \Gamma \vdash t[t_1, \ldots, t_n]}{\Gamma_1 \cup \cdots \cup \Gamma_n \cup \Gamma \vdash t[t_1', \ldots, t_n']}$$

$$\frac{\Gamma \vdash t}{\Gamma[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n] \vdash t[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n]}$$

Primitive inferences

Eight primitive rules of inference:

$$\frac{}{t \ \vdash \ t} \qquad \frac{}{\vdash \ t = t} \qquad \frac{\Gamma \ \vdash \ t_2}{\Gamma - \{t_1\} \ \vdash \ t_1 \Rightarrow t_2} \qquad \frac{\Gamma_1 \ \vdash \ t_1 \Rightarrow t_2 \qquad\qquad \Gamma_2 \ \vdash \ t_1}{\Gamma_1 \cup \Gamma_2 \ \vdash \ t_2}$$

$$\frac{}{\vdash \ (\lambda x.\ t_1)t_2 = t_1[t_2/x]} \qquad \frac{\Gamma \ \vdash \ t_1 = t_2}{\Gamma \ \vdash \ (\lambda x.\ t_1) = (\lambda x.\ t_2)}$$

beta reduction

$$\frac{\Gamma_1 \ \vdash \ t \quad \cdots \quad t_n = t'_n \qquad\qquad \Gamma \ \vdash \ t[t_1, \ldots, t_n]}{\Gamma_1 \cup \cdots \cup \Gamma_n \cup \Gamma \ \vdash \ t[t'_1, \ldots, t'_n]}$$

$$\frac{\Gamma \ \vdash \ t}{\Gamma[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n] \ \vdash \ t[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n]}$$

Primitive inferences

Eight primitive rules of inference:

$$\frac{}{t \vdash t} \qquad \frac{}{\vdash t = t} \qquad \frac{\Gamma \vdash t_2}{\Gamma - \{t_1\} \vdash t_1 \Rightarrow t_2} \qquad \frac{\Gamma_1 \vdash t_1 \Rightarrow t_2 \qquad\qquad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$$

$$\frac{}{\vdash (\lambda x.\ t_1)t_2 = t_1[t_2/x]} \qquad \frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.\ t_1) = (\lambda x.\ t_2)}$$

beta reduction

abstraction

$$\frac{\Gamma_1 \vdash t \;\cdots\; t_n = \;\cdots\; , \ldots, t_n]}{\Gamma_1 \cup \cdots \cup \Gamma_n \cup \Gamma \vdash t[t_1, \ldots, t_n]}$$

$$\frac{\Gamma \vdash t}{\Gamma[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n] \vdash t[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n]}$$

Primitive inferences

Eight primitive rules of inference:

$$\frac{}{t \vdash t} \qquad \frac{}{\vdash t = t} \qquad \frac{\Gamma \vdash t_2}{\Gamma - \{t_1\} \vdash t_1 \Rightarrow t_2} \qquad \frac{\Gamma_1 \vdash t_1 \Rightarrow t_2 \qquad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$$

$$\frac{}{\vdash (\lambda x.\ t_1)t_2 = t_1[t_2/x]} \qquad \frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.\ t_1) = (\lambda x.\ t_2)}$$

beta reduction

abstraction

$$\frac{\Gamma_1 \vdash t \quad \cdots \quad t_n = \quad \cdots \quad , \ldots, t_n]}{\Gamma_1 \cup \cdots \cup \Gamma_n \cup \Gamma \vdash t[t_1, \ldots, t_n]}$$

$$\frac{\Gamma \vdash t}{\Gamma[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n] \vdash t[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n]}$$

A formal proof in HOL must follow the primitive inferences.

For practical proof work, we have proof assistants, e.g. HOL4, and Isabelle/HOL (these tools are not examinable).

FP language definition

How to write the formal specification of a FP language?

FP language definition

How to write the formal specification of a FP language?

Options:

- operational semantics (syntactic operations)
- denotational semantics (meaning of programs)
- axiomatic semantics (in terms of program logic)

FP language definition

How to write the formal specification of a FP language?

Options:

- operational semantics (syntactic operations)
- denotational semantics (meaning of programs)
- axiomatic semantics (in terms of program logic)

In this course, we use operational semantics.

FP language definition

How to write the formal specification of a FP language?

Options:

- operational semantics (syntactic operations)
- denotational semantics (meaning of programs)
- axiomatic semantics (in terms of program logic)

In this course, we use operational semantics.

The language definition will be the specification for the implementation (i.e. what we verify, formally prove).

The definition in practice

The definition in practice

We define the language using higher-order logic (HOL).

In practice, we define:

- the type of values in the FP language,
- the syntax of program expressions, and
- how program expressions evaluate.

The definition in practice

We define the language using higher-order logic (HOL).

In practice, we define:

- the type of values in the FP language,
- the syntax of program expressions, and
- how program expressions evaluate.

We will specify the semantics of

- a simple first-order Lisp
- a subset of SML (next lecture)

Lisp examples

```
> 1
1

> (+ 1 2)
3

> '(1 2 3)
(1 2 3)

> (cdr '(1 2 3))
(2 3)

> (defun app (x y)
    (if (consp x)
        (cons (car x) (app (cdr x) y))
      y))

> (app '(1 2 3) '(4 5 6))
(1 2 3 4 5 6)
```

Lisp values: s-expressions

We start by defining a type for Lisp values.

Lisp values: s-expressions

We start by defining a type for Lisp values.

Values in our Lisp are s-expressions:

- a number,
- a symbol (immutable string), or
- a pair of s-expressions.

Lisp values: s-expressions

We start by defining a type for Lisp values.

Values in our Lisp are s-expressions:

- a number,
- a symbol (immutable string), or
- a pair of s-expressions.

We can define this in HOL using a datatype declaration:

```
SExp ::= Dot of SExp SExp | Val of num | Sym of string
```

Lisp values: s-expressions

We start by defining a type for Lisp values.

Values in our Lisp are s-expressions:

- a number,
- a symbol (immutable string), or
- a pair of s-expressions.

We can define this in HOL using a datatype declaration:

```
SExp ::= Dot of SExp SExp | Val of num | Sym of string
```

Such a definition introduces a new type, SExp, and constructor functions in HOL:

```
Dot : SExp -> SExp -> SExp
Val : num -> SExp
Sym : string -> SExp
```

Syntax of programs

The concrete syntax of Lisp programs consists of strings:

```
(defun app (x y)
    (if (consp x)
        (cons (car x) (app (cdr x) y))
      y))
```

Syntax of programs

The concrete syntax of Lisp programs consists of strings:

```
(defun app (x y)
    (if (consp x)
        (cons (car x) (app (cdr x) y))
      y))
```

but the semantics is best defined in terms of abstract syntax.

We want a datatype for this...

Syntax of programs (cont.)

The datatype for the abstract syntax (AST) of our Lisp programs:

```
lisp_primitive_op ::=
          Cons | Car | Cdr | Equal | Less
        | Add | Sub | Consp | Natp | Symbolp

func ::= PrimitiveFun of lisp_primitive_op
        | Funcall | Fun of string

term ::= Const of SExp
        | Var of string
        | App of func (term list)
        | If of term term term
```

Syntax of programs (cont.)

The datatype for the abstract syntax (AST) of our Lisp programs:

```
lisp_primitive_op ::=
          Cons | Car | Cdr | Equal | Less
        | Add | Sub | Consp | Natp | Symbolp

func ::= PrimitiveFun of lisp_primitive_op
        | Funcall | Fun of string

term ::= Const of SExp
        | Var of string
        | App of func (term list)
        | If of term term term
```

Example: the program `(cons ‘1 ‘nil)` is represented as:

```
App (PrimitiveFun Cons) [Const (Val 1), Const (Sym “nil”)]
```

Modelling evaluation

Next, we define an big-step operational semantics (op.sem.) that defines how programs evaluate (i.e. execute).

The op.sem. is expressed as an inductive predicate/relation.

Example: inductive definition of the even natural numbers,

$$\frac{}{\textsf{Even}\ 0} \qquad \frac{\textsf{Even}\ n}{\textsf{Even}\ (n+2)}$$

Here Even n is true if and only if n is an even natural numbers.

Aside: defining ind. pred. in HOL

HOL allows only abbreviating definitions, of the form:

$$new\_constant \equiv closed\_term$$

Aside: defining ind. pred. in HOL

HOL allows only abbreviating definitions, of the form:

$$new\_constant \equiv closed\_term$$

How is Even defined?

Aside: defining ind. pred. in HOL

HOL allows only abbreviating definitions, of the form:

$$new\_constant \equiv closed\_term$$

How is Even defined?

$$\texttt{Even}\ n \equiv$$
$$(\forall P.\ (P\ 0) \land (\forall n.\ P\ n \implies P\ (n+2)) \implies P\ n)$$

Evaluation in our Lisp

Evaluation is defined as a mutually rec. inductive relations:

Evaluation in our Lisp

Evaluation is defined as a mutually rec. inductive relations:

expression evaluation:

$$(exp, a, fns) \Downarrow_{\mathrm{ev}} result$$

Evaluation in our Lisp

Evaluation is defined as a mutually rec. inductive relations:

expression evaluation:

$$(\mathit{exp}, a, \mathit{fns}) \Downarrow_{\mathrm{ev}} \mathit{result}$$

expression-list evaluation:

$$(\mathit{exp\_list}, a, \mathit{fns}) \Downarrow_{\mathrm{evl}} \mathit{result\_list}$$

Evaluation in our Lisp

Evaluation is defined as a mutually rec. inductive relations:

expression evaluation:

$$(exp, a, fns) \Downarrow_{\mathrm{ev}} result$$

expression-list evaluation:

$$(exp\_list, a, fns) \Downarrow_{\mathrm{evl}} result\_list$$

evaluation of function application:

$$(func, args, a, fns) \Downarrow_{\mathrm{ap}} result$$

Evaluation in our Lisp

Evaluation is defined as a mutually rec. inductive relations:

expression evaluation:

$$(\mathit{exp}, a, \mathit{fns}) \Downarrow_{\mathrm{ev}} \mathit{result}$$

expression to evaluate

tion:

$$(\mathit{exp\_list}, a, \mathit{fns}) \Downarrow_{\mathrm{evl}} \mathit{result\_list}$$

evaluation of function application:

$$(\mathit{func}, \mathit{args}, a, \mathit{fns}) \Downarrow_{\mathrm{ap}} \mathit{result}$$

Evaluation in our Lisp

Evaluation is defined as a mutually rec. inductive relations:

expression evaluation:

$$(exp, a, fns) \Downarrow_{\text{ev}} result$$

expression to evaluate

tion:

(

mapping from variables to values

evaluation of function application:

$$(func, args, a, fns) \Downarrow_{\text{ap}} result$$

Evaluation in our Lisp

Evaluation is defined as a mutually rec. inductive relations:

expression evaluation:

$$(exp, a, fns) \Downarrow_{\mathrm{ev}} result$$

expression to evaluate

tion:

(

mapping from variables to values

evaluat

function definitions

$$(func, args, a, fns) \Downarrow_{\mathrm{ap}} result$$

Evaluation in our Lisp

Evaluation is defined as a mutually rec. inductive relations:

expression evaluation:

$(exp, a, fns) \Downarrow_{\mathrm{ev}} result$

expression to evaluate

tion

(

mapping from variables to values

evaluat

function definitions

$(func, args, a,$

result value for terminating eval.

Big-step semantics

$$\frac{}{(\mathsf{Const}\ v, a, fns) \Downarrow_{\mathrm{ev}} v}$$

$$\frac{a(n) = v}{(\mathsf{Var}\ n, a, fns) \Downarrow_{\mathrm{ev}} v}$$

If-expressions

$$\frac{(e_1, a, \mathit{fns}) \Downarrow_{\mathrm{ev}} s_1 \quad (e_2, a, \mathit{fns}) \Downarrow_{\mathrm{ev}} s_2 \quad \mathsf{isTrue}\ s_1}{(\mathsf{If}\ e_1\ e_2\ e_3, a, \mathit{fns}) \Downarrow_{\mathrm{ev}} s_2}$$

$$\frac{(e_1, a, \mathit{fns}) \Downarrow_{\mathrm{ev}} s_1 \quad (e_3, a, \mathit{fns}) \Downarrow_{\mathrm{ev}} s_3 \quad \neg\mathsf{isTrue}\ s_1}{(\mathsf{If}\ e_1\ e_2\ e_3, a, \mathit{fns}) \Downarrow_{\mathrm{ev}} s_3}$$

App: function application

$$\frac{(el, a, fns) \Downarrow_{\mathrm{evl}} sl \qquad (f, sl, a, fns) \Downarrow_{\mathrm{ap}} s}{(\mathsf{App}\ f\ \ el, a, fns) \Downarrow_{\mathrm{ev}} s}$$

App: function application

$$\frac{(el, a, fns) \Downarrow_{\text{evl}} sl \qquad (f, sl, a, fns) \Downarrow_{\text{ap}} s}{(\textsf{App}\ f\ \ el, a, fns) \Downarrow_{\text{ev}} s}$$

$$\frac{}{([], a, fns) \Downarrow_{\text{evl}} []}$$

$$\frac{(e, a, fns) \Downarrow_{\text{ev}} s \qquad (el, a, fns) \Downarrow_{\text{evl}} sl}{(e :: el, a, fns) \Downarrow_{\text{evl}} s :: sl}$$

App continued

$$\frac{\textsf{eval\_primitive}\ (op, args) = s}{(\textsf{PrimitiveFun}\ op, args, a, fns) \Downarrow_{\mathrm{ap}} s}$$

$$\frac{fns(name) = (params, body) \qquad (body, params \mapsto args, fns) \Downarrow_{\mathrm{ev}} s}{(\textsf{Fun}\ name, args, a, fns) \Downarrow_{\mathrm{ap}} s}$$

$$\frac{(\textsf{Fun}\ name, args, a, fns) \Downarrow_{\mathrm{ap}} s}{(\textsf{Funcall}, \textsf{Sym}\ name :: args, a, fns) \Downarrow_{\mathrm{ap}} s}$$

Big-step op.sem summary

Evaluation is defined as an inductively defined relation:

$$(exp, a, fns) \Downarrow_{\mathrm{ev}} result$$

Big-step op.sem summary

Evaluation is defined as an inductively defined relation:

$$(exp, a, fns) \Downarrow_{\mathrm{ev}} result$$

- describes terminating evaluations
- relates expressions in one step to result value
- defined using 'inference-like' rules

Big-step op.sem summary

Evaluation is defined as an inductively defined relation:

$$(\mathit{exp}, a, \mathit{fns}) \Downarrow_{\mathrm{ev}} \mathit{result}$$

- describes terminating evaluations
- relates expressions in one step to result value
- defined using 'inference-like' rules

Criticism: what about non-terminating evaluations?
In later lectures:

- small-step semantics (models evaluation in steps)
- clocked big-step semantics

Summary

Formal specification & verification

- 'formal' i.e. some form of formal logic or calculus is used
- e.g. higher-order logic

Summary

Formal specification & verification

- ‘formal’ i.e. some form of formal logic or calculus is used
- e.g. higher-order logic

Language definitions:

- operational semantics (syntactic operations)
- denotational semantics (meaning of programs)
- axiomatic semantics (defines a programming logic)

Summary

Formal specification & verification

- ‘formal’ i.e. some form of formal logic or calculus is used
- e.g. higher-order logic

Language definitions:

- operational semantics (syntactic operations)
- denotational semantics (meaning of programs)
- axiomatic semantics (defines a programming logic)

Big-step operational semantics

- inductive relation describes evaluation
- big-step i.e. term-to-result evaluation is described by a single transition