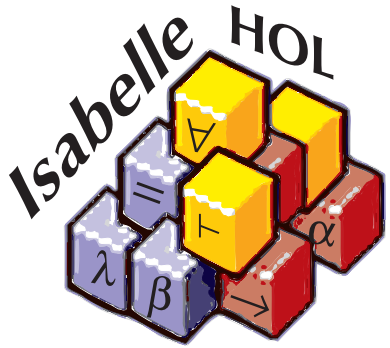
New trends in code generation: Haskabelle and the Predicate Compiler



Florian Haftmann
joint work with Stefan Berghofer and Lukas Bulwahn
Technische Universität München

Workshop in Belgrade – Jan. 2010

Part one: Haskabelle

Haskabelle

- pragmatic translator from Haskell programs to Isabelle theories

Haskabelle

- pragmatic translator from Haskell programs to Isabelle theories
- pragmatic: parsing Haskell to abstract syntax, printing Isabelle from abstract syntax
- tries to translate things one-to-one as far as possible

Haskabelle

- pragmatic translator from Haskell programs to Isabelle theories
- pragmatic: parsing Haskell to abstract syntax, printing Isabelle from abstract syntax
- tries to translate things one-to-one as far as possible
- restrictions mainly due to restrictions of Isabelle:
  - less expressive type system (e.g. no constructor classes)
  - no local function bindings
  - only provably terminating function definitions

Haskabelle

- pragmatic translator from Haskell programs to Isabelle theories
- pragmatic: parsing Haskell to abstract syntax, printing Isabelle from abstract syntax
- tries to translate things one-to-one as far as possible
- restrictions mainly due to restrictions of Isabelle:
  - less expressive type system (e.g. no constructor classes)
  - no local function bindings
  - only provably terminating function definitions
- simple example: printing radix representations ▶ EXAMPLE

Haskabelle

- pragmatic translator from Haskell programs to Isabelle theories
- pragmatic: parsing Haskell to abstract syntax, printing Isabelle from abstract syntax
- tries to translate things one-to-one as far as possible
- restrictions mainly due to restrictions of Isabelle:
  - less expressive type system (e.g. no constructor classes)
  - no local function bindings
  - only provably terminating function definitions
- simple example: printing radix representations ▶ EXAMPLE
- a realistic example: finite maps
  - taken from http://hackage.haskell.org/cgi-bin/hackage-scr package/FiniteMap-0.1
    ▶ EXAMPLE

Adaptation

Excerpt from `default/adapt.txt` in the Haskabelle distribution:

```
classes
  "Prelude.Eq"          "Prelude.eq"
types
  "Prelude.Bool"        "bool"
  "Prelude.PairTyCon"   "*"
  "Prelude.Maybe"       "option"
  "Prelude.String"      "string"
  "Prelude.Int"         "int"
  "Prelude.Integer"     "int"
consts
  "Prelude.True"        "True"
  "Prelude.False"       "False"
  "Prelude.(&&)"        "op &"
  "Prelude._|_"         "HOL.undefined"
```

Adaptation

Excerpt from `default/adapt.txt` in the Haskabelle distribution:

```
classes
  "Prelude.Eq"          "Prelude.eq"
types
  "Prelude.Bool"        "bool"
  "Prelude.PairTyCon"   "*"
  "Prelude.Maybe"       "option"
  "Prelude.String"      "string"
  "Prelude.Int"         "int"
  "Prelude.Integer"     "int"
consts
  "Prelude.True"        "True"
  "Prelude.False"       "False"
  "Prelude.(&&)"        "op &"
  "Prelude._|_"         "HOL.undefined"
```

Purpose:

- provide counterparts for fundamental Haskell definitions
- reuse existing Isabelle definitions (proofs!)

State of the art

What we have

- promising preliminary experiments with the seL4 operating system kernel http://ertos.nicta.com.au/research/l4.verified/
- used by secunet http://www.secunet.com/en/ "IT security beyond expectations"

What we encourage

- give it a try
- extend Haskabelle to fit your needs

Part two: Predicate Compiler

Executing Inductive Predicates in a functional language

Executing Inductive Predicates in a functional language

Inductive Predicates are everywhere

- Programming language semantics
- Type systems
- Logical calculi

Executing Inductive Predicates in a functional language

Inductive Predicates are everywhere

- Programming language semantics $\rightsquigarrow$ Interpreters
- Type systems $\rightsquigarrow$ Type checkers
- Logical calculi $\rightsquigarrow$ Inference mechanisms

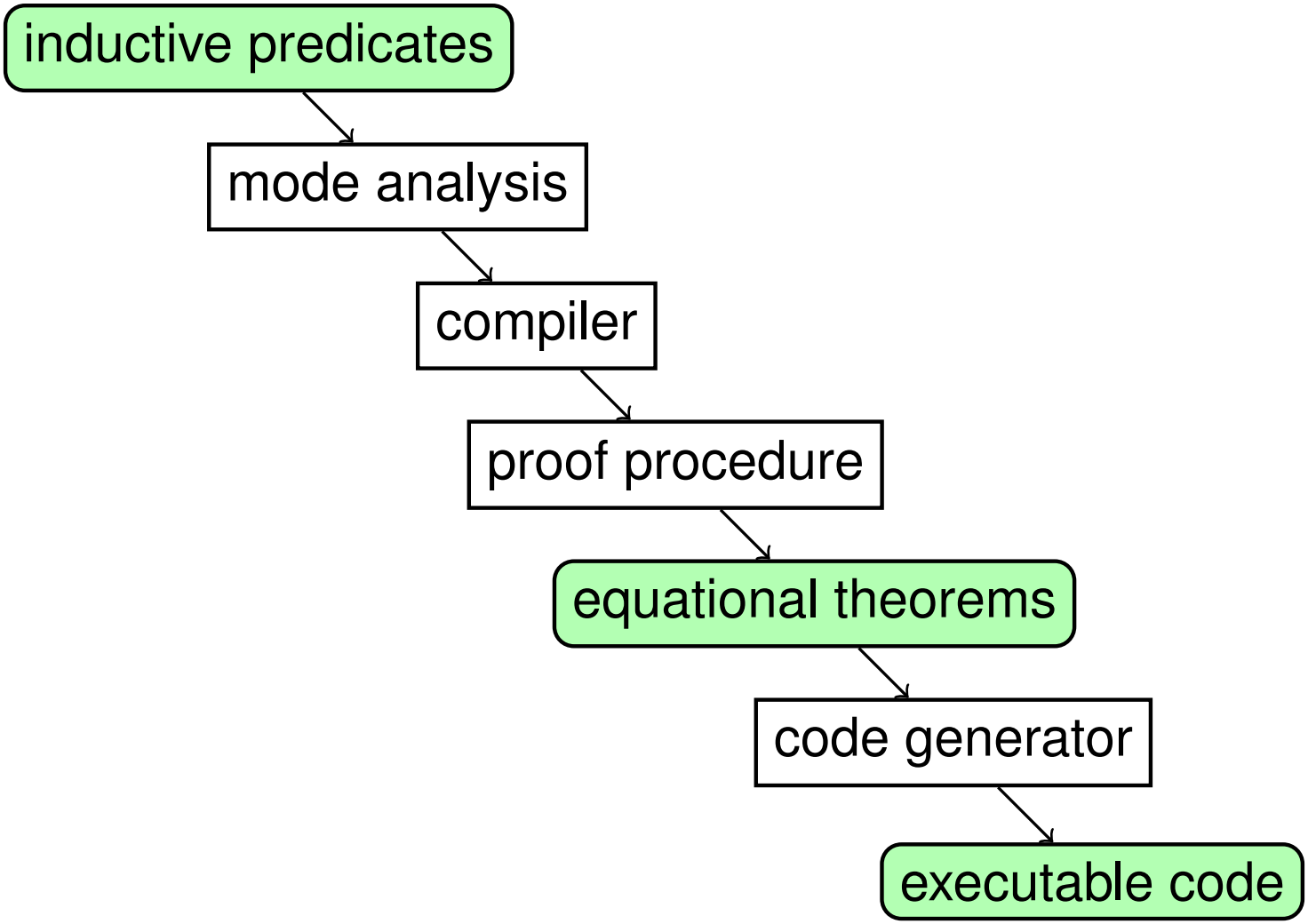
Executing Inductive Predicates in a functional language

Inductive Predicates are everywhere

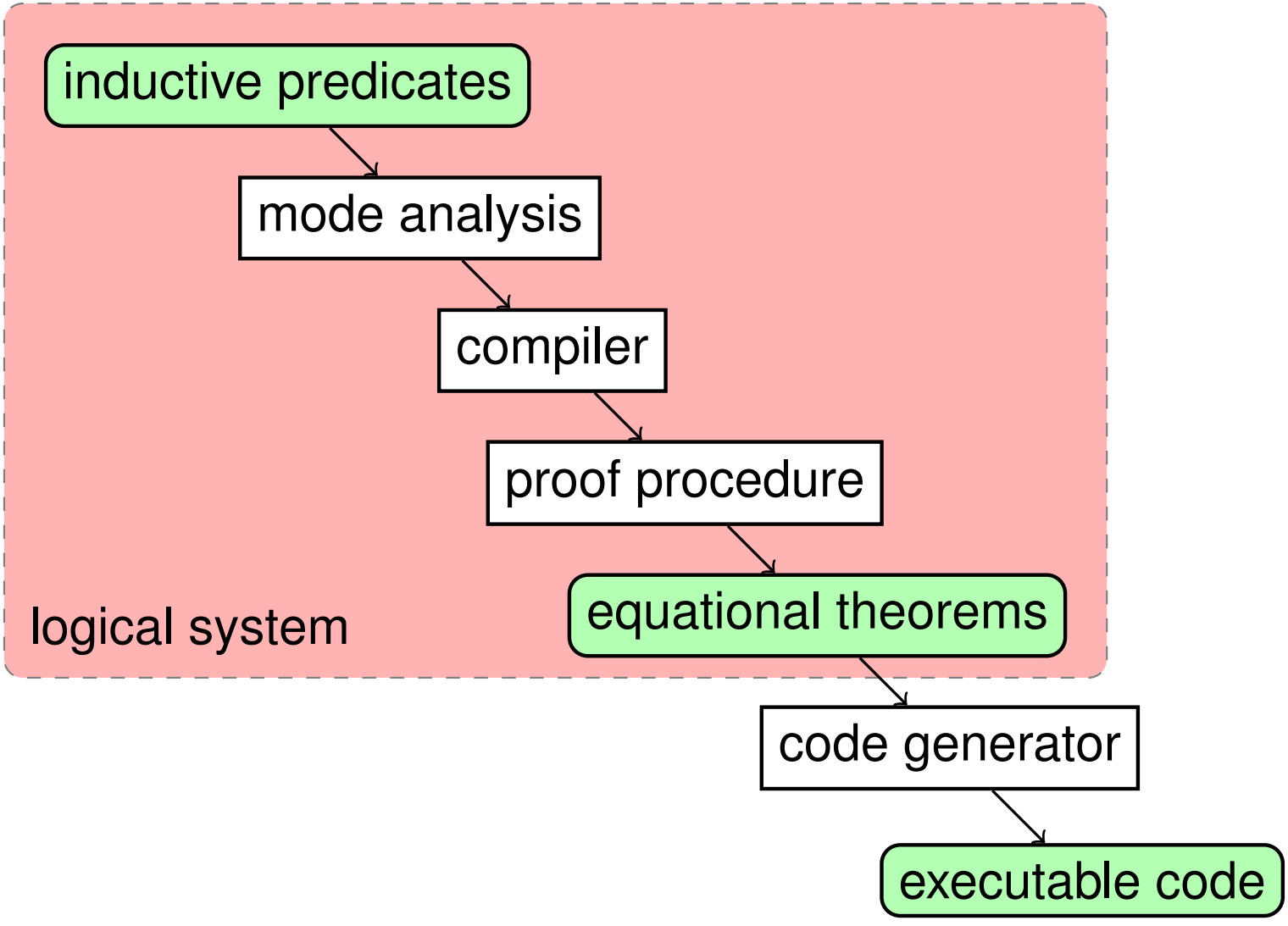
- Programming language semantics $\rightsquigarrow$ Interpreters
- Type systems $\rightsquigarrow$ Type checkers
- Logical calculi $\rightsquigarrow$ Inference mechanisms

Important for Prototypes and Counterexample generation

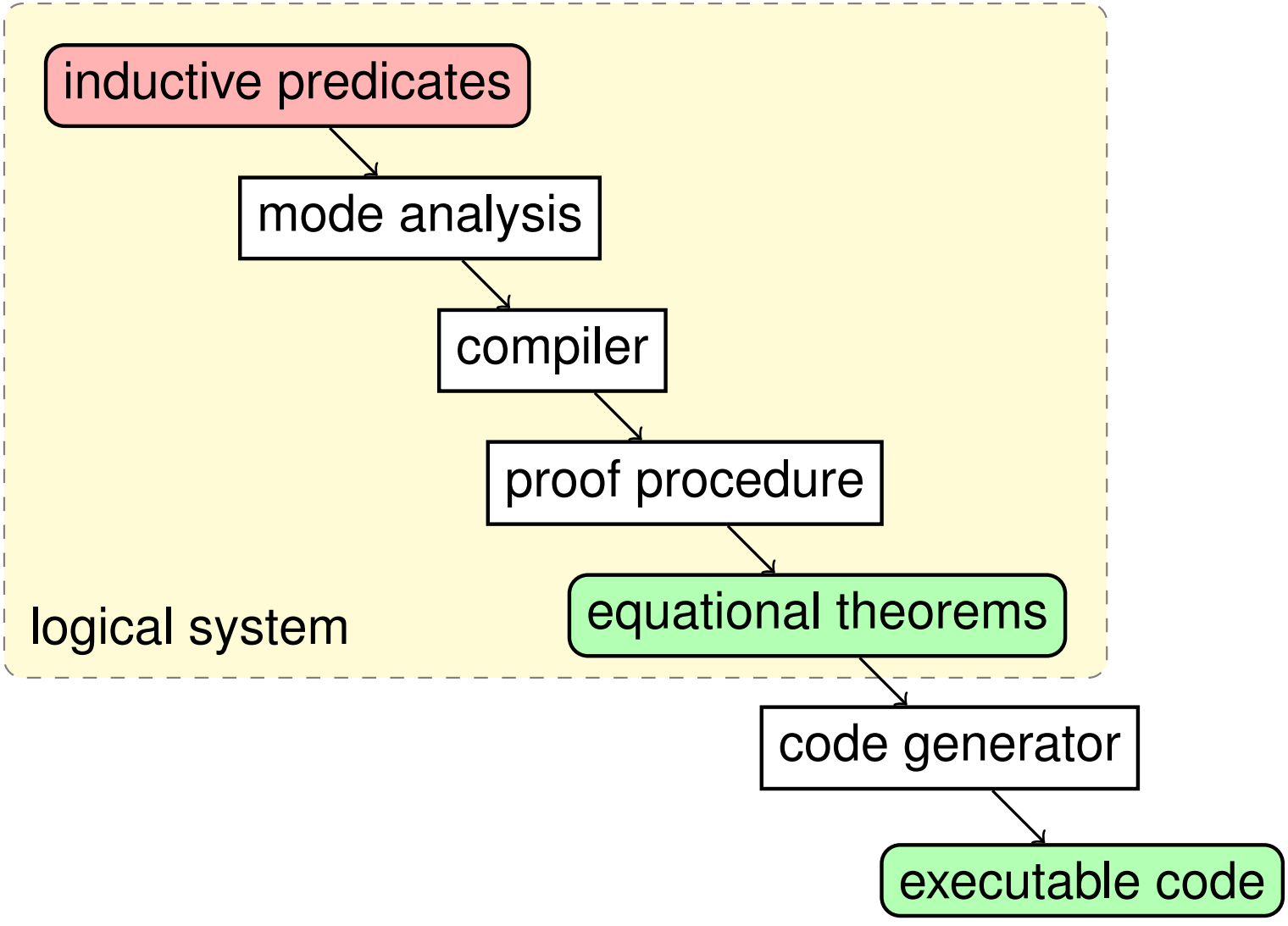
Overview


inductive predicates
mode analysis
compiler
proof procedure
equational theorems
code generator
executable code

Overview


inductive predicates
mode analysis
compiler
proof procedure
logical system
equational theorems
code generator
executable code

Overview


inductive predicates
mode analysis
compiler
proof procedure
logical system
equational theorems
code generator
executable code

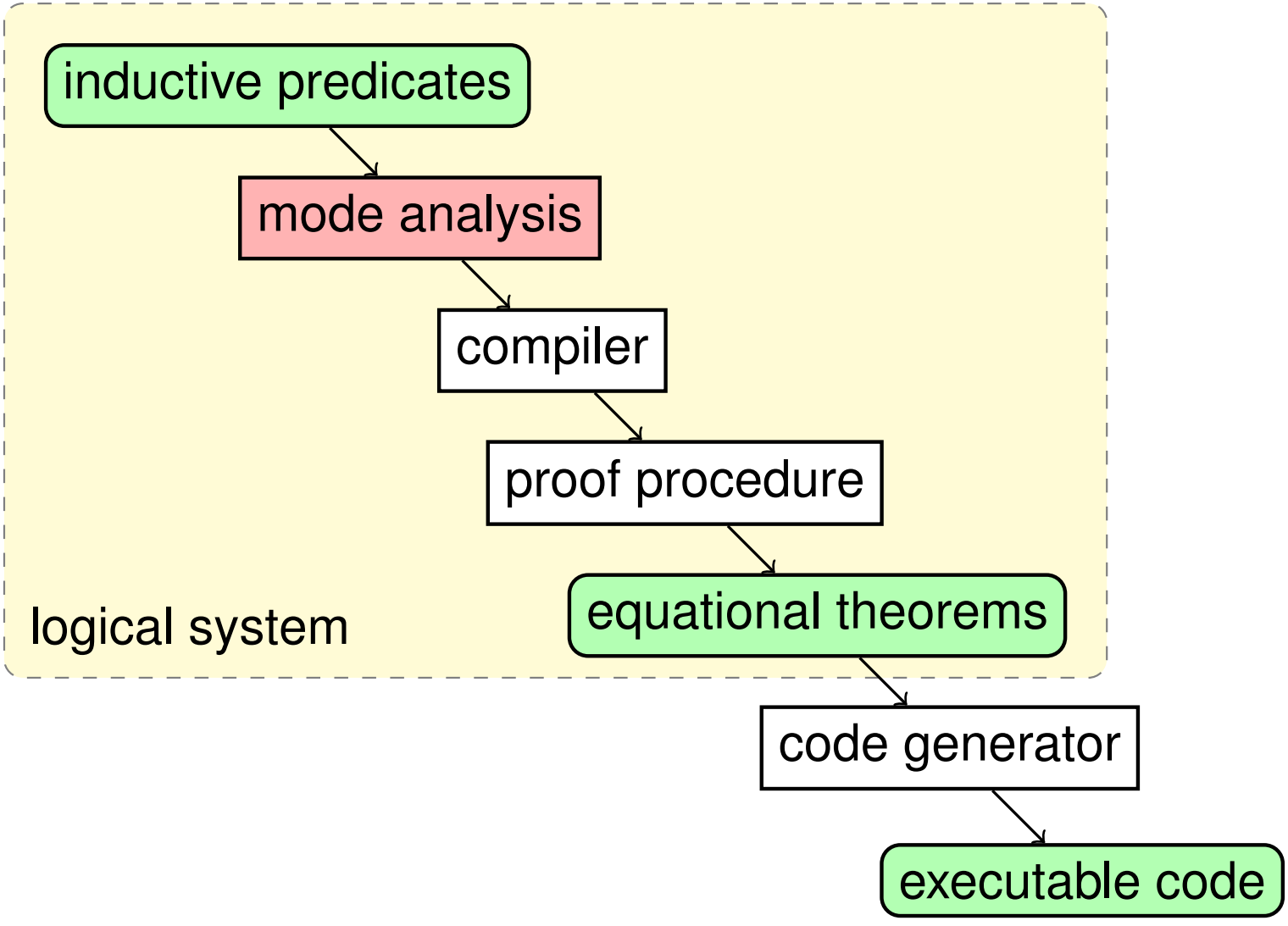
Inductive Predicates

Example: append

$$\frac{xs = [] \qquad ys = zs}{append\ xs\ ys\ zs}$$

$$\frac{xs = x \cdot xs' \qquad append\ xs'\ ys\ zs' \qquad x \cdot zs' = zs}{append\ xs\ ys\ zs}$$

Overview


inductive predicates
mode analysis
compiler
proof procedure
equational theorems
logical system
code generator
executable code

Modes

Inductive Predicates describe Relations.

Modes

Inductive Predicates describe Relations.

Inductive Predicates describe Functions which return Sets of solutions.

Modes

Inductive Predicates describe Relations.

Inductive Predicates describe Functions which return Sets of solutions.
A Mode describes a way of querying an Inductive Predicate.

Modes

Inductive Predicates describe Relations.

Inductive Predicates describe Functions which return Sets of solutions.
A Mode describes a way of querying an Inductive Predicate.

Example: *append*

- What is the concatenation of two given lists? (Mode {1, 2})

Modes

Inductive Predicates describe Relations.

Inductive Predicates describe Functions which return Sets of solutions.
A Mode describes a way of querying an Inductive Predicate.

Example: *append*

- What is the concatenation of two given lists? (Mode {1, 2})
  $append_{\{1,2\}}\ xs\ ys = \{zs.\ append\ xs\ ys\ zs\}$

Modes

Inductive Predicates describe Relations.

Inductive Predicates describe Functions which return Sets of solutions.
A Mode describes a way of querying an Inductive Predicate.

Example: *append*

- What is the concatenation of two given lists? (Mode {1, 2})
  $append_{\{1,2\}}\ xs\ ys = \{zs.\ append\ xs\ ys\ zs\}$
  $append_{\{1,2\}}\ [a, b]\ [c, d] = \{\ [a, b, c, d]\ \}$

Modes

Inductive Predicates describe Relations.

Inductive Predicates describe Functions which return Sets of solutions.
A Mode describes a way of querying an Inductive Predicate.

Example: *append*

- What is the concatenation of two given lists? (Mode {1, 2})
  $append_{\{1,2\}}\ xs\ ys = \{zs.\ append\ xs\ ys\ zs\}$
  $append_{\{1,2\}}\ [a, b]\ [c, d] = \{\ [a, b, c, d]\ \}$
- In which two lists can a given list be split? (Mode {3})

Modes

Inductive Predicates describe Relations.

Inductive Predicates describe Functions which return Sets of solutions.
A Mode describes a way of querying an Inductive Predicate.

Example: *append*

- What is the concatenation of two given lists? (Mode {1, 2})
  $append_{\{1,2\}}\ xs\ ys = \{zs.\ append\ xs\ ys\ zs\}$
  $append_{\{1,2\}}\ [a, b]\ [c, d] = \{\ [a, b, c, d]\ \}$
- In which two lists can a given list be split? (Mode {3})
  $append_{\{3\}}\ zs = \{(xs, ys).\ append\ xs\ ys\ zs\}$

Modes

Inductive Predicates describe Relations.

Inductive Predicates describe Functions which return Sets of solutions.
A Mode describes a way of querying an Inductive Predicate.

Example: *append*

- What is the concatenation of two given lists? (Mode {1, 2})
  $append_{\{1,2\}}\ xs\ ys = \{zs.\ append\ xs\ ys\ zs\}$
  $append_{\{1,2\}}\ [a, b]\ [c, d] = \{\ [a, b, c, d]\ \}$
- In which two lists can a given list be split? (Mode {3})
  $append_{\{3\}}\ zs = \{(xs, ys).\ append\ xs\ ys\ zs\}$
  $append_{\{3\}}\ [a, b, c, d] = \{([], [a, b, c, d]), ([a], [b, c, d]), \ldots$
  $\ldots, ([a, b, c, d], [])\}$

Modes

Inductive Predicates describe Relations.

Inductive Predicates describe Functions which return Sets of solutions.
A Mode describes a way of querying an Inductive Predicate.

Example: *append*

- What is the concatenation of two given lists? (Mode {1, 2})
  $append_{\{1,2\}}\ xs\ ys = \{zs.\ append\ xs\ ys\ zs\}$
  $append_{\{1,2\}}\ [a, b]\ [c, d] = \{\ [a, b, c, d]\ \}$
- In which two lists can a given list be split? (Mode {3})
  $append_{\{3\}}\ zs = \{(xs, ys).\ append\ xs\ ys\ zs\}$
  $append_{\{3\}}\ [a, b, c, d] = \{([], [a, b, c, d]), ([a], [b, c, d]), \ldots$
  $\ldots, ([a, b, c, d], [])\}$

Mode Analysis

A mode requires a dataflow of ground values in the introduction rules.

Example: *append* for mode {1, 2}

$$\frac{xs = [] \qquad ys = zs}{append\ xs\ ys\ zs}$$

$$\frac{xs = x \cdot xs' \qquad append\ xs'\ ys\ zs' \qquad x \cdot zs' = zs}{append\ xs\ ys\ zs}$$

Mode Analysis

A mode requires a dataflow of ground values in the introduction rules.

Example: *append* for mode {1, 2}

$$\frac{xs = [] \qquad ys = zs}{append\ xs\ ys\ zs}$$

$$\frac{xs = x \cdot xs' \qquad append\ xs'\ ys\ zs' \qquad x \cdot zs' = zs}{append\ xs\ ys\ zs}$$

Mode Analysis

A mode requires a dataflow of ground values in the introduction rules.

Example: *append* for mode {1, 2}

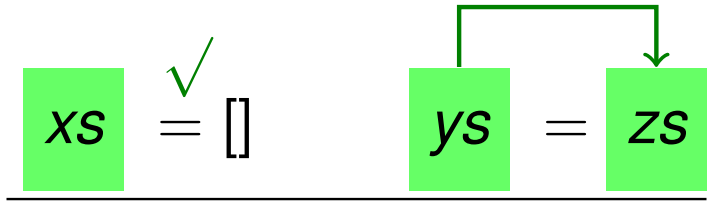
$$\frac{xs \overset{\surd}{=} [] \qquad ys = zs}{append\ xs\ ys\ zs}$$

$$\frac{xs = x \cdot xs' \qquad append\ xs'\ ys\ zs' \qquad x \cdot zs' = zs}{append\ xs\ ys\ zs}$$

Mode Analysis

A mode requires a dataflow of ground values in the introduction rules.

Example: *append* for mode {1, 2}

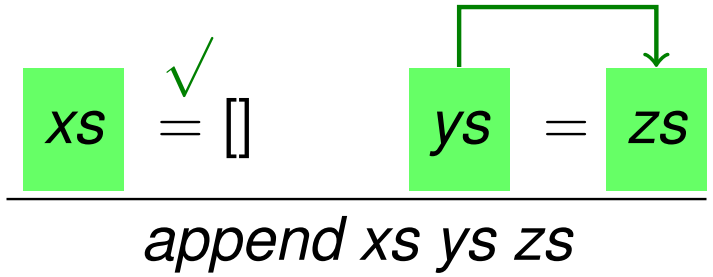


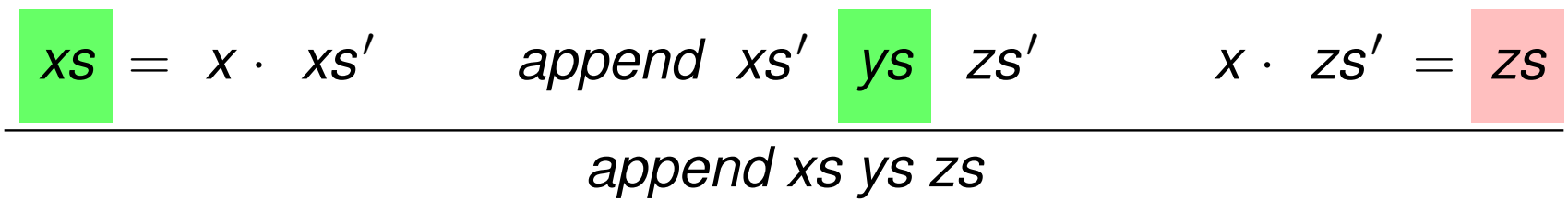
$$\frac{xs = x \cdot xs' \qquad append\ xs'\ ys\ zs' \qquad x \cdot zs' = zs}{append\ xs\ ys\ zs}$$

Mode Analysis

A mode requires a dataflow of ground values in the introduction rules.

Example: *append* for mode {1, 2}

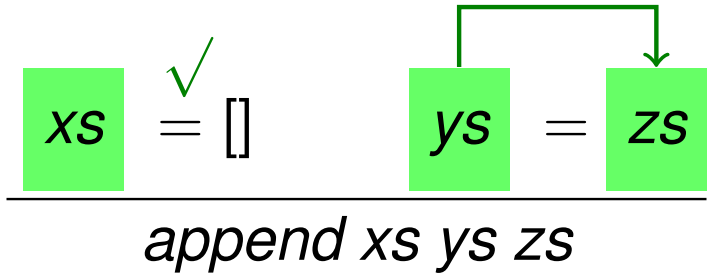


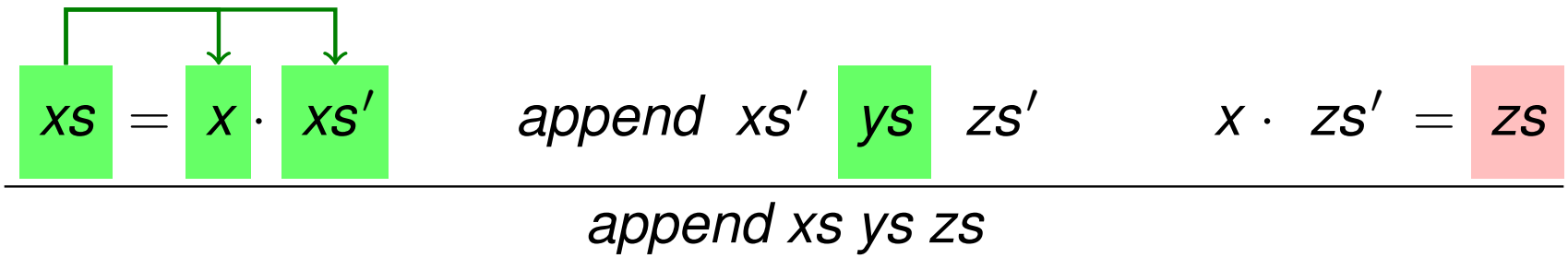


Mode Analysis

A mode requires a dataflow of ground values in the introduction rules.

Example: *append* for mode {1, 2}

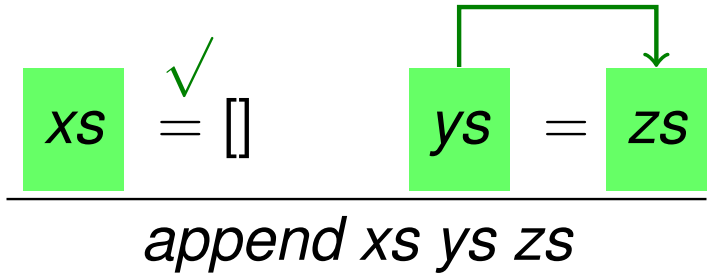


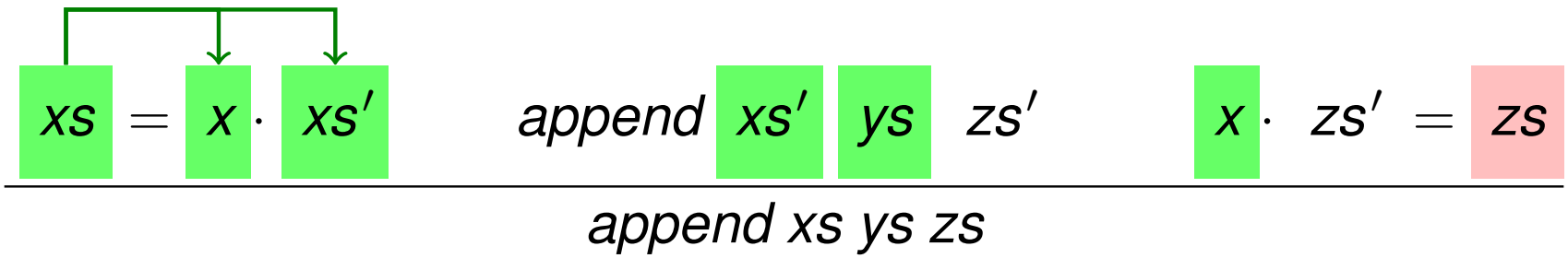


Mode Analysis

A mode requires a dataflow of ground values in the introduction rules.

Example: *append* for mode {1, 2}

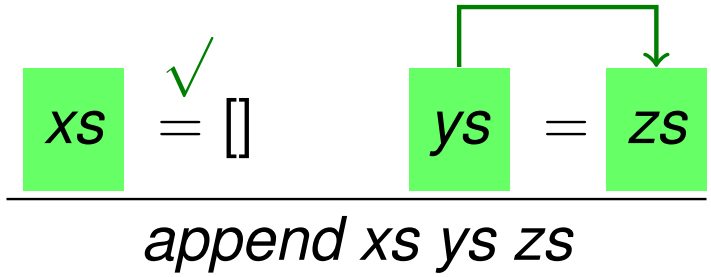


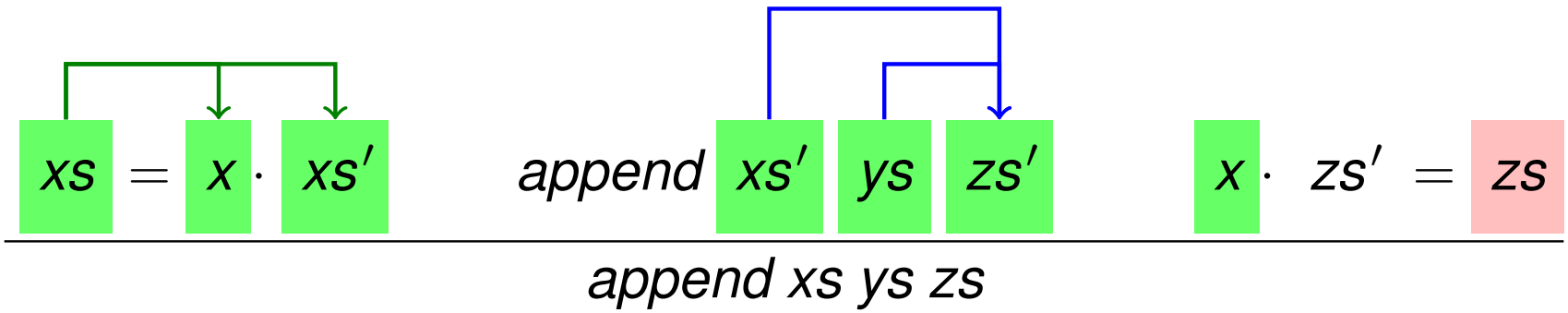


Mode Analysis

A mode requires a dataflow of ground values in the introduction rules.

Example: *append* for mode {1, 2}

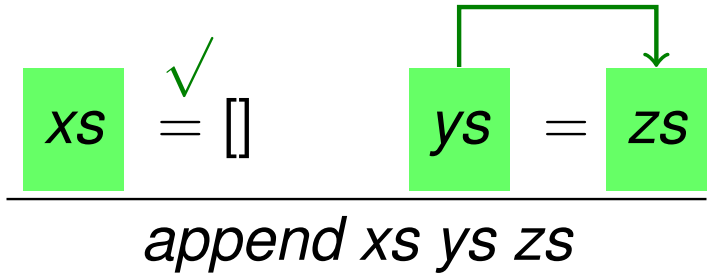


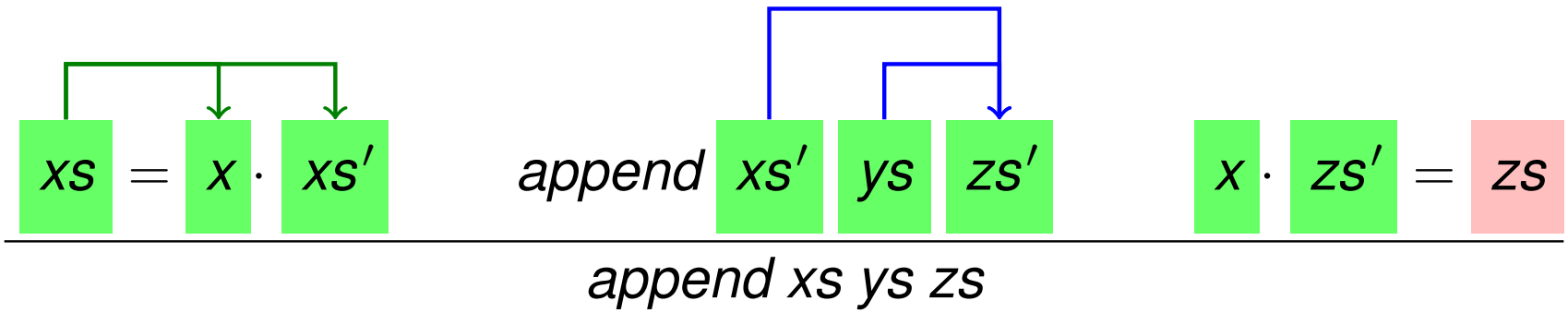


Mode Analysis

A mode requires a dataflow of ground values in the introduction rules.

Example: *append* for mode {1, 2}

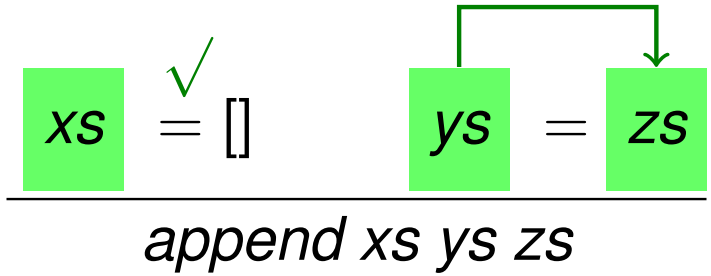


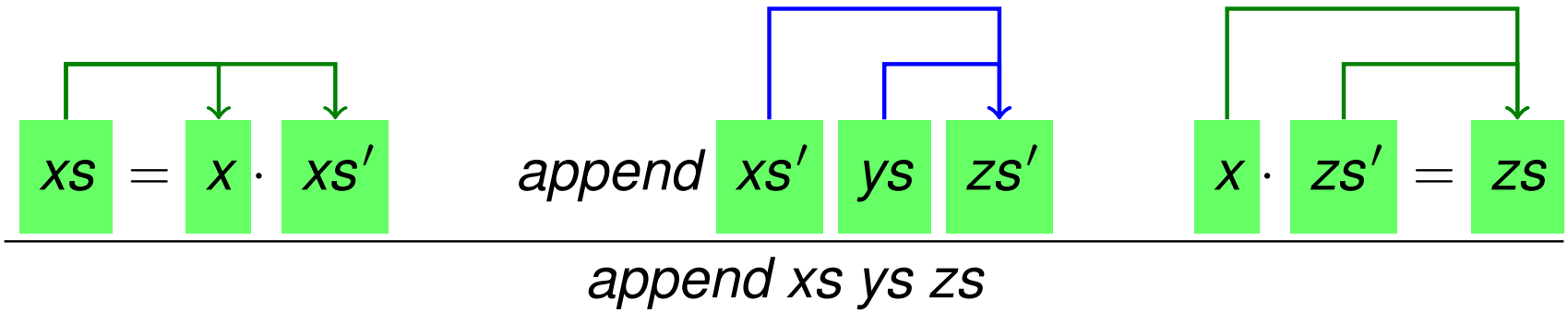


Mode Analysis

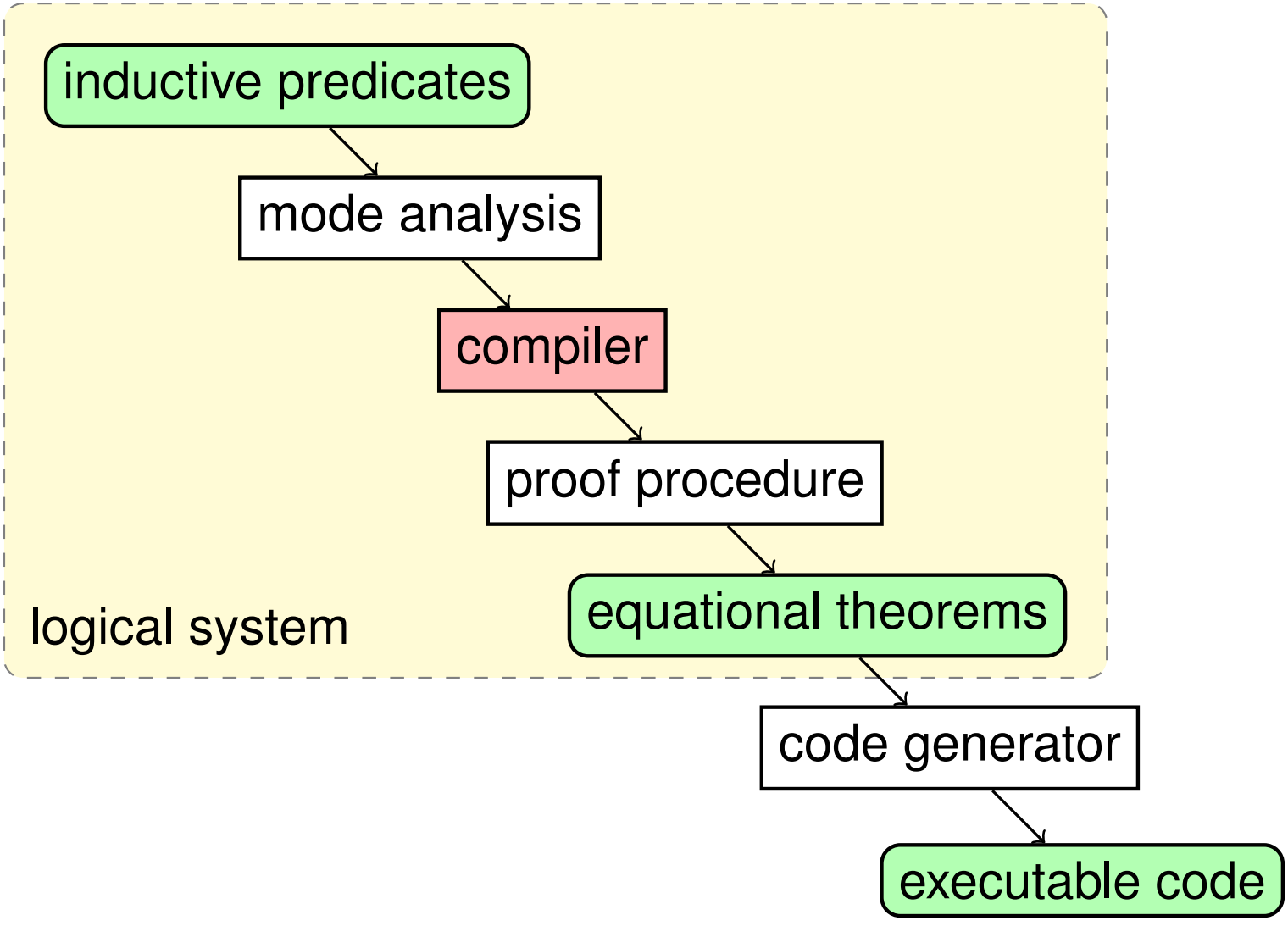
A mode requires a dataflow of ground values in the introduction rules.

Example: *append* for mode {1, 2}





Overview


inductive predicates
mode analysis
compiler
proof procedure
equational theorems
logical system
code generator
executable code

Compilation of code equations

Basic set operations

- empty set: $\emptyset$
- singleton set: $\{x\}$
- union operation: $A \cup B$
- bind operation: $S \gg\!= f$

Compilation of code equations

Basic set operations

- empty set: $\emptyset$
- singleton set: $\{x\}$
- union operation: $A \cup B$
- bind operation: $S \gg\!= f$
  $(\gg\!=) :: \alpha\ set \Rightarrow (\alpha \Rightarrow \beta\ set) \Rightarrow \beta\ set$

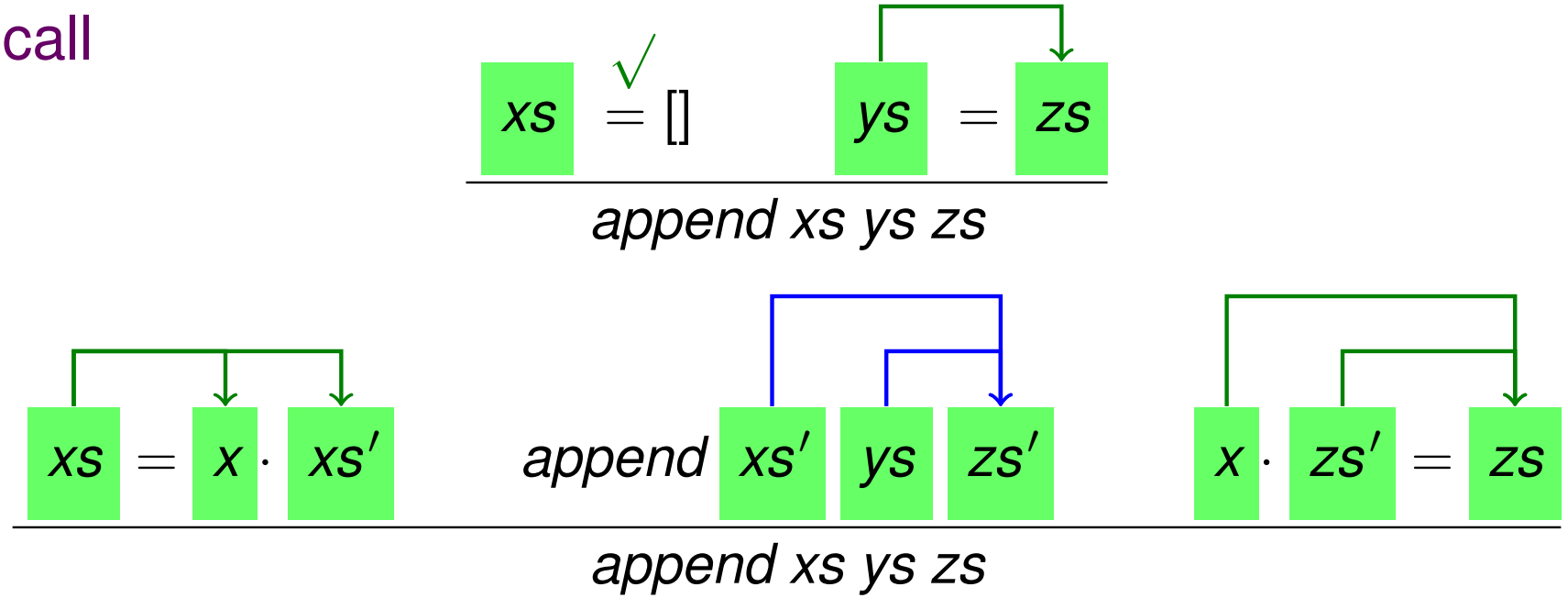
Compilation of code equations

Basic set operations

- empty set: $\emptyset$
- singleton set: $\{x\}$
- union operation: $A \cup B$
- bind operation: $S \gg\!= f$
  $(\gg\!=) :: \alpha\ set \Rightarrow (\alpha \Rightarrow \beta\ set) \Rightarrow \beta\ set$
  applying function $f :: \alpha \Rightarrow \beta\ set$
  to all elements of $S :: \alpha\ set$ and merging the results, i.e. $\bigcup f\ `\ S$

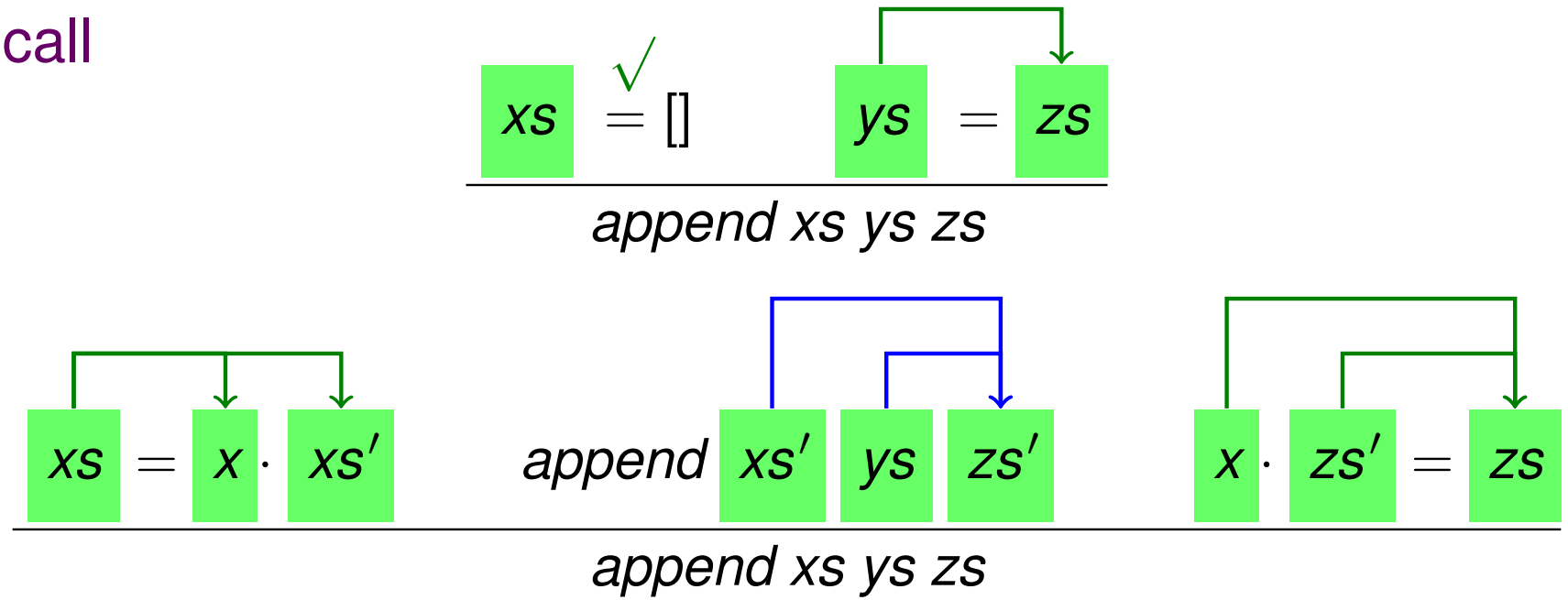
Compilation of code equations

Recall



Compilation of code equations

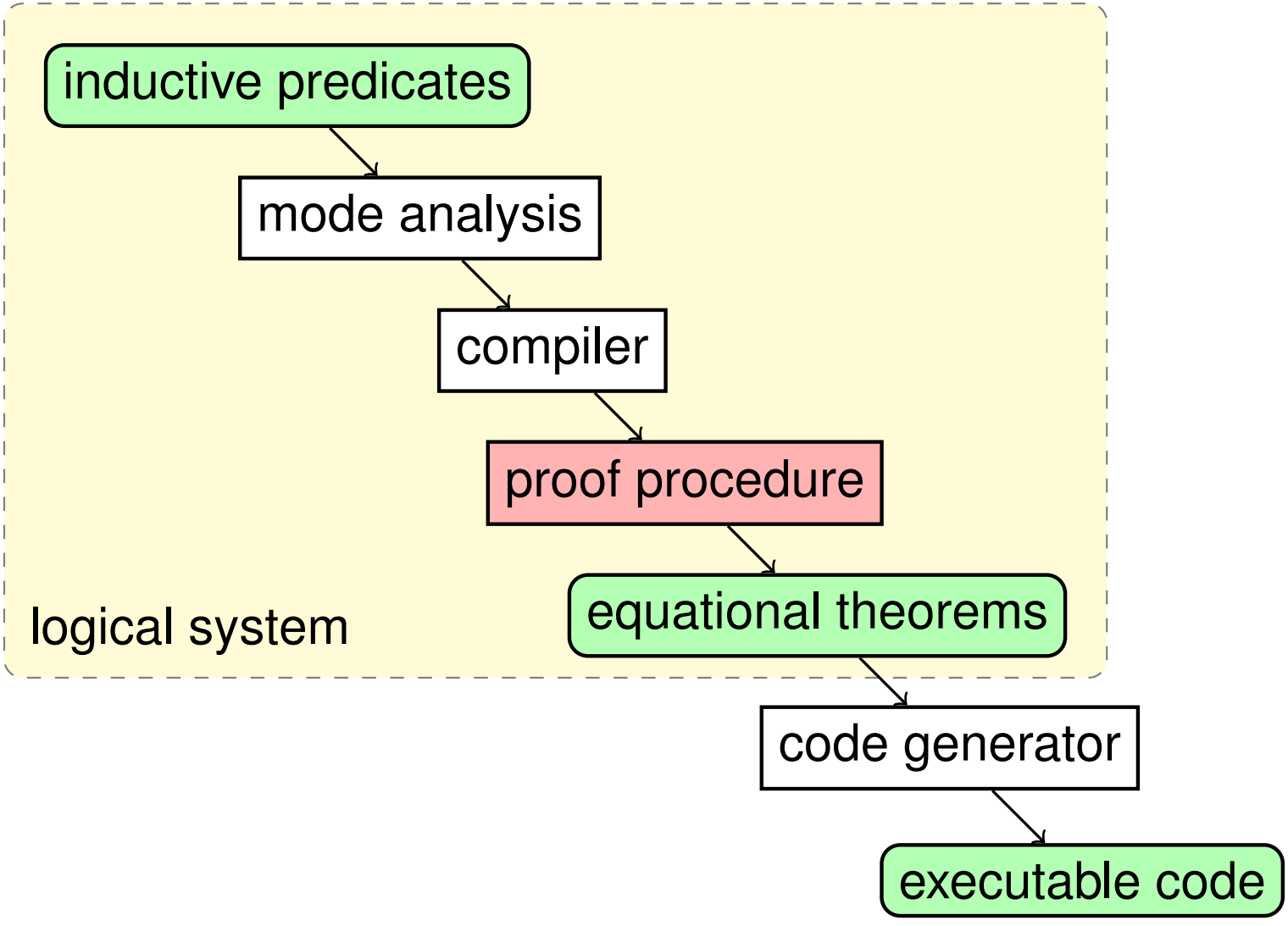
Recall



Code equation of *append*

$$append_{\{1,\,2\}}\ xs\ ys =$$
$$(case\ xs\ of\ [] \Rightarrow \{ys\} \mid x{\cdot}xs' \Rightarrow \emptyset)$$
$$\cup\ (case\ xs\ of\ [] \Rightarrow \emptyset \mid x{\cdot}xs' \Rightarrow append_{\{1,\,2\}}\ xs'\ ys \gg\!= (\lambda zs'.$$
$$\{x{\cdot}zs'\}))$$

Overview


inductive predicates
mode analysis
compiler
proof procedure
logical system
equational theorems
code generator
executable code

Correctness Proof of Construction

Definition:

$append_{\{1,\ 2\}}\ xs\ ys = \{zs.\ append\ xs\ ys\ zs\}$

Correctness Proof of Construction

Definition:

$append_{\{1,\,2\}}\ xs\ ys = \{zs.\ append\ xs\ ys\ zs\}$

Lemma:

$append_{\{1,\,2\}}\ xs\ ys =$

$(case\ xs\ of\ [] \Rightarrow \{ys\} \mid x{\cdot}xs' \Rightarrow \emptyset)$

$\cup\ (case\ xs\ of\ [] \Rightarrow \emptyset \mid x{\cdot}xs' \Rightarrow append_{\{1,\,2\}}\ xs'\ ys \gg\!= (\lambda zs'.$

$\{x{\cdot}zs'\}))$

Correctness Proof of Construction

Definition:

$append_{\{1,\,2\}}\ xs\ ys = \{zs.\ append\ xs\ ys\ zs\}$

Lemma:

$append_{\{1,\,2\}}\ xs\ ys =$
$(case\ xs\ of\ [] \Rightarrow \{ys\} \mid x{\cdot}xs' \Rightarrow \emptyset)$
$\cup\ (case\ xs\ of\ [] \Rightarrow \emptyset \mid x{\cdot}xs' \Rightarrow append_{\{1,\,2\}}\ xs'\ ys \gg\!= (\lambda zs'.$
$\{x{\cdot}zs'\}))$

Highlights of the proof procedure:

- Functions are correct by definition.

Correctness Proof of Construction

Definition:

$append_{\{1,\ 2\}}\ xs\ ys = \{zs.\ append\ xs\ ys\ zs\}$

Lemma:

$append_{\{1,\ 2\}}\ xs\ ys =$

$(case\ xs\ of\ [] \Rightarrow \{ys\} \mid x{\cdot}xs' \Rightarrow \emptyset)$

$\cup\ (case\ xs\ of\ [] \Rightarrow \emptyset \mid x{\cdot}xs' \Rightarrow append_{\{1,\ 2\}}\ xs'\ ys \gg\!= (\lambda zs'.$

$\{x{\cdot}zs'\}))$

Highlights of the proof procedure:

- Functions are correct by definition.
- No induction, only introduction and elimination rules of the inductive predicate.

Correctness Proof of Construction

Definition:

$append_{\{1,\ 2\}}\ xs\ ys = \{zs.\ append\ xs\ ys\ zs\}$

Lemma:

$append_{\{1,\ 2\}}\ xs\ ys =$

$(case\ xs\ of\ [] \Rightarrow \{ys\} \mid x{\cdot}xs' \Rightarrow \emptyset)$

$\cup\ (case\ xs\ of\ [] \Rightarrow \emptyset \mid x{\cdot}xs' \Rightarrow append_{\{1,\ 2\}}\ xs'\ ys \gg\!= (\lambda zs'.$

$\{x{\cdot}zs'\}))$

Highlights of the proof procedure:

- Functions are correct by definition.
- No induction, only introduction and elimination rules of the inductive predicate.
- No termination proof for recursive functions

Correctness Proof of Construction

Definition:

$append_{\{1,\ 2\}}\ xs\ ys = \{zs.\ append\ xs\ ys\ zs\}$

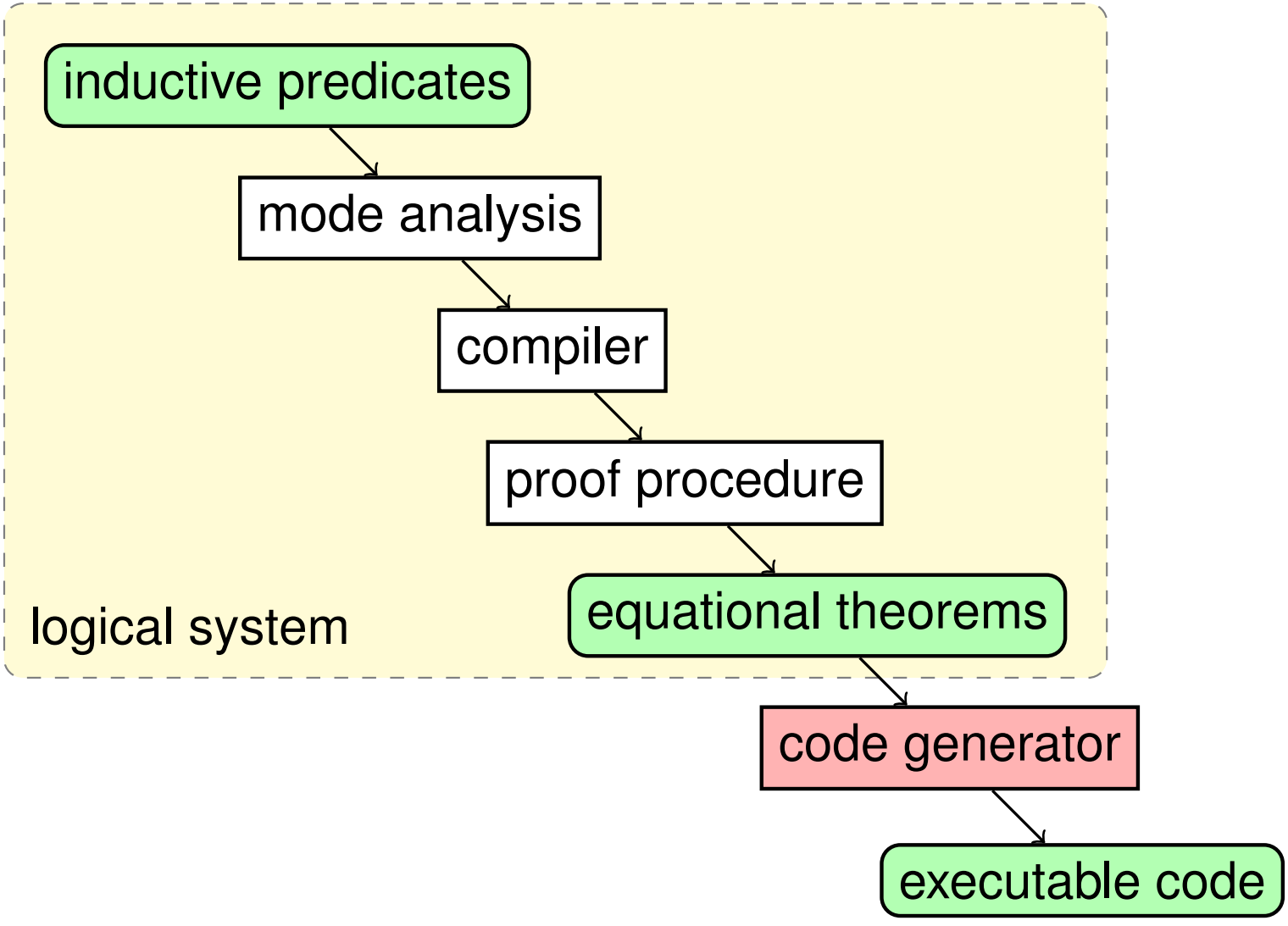
Lemma:

$append_{\{1,\ 2\}}\ xs\ ys =$
$(case\ xs\ of\ [] \Rightarrow \{ys\} \mid x{\cdot}xs' \Rightarrow \emptyset)$
$\cup\ (case\ xs\ of\ [] \Rightarrow \emptyset \mid x{\cdot}xs' \Rightarrow append_{\{1,\ 2\}}\ xs'\ ys \gg\!= (\lambda zs'.$
$\{x{\cdot}zs'\}))$

Highlights of the proof procedure:

- Functions are correct by definition.
- No induction, only introduction and elimination rules of the inductive predicate.
- No termination proof for recursive functions
- No interdependence of mutually recursive functions

Overview


inductive predicates
mode analysis
compiler
proof procedure
equational theorems
logical system
code generator
executable code

Lazy-Evaluation Model of Sets

We develop a model for sets with two properties:

Lazy-Evaluation Model of Sets

We develop a model for sets with two properties:

- executable, i.e. only uses executable functions (no choice-operator)

Lazy-Evaluation Model of Sets

We develop a model for sets with two properties:

- executable, i.e. only uses executable functions
  (no choice-operator)
- lazy evaluation in eager language, i.e. use explicit closures to delay computation

Lazy-Evaluation Model of Sets

Representation for Sets

datatype α *seq* $=$ *Empty* | *Insert* α (α *set*) | *Union* (α *set list*)

Lazy-Evaluation Model of Sets

Representation for Sets

datatype α *seq* $=$ *Empty* | *Insert* α (α *set*) | *Union* (α *set list*)

Seq :: (*unit* $\Rightarrow$ α *seq*) $\Rightarrow$ α *set*

Lazy-Evaluation Model of Sets

Representation for Sets

datatype α *seq* $=$ *Empty* | *Insert* α (α *set*) | *Union* (α *set list*)

Seq :: (*unit* $\Rightarrow$ α *seq*) $\Rightarrow$ α *set*
Seq f $=$ (*case f* () *of*
Empty $\Rightarrow$ $\emptyset$ | *Insert x xq* $\Rightarrow$ $\{x\} \cup$ *xq* | *Union xqs* $\Rightarrow$ $\bigcup$ *xqs*)

Lazy-Evaluation Model of Sets

Representation for Sets

datatype α *seq* $=$ *Empty* | *Insert* α (α *set*) | *Union* (α *set list*)

Seq :: (*unit* $\Rightarrow$ α *seq*) $\Rightarrow$ α *set*
Seq f $=$ (*case f* () *of*
Empty $\Rightarrow$ $\emptyset$ | *Insert x xq* $\Rightarrow$ $\{x\} \cup$ *xq* | *Union xqs* $\Rightarrow$ $\bigcup$ *xqs*)

Mapping to Standard ML

Empty :: α *seq*
Insert :: $\alpha \Rightarrow \alpha$ *set* $\Rightarrow \alpha$ *seq*
Union :: α *set list* $\Rightarrow \alpha$ *seq*
Seq :: (*unit* $\Rightarrow$ α *seq*) $\Rightarrow$ α *set*

Lazy-Evaluation Model of Sets

Representation for Sets

datatype α *seq* = *Empty* | *Insert* α (α *set*) | *Union* (α *set list*)

Seq :: (*unit* $\Rightarrow$ α *seq*) $\Rightarrow$ α *set*
Seq f = (*case f* () *of*
Empty $\Rightarrow$ $\emptyset$ | *Insert x xq* $\Rightarrow$ $\{x\} \cup$ *xq* | *Union xqs* $\Rightarrow$ $\bigcup$ *xqs*)

Mapping to Standard ML

Empty :: α *seq*
Insert :: $\alpha \Rightarrow \alpha$ *set* $\Rightarrow \alpha$ *seq*
Union :: α *set list* $\Rightarrow \alpha$ *seq*
Seq :: (*unit* $\Rightarrow \alpha$ *seq*) $\Rightarrow \alpha$ *set*

are chosen to be uninterpreted in SML:

```
datatype 'a seq = Empty | Insert of 'a * 'a set | Union of 'a set list
and 'a set = Seq of (unit -> 'a seq);
```

Executable Equations of Set Operations

We derive executable equations for the four set operations:

Executable Equations of Set Operations

We derive executable equations for the four set operations:

$$\emptyset = \mathit{Seq}\ (\lambda u.\ \mathit{Empty})$$

$$\{x\} = \mathit{Seq}\ (\lambda u.\ \mathit{Insert}\ x\ \emptyset)$$

Executable Equations of Set Operations

We derive executable equations for the four set operations:

$$\emptyset = \mathit{Seq}\ (\lambda u.\ \mathit{Empty})$$

$$\{x\} = \mathit{Seq}\ (\lambda u.\ \mathit{Insert}\ x\ \emptyset)$$

$$\begin{aligned}
&\mathit{Seq}\ g \gg\!= f = \\
&\quad \mathit{Seq}\ (\lambda u.\ \mathit{case}\ g\ ()\ \mathit{of}\ \mathit{Empty} \Rightarrow \mathit{Empty} \\
&\qquad\qquad \mid \mathit{Insert}\ x\ xq \Rightarrow \mathit{Union}\ [f\ x,\ xq \gg\!= f] \\
&\qquad\qquad \mid \mathit{Union}\ xqs \Rightarrow \mathit{Union}\ (\mathit{map}\ (\lambda x.\ x \gg\!= f)\ xqs))
\end{aligned}$$

Executable Equations of Set Operations

We derive executable equations for the four set operations:

$$\emptyset = \mathit{Seq}\ (\lambda u.\ \mathit{Empty})$$

$$\{x\} = \mathit{Seq}\ (\lambda u.\ \mathit{Insert}\ x\ \emptyset)$$

$$\begin{aligned}
&\mathit{Seq}\ g \gg\!= f = \\
&\quad \mathit{Seq}\ (\lambda u.\ \mathit{case}\ g\ ()\ \mathit{of}\ \mathit{Empty} \Rightarrow \mathit{Empty} \\
&\qquad\qquad \mid \mathit{Insert}\ x\ xq \Rightarrow \mathit{Union}\ [f\ x,\ xq \gg\!= f] \\
&\qquad\qquad \mid \mathit{Union}\ xqs \Rightarrow \mathit{Union}\ (\mathit{map}\ (\lambda x.\ x \gg\!= f)\ xqs))
\end{aligned}$$

$$\begin{aligned}
&\mathit{Seq}\ f \cup \mathit{Seq}\ g = \\
&\quad \mathit{Seq}\ (\lambda u.\ \mathit{case}\ f\ ()\ \mathit{of}\ \mathit{Empty} \Rightarrow g\ () \\
&\qquad\qquad \mid \mathit{Insert}\ x\ xq \Rightarrow \mathit{Insert}\ x\ (xq \cup \mathit{Seq}\ g) \\
&\qquad\qquad \mid \mathit{Union}\ xqs \Rightarrow \mathit{Union}\ (xqs\ @\ [\mathit{Seq}\ g]))
\end{aligned}$$

Executable Equations of Set Operations

We derive executable equations for the four set operations:

$$\emptyset = \mathit{Seq}\ (\lambda u.\ \mathit{Empty})$$

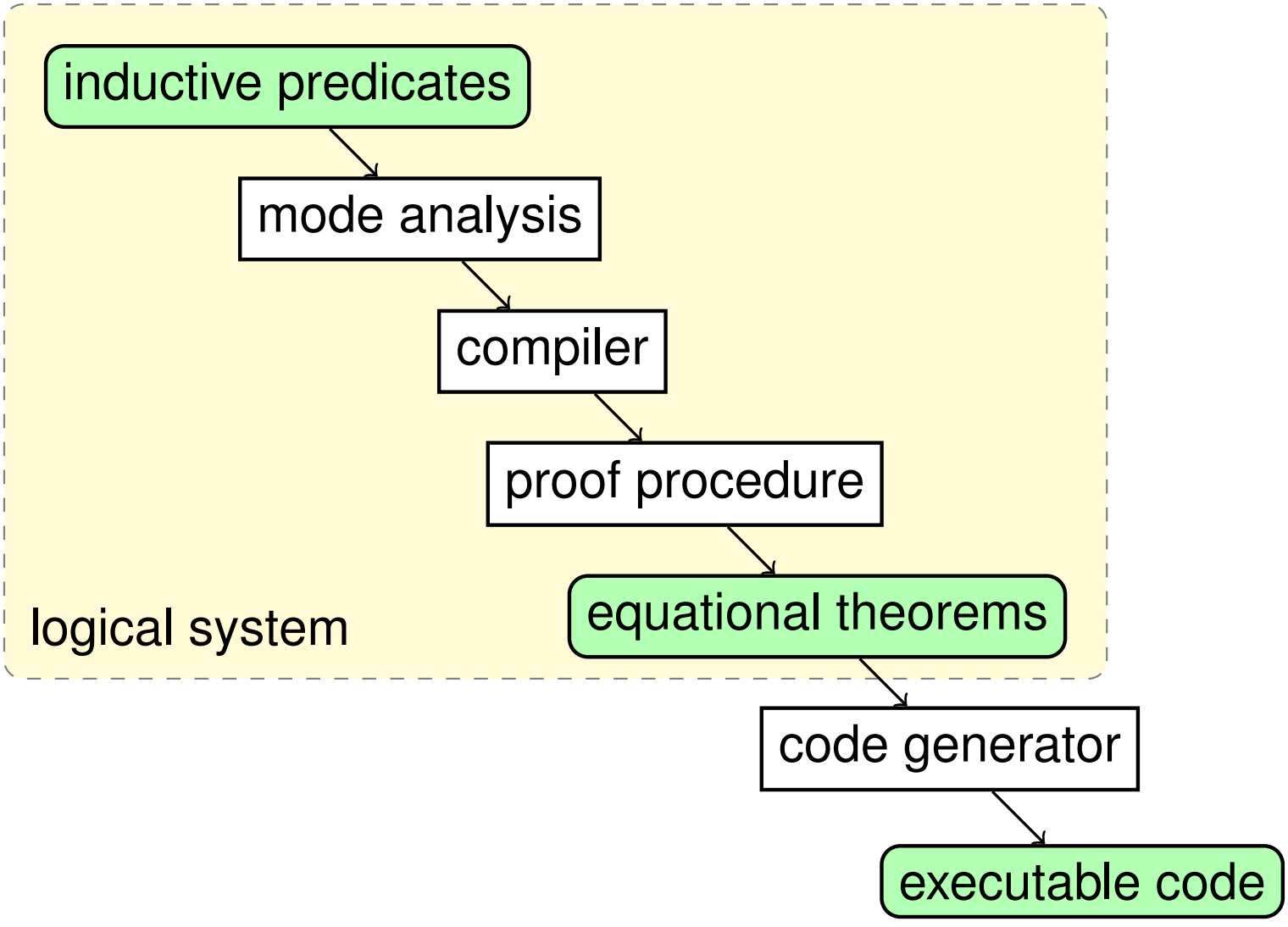
$$\{x\} = \mathit{Seq}\ (\lambda u.\ \mathit{Insert}\ x\ \emptyset)$$

$$\begin{aligned}
&\mathit{Seq}\ g \gg\!= f = \\
&\quad \mathit{Seq}\ (\lambda u.\ \mathit{case}\ g\ ()\ \mathit{of}\ \mathit{Empty} \Rightarrow \mathit{Empty} \\
&\qquad\qquad \mid \mathit{Insert}\ x\ xq \Rightarrow \mathit{Union}\ [f\ x,\ xq \gg\!= f] \\
&\qquad\qquad \mid \mathit{Union}\ xqs \Rightarrow \mathit{Union}\ (\mathit{map}\ (\lambda x.\ x \gg\!= f)\ xqs))
\end{aligned}$$

$$\begin{aligned}
&\mathit{Seq}\ f \cup \mathit{Seq}\ g = \\
&\quad \mathit{Seq}\ (\lambda u.\ \mathit{case}\ f\ ()\ \mathit{of}\ \mathit{Empty} \Rightarrow g\ () \\
&\qquad\qquad \mid \mathit{Insert}\ x\ xq \Rightarrow \mathit{Insert}\ x\ (xq \cup \mathit{Seq}\ g) \\
&\qquad\qquad \mid \mathit{Union}\ xqs \Rightarrow \mathit{Union}\ (xqs\ @\ [\mathit{Seq}\ g]))
\end{aligned}$$

Note: All equations are guarded by $\mathit{Seq}\ (\lambda u.\ \ldots)$!

Overview


inductive predicates
mode analysis
compiler
proof procedure
equational theorems
logical system
code generator
executable code