(this file is in presentation format; view it in Adobe Acrobat Reader DC, not in browser and not in Adobe Touch)

Formal Software Development

Adrian Zidaritz

zidaritz @ berkeley.edu

March 8, 2011

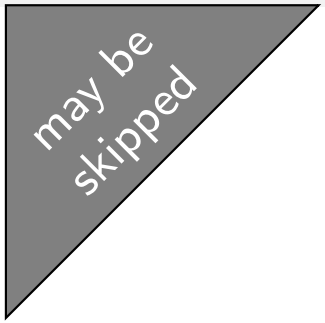
Introduction

This is an overview of the 'Formal Software Development' program. It is written for software professionals, and as such, it assumes some familiarity with symbolic manipulation.

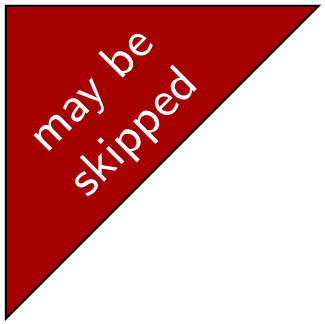
In this overview, we:

- Explain what a mathematical formal system is
- Show why formal systems are the foundation of software
- Give some concrete examples of formal software development
- Present the structure and the goals of the program

Introduction



Some slides are marked with this gray sticker; they contain some fairly elementary mathematical concepts, essentially *induction*. These slides may be skipped.



Other slides are marked with this red sticker; they assume knowledge of higher level algebra and treat *induction* at a more advanced level. These slides may also be skipped.

We include these optional slides because the program itself is a progression from basic concepts to more advanced ones, and we want to give all prospective students a bird's-eye view of the entire program and a roadmap to follow. Moreover, *induction is the central proof technique of the program*.

The use of mathematical logic

The use of mathematical logic

Formal software development is the use of *mathematical logic* to specify, design, build, analyze and test software.

What does it mean to use mathematical logic?

What does it mean to use mathematical logic?

It means that one has to

What does it mean to use mathematical logic?

It means that one has to

- study mathematical logic

What does it mean to use mathematical logic?

It means that one has to

- study mathematical logic
- study the tools that implement various logics

What does it mean to use mathematical logic?

It means that one has to

- study mathematical logic
- study the tools that implement various logics
- understand how to apply these tools to software engineering

How much mathematical logic?

How much mathematical logic?

- Mathematical logic is a large field in itself; it consists of proof theory, model theory, and recursive functions (=computability); set theory is regarded by many as belonging to logic too

How much mathematical logic?

- Mathematical logic is a large field in itself; it consists of proof theory, model theory, and recursive functions (=computability); set theory is regarded by many as belonging to logic too
- Logic is also the basis of many fields of computer science: type theory, specification languages, theory of computation, term rewriting, various program logics, automatic and interactive provers, etc ...

How much mathematical logic?

- Mathematical logic is a large field in itself; it consists of proof theory, model theory, and recursive functions (=computability); set theory is regarded by many as belonging to logic too
- Logic is also the basis of many fields of computer science: type theory, specification languages, theory of computation, term rewriting, various program logics, automatic and interactive provers, etc ...
- Formal software development needs results from all these subfields of mathematical logic and from many of its applications to computer science

How much mathematical logic?

- Mathematical logic is a large field in itself; it consists of proof theory, model theory, and recursive functions (=computability); set theory is regarded by many as belonging to logic too
- Logic is also the basis of many fields of computer science: type theory, specification languages, theory of computation, term rewriting, various program logics, automatic and interactive provers, etc ...
- Formal software development needs results from all these subfields of mathematical logic and from many of its applications to computer science
- So the answer to the title question would be: *an awful lot*

How much mathematical logic?

- Mathematical logic is a large field in itself; it consists of proof theory, model theory, and recursive functions (=computability); set theory is regarded by many as belonging to logic too
- Logic is also the basis of many fields of computer science: type theory, specification languages, theory of computation, term rewriting, various program logics, automatic and interactive provers, etc ...
- Formal software development needs results from all these subfields of mathematical logic and from many of its applications to computer science
- So the answer to the title question would be: *an awful lot*
- Part of the motivation behind this overview is to show you the program's road map through this large body of knowledge, a road that should lead to significant applications in software development

How much mathematical logic?

- Mathematical logic is a large field in itself; it consists of proof theory, model theory, and recursive functions (=computability); set theory is regarded by many as belonging to logic too
- Logic is also the basis of many fields of computer science: type theory, specification languages, theory of computation, term rewriting, various program logics, automatic and interactive provers, etc ...
- Formal software development needs results from all these subfields of mathematical logic and from many of its applications to computer science
- So the answer to the title question would be: *an awful lot*
- Part of the motivation behind this overview is to show you the program's road map through this large body of knowledge, a road that should lead to significant applications in software development
- *At the core of this use of logic are the formal systems*

What is a mathematical formal system?

What is a mathematical formal system?

- A (mathematical) formal system is a *language* and a set of *rules*

What is a mathematical formal system?

- A (mathematical) formal system is a *language* and a set of *rules*
- The word 'formal' is meant to embody the rigidity and the precision of language and rules

What is a mathematical formal system?

- A (mathematical) formal system is a *language* and a set of *rules*
- The word 'formal' is meant to embody the rigidity and the precision of language and rules
- . . . in opposition to 'informal' systems based on natural languages, which are flexible and ambiguous

What is a mathematical formal system?

- A (mathematical) formal system is a *language* and a set of *rules*
- The word 'formal' is meant to embody the rigidity and the precision of language and rules
- . . . in opposition to 'informal' systems based on natural languages, which are flexible and ambiguous
- *You may think of a formal system as a recipe, to be applied mechanically, without any creative thinking*

What is a mathematical formal system?

- A (mathematical) formal system is a *language* and a set of *rules*
- The word 'formal' is meant to embody the rigidity and the precision of language and rules
- . . . in opposition to 'informal' systems based on natural languages, which are flexible and ambiguous
- *You may think of a formal system as a recipe, to be applied mechanically, without any creative thinking*
- So by itself, a formal system is inert, it does not do anything

What is a mathematical formal system?

- A (mathematical) formal system is a *language* and a set of *rules*
- The word 'formal' is meant to embody the rigidity and the precision of language and rules
- . . . in opposition to 'informal' systems based on natural languages, which are flexible and ambiguous
- *You may think of a formal system as a recipe, to be applied mechanically, without any creative thinking*
- So by itself, a formal system is inert, it does not do anything
- We are not concerned for now with a *specific meaning* of such a system

What can we do with such a system?

What can we do with such a system?

- We can reason based on the rules

What can we do with such a system?

- We can reason based on the rules
- Reason means nothing but the blind application of the rules to sentences of the language in order to derive other sentences

What can we do with such a system?

- We can reason based on the rules
- Reason means nothing but the blind application of the rules to sentences of the language in order to derive other sentences
- We begin with a look at a simple formal system

What can we do with such a system?

- We can reason based on the rules
- Reason means nothing but the blind application of the rules to sentences of the language in order to derive other sentences
- We begin with a look at a simple formal system
- *The concepts introduced while looking at this simple system are the nuts and bolts of the program*

What can we do with such a system?

- We can reason based on the rules
- Reason means nothing but the blind application of the rules to sentences of the language in order to derive other sentences
- We begin with a look at a simple formal system
- *The concepts introduced while looking at this simple system are the nuts and bolts of the program*
- Try to understand this simple system as much as possible

What can we do with such a system?

- We can reason based on the rules
- Reason means nothing but the blind application of the rules to sentences of the language in order to derive other sentences
- We begin with a look at a simple formal system
- *The concepts introduced while looking at this simple system are the nuts and bolts of the program*
- Try to understand this simple system as much as possible
- Otherwise parts of the overview will sound gibberish

Example of a formal system

Example of a formal system

The language

Example of a formal system

The language

- The alphabet of our language consists of three symbols: a, b, and *.

Example of a formal system

The language

- The alphabet of our language consists of three symbols: a, b, and *.
- Symbol combinations are called *formulas*.

Example of a formal system

The language

- The alphabet of our language consists of three symbols: a, b, and *.
- Symbol combinations are called *formulas*.
- So ab, baa*, **a*bbb are examples of formulas.

Example of a formal system

The language

- The alphabet of our language consists of three symbols: a, b, and *.
- Symbol combinations are called *formulas*.
- So ab, baa*, **a*bbb are examples of formulas.
- A language has a grammar: a way to specify which formulas are acceptable; the technical term is well-formed formulas, *wffs* in short.

Example of a formal system

The language

- The alphabet of our language consists of three symbols: a, b, and *.
- Symbol combinations are called *formulas*.
- So ab, baa*, **a*bbb are examples of formulas.
- A language has a grammar: a way to specify which formulas are acceptable; the technical term is well-formed formulas, *wffs* in short.
- The rules of the grammar are known as formation rules, as opposed to the rules of system, which are known as *rules of inference*.

The grammar

The grammar

The grammar (or, the formation rules):

The grammar

The grammar (or, the formation rules):

1. * is a wff

The grammar

The grammar (or, the formation rules):

1. * is a wff
2. if X and Y are formulas containing no *, then X*Y is a wff

The grammar

The grammar (or, the formation rules):

1. * is a wff
2. if X and Y are formulas containing no *, then X*Y is a wff
3. no other formulas are wffs

The grammar

The grammar (or, the formation rules):

1. * is a wff
2. if X and Y are formulas containing no *, then X*Y is a wff
3. no other formulas are wffs

The grammar

The grammar (or, the formation rules):

1. * is a wff
2. if X and Y are formulas containing no *, then X*Y is a wff
3. no other formulas are wffs

The symbols X and Y do not belong to the language, they are just placeholders for arbitrary formulas. We'll talk about them later.

The grammar

The grammar (or, the formation rules):

1. * is a wff
2. if X and Y are formulas containing no *, then X*Y is a wff
3. no other formulas are wffs

The symbols X and Y do not belong to the language, they are just placeholders for arbitrary formulas. We'll talk about them later.

- For example, if X stands for a and Y for b, since neither X nor Y contain a *, then by the second formation rule, a*b is a wff

The grammar

The grammar (or, the formation rules):

1. * is a wff
2. if X and Y are formulas containing no *, then X*Y is a wff
3. no other formulas are wffs

The symbols X and Y do not belong to the language, they are just placeholders for arbitrary formulas. We'll talk about them later.

- For example, if X stands for a and Y for b, since neither X nor Y contain a *, then by the second formation rule, a*b is a wff
- Similarly, aa*bb, aba*bbbbb, baba*abab are wffs (think of what X and Y are in these cases)

The grammar

The grammar (or, the formation rules):

1. * is a wff
2. if X and Y are formulas containing no *, then X*Y is a wff
3. no other formulas are wffs

The symbols X and Y do not belong to the language, they are just placeholders for arbitrary formulas. We'll talk about them later.

- For example, if X stands for a and Y for b, since neither X nor Y contain a *, then by the second formation rule, a*b is a wff
- Similarly, aa*bb, aba*bbbbb, baba*abab are wffs (think of what X and Y are in these cases)
- Whereas **, *aba*, a*b*b are not wffs, because we cannot find any X and Y to fit the grammar requirements

Rules of inference, in general

Rules of inference, in general

The rules of inference dictate how a new wff, called a *conclusion*, can be deduced (or inferred) from a set (possibly empty) of other wffs, called *premises*.

Rules of inference, in general

The rules of inference dictate how a new wff, called a *conclusion*, can be deduced (or inferred) from a set (possibly empty) of other wffs, called *premises*.

- A rule with an empty set of premises is called an *axiom*.

Rules of inference, in general

The rules of inference dictate how a new wff, called a *conclusion*, can be deduced (or inferred) from a set (possibly empty) of other wffs, called *premises*.

- A rule with an empty set of premises is called an *axiom*.
- It is helpful to make this distinction because axioms are what get the system up and going, i.e. you have to start the deduction process somewhere.

Rules of inference, in general

The rules of inference dictate how a new wff, called a *conclusion*, can be deduced (or inferred) from a set (possibly empty) of other wffs, called *premises*.

- A rule with an empty set of premises is called an *axiom*.
- It is helpful to make this distinction because axioms are what get the system up and going, i.e. you have to start the deduction process somewhere.
- Otherwise, axioms are nothing but special rules.

Rules of inference, in general

The rules of inference dictate how a new wff, called a *conclusion*, can be deduced (or inferred) from a set (possibly empty) of other wffs, called *premises*.

- A rule with an empty set of premises is called an *axiom*.
- It is helpful to make this distinction because axioms are what get the system up and going, i.e. you have to start the deduction process somewhere.
- Otherwise, axioms are nothing but special rules.

Rules of inference, in general

The rules of inference dictate how a new wff, called a *conclusion*, can be deduced (or inferred) from a set (possibly empty) of other wffs, called *premises*.

- A rule with an empty set of premises is called an *axiom*.
- It is helpful to make this distinction because axioms are what get the system up and going, i.e. you have to start the deduction process somewhere.
- Otherwise, axioms are nothing but special rules.

A finite sequence of deductions is called a *proof*. If a proof uses no premises other than axioms, then every wff in the sequence of the proof is a *theorem*. In particular, axioms are theorems, their proofs being sequences of length 1. Although the rules can be used to build proofs from any premises, we will not be using this facility in these slides. For us, proofs begin with axioms and dish out theorems.

The rules of our system

The rules of our system

Our system has the following rules:

The rules of our system

Our system has the following rules:

- (r1) we can always deduce *

The rules of our system

Our system has the following rules:

- (r1) we can always deduce *
- (r2) from X we can deduce Xb

The rules of our system

Our system has the following rules:

- (r1) we can always deduce *
- (r2) from X we can deduce Xb
- (r3) from X*Yb we can deduce aX*Yb

The rules of our system

Our system has the following rules:

- (r1) we can always deduce *
- (r2) from X we can deduce Xb
- (r3) from X*Yb we can deduce aX*Yb
- (r4) from Xa*bY we can deduce X*Y

The rules of our system

Our system has the following rules:

- (r1) we can always deduce *
- (r2) from X we can deduce Xb
- (r3) from X*Yb we can deduce aX*Yb
- (r4) from Xa*bY we can deduce X*Y

The rules of our system

Our system has the following rules:

- (r1) we can always deduce *
- (r2) from X we can deduce Xb
- (r3) from X*Yb we can deduce aX*Yb
- (r4) from Xa*bY we can deduce X*Y

Since it has no premises, rule 1 is an *axiom*. *You can apply the rules in any order you wish*, you do not have to go 2-3-4. This feature of formal systems is called *nondeterminism*; we'll review nondeterminism later, and a few proofs that we build will also emphasize it.

The rules of our system

Our system has the following rules:

- (r1) we can always deduce *
- (r2) from X we can deduce Xb
- (r3) from X*Yb we can deduce aX*Yb
- (r4) from Xa*bY we can deduce X*Y

Since it has no premises, rule 1 is an *axiom*. *You can apply the rules in any order you wish*, you do not have to go 2-3-4. This feature of formal systems is called *nondeterminism*; we'll review nondeterminism later, and a few proofs that we build will also emphasize it. We'll call our example formal system DANS.

Proving theorems with our formal system

Proving theorems with our formal system

Let's look at some proofs in DANS; we'll number the steps of a proof as #1, #2, and so on. Each step will contain the theorem proved at that step and, in parentheses, how we proved it (which must be a rule applied to previously proved steps).

Proving theorems with our formal system

Let's look at some proofs in DANS; we'll number the steps of a proof as #1, #2, and so on. Each step will contain the theorem proved at that step and, in parentheses, how we proved it (which must be a rule applied to previously proved steps).

Theorem: a*bb

Proving theorems with our formal system

Let's look at some proofs in DANS; we'll number the steps of a proof as #1, #2, and so on. Each step will contain the theorem proved at that step and, in parentheses, how we proved it (which must be a rule applied to previously proved steps).

Theorem: a*bb

Proof.

- #1: * (r1)

Proving theorems with our formal system

Let's look at some proofs in DANS; we'll number the steps of a proof as #1, #2, and so on. Each step will contain the theorem proved at that step and, in parentheses, how we proved it (which must be a rule applied to previously proved steps).

Theorem: a*bb

Proof.

- #1: * (r1)
- #2: *b (#1,r2)

Proving theorems with our formal system

Let's look at some proofs in DANS; we'll number the steps of a proof as #1, #2, and so on. Each step will contain the theorem proved at that step and, in parentheses, how we proved it (which must be a rule applied to previously proved steps).

Theorem: a*bb

Proof.

- #1: * (r1)
- #2: *b (#1,r2)
- #3: a*b (#2,r3)

Proving theorems with our formal system

Let's look at some proofs in DANS; we'll number the steps of a proof as #1, #2, and so on. Each step will contain the theorem proved at that step and, in parentheses, how we proved it (which must be a rule applied to previously proved steps).

Theorem: a*bb

Proof.

- #1: * (r1)
- #2: *b (#1,r2)
- #3: a*b (#2,r3)
- #4: a*bb (#2,r2)

□

Proof objects

Proof objects

This form of a proof is known as a *proof object*.

Proof objects

This form of a proof is known as a *proof object*. So

{#1: * (r1); #2: *b (#1,r2); #3: a*b (#2,r3); #4: a*bb (#2,r2)}

is a proof object.

Proof objects

This form of a proof is known as a *proof object*. So

{#1: * (r1); #2: *b (#1,r2); #3: a*b (#2,r3); #4: a*bb (#2,r2)}

is a proof object. Proof objects are important because anyone who is given the language and rules of DANS could verify the steps of the proof.

Proof objects

This form of a proof is known as a *proof object*. So

{#1: * (r1); #2: *b (#1,r2); #3: a*b (#2,r3); #4: a*bb (#2,r2)}

is a proof object. Proof objects are important because anyone who is given the language and rules of DANS could verify the steps of the proof. Let's remark that a theorem can have many proofs; we could have proved a*bb with this proof object:

{#1: * (r1); #2: *b (#1,r2);#3: *bb (#2,r2); #4: a*bb (#2,r3)}

Proof objects

This form of a proof is known as a *proof object*. So

{#1: * (r1); #2: *b (#1,r2); #3: a*b (#2,r3); #4: a*bb (#2,r2)}

is a proof object. Proof objects are important because anyone who is given the language and rules of DANS could verify the steps of the proof. Let's remark that a theorem can have many proofs; we could have proved a*bb with this proof object:

{#1: * (r1); #2: *b (#1,r2);#3: *bb (#2,r2); #4: a*bb (#2,r3)}

A more complete description of a proof would include not just the proof object, but the entire formal system attached. This is sometimes called a proof certificate.

Proof objects

This form of a proof is known as a *proof object*. So

{#1: * (r1); #2: *b (#1,r2); #3: a*b (#2,r3); #4: a*bb (#2,r2)}

is a proof object. Proof objects are important because anyone who is given the language and rules of DANS could verify the steps of the proof. Let's remark that a theorem can have many proofs; we could have proved a*bb with this proof object:

{#1: * (r1); #2: *b (#1,r2);#3: *bb (#2,r2); #4: a*bb (#2,r3)}

A more complete description of a proof would include not just the proof object, but the entire formal system attached. This is sometimes called a proof certificate. So {[DANS] #1: * (r1); #2: *b (#1,r2); #3: a*b (#2,r3); #4: a*bb (#2,r2)} would be a proof certificate. If you emailed me this proof certificate, I would know exactly what you proved.

Proof objects

This form of a proof is known as a *proof object*. So

{#1: * (r1); #2: *b (#1,r2); #3: a*b (#2,r3); #4: a*bb (#2,r2)}

is a proof object. Proof objects are important because anyone who is given the language and rules of DANS could verify the steps of the proof. Let's remark that a theorem can have many proofs; we could have proved a*bb with this proof object:

{#1: * (r1); #2: *b (#1,r2);#3: *bb (#2,r2); #4: a*bb (#2,r3)}

A more complete description of a proof would include not just the proof object, but the entire formal system attached. This is sometimes called a proof certificate. So {[DANS] #1: * (r1); #2: *b (#1,r2); #3: a*b (#2,r3); #4: a*bb (#2,r2)} would be a proof certificate. If you emailed me this proof certificate, I would know exactly what you proved. But most of the time, the formal system is part of the discourse, so proof objects are sufficient.

Moving up a level

Moving up a level

- We now know how to do proofs (in other words, how to reason within the system)

Moving up a level

- We now know how to do proofs (in other words, how to reason within the system)
- There is a large variety of such formal systems

Moving up a level

- We now know how to do proofs (in other words, how to reason within the system)
- There is a large variety of such formal systems
- Most of the mathematics you ever learned can be cranked out by such formal systems (see mizar.org)

Moving up a level

- We now know how to do proofs (in other words, how to reason within the system)
- There is a large variety of such formal systems
- Most of the mathematics you ever learned can be cranked out by such formal systems (see mizar.org)
- The foundations of software can also be cranked out by formal systems, as we'll see soon

Moving up a level

- We now know how to do proofs (in other words, how to reason within the system)
- There is a large variety of such formal systems
- Most of the mathematics you ever learned can be cranked out by such formal systems (see mizar.org)
- The foundations of software can also be cranked out by formal systems, as we'll see soon
- Now comes a big switch of perspective ...

Moving up a level

- We now know how to do proofs (in other words, how to reason within the system)
- There is a large variety of such formal systems
- Most of the mathematics you ever learned can be cranked out by such formal systems (see mizar.org)
- The foundations of software can also be cranked out by formal systems, as we'll see soon
- Now comes a big switch of perspective ...
- *These formal systems have some important properties, which can be studied with the use of mathematics*

Moving up a level

- We now know how to do proofs (in other words, how to reason within the system)
- There is a large variety of such formal systems
- Most of the mathematics you ever learned can be cranked out by such formal systems (see mizar.org)
- The foundations of software can also be cranked out by formal systems, as we'll see soon
- Now comes a big switch of perspective ...
- *These formal systems have some important properties, which can be studied with the use of mathematics*
- This study comes with a grand name: *metamathematics*, i.e. the study of mathematics itself

Moving up a level

- We now know how to do proofs (in other words, how to reason within the system)
- There is a large variety of such formal systems
- Most of the mathematics you ever learned can be cranked out by such formal systems (see mizar.org)
- The foundations of software can also be cranked out by formal systems, as we'll see soon
- Now comes a big switch of perspective ...
- *These formal systems have some important properties, which can be studied with the use of mathematics*
- This study comes with a grand name: *metamathematics*, i.e. the study of mathematics itself
- We'll do a lot of metamathematics in this program

Moving up a level

- We now know how to do proofs (in other words, how to reason within the system)
- There is a large variety of such formal systems
- Most of the mathematics you ever learned can be cranked out by such formal systems (see mizar.org)
- The foundations of software can also be cranked out by formal systems, as we'll see soon
- Now comes a big switch of perspective ...
- *These formal systems have some important properties, which can be studied with the use of mathematics*
- This study comes with a grand name: *metamathematics*, i.e. the study of mathematics itself
- We'll do a lot of metamathematics in this program
- And metasoftware, if you wish

Moving up a level

- We now know how to do proofs (in other words, how to reason within the system)
- There is a large variety of such formal systems
- Most of the mathematics you ever learned can be cranked out by such formal systems (see mizar.org)
- The foundations of software can also be cranked out by formal systems, as we'll see soon
- Now comes a big switch of perspective ...
- *These formal systems have some important properties, which can be studied with the use of mathematics*
- This study comes with a grand name: *metamathematics*, i.e. the study of mathematics itself
- We'll do a lot of metamathematics in this program
- And metasoftware, if you wish
- Meta just means 'outside' or 'about'

Meta level and object level

Meta level and object level

To distinguish it from metamathematics, mathematics inside the formal system is called *object level* mathematics; the language of the formal system is the object language.

Meta level and object level

To distinguish it from metamathematics, mathematics inside the formal system is called *object level* mathematics; the language of the formal system is the object language. We'll use mostly informal language when working at the meta level in this overview.

Meta level and object level

To distinguish it from metamathematics, mathematics inside the formal system is called *object level* mathematics; the language of the formal system is the object language. We'll use mostly informal language when working at the meta level in this overview. But keep in mind that some of the tools we study are able to do metamathematics formally, i.e. we can *create and analyze formal systems within another formal system*.

Meta level and object level

To distinguish it from metamathematics, mathematics inside the formal system is called *object level* mathematics; the language of the formal system is the object language. We'll use mostly informal language when working at the meta level in this overview. But keep in mind that some of the tools we study are able to do metamathematics formally, i.e. we can *create and analyze formal systems within another formal system*. The ML (Meta Language) programming language was created precisely for the purpose of doing metamathematics. OCaml and F# are its descendants.

Meta level and object level

To distinguish it from metamathematics, mathematics inside the formal system is called *object level* mathematics; the language of the formal system is the object language. We'll use mostly informal language when working at the meta level in this overview. But keep in mind that some of the tools we study are able to do metamathematics formally, i.e. we can *create and analyze formal systems within another formal system*. The ML (Meta Language) programming language was created precisely for the purpose of doing metamathematics. OCaml and F# are its descendants. We'll study them in the 'ML Languages and Provers' course.

Meta level and object level

To distinguish it from metamathematics, mathematics inside the formal system is called *object level* mathematics; the language of the formal system is the object language. We'll use mostly informal language when working at the meta level in this overview. But keep in mind that some of the tools we study are able to do metamathematics formally, i.e. we can *create and analyze formal systems within another formal system*. The ML (Meta Language) programming language was created precisely for the purpose of doing metamathematics. OCaml and F# are its descendants. We'll study them in the 'ML Languages and Provers' course. So let's get used to some of this 'meta' terminology:

Meta level and object level

To distinguish it from metamathematics, mathematics inside the formal system is called *object level* mathematics; the language of the formal system is the object language. We'll use mostly informal language when working at the meta level in this overview. But keep in mind that some of the tools we study are able to do metamathematics formally, i.e. we can *create and analyze formal systems within another formal system*. The ML (Meta Language) programming language was created precisely for the purpose of doing metamathematics. OCaml and F# are its descendants. We'll study them in the 'ML Languages and Provers' course. So let's get used to some of this 'meta' terminology:

- metalanguage (the language in which we reason about the formal system)

Meta level and object level

To distinguish it from metamathematics, mathematics inside the formal system is called *object level* mathematics; the language of the formal system is the object language. We'll use mostly informal language when working at the meta level in this overview. But keep in mind that some of the tools we study are able to do metamathematics formally, i.e. we can *create and analyze formal systems within another formal system*. The ML (Meta Language) programming language was created precisely for the purpose of doing metamathematics. OCaml and F# are its descendants. We'll study them in the 'ML Languages and Provers' course. So let's get used to some of this 'meta' terminology:

- metalanguage (the language in which we reason about the formal system)
- metavariables (variables that do not belong to the object language)

Meta level and object level

To distinguish it from metamathematics, mathematics inside the formal system is called *object level* mathematics; the language of the formal system is the object language. We'll use mostly informal language when working at the meta level in this overview. But keep in mind that some of the tools we study are able to do metamathematics formally, i.e. we can *create and analyze formal systems within another formal system*. The ML (Meta Language) programming language was created precisely for the purpose of doing metamathematics. OCaml and F# are its descendants. We'll study them in the 'ML Languages and Provers' course. So let's get used to some of this 'meta' terminology:

- metalanguage (the language in which we reason about the formal system)
- metavariables (variables that do not belong to the object language)
- metatheorem (a theorem about the system, not a theorem of the system)

Meta level and object level

To distinguish it from metamathematics, mathematics inside the formal system is called *object level* mathematics; the language of the formal system is the object language. We'll use mostly informal language when working at the meta level in this overview. But keep in mind that some of the tools we study are able to do metamathematics formally, i.e. we can *create and analyze formal systems within another formal system*. The ML (Meta Language) programming language was created precisely for the purpose of doing metamathematics. OCaml and F# are its descendants. We'll study them in the 'ML Languages and Provers' course. So let's get used to some of this 'meta' terminology:

- metalanguage (the language in which we reason about the formal system)
- metavariables (variables that do not belong to the object language)
- metatheorem (a theorem about the system, not a theorem of the system)
- metamathematics (study of formal systems themselves, from the outside)

The first big metamathematical question: Consistency

The first big metamathematical question: Consistency

Let's do some metamathematics.

The first big metamathematical question: Consistency

Let's do some metamathematics. Let's say that *all* the wffs of a formal system are theorems.

The first big metamathematical question: Consistency

Let's do some metamathematics. Let's say that *all* the wffs of a formal system are theorems. In this case the formal system is worthless, the rules do not accomplish anything useful.

The first big metamathematical question: Consistency

Let's do some metamathematics. Let's say that *all* the wffs of a formal system are theorems. In this case the formal system is worthless, the rules do not accomplish anything useful. So a formal system is said to be *consistent* if not all wffs are theorems.

The first big metamathematical question: Consistency

Let's do some metamathematics. Let's say that *all* the wffs of a formal system are theorems. In this case the formal system is worthless, the rules do not accomplish anything useful. So a formal system is said to be *consistent* if not all wffs are theorems.

Metatheorem: DANS is a consistent system

The first big metamathematical question: Consistency

Let's do some metamathematics. Let's say that *all* the wffs of a formal system are theorems. In this case the formal system is worthless, the rules do not accomplish anything useful. So a formal system is said to be *consistent* if not all wffs are theorems.

Metatheorem: DANS is a consistent system

Proof. We show that b* is not a theorem. The axiom * has no b on the left. No rules allow the introduction of a b on the left of the *. □

Decidability of grammar and rules

Decidability of grammar and rules

- Let's look at the grammar and the inference rules of our example

Decidability of grammar and rules

- Let's look at the grammar and the inference rules of our example
- We see that the grammar is *decidable*, i.e. there is an *algorithm* that, given a formula, it will answer YES if the formula is a wff and NO if it is not

Decidability of grammar and rules

- Let's look at the grammar and the inference rules of our example
- We see that the grammar is *decidable*, i.e. there is an *algorithm* that, given a formula, it will answer YES if the formula is a wff and NO if it is not
- Here is the grammar algorithm: given a formula X, count the number of *. If the count is 1, answer YES, otherwise answer NO.

Decidability of grammar and rules

- Let's look at the grammar and the inference rules of our example
- We see that the grammar is *decidable*, i.e. there is an *algorithm* that, given a formula, it will answer YES if the formula is a wff and NO if it is not
- Here is the grammar algorithm: given a formula X, count the number of *. If the count is 1, answer YES, otherwise answer NO.
- The inference rules are also rigged in this special way, i.e. they are also decidable

Decidability of grammar and rules

- Let's look at the grammar and the inference rules of our example
- We see that the grammar is *decidable*, i.e. there is an *algorithm* that, given a formula, it will answer YES if the formula is a wff and NO if it is not
- Here is the grammar algorithm: given a formula X, count the number of *. If the count is 1, answer YES, otherwise answer NO.
- The inference rules are also rigged in this special way, i.e. they are also decidable
- By this we mean that for each rule, there is an *algorithm* that, given a finite set of wffs, it will answer YES if the rule is applicable to them and NO if it is not

Decidability of grammar and rules

- Let's look at the grammar and the inference rules of our example
- We see that the grammar is *decidable*, i.e. there is an *algorithm* that, given a formula, it will answer YES if the formula is a wff and NO if it is not
- Here is the grammar algorithm: given a formula X, count the number of *. If the count is 1, answer YES, otherwise answer NO.
- The inference rules are also rigged in this special way, i.e. they are also decidable
- By this we mean that for each rule, there is an *algorithm* that, given a finite set of wffs, it will answer YES if the rule is applicable to them and NO if it is not
- For example, the algorithm for rule 4 is: find *. If we can find an a on its left and a b on its right, then the answer is YES, otherwise it is NO.

Decidability of grammar and rules

- Let's look at the grammar and the inference rules of our example
- We see that the grammar is *decidable*, i.e. there is an *algorithm* that, given a formula, it will answer YES if the formula is a wff and NO if it is not
- Here is the grammar algorithm: given a formula X, count the number of *. If the count is 1, answer YES, otherwise answer NO.
- The inference rules are also rigged in this special way, i.e. they are also decidable
- By this we mean that for each rule, there is an *algorithm* that, given a finite set of wffs, it will answer YES if the rule is applicable to them and NO if it is not
- For example, the algorithm for rule 4 is: find *. If we can find an a on its left and a b on its right, then the answer is YES, otherwise it is NO.
- Only one thing is left unclear: *what exactly is an algorithm?*

What is an algorithm?

What is an algorithm?

- Informally we know it is something that can be expressed in a finite number of steps, some sort of program maybe containing loops

What is an algorithm?

- Informally we know it is something that can be expressed in a finite number of steps, some sort of program maybe containing loops
- Other informal ways to describe algorithms: *computable* or, the original term used by Hilbert, *finitary*

What is an algorithm?

- Informally we know it is something that can be expressed in a finite number of steps, some sort of program maybe containing loops
- Other informal ways to describe algorithms: *computable* or, the original term used by Hilbert, *finitary*
- It took a lot of work to come up with a convincing *formal definition*

What is an algorithm?

- Informally we know it is something that can be expressed in a finite number of steps, some sort of program maybe containing loops
- Other informal ways to describe algorithms: *computable* or, the original term used by Hilbert, *finitary*
- It took a lot of work to come up with a convincing *formal definition*
- The astonishing thing is that many alternative definitions kept coming, and ...

What is an algorithm?

- Informally we know it is something that can be expressed in a finite number of steps, some sort of program maybe containing loops
- Other informal ways to describe algorithms: *computable* or, the original term used by Hilbert, *finitary*
- It took a lot of work to come up with a convincing *formal definition*
- The astonishing thing is that many alternative definitions kept coming, and ...
- *All formal definitions of 'algorithm' were proven to be equivalent!*

What is an algorithm?

- Informally we know it is something that can be expressed in a finite number of steps, some sort of program maybe containing loops
- Other informal ways to describe algorithms: *computable* or, the original term used by Hilbert, *finitary*
- It took a lot of work to come up with a convincing *formal definition*
- The astonishing thing is that many alternative definitions kept coming, and ...
- *All formal definitions of 'algorithm' were proven to be equivalent!*
- This equivalence theorem is one of the great accomplishments of the 20th century mathematics

What is an algorithm?

- Informally we know it is something that can be expressed in a finite number of steps, some sort of program maybe containing loops
- Other informal ways to describe algorithms: *computable* or, the original term used by Hilbert, *finitary*
- It took a lot of work to come up with a convincing *formal definition*
- The astonishing thing is that many alternative definitions kept coming, and ...
- *All formal definitions of 'algorithm' were proven to be equivalent!*
- This equivalence theorem is one of the great accomplishments of the 20th century mathematics
- The best known definitions are: Gödel's recursive functions, Turing machines and Church's Lambda Calculus

What is an algorithm?

- Informally we know it is something that can be expressed in a finite number of steps, some sort of program maybe containing loops
- Other informal ways to describe algorithms: *computable* or, the original term used by Hilbert, *finitary*
- It took a lot of work to come up with a convincing *formal definition*
- The astonishing thing is that many alternative definitions kept coming, and ...
- *All formal definitions of 'algorithm' were proven to be equivalent!*
- This equivalence theorem is one of the great accomplishments of the 20th century mathematics
- The best known definitions are: Gödel's recursive functions, Turing machines and Church's Lambda Calculus
  - Turing machines led to imperative languages like C, Java, C#

What is an algorithm?

- Informally we know it is something that can be expressed in a finite number of steps, some sort of program maybe containing loops
- Other informal ways to describe algorithms: *computable* or, the original term used by Hilbert, *finitary*
- It took a lot of work to come up with a convincing *formal definition*
- The astonishing thing is that many alternative definitions kept coming, and ...
- *All formal definitions of 'algorithm' were proven to be equivalent!*
- This equivalence theorem is one of the great accomplishments of the 20th century mathematics
- The best known definitions are: Gödel's recursive functions, Turing machines and Church's Lambda Calculus
  - Turing machines led to imperative languages like C, Java, C#
  - Church's Lambda Calculus led to functional languages like Lisp, ML, Haskell

The Church-Turing thesis

The Church-Turing thesis

The above equivalent formal definitions capture the informal notion of computability or algorithm.

The Church-Turing thesis

The above equivalent formal definitions capture the informal notion of computability or algorithm.

This is not a theorem, it is a thesis, so there is no question of proving it. It captures the belief that humanity has nailed the concept of computability, that there exists no other mechanism that deserves to be called an algorithm.

Formal systems are rigged in a very special way

Formal systems are rigged in a very special way

- This overview is not the place to describe any of the above formal definitions of computability

Formal systems are rigged in a very special way

- This overview is not the place to describe any of the above formal definitions of computability
- (we'll cover this in the 'Logic and Computation' course)

Formal systems are rigged in a very special way

- This overview is not the place to describe any of the above formal definitions of computability
- (we'll cover this in the 'Logic and Computation' course)
- All we need to know for now is that such a rigorous definition exists

Formal systems are rigged in a very special way

- This overview is not the place to describe any of the above formal definitions of computability
- (we'll cover this in the 'Logic and Computation' course)
- All we need to know for now is that such a rigorous definition exists
- Therefore, decidability, which lacked the rigorous definition of algorithm, is now rigorously defined

Formal systems are rigged in a very special way

- This overview is not the place to describe any of the above formal definitions of computability
- (we'll cover this in the 'Logic and Computation' course)
- All we need to know for now is that such a rigorous definition exists
- Therefore, decidability, which lacked the rigorous definition of algorithm, is now rigorously defined
- *For all formal systems, the grammar is always rigged so that it is decidable*

Formal systems are rigged in a very special way

- This overview is not the place to describe any of the above formal definitions of computability
- (we'll cover this in the 'Logic and Computation' course)
- All we need to know for now is that such a rigorous definition exists
- Therefore, decidability, which lacked the rigorous definition of algorithm, is now rigorously defined
- *For all formal systems, the grammar is always rigged so that it is decidable*
- *The inference rules are also rigged so that they are decidable*

Formal systems are rigged in a very special way

- This overview is not the place to describe any of the above formal definitions of computability
- (we'll cover this in the 'Logic and Computation' course)
- All we need to know for now is that such a rigorous definition exists
- Therefore, decidability, which lacked the rigorous definition of algorithm, is now rigorously defined
- *For all formal systems, the grammar is always rigged so that it is decidable*
- *The inference rules are also rigged so that they are decidable*
- *In one sentence, formal systems have computability built-in*

The power of formal systems

The power of formal systems

- Not only do they have computability built-in

The power of formal systems

- Not only do they have computability built-in
- They do not have any *more* computational power than Turing machines have, they are yet another formalism equivalent to the other three

The power of formal systems

- Not only do they have computability built-in
- They do not have any *more* computational power than Turing machines have, they are yet another formalism equivalent to the other three
- Just as in the case of Turing machines, nondeterminism does not add to this power, i.e. you can build an equivalent deterministic system that produces the same theorems

The power of formal systems

- Not only do they have computability built-in
- They do not have any *more* computational power than Turing machines have, they are yet another formalism equivalent to the other three
- Just as in the case of Turing machines, nondeterminism does not add to this power, i.e. you can build an equivalent deterministic system that produces the same theorems

The power of formal systems

- Not only do they have computability built-in
- They do not have any *more* computational power than Turing machines have, they are yet another formalism equivalent to the other three
- Just as in the case of Turing machines, nondeterminism does not add to this power, i.e. you can build an equivalent deterministic system that produces the same theorems

So, for any theory that can be described algorithmically (and that's all we really need, but that's philosophy), whether a mathematical theory, a software theory, a physics theory, etc . . . , there is a formal system that can describe it.

Back to our formal system

Back to our formal system

- Because DANS has computability built in, we can do two things

Back to our formal system

- Because DANS has computability built in, we can do two things
- We can design a mechanical strategy that builds all the theorems of the system, one after the other (exercise)

Back to our formal system

- Because DANS has computability built in, we can do two things
- We can design a mechanical strategy that builds all the theorems of the system, one after the other (exercise)
- We can design a mechanical strategy that, given a proof, answers YES if the proof is correct, NO otherwise (exercise)

Back to our formal system

- Because DANS has computability built in, we can do two things
- We can design a mechanical strategy that builds all the theorems of the system, one after the other (exercise)
- We can design a mechanical strategy that, given a proof, answers YES if the proof is correct, NO otherwise (exercise)
- Notice that our formal system produces an infinite number of theorems, so the procedure above *potentially* produces all the theorems, we can never have all the theorems collected together as that set would be an *actual* infinity, which is impossible to reach computationally

Back to our formal system

- Because DANS has computability built in, we can do two things
- We can design a mechanical strategy that builds all the theorems of the system, one after the other (exercise)
- We can design a mechanical strategy that, given a proof, answers YES if the proof is correct, NO otherwise (exercise)
- Notice that our formal system produces an infinite number of theorems, so the procedure above *potentially* produces all the theorems, we can never have all the theorems collected together as that set would be an *actual* infinity, which is impossible to reach computationally
- *In formal systems, the infinite is always understood as this potential, not actual, representation*

Back to our formal system

- Because DANS has computability built in, we can do two things
- We can design a mechanical strategy that builds all the theorems of the system, one after the other (exercise)
- We can design a mechanical strategy that, given a proof, answers YES if the proof is correct, NO otherwise (exercise)
- Notice that our formal system produces an infinite number of theorems, so the procedure above *potentially* produces all the theorems, we can never have all the theorems collected together as that set would be an *actual* infinity, which is impossible to reach computationally
- *In formal systems, the infinite is always understood as this potential, not actual, representation*
- If this sounds difficult, think of the way natural numbers or lists are introduced in your favorite functional language. The constructors embody this potential infinity.

Decidability of our example

Decidability of our example

- If we have a wff, how can we tell if it is a theorem or not?

Decidability of our example

- If we have a wff, how can we tell if it is a theorem or not?
- Does DANS have an algorithm that answers YES or NO to such a question?

Decidability of our example

- If we have a wff, how can we tell if it is a theorem or not?
- Does DANS have an algorithm that answers YES or NO to such a question?
- This is a big fork in the road for the formal systems we study in the program

Decidability of our example

- If we have a wff, how can we tell if it is a theorem or not?
- Does DANS have an algorithm that answers YES or NO to such a question?
- This is a big fork in the road for the formal systems we study in the program
- Some systems do admit such an algorithm (called a *decision procedure*), others don't

Decidability of our example

- If we have a wff, how can we tell if it is a theorem or not?
- Does DANS have an algorithm that answers YES or NO to such a question?
- This is a big fork in the road for the formal systems we study in the program
- Some systems do admit such an algorithm (called a *decision procedure*), others don't
- Finding a decision procedure for a useful formal system (not a toy like DANS) is a non-trivial effort

Decidability of our example

- If we have a wff, how can we tell if it is a theorem or not?
- Does DANS have an algorithm that answers YES or NO to such a question?
- This is a big fork in the road for the formal systems we study in the program
- Some systems do admit such an algorithm (called a *decision procedure*), others don't
- Finding a decision procedure for a useful formal system (not a toy like DANS) is a non-trivial effort
- The more expressive the system (i.e. the more powerful its language and rules of inference are), the less likely it is that it has such a procedure

DANS is decidable

may be skipped

DANS is decidable

may be skipped

Metatheorem: Dans is a decidable system

The proof is by induction. *Proofs by induction are typical of the metamathematics of formal systems*, that's why we include one here. They will be seen over and over during the program. Those unfamiliar with mathematical induction may skip the proof.

We treat induction at length in the course 'Formal semantics of programming languages'. *We also give a more detailed presentation of induction and recursion, as they relate to categories, in a more advanced section of this overview*. For those who need a quick brush-up, we include the definition of mathematical induction and one typical example of its use.

Mathematical Induction

may be skipped

Mathematical Induction

may be skipped

Mathematical Induction

Let $P(n)$ be a predicate on the set $\mathbb{N}$ of natural numbers. Then

$$(P(0) \wedge (\forall k \in \mathbb{N}.P(k) \rightarrow P(k+1))) \rightarrow \forall n \in \mathbb{N}.P(n)$$

A typical proof by mathematical induction

may be skipped

A typical proof by mathematical induction

may be skipped

The sum of the first n natural numbers is given by

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

A typical proof by mathematical induction

may be skipped

The sum of the first n natural numbers is given by

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

Proof:

A typical proof by mathematical induction

may be skipped

The sum of the first n natural numbers is given by

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

Proof:

1. Let $P(n)$ be the property given by (1)

A typical proof by mathematical induction

may be skipped

The sum of the first n natural numbers is given by

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

Proof:

1. Let $P(n)$ be the property given by (1)
2. $P(0)$ is true, because $0 = \frac{0(0+1)}{2}$

A typical proof by mathematical induction

may be skipped

The sum of the first n natural numbers is given by

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

Proof:

1. Let $P(n)$ be the property given by (1)
2. $P(0)$ is true, because $0 = \frac{0(0+1)}{2}$
3. Assume $P(k)$ is true, i.e. $\sum_{i=0}^{k} i = \frac{k(k+1)}{2}$

A typical proof by mathematical induction

may be skipped

The sum of the first n natural numbers is given by

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

Proof:

1. Let $P(n)$ be the property given by (1)
2. $P(0)$ is true, because $0 = \frac{0(0+1)}{2}$
3. Assume $P(k)$ is true, i.e. $\sum_{i=0}^{k} i = \frac{k(k+1)}{2}$
4. Then $\sum_{i=0}^{k+1} i = \frac{k(k+1)}{2} + (k+1) = \frac{k(k+1)+2(k+1)}{2} = \frac{(k+1)(k+2)}{2}$

A typical proof by mathematical induction

may be skipped

The sum of the first n natural numbers is given by

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

Proof:

1. Let $P(n)$ be the property given by (1)
2. $P(0)$ is true, because $0 = \frac{0(0+1)}{2}$
3. Assume $P(k)$ is true, i.e. $\sum_{i=0}^{k} i = \frac{k(k+1)}{2}$
4. Then $\sum_{i=0}^{k+1} i = \frac{k(k+1)}{2} + (k+1) = \frac{k(k+1)+2(k+1)}{2} = \frac{(k+1)(k+2)}{2}$
5. But $\sum_{i=0}^{k+1} i = \frac{(k+1)(k+2)}{2}$ is exactly $P(k+1)$

A typical proof by mathematical induction

may be skipped

The sum of the first n natural numbers is given by

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

Proof:

1. Let $P(n)$ be the property given by (1)
2. $P(0)$ is true, because $0 = \frac{0(0+1)}{2}$
3. Assume $P(k)$ is true, i.e. $\sum_{i=0}^{k} i = \frac{k(k+1)}{2}$
4. Then $\sum_{i=0}^{k+1} i = \frac{k(k+1)}{2} + (k+1) = \frac{k(k+1)+2(k+1)}{2} = \frac{(k+1)(k+2)}{2}$
5. But $\sum_{i=0}^{k+1} i = \frac{(k+1)(k+2)}{2}$ is exactly $P(k+1)$
6. By mathematical induction, the formula (1) is true for all $n \in \mathbb{N}$ □

Proof that DANS is decidable

may be skipped

Proof that DANS is decidable

may be skipped

Proof that DANS is decidable

may be skipped

Proof.

- Let's consider the wffs having the property (called P) that they contain

Proof that DANS is decidable

may be skipped

Proof.

- Let's consider the wffs having the property (called P) that they contain
  - a single *

Proof that DANS is decidable

may be skipped

Proof.

- Let's consider the wffs having the property (called P) that they contain
  - a single *
  - only a's on the left, only b's on the right

Proof that DANS is decidable

may be skipped

Proof.

- Let's consider the wffs having the property (called P) that they contain
  - a single *
  - only a's on the left, only b's on the right
  - less a's than b's

Proof that DANS is decidable

may be skipped

Proof.

- Let's consider the wffs having the property (called P) that they contain
  - a single *
  - only a's on the left, only b's on the right
  - less a's than b's
- We will prove that all theorems have this property. The metaproof is by induction on the length of the object proof.

Proof that DANS is decidable

may be skipped

Proof that DANS is decidable

may be skipped

- If the proof has length 1, then the theorem must be * and this has the property P

Proof that DANS is decidable

may be skipped

- If the proof has length 1, then the theorem must be * and this has the property P
- Assume all theorems whose proofs have length smaller or equal to n have this property P

Proof that DANS is decidable

may be skipped

- If the proof has length 1, then the theorem must be * and this has the property P
- Assume all theorems whose proofs have length smaller or equal to n have this property P
- Let $p_1, \ldots, p_{n+1}$ be a proof of length $n+1$. Then p_{n+1} must have been deduced from p_n by one of the rules 2,3, or 4. Now since the proof of p_n has length n, by the induction hypothesis, p_n has property P. But then p_{n+1} must also have property P because rules 2,3, and 4 all preserve this property. □

The central role of induction

The central role of induction

- The most important proof technique of metamathematics

The central role of induction

- The most important proof technique of metamathematics
- (we already saw some of its uses)

The central role of induction

- The most important proof technique of metamathematics
- (we already saw some of its uses)
- Induction itself can be formalized, for example, inside one of the most basic formal systems, Primitive Recursive Arithmetic (PRA), a sublogic of the better-known Peano Arithmetic

The central role of induction

- The most important proof technique of metamathematics
- (we already saw some of its uses)
- Induction itself can be formalized, for example, inside one of the most basic formal systems, Primitive Recursive Arithmetic (PRA), a sublogic of the better-known Peano Arithmetic
- We study PRA in the 'Main Concepts of Logic' course

The central role of induction

The central role of induction

- Induction is so ubiquitous in our program because

The central role of induction

- Induction is so ubiquitous in our program because
  - the formation rules are defined inductively

The central role of induction

- Induction is so ubiquitous in our program because
  - the formation rules are defined inductively
  - the inference rules are defined inductively

The central role of induction

- Induction is so ubiquitous in our program because
  - the formation rules are defined inductively
  - the inference rules are defined inductively
  - the grammar of programing languages is defined inductively

The central role of induction

- Induction is so ubiquitous in our program because
  - the formation rules are defined inductively
  - the inference rules are defined inductively
  - the grammar of programing languages is defined inductively
  - algebraic data is defined inductively (and it is known that *any computational form of data can be described algebraically*)

So should we pick PRA as the base logic and be done?

So should we pick PRA as the base logic and be done?

We will also see in the section on code verification, that the main method of verification is using an axiomatic semantics of the programming language to do *induction over the structure of the program*.

So should we pick PRA as the base logic and be done?

We will also see in the section on code verification, that the main method of verification is using an axiomatic semantics of the programming language to do *induction over the structure of the program*. So if induction is so important for both the study of formal systems themselves and their applications to software verification, why not do everything with an inductive prover?

So should we pick PRA as the base logic and be done?

We will also see in the section on code verification, that the main method of verification is using an axiomatic semantics of the programming language to do *induction over the structure of the program*. So if induction is so important for both the study of formal systems themselves and their applications to software verification, why not do everything with an inductive prover?

- Some software development can indeed be based on PRA and inductive provers

So should we pick PRA as the base logic and be done?

We will also see in the section on code verification, that the main method of verification is using an axiomatic semantics of the programming language to do *induction over the structure of the program*. So if induction is so important for both the study of formal systems themselves and their applications to software verification, why not do everything with an inductive prover?

- Some software development can indeed be based on PRA and inductive provers
- We study them in the 'Lisp and Inductive Provers' course

So should we pick PRA as the base logic and be done?

We will also see in the section on code verification, that the main method of verification is using an axiomatic semantics of the programming language to do *induction over the structure of the program*. So if induction is so important for both the study of formal systems themselves and their applications to software verification, why not do everything with an inductive prover?

- Some software development can indeed be based on PRA and inductive provers
- We study them in the 'Lisp and Inductive Provers' course
- Best known such prover: ACL2

Choose the right prover for the job

Choose the right prover for the job

- But we need a larger set of provers, some more automatic, some more interactive

Choose the right prover for the job

- But we need a larger set of provers, some more automatic, some more interactive
- This is an engineering job, look at many provers and decide what you need

Choose the right prover for the job

- But we need a larger set of provers, some more automatic, some more interactive
- This is an engineering job, look at many provers and decide what you need
- There are big differences between them

Choose the right prover for the job

- But we need a larger set of provers, some more automatic, some more interactive
- This is an engineering job, look at many provers and decide what you need
- There are big differences between them
- FOL Provers: more automatic, many efficient decision procedures can be added

Choose the right prover for the job

- But we need a larger set of provers, some more automatic, some more interactive
- This is an engineering job, look at many provers and decide what you need
- There are big differences between them
- FOL Provers: more automatic, many efficient decision procedures can be added
- HOL provers: powerful, but require more trust and more theory

Choose the right prover for the job

- But we need a larger set of provers, some more automatic, some more interactive
- This is an engineering job, look at many provers and decide what you need
- There are big differences between them
- FOL Provers: more automatic, many efficient decision procedures can be added
- HOL provers: powerful, but require more trust and more theory
- Type theory provers: even more powerful, but require even more trust and theory

Choose the right prover for the job

- But we need a larger set of provers, some more automatic, some more interactive
- This is an engineering job, look at many provers and decide what you need
- There are big differences between them
- FOL Provers: more automatic, many efficient decision procedures can be added
- HOL provers: powerful, but require more trust and more theory
- Type theory provers: even more powerful, but require even more trust and theory
- (why is trust an issue? Because the more powerful the logic, the easier it is to become inconsistent)

Other terms used for formal systems

Other terms used for formal systems

- In papers, books, www, etc . . . the terms *formal system*, *logic system*, *logic calculus*, *logic* are used interchangeably (e.g., lambda *calculus*, but combinatory *logic*). We'll follow this loose tradition.

Other terms used for formal systems

- In papers, books, www, etc . . . the terms *formal system*, *logic system*, *logic calculus*, *logic* are used interchangeably (e.g., lambda *calculus*, but combinatory *logic*). We'll follow this loose tradition.

Other terms used for formal systems

- In papers, books, www, etc . . . the terms *formal system*, *logic system*, *logic calculus*, *logic* are used interchangeably (e.g., lambda *calculus*, but combinatory *logic*). We'll follow this loose tradition. On the other hand the term *theory* is generally used in a more precise sense; namely it is a specification of a special set of symbols (called a signature) and additional rules, on top of a base formal system.

Other terms used for formal systems

- In papers, books, www, etc . . . the terms *formal system*, *logic system*, *logic calculus*, *logic* are used interchangeably (e.g., lambda *calculus*, but combinatory *logic*). We'll follow this loose tradition. On the other hand the term *theory* is generally used in a more precise sense; namely it is a specification of a special set of symbols (called a signature) and additional rules, on top of a base formal system.
- Often a *logic* includes a meaning for the language (i.e. a semantics); our formal systems don't.

Other terms used for formal systems

- In papers, books, www, etc . . . the terms *formal system*, *logic system*, *logic calculus*, *logic* are used interchangeably (e.g., lambda *calculus*, but combinatory *logic*). We'll follow this loose tradition. On the other hand the term *theory* is generally used in a more precise sense; namely it is a specification of a special set of symbols (called a signature) and additional rules, on top of a base formal system.
- Often a *logic* includes a meaning for the language (i.e. a semantics); our formal systems don't.
- Sometimes a *logic* does not include a proof calculus, it includes only a semantics!

Other terms used for formal systems

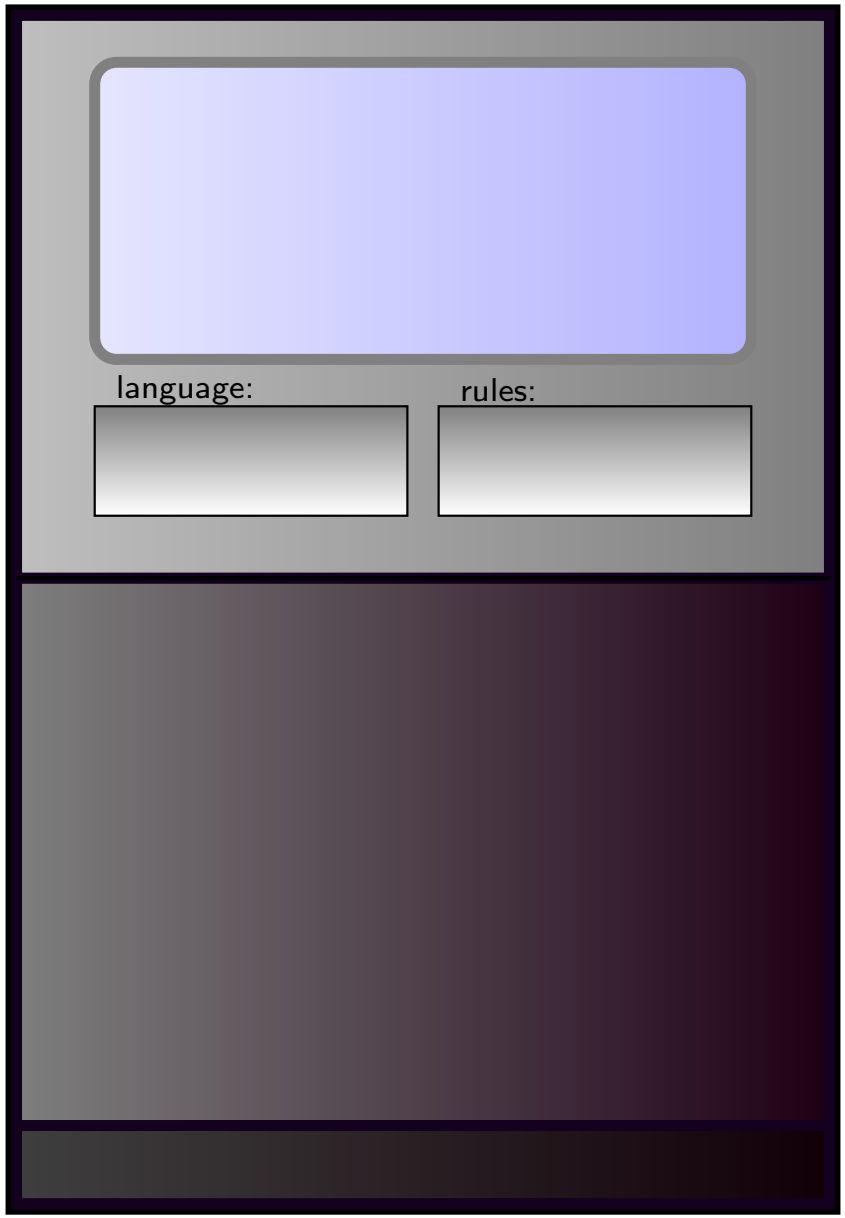
Other terms used for formal systems

- Some people make no difference between metamathematics and mathematical logic; originally, when Hilbert coined the term metamathematics, the two fields were the same. But right now, most people consider mathematical logic to be a much broader field, and metamathematics to be just the mathematical study of the rules of a formal system.

Other terms used for formal systems

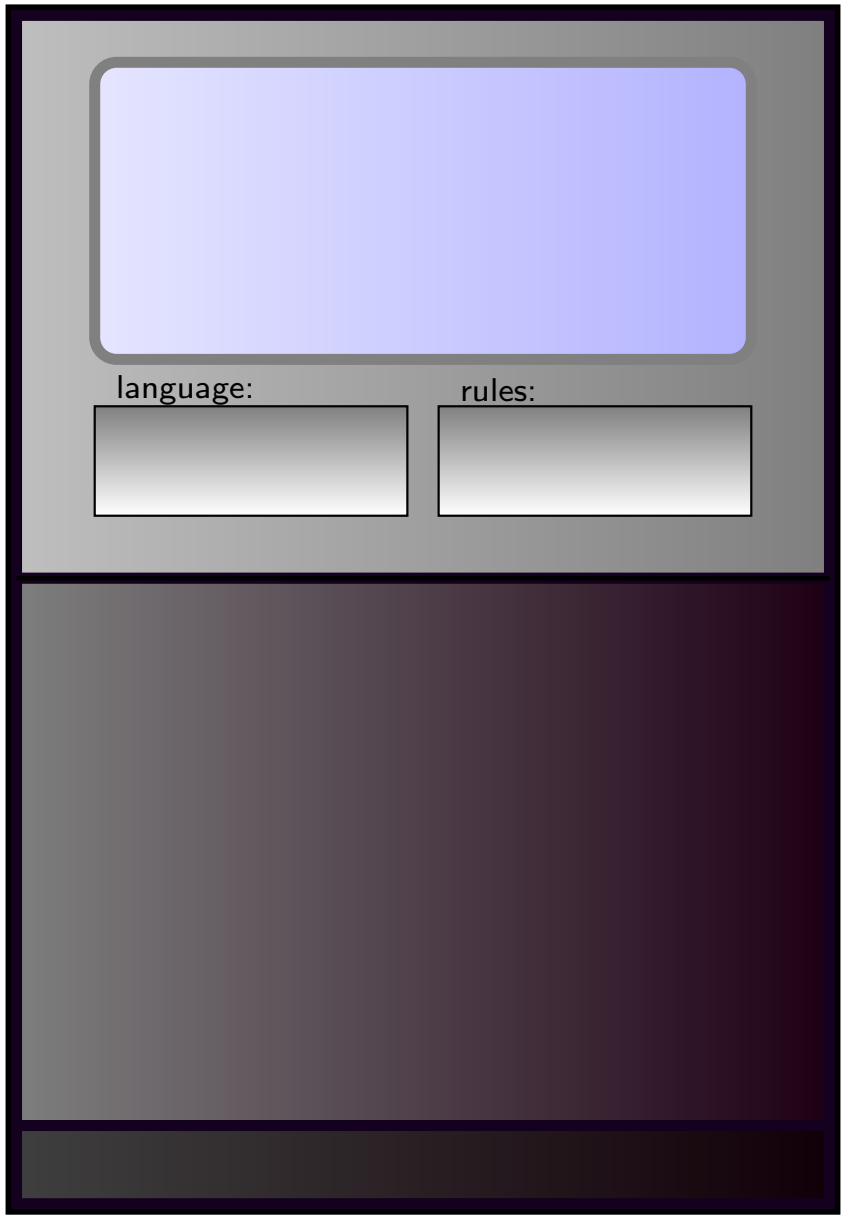
- Some people make no difference between metamathematics and mathematical logic; originally, when Hilbert coined the term metamathematics, the two fields were the same. But right now, most people consider mathematical logic to be a much broader field, and metamathematics to be just the mathematical study of the rules of a formal system.
- By the way, logic for us is always *mathematical* logic. Otherwise, logic is a field of philosophy and we have no use for that.

Machines



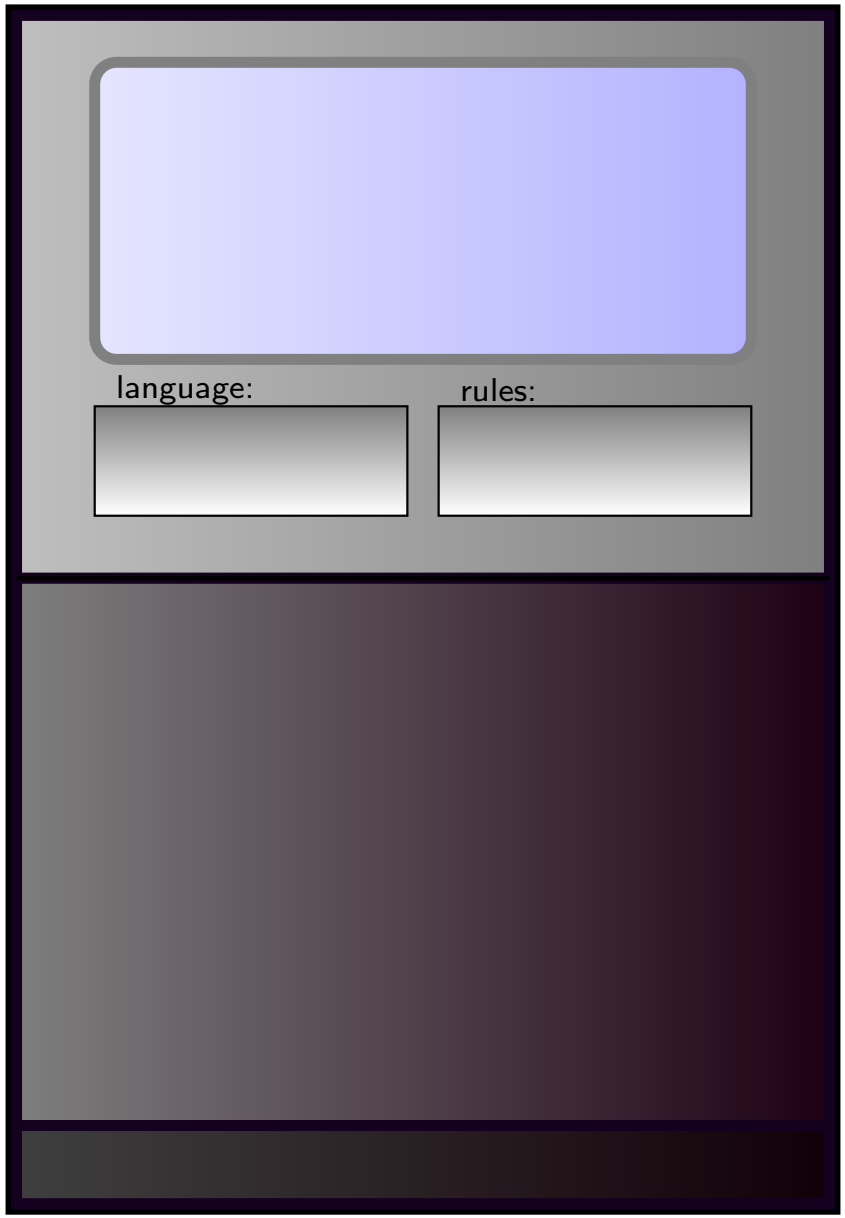
Machines

- We have learned how to produce proofs manually



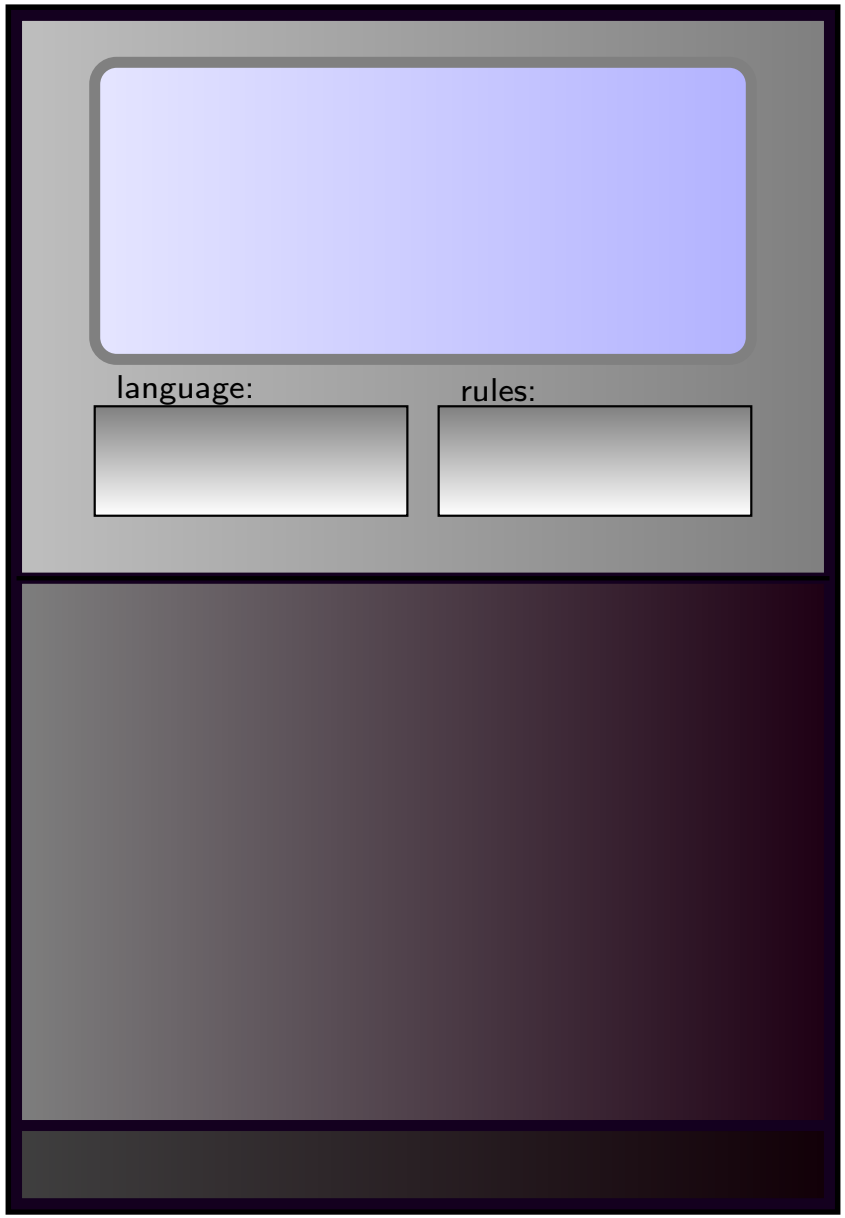
Machines

- We have learned how to produce proofs manually
- But a formal system is obviously meant to be used by a machine



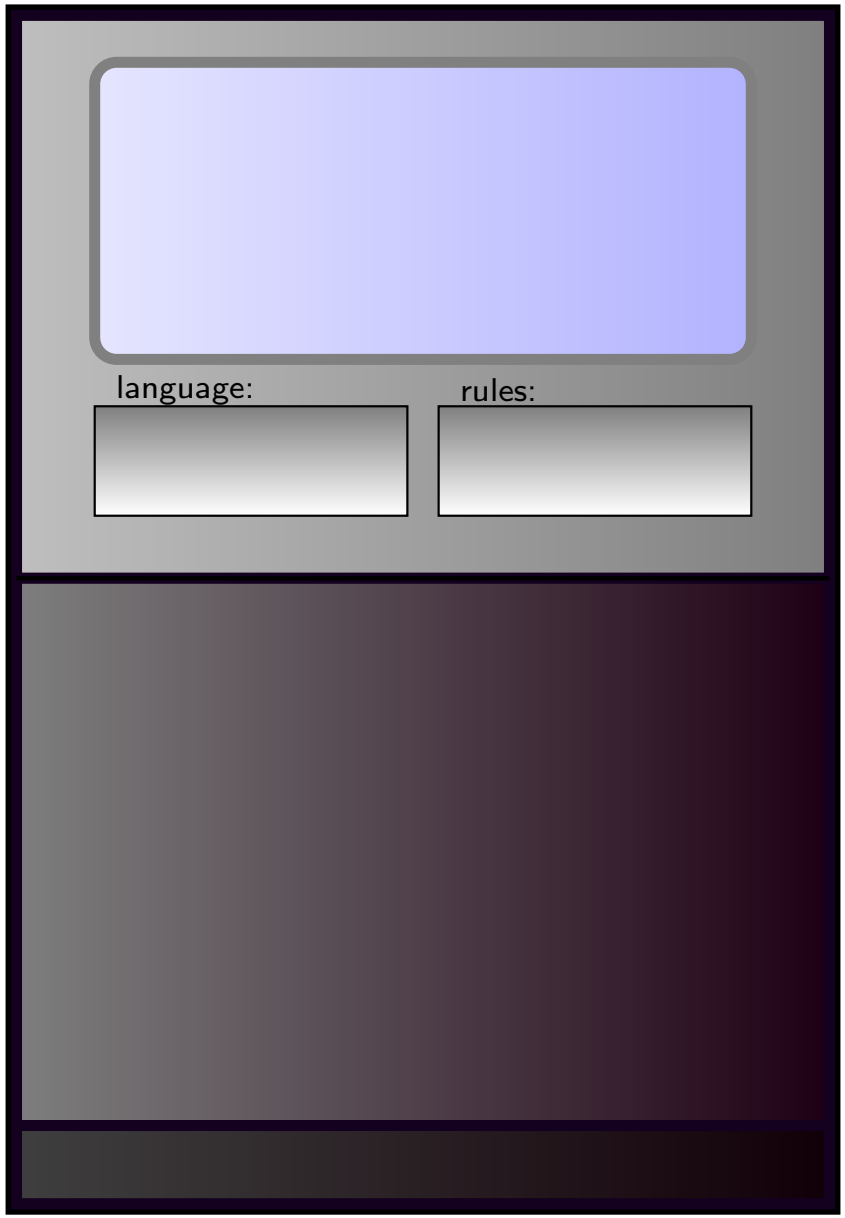
Machines

- We have learned how to produce proofs manually
- But a formal system is obviously meant to be used by a machine
- We now show how to build such a machine



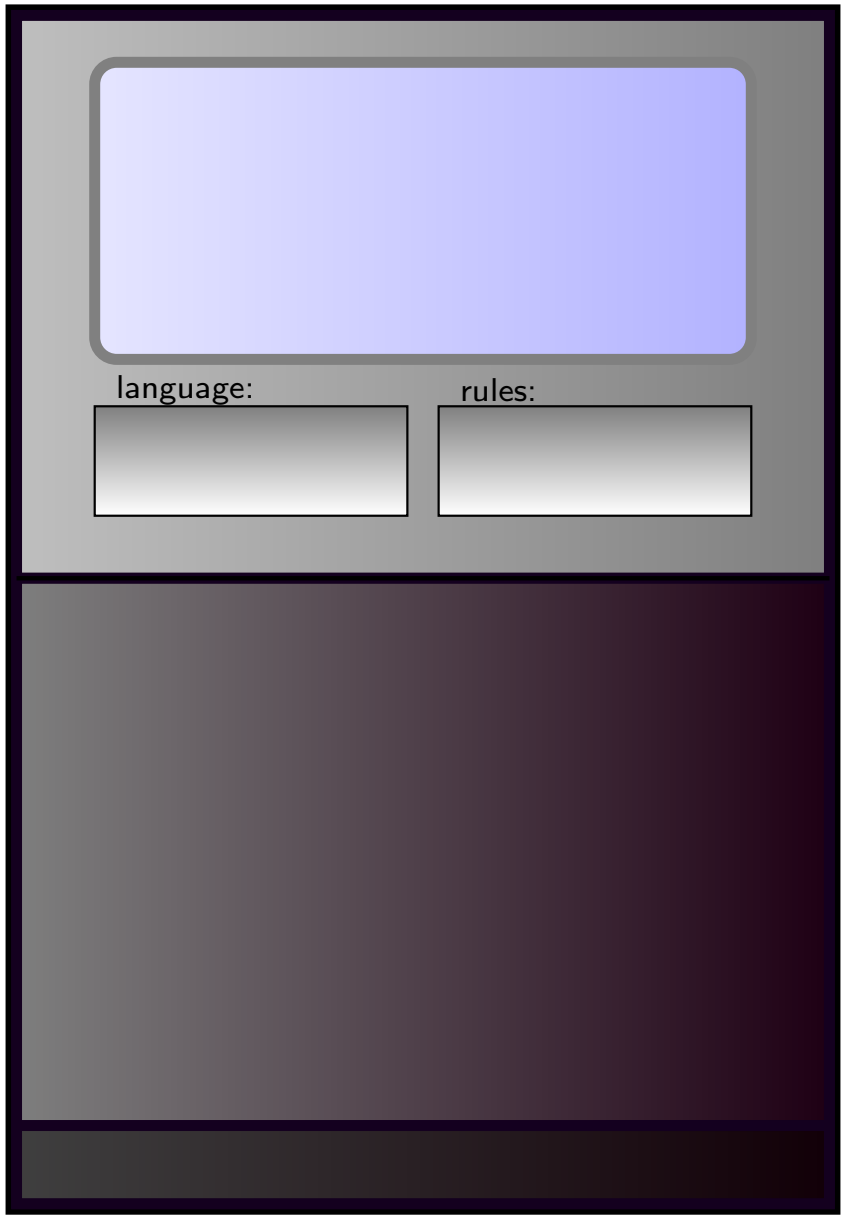
Machines

- We have learned how to produce proofs manually
- But a formal system is obviously meant to be used by a machine
- We now show how to build such a machine
- The formal system appears at the top of the machine



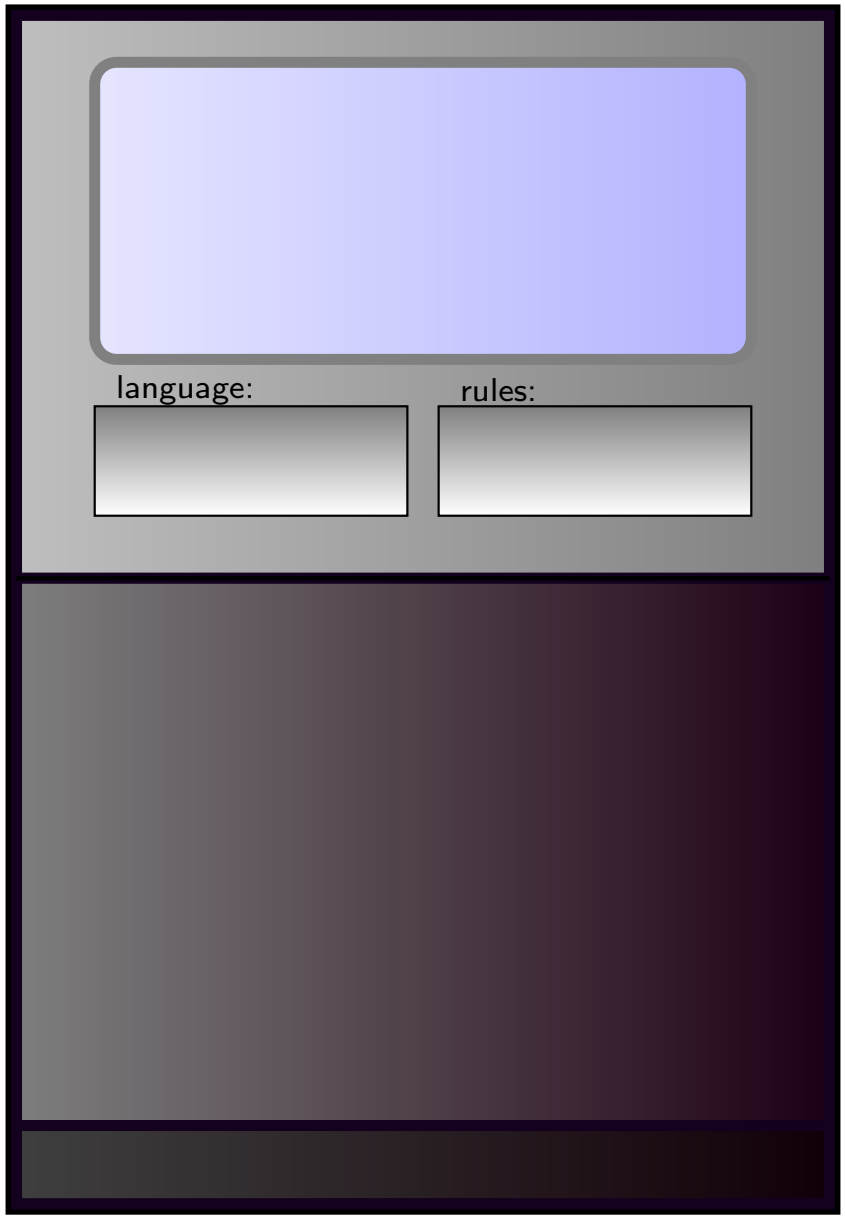
Machines

- We have learned how to produce proofs manually
- But a formal system is obviously meant to be used by a machine
- We now show how to build such a machine
- The formal system appears at the top of the machine



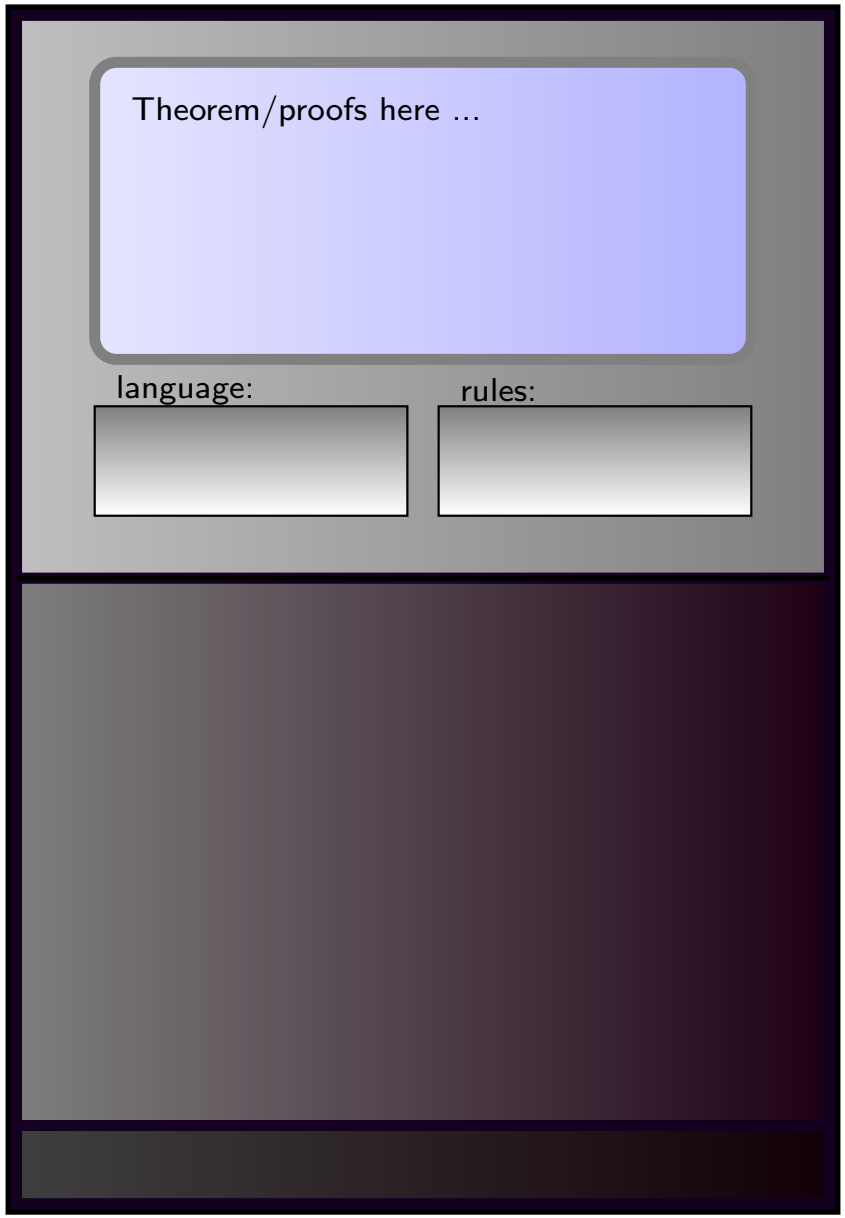
Machines

- We have learned how to produce proofs manually
- But a formal system is obviously meant to be used by a machine
- We now show how to build such a machine
- The formal system appears at the top of the machine
- The output of the machine is also at the top



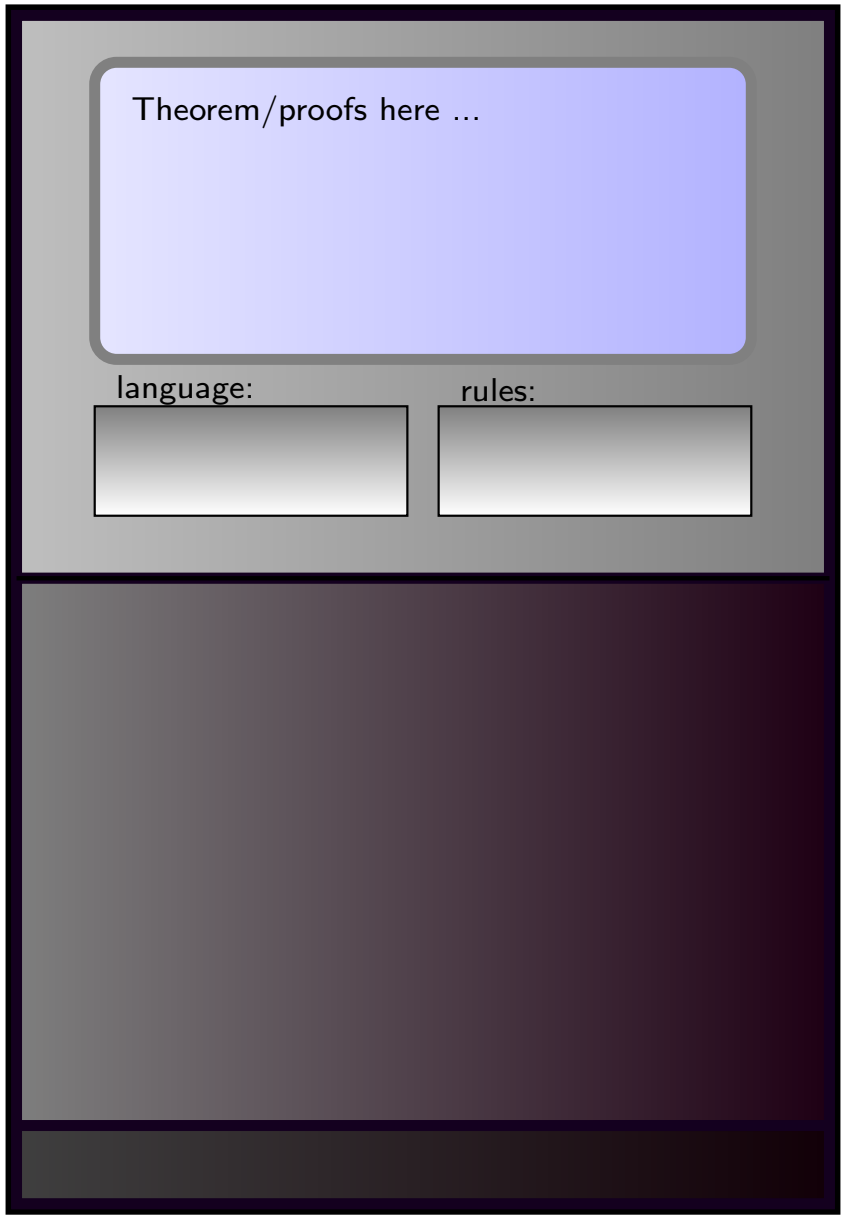
Machines

- We have learned how to produce proofs manually
- But a formal system is obviously meant to be used by a machine
- We now show how to build such a machine
- The formal system appears at the top of the machine
- The output of the machine is also at the top



Machines

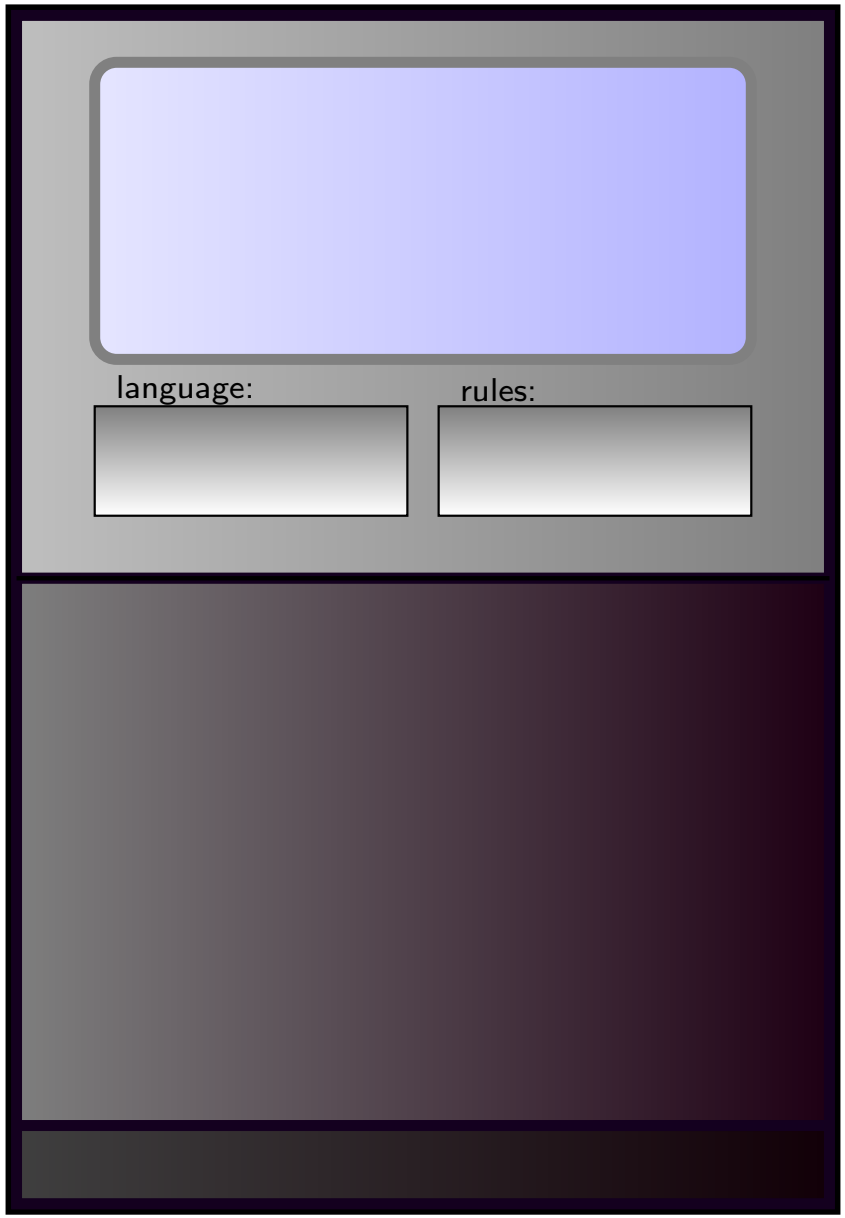
- We have learned how to produce proofs manually
- But a formal system is obviously meant to be used by a machine
- We now show how to build such a machine
- The formal system appears at the top of the machine
- The output of the machine is also at the top
- The top is the 'object level' of the machine



Building the DANS machine

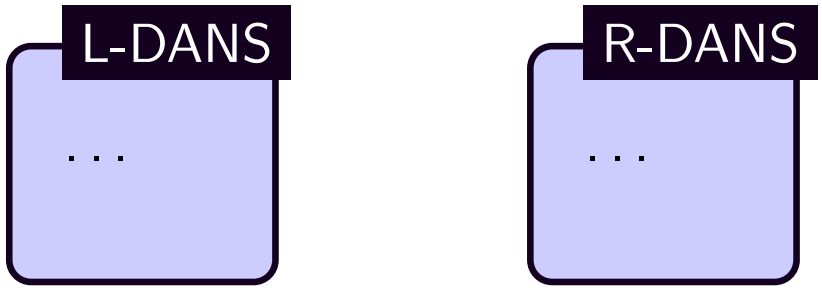
Building the DANS machine

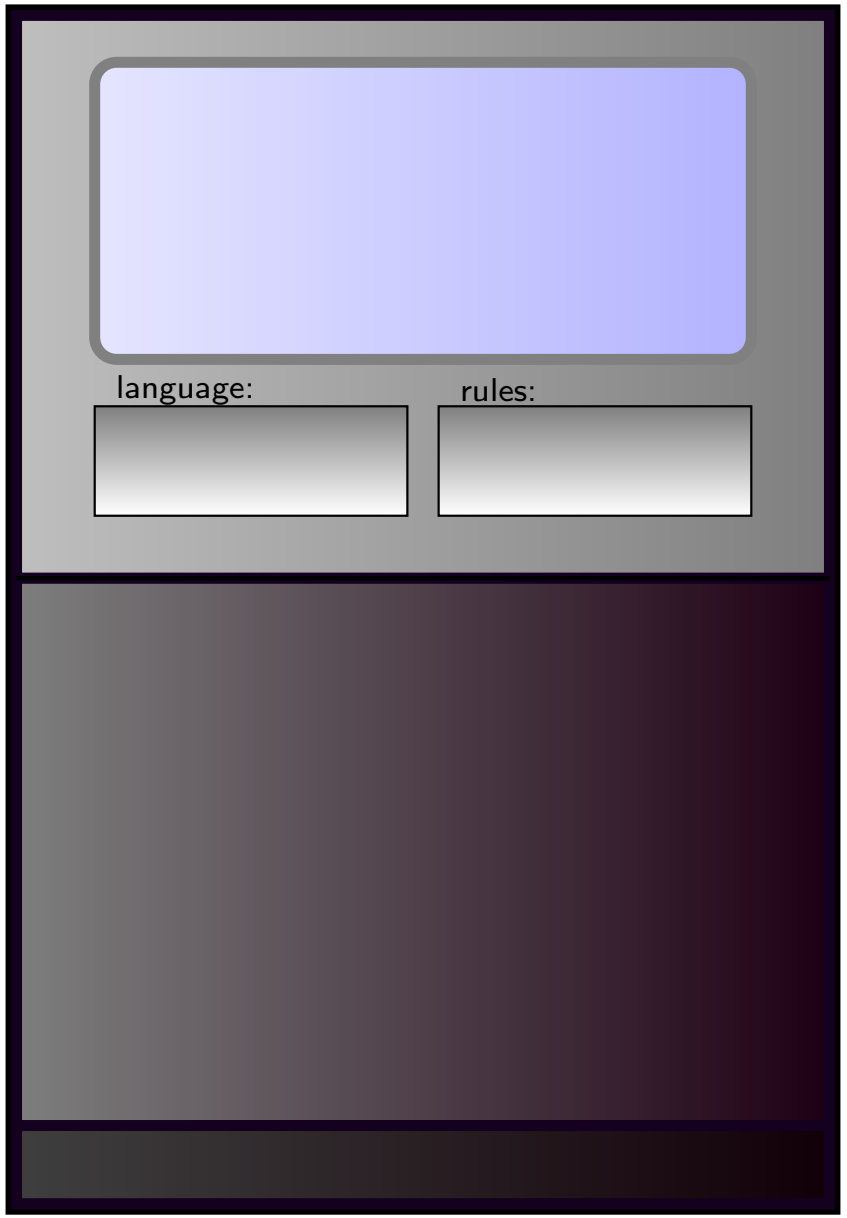
So let's take the DANS language (L-DANS) and the rules (R-DANS) and build a machine around this formal system.



Building the DANS machine

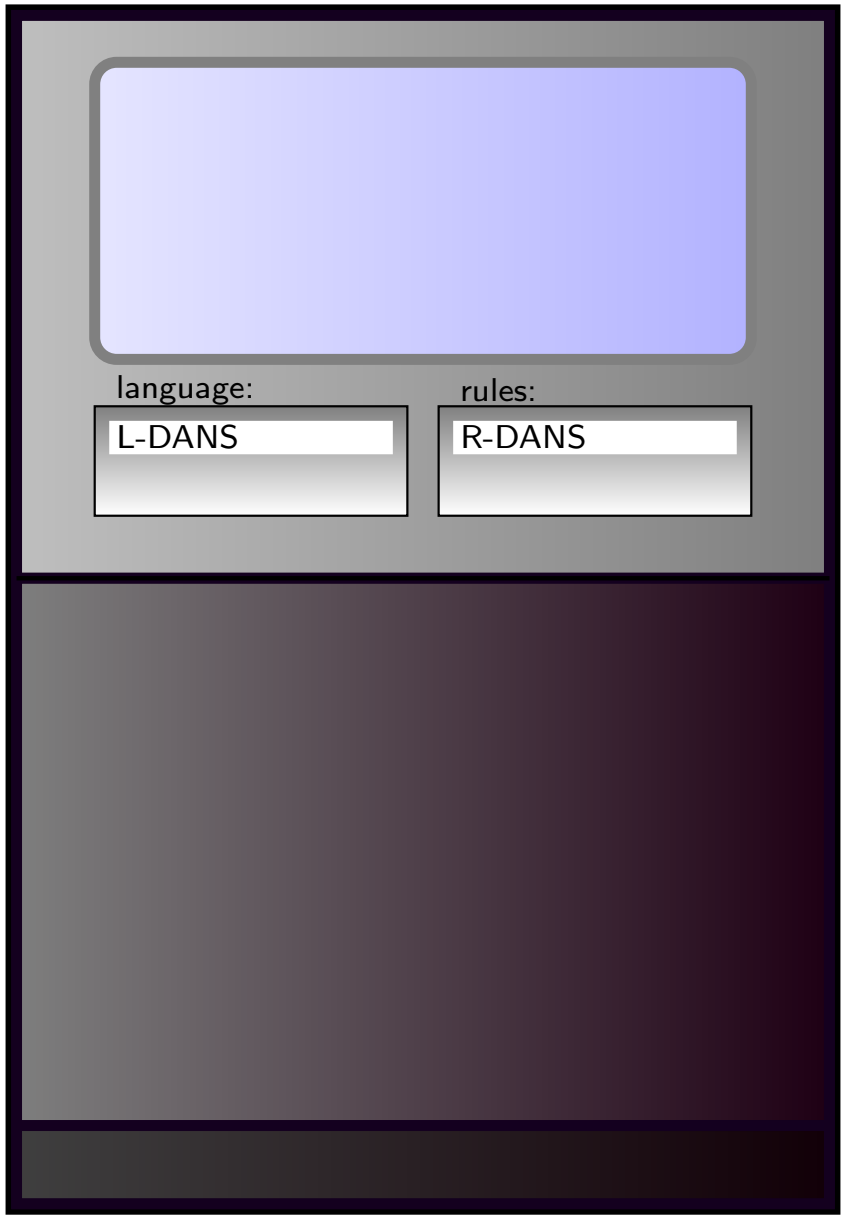
So let's take the DANS language (L-DANS) and the rules (R-DANS) and build a machine around this formal system.





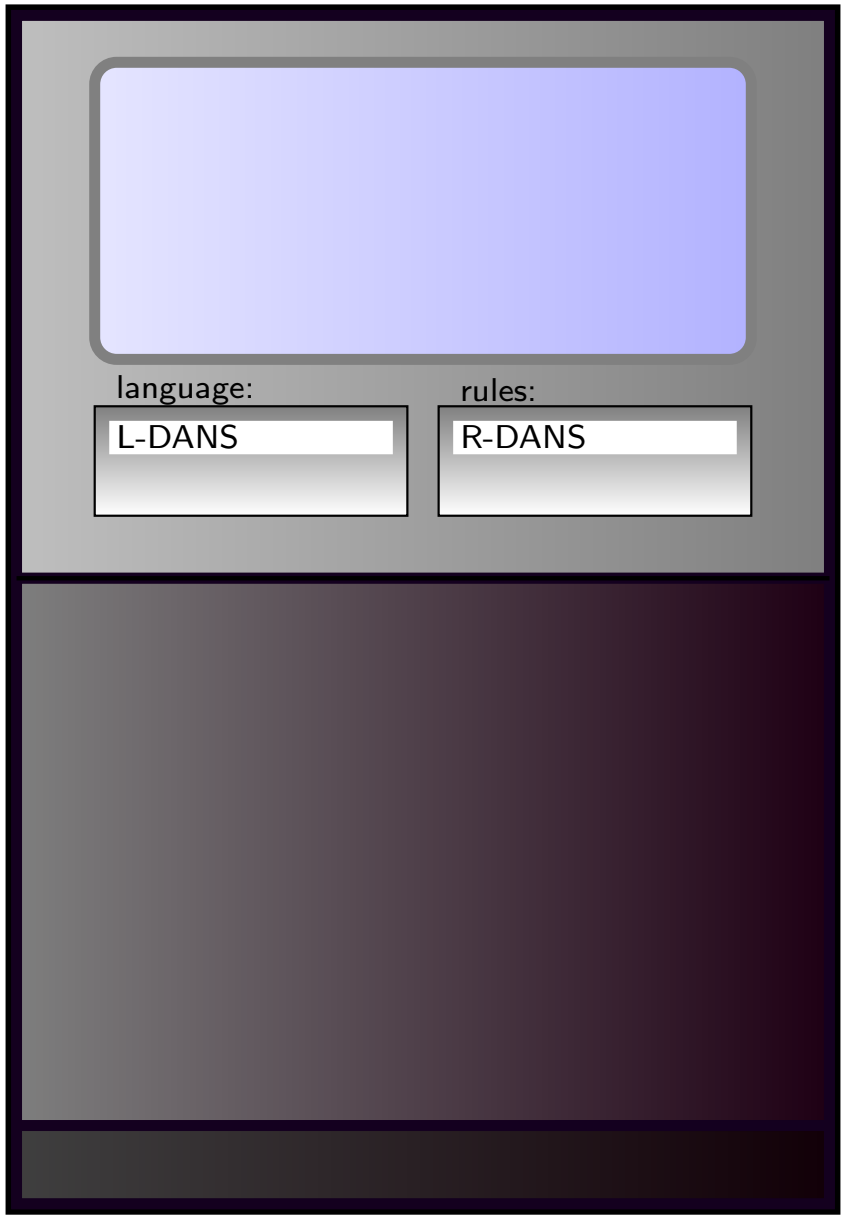
Building the DANS machine

So let's take the DANS language (L-DANS) and the rules (R-DANS) and build a machine around this formal system.



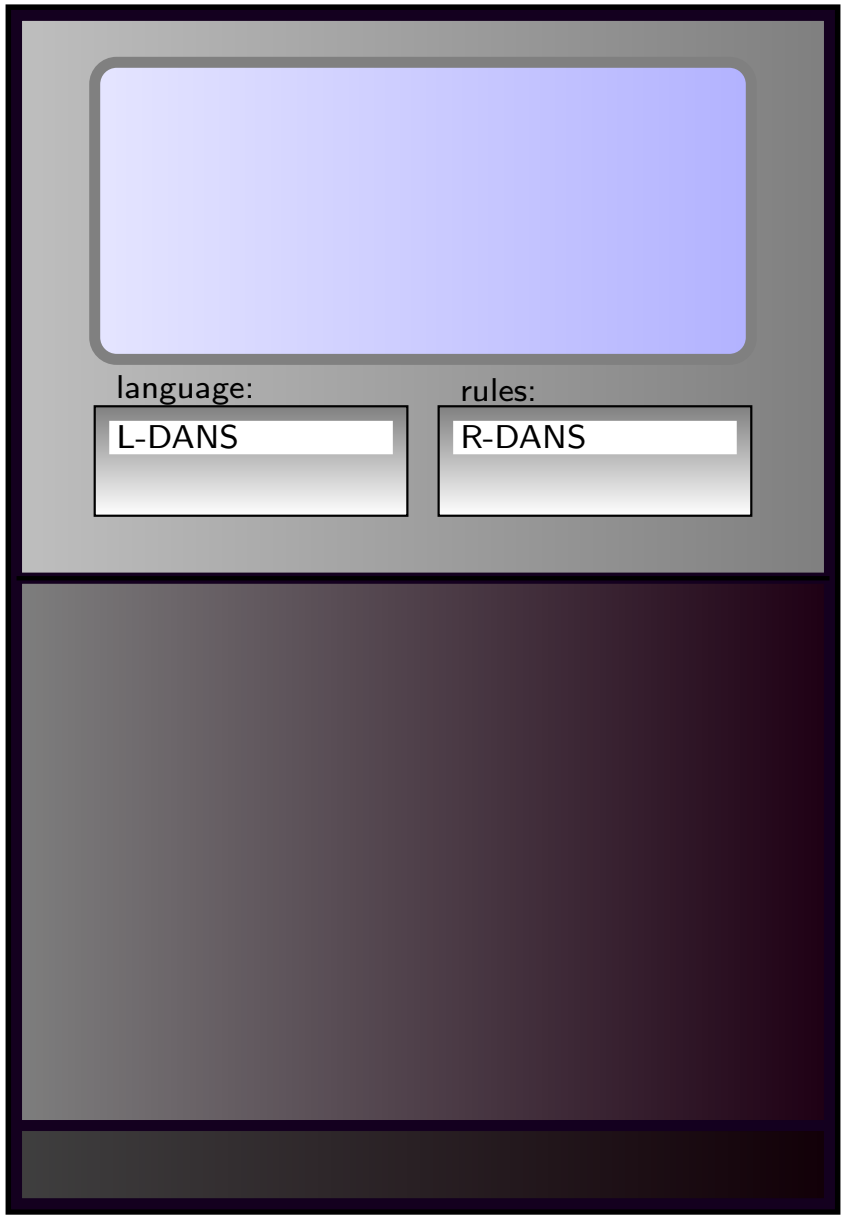
Proof checking with the machine

- Programming our machine is done in metalanguage



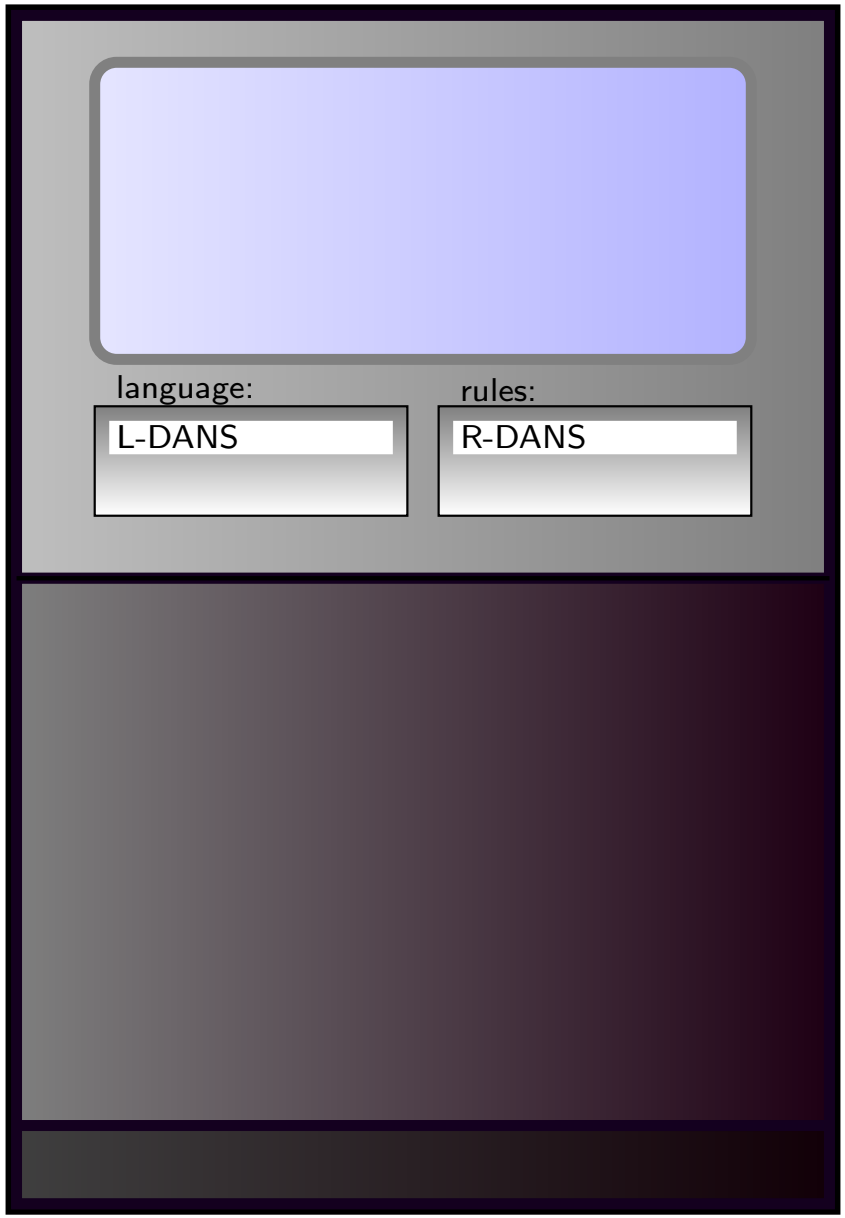
Proof checking with the machine

- Programming our machine is done in metalanguage
- We do not specify this metalanguage for now



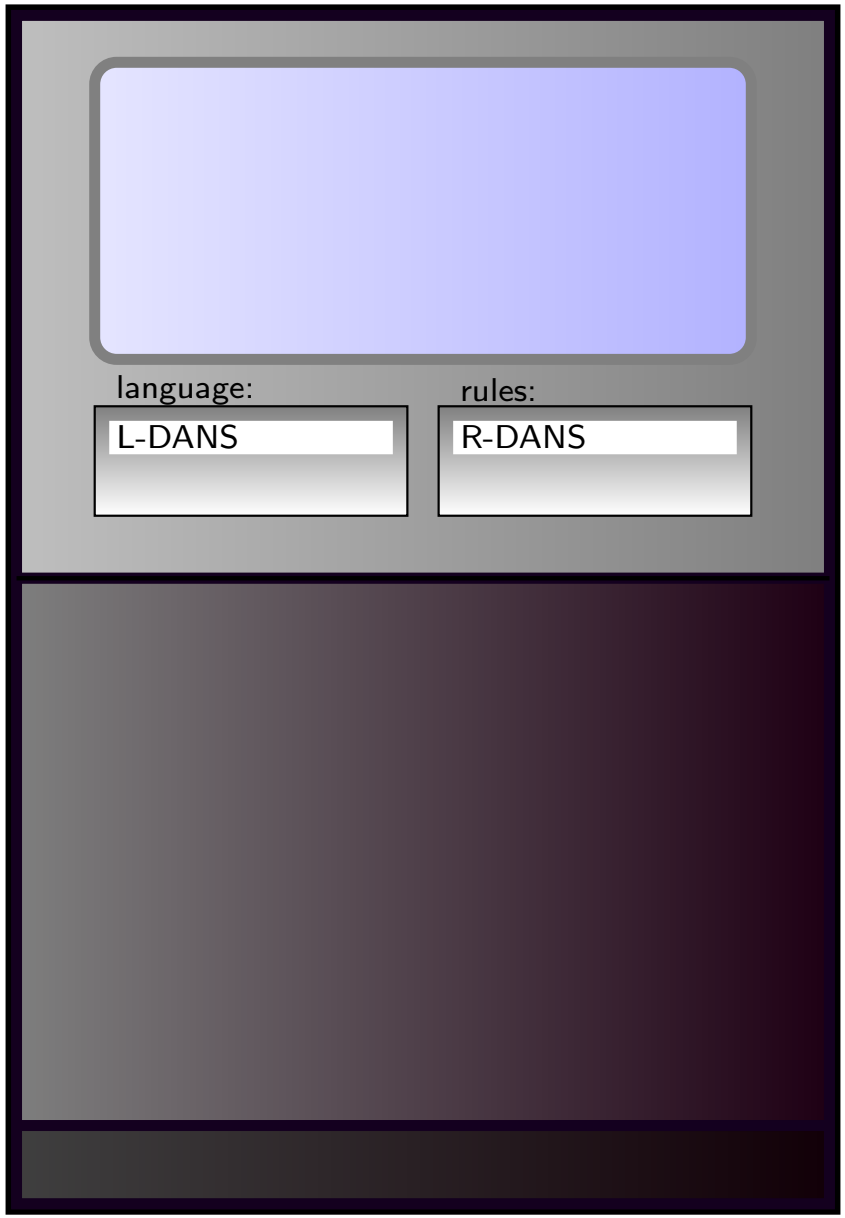
Proof checking with the machine

- Programming our machine is done in metalanguage
- We do not specify this metalanguage for now
- (many programming languages could be used, ML in particular)



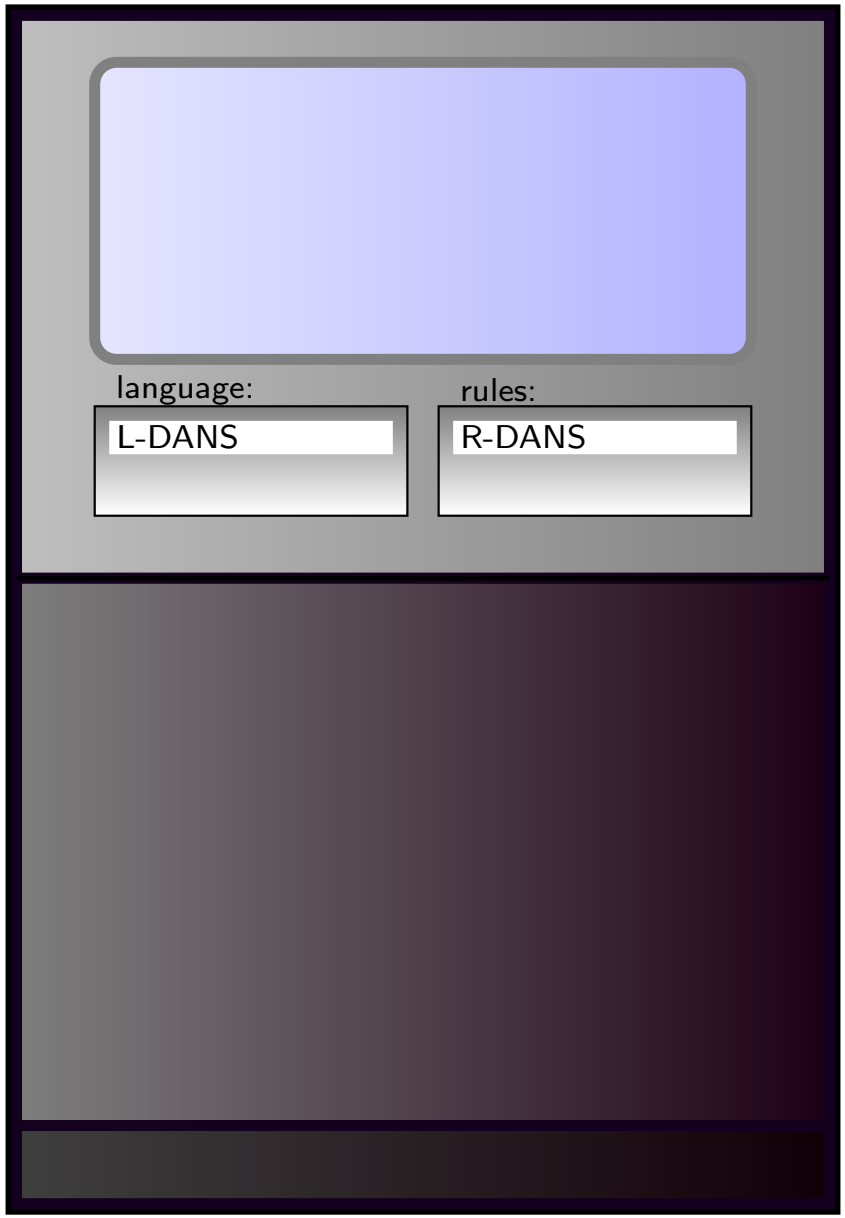
Proof checking with the machine

- Programming our machine is done in metalanguage
- We do not specify this metalanguage for now
- (many programming languages could be used, ML in particular)
- The bottom half is the 'metalevel'



Proof checking with the machine

- Programming our machine is done in metalanguage
- We do not specify this metalanguage for now
- (many programming languages could be used, ML in particular)
- The bottom half is the 'metalevel'
- First thing we build is *proof checking*

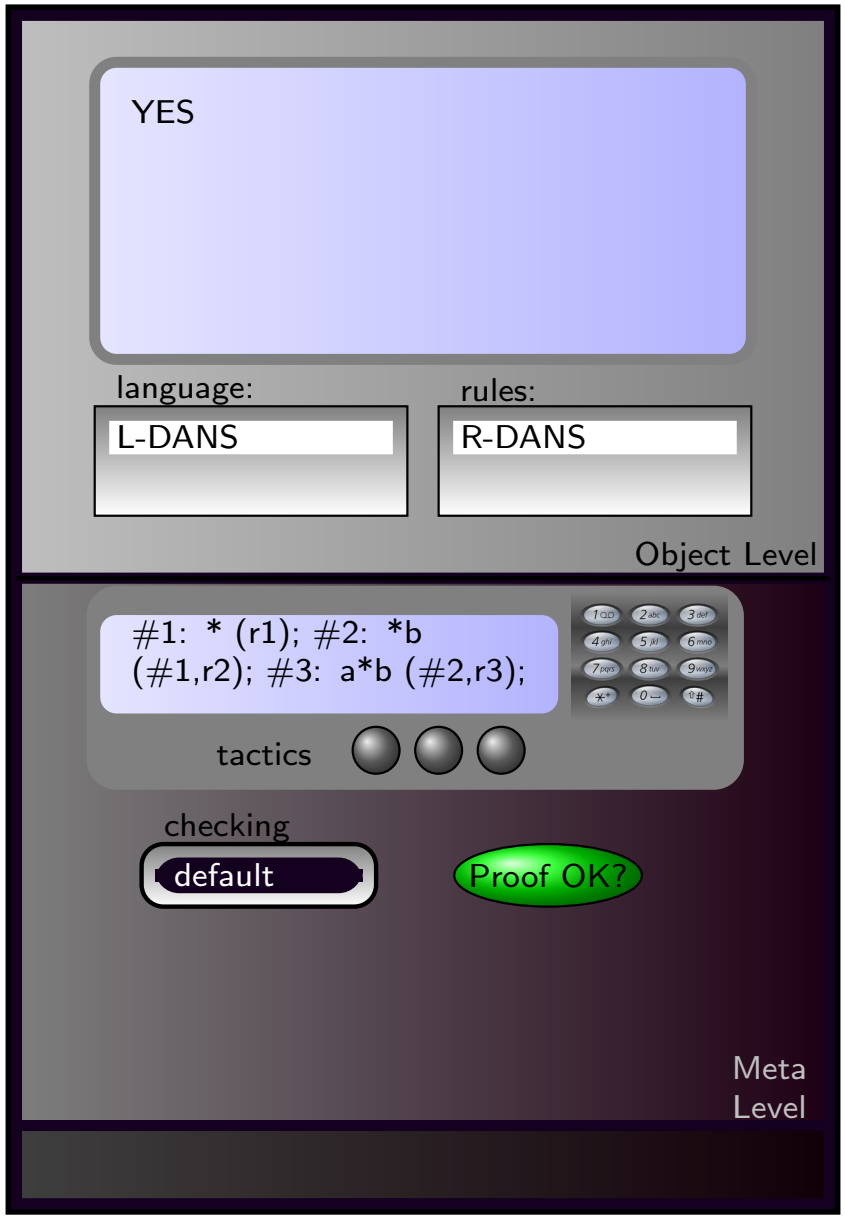


Proof checking with the machine

- Programming our machine is done in metalanguage
- We do not specify this metalanguage for now
- (many programming languages could be used, ML in particular)
- The bottom half is the 'metalevel'
- First thing we build is *proof checking*

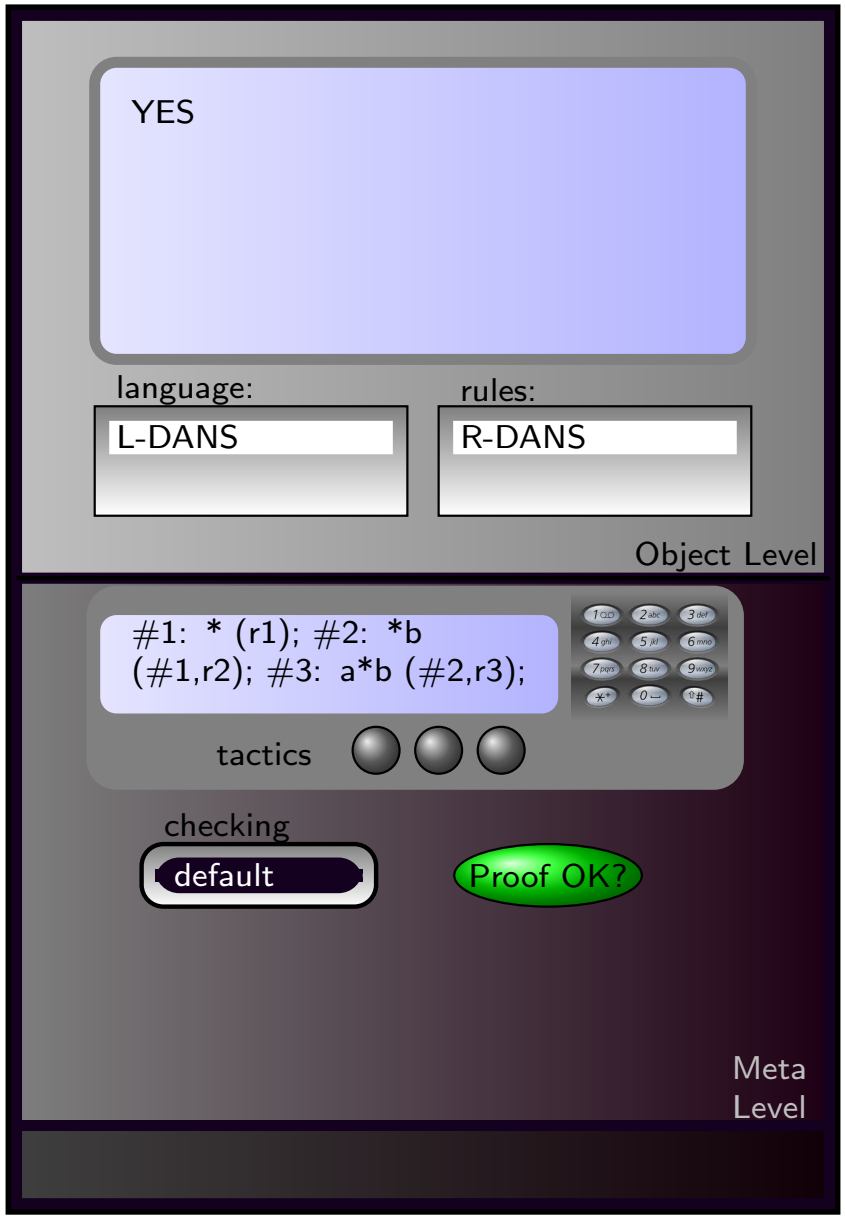
Proof checking with the machine

- Programming our machine is done in metalanguage
- We do not specify this metalanguage for now
- (many programming languages could be used, ML in particular)
- The bottom half is the 'metalevel'
- First thing we build is *proof checking*
- If we enter the proof object for a*b and click on the green button, the machine will check the proof and answer YES



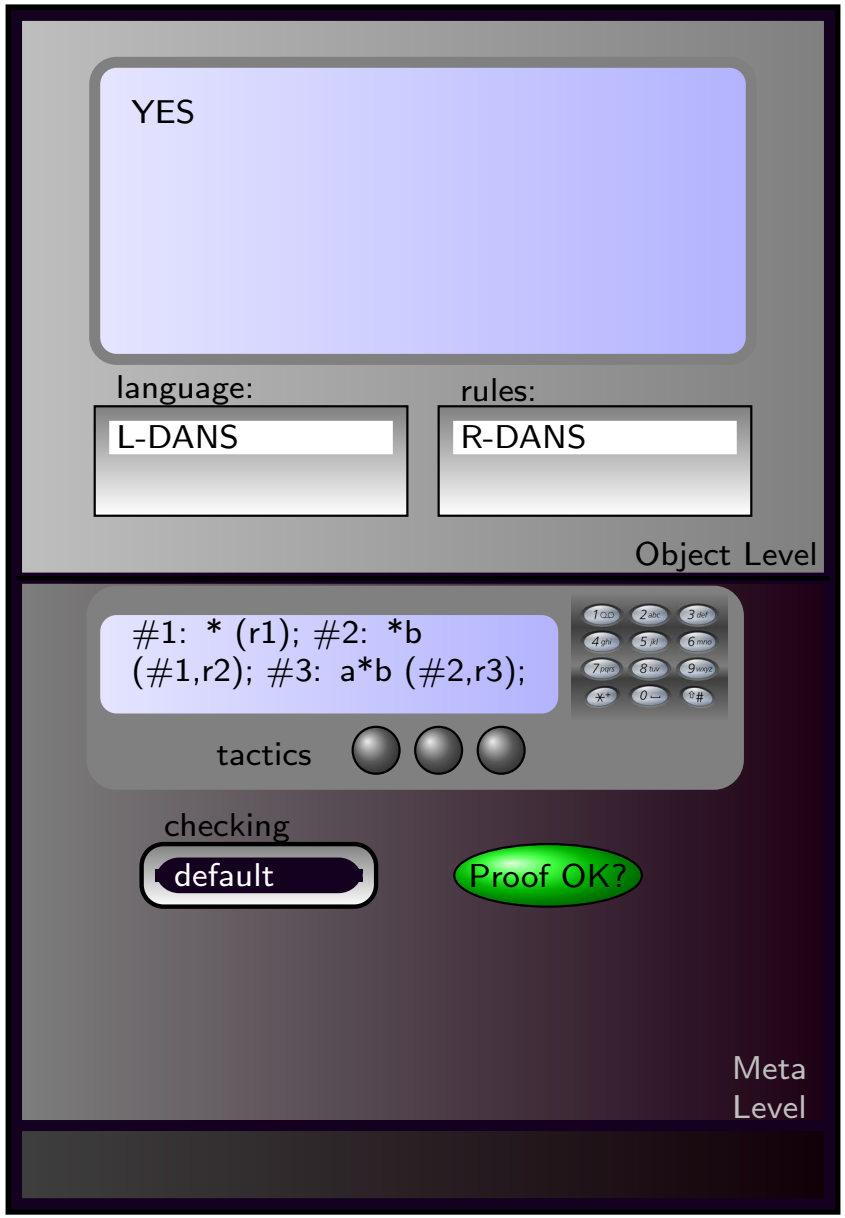
Proof checking algorithms

- Notice that the proof checking algorithm is marked default



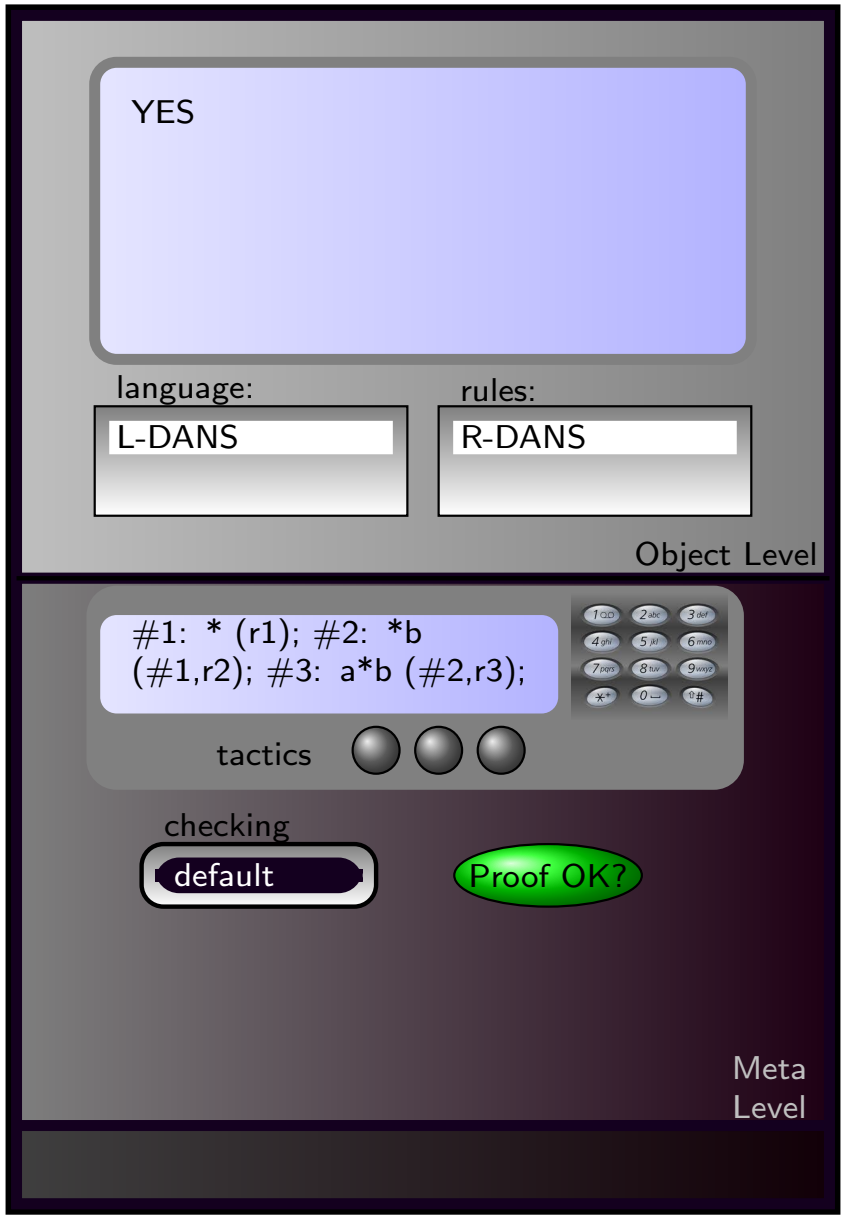
Proof checking algorithms

- Notice that the proof checking algorithm is marked default
- DANS is a very simple system, so default checking is OK



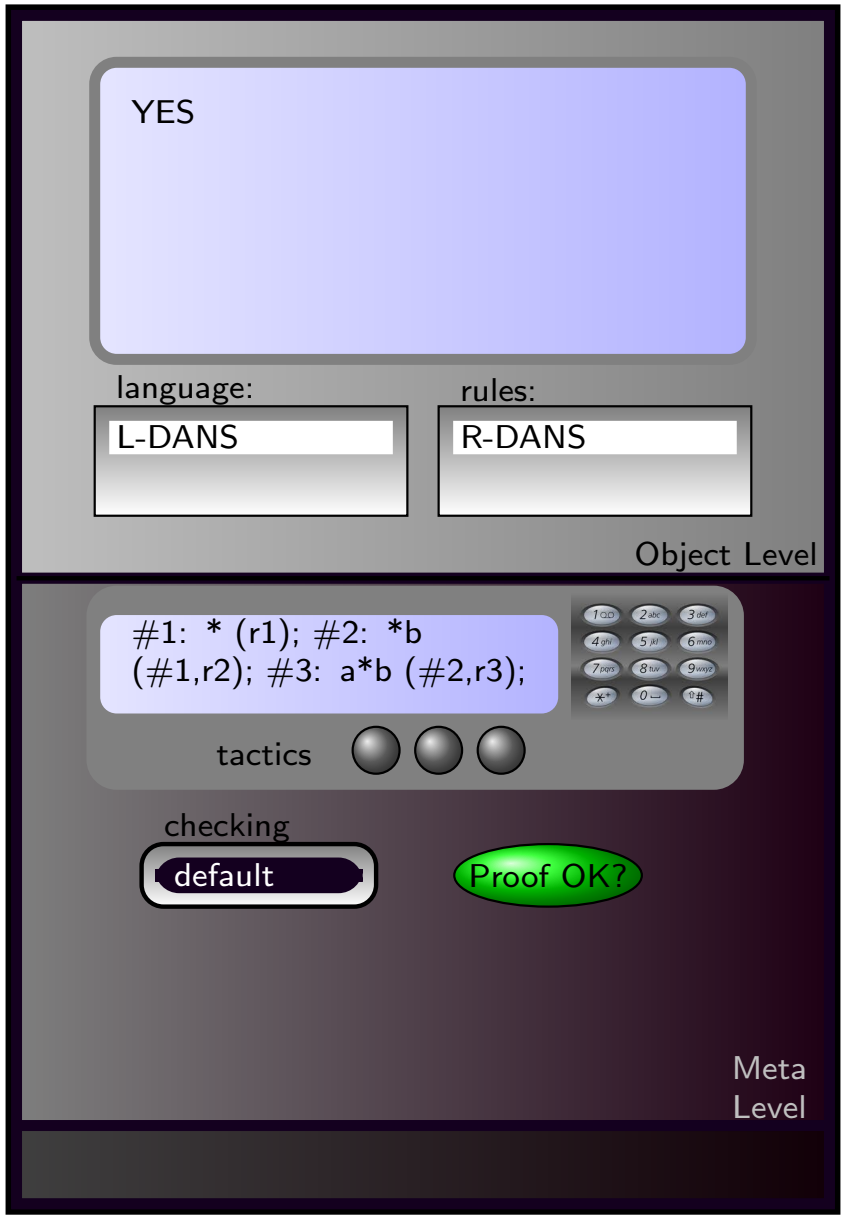
Proof checking algorithms

- Notice that the proof checking algorithm is marked default
- DANS is a very simple system, so default checking is OK
- Default checking: for each step, just check that the rule in parentheses has been applied correctly



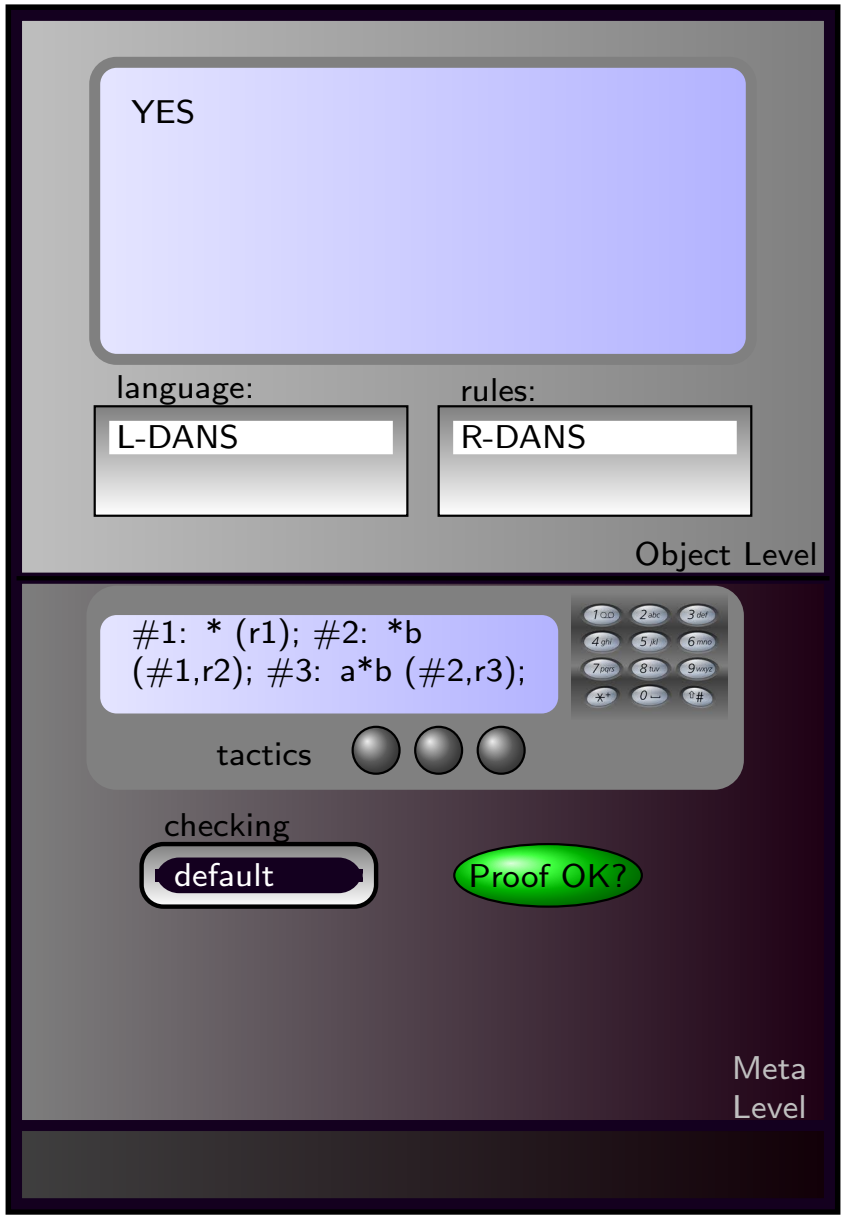
Proof checking algorithms

- Notice that the proof checking algorithm is marked default
- DANS is a very simple system, so default checking is OK
- Default checking: for each step, just check that the rule in parentheses has been applied correctly
- For more complicated formal systems, proof checking is more sophisticated



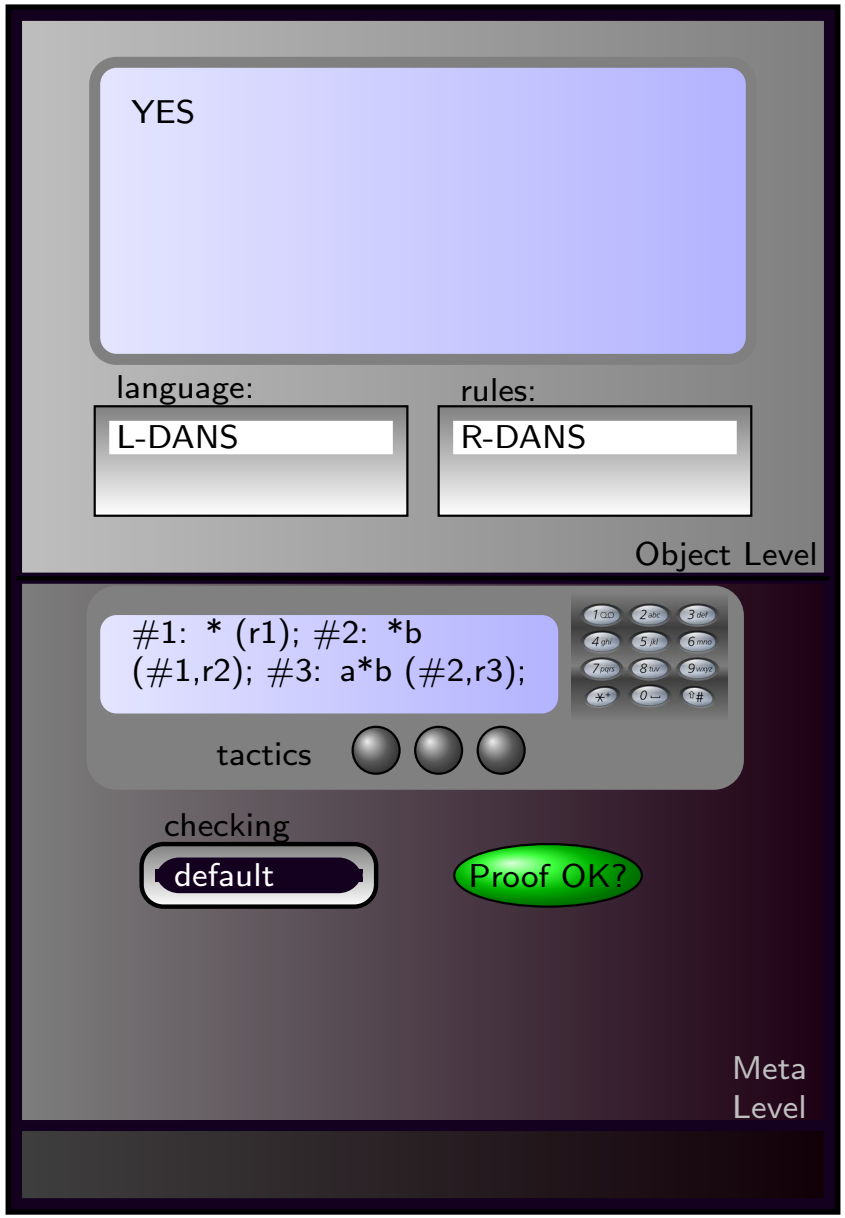
Proof checking algorithms

- Notice that the proof checking algorithm is marked default
- DANS is a very simple system, so default checking is OK
- Default checking: for each step, just check that the rule in parentheses has been applied correctly
- For more complicated formal systems, proof checking is more sophisticated
- Metamathematical studies of the rules may show that some rules are redundant or that more efficient helper rules can be introduced



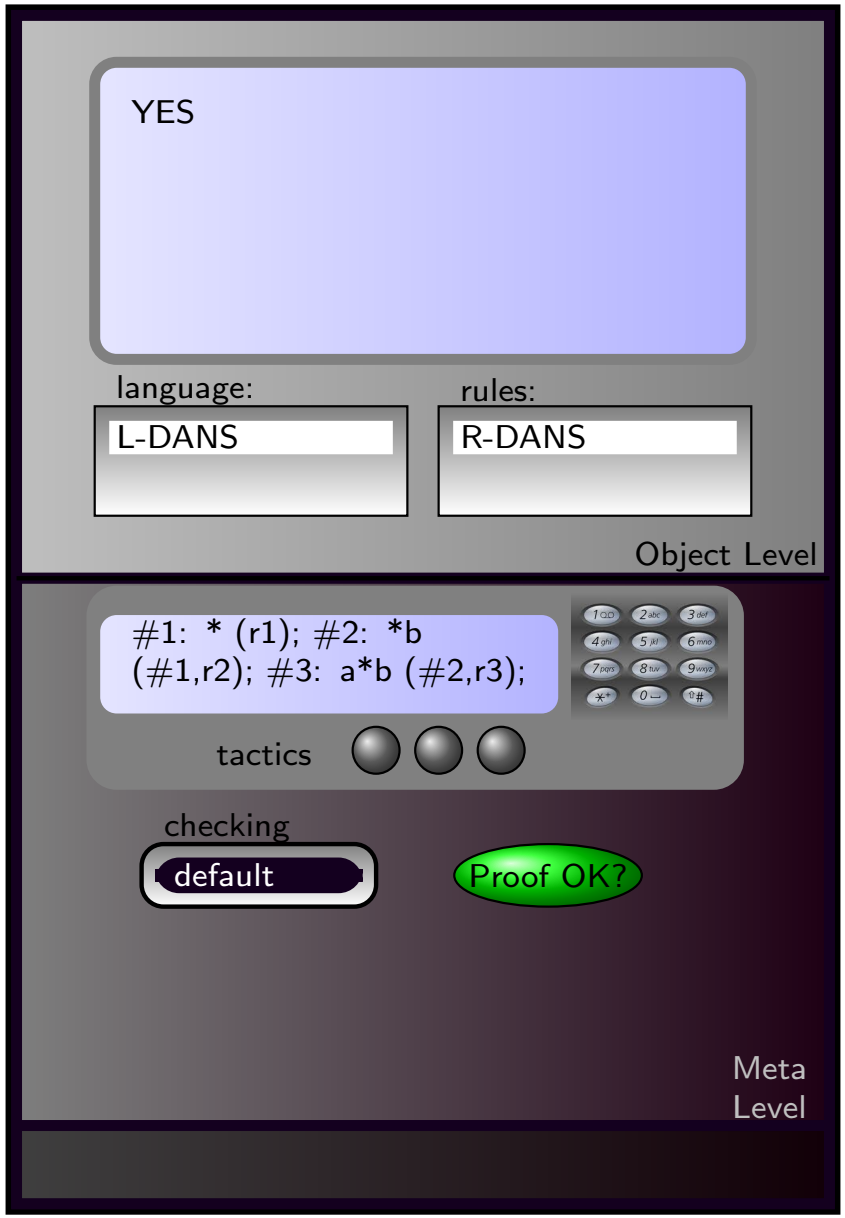
Proof construction with tactics

- Notice the *tactics* buttons



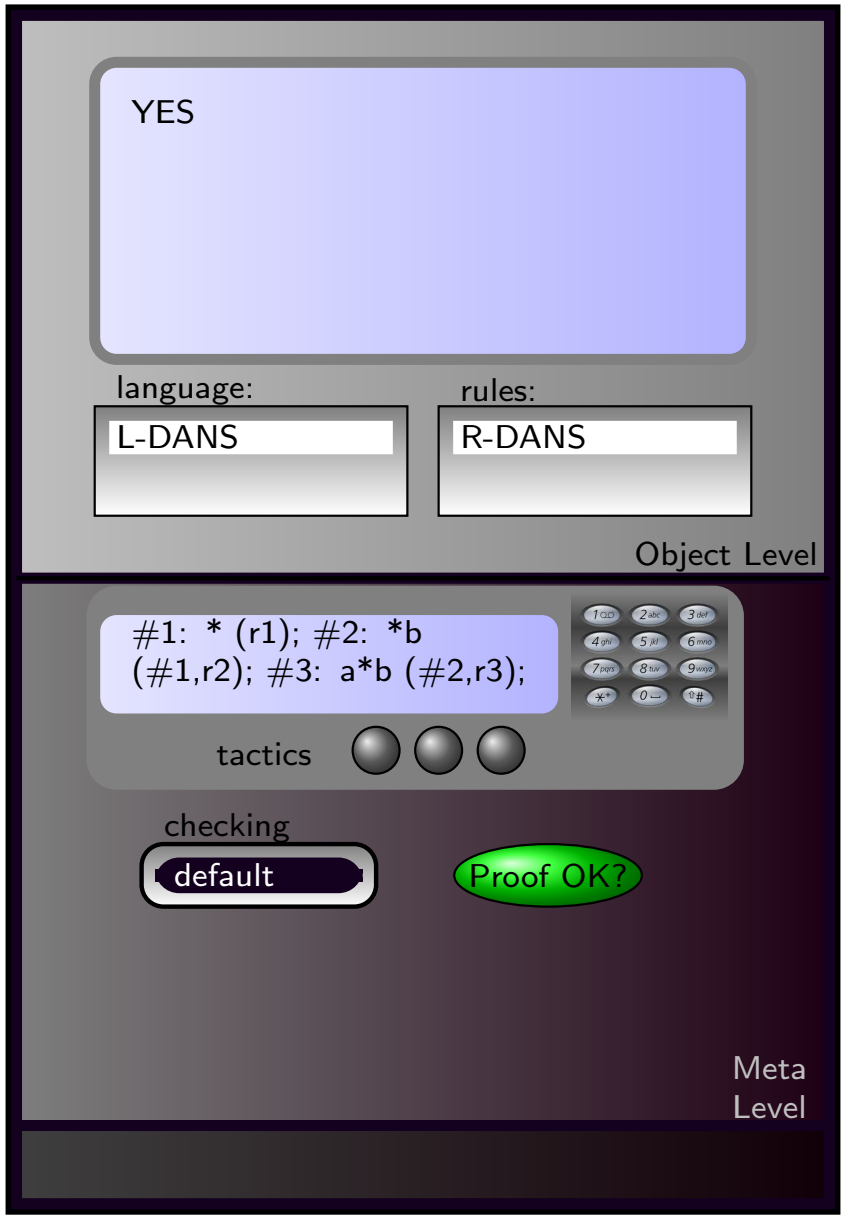
Proof construction with tactics

- Notice the *tactics* buttons
- Some of the most powerful machines are used interactively



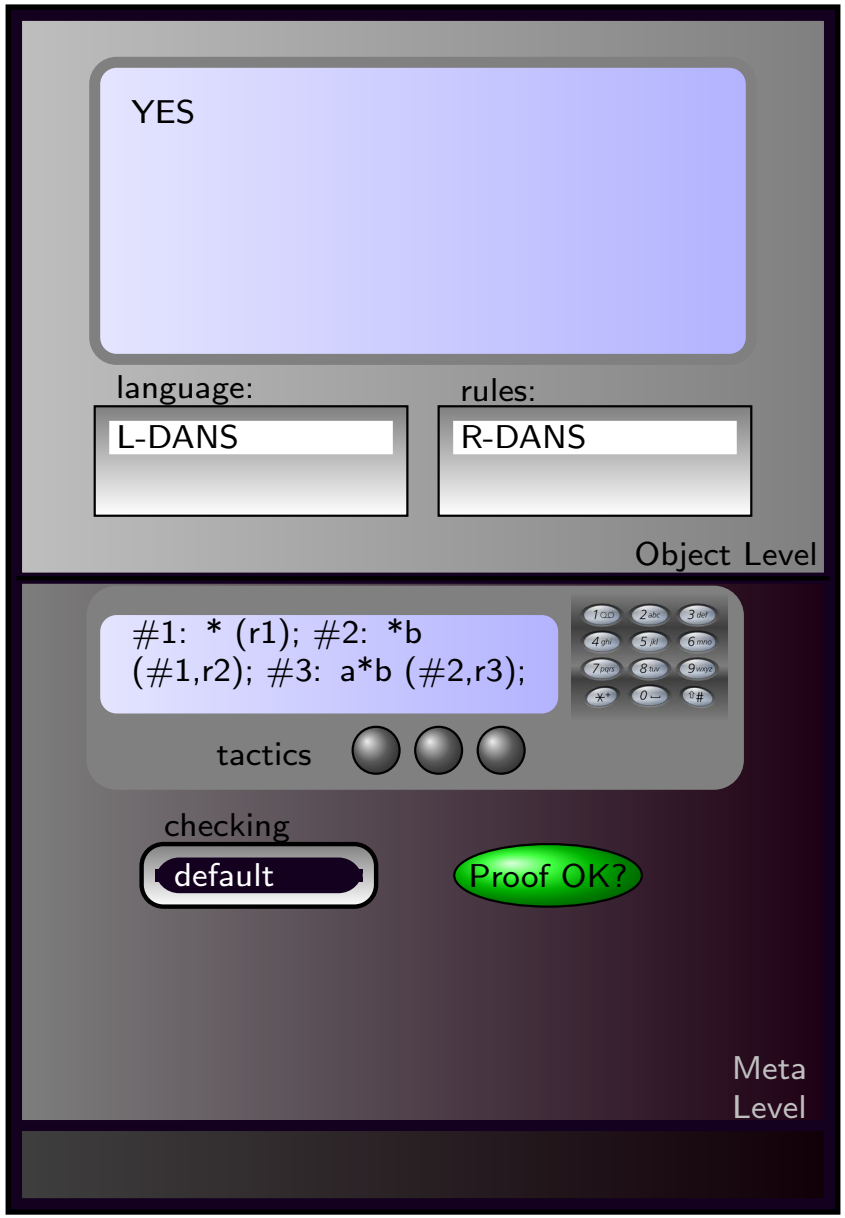
Proof construction with tactics

- Notice the *tactics* buttons
- Some of the most powerful machines are used interactively
- The built-in tactics help you build the proof in the input area



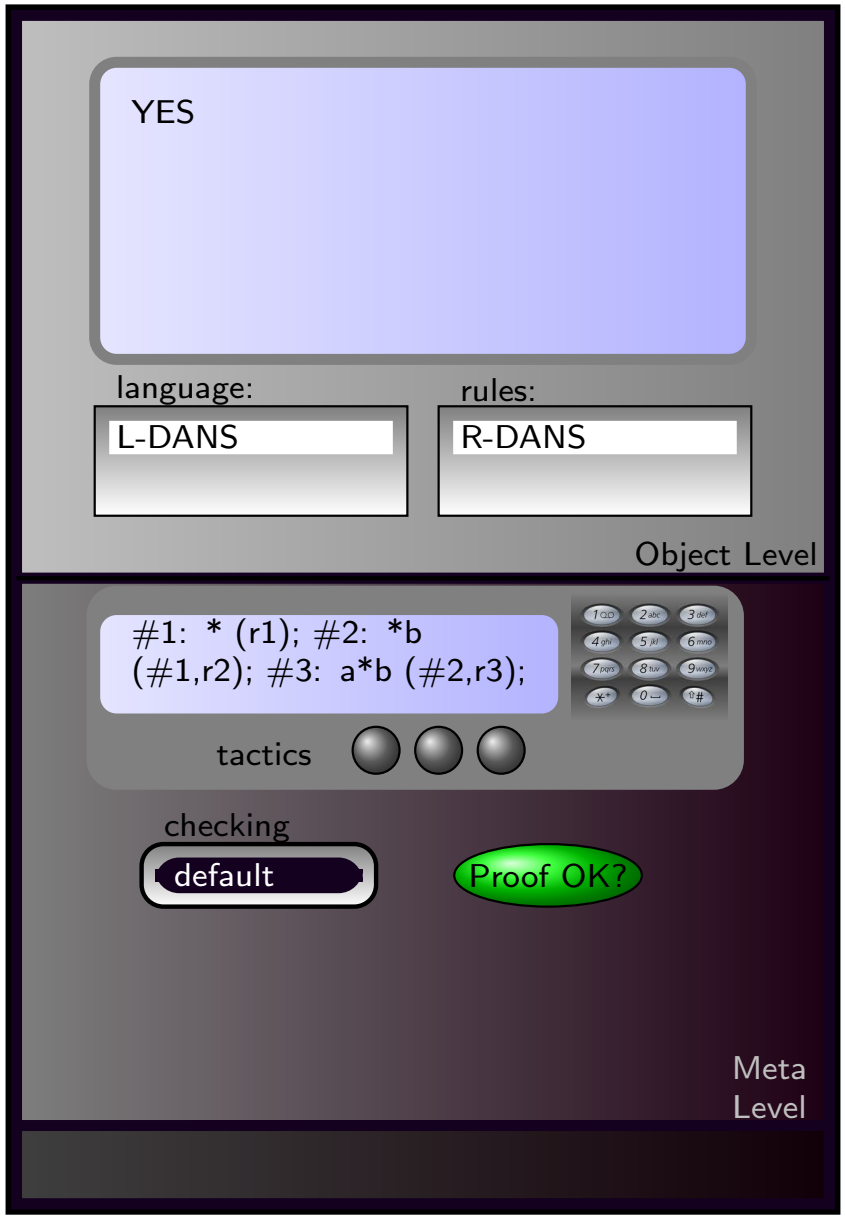
Proof construction with tactics

- Notice the *tactics* buttons
- Some of the most powerful machines are used interactively
- The built-in tactics help you build the proof in the input area
- These tactics pack a lot of punch



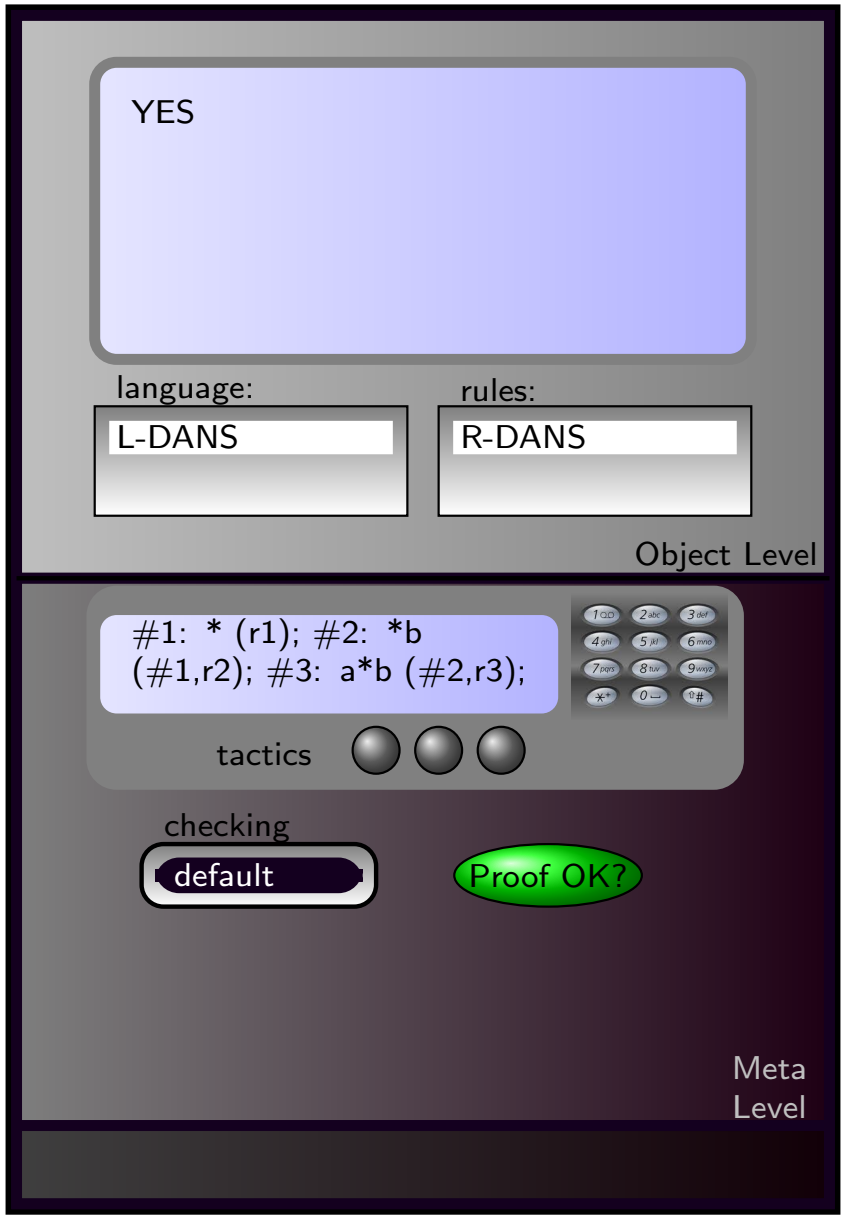
Proof construction with tactics

- Notice the *tactics* buttons
- Some of the most powerful machines are used interactively
- The built-in tactics help you build the proof in the input area
- These tactics pack a lot of punch
- Many provers allow you to write your own tactics in specialized 'tactics languages'



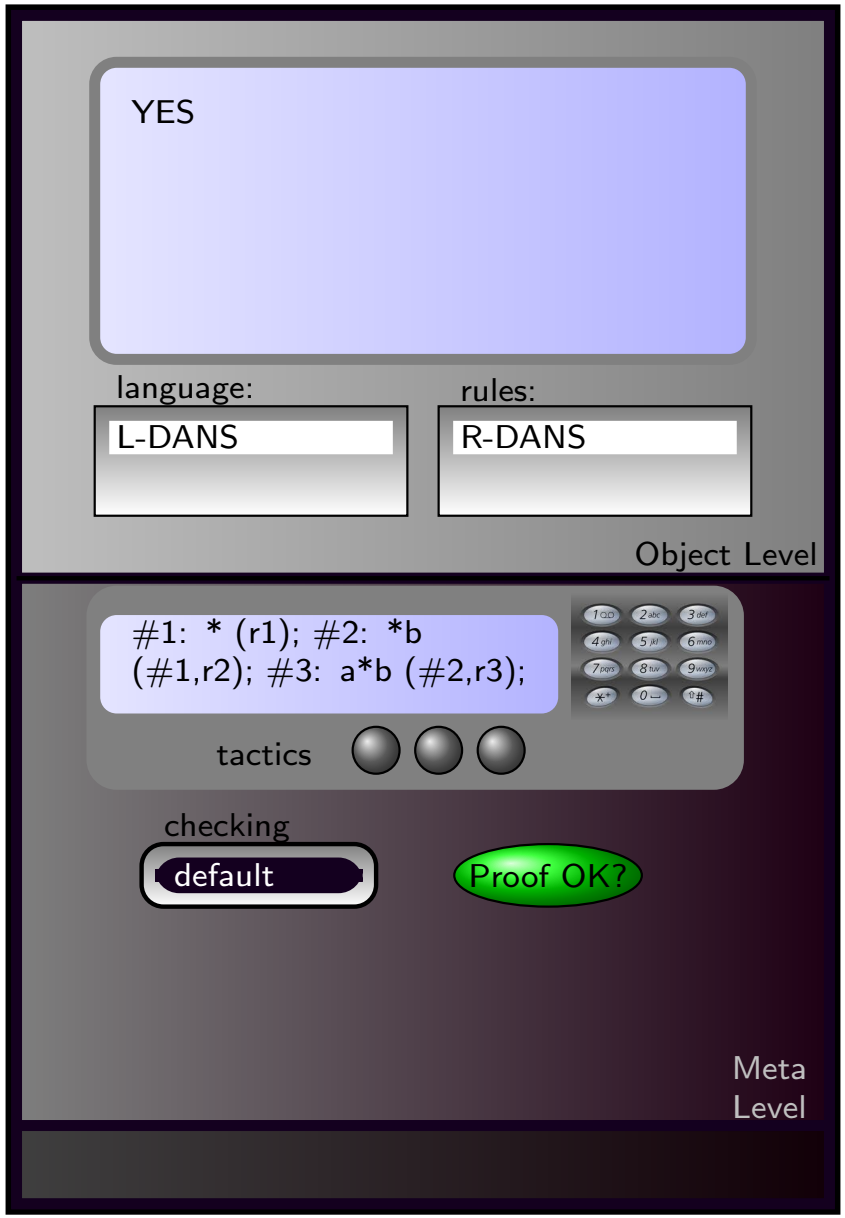
Proof construction with tactics

- Notice the *tactics* buttons
- Some of the most powerful machines are used interactively
- The built-in tactics help you build the proof in the input area
- These tactics pack a lot of punch
- Many provers allow you to write your own tactics in specialized 'tactics languages'
- (instead of writing them in the metalanguage)



Proof construction with tactics

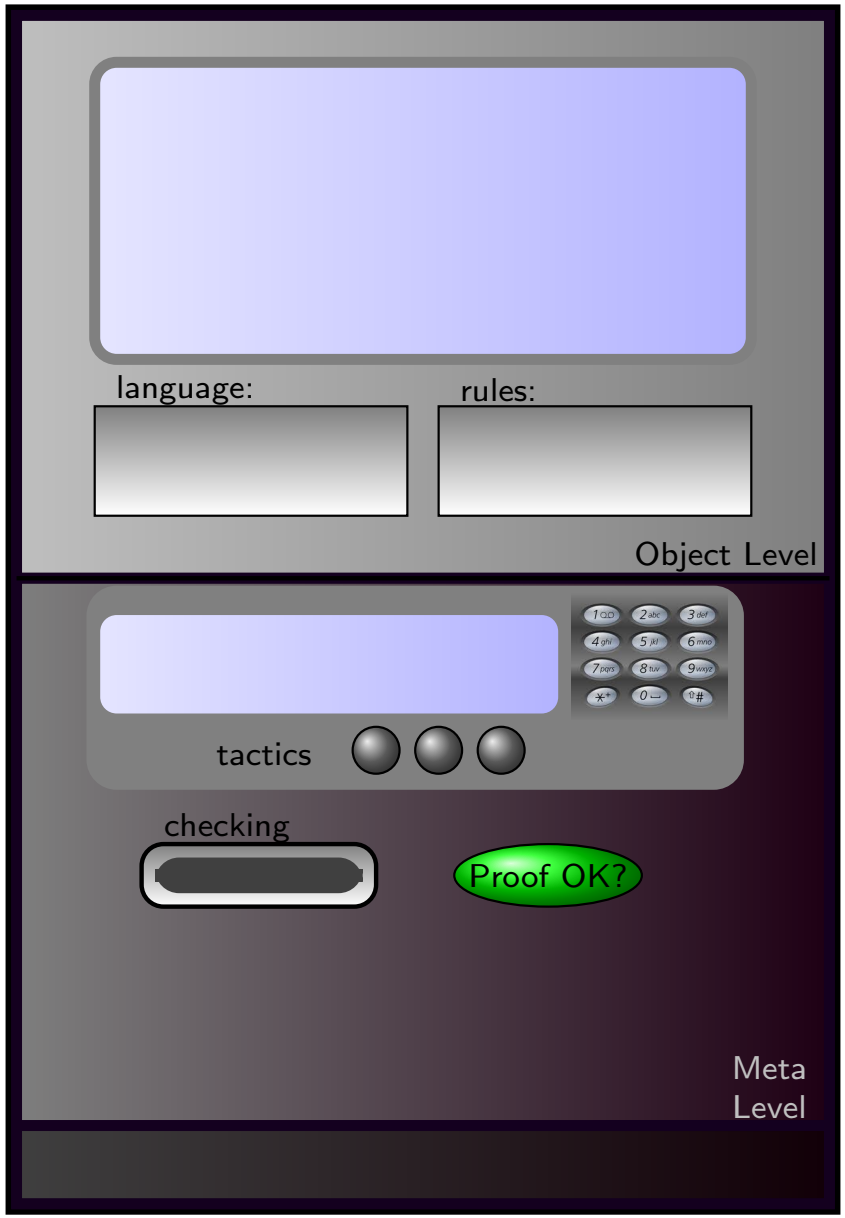
- Notice the *tactics* buttons
- Some of the most powerful machines are used interactively
- The built-in tactics help you build the proof in the input area
- These tactics pack a lot of punch
- Many provers allow you to write your own tactics in specialized 'tactics languages'
- (instead of writing them in the metalanguage)
- We will use this facility when we study Coq and Isabelle in the 'Interactive Provers' course



Asking the machine if a formula is a theorem

Asking the machine if a formula is a theorem

- Can we just enter a wff (not a proof) in the input area and click some button that will answer YES if the wff is a theorem?



Asking the machine if a formula is a theorem

- Can we just enter a wff (not a proof) in the input area and click some button that will answer YES if the wff is a theorem?
- The short answer is yes, recall that all formal systems are rigged in a special way, precisely so that they can do this by default

Asking the machine if a formula is a theorem

- Can we just enter a wff (not a proof) in the input area and click some button that will answer YES if the wff is a theorem?
- The short answer is yes, recall that all formal systems are rigged in a special way, precisely so that they can do this by default
- The default algorithm for checking whether a wff is a theorem is to let the machine run and record all theorems; if the wff is a theorem, eventually the machine will match it and output YES

Is this default strategy acceptable?

Is this default strategy acceptable?

- With the default strategy, if a wff is a theorem, we have no clue how long the machine might take to prove it and ...

Is this default strategy acceptable?

- With the default strategy, if a wff is a theorem, we have no clue how long the machine might take to prove it and . . .
- . . . *if the wff is not a theorem, the machine will run forever*

Is this default strategy acceptable?

- With the default strategy, if a wff is a theorem, we have no clue how long the machine might take to prove it and . . .
- . . . *if the wff is not a theorem, the machine will run forever*
- The default strategy knows provability but it cannot determine that a wff is *not provable* and stop

Is this default strategy acceptable?

- With the default strategy, if a wff is a theorem, we have no clue how long the machine might take to prove it and . . .
- . . . *if the wff is not a theorem, the machine will run forever*
- The default strategy knows provability but it cannot determine that a wff is *not provable* and stop
- For certain formal systems, no matter what proving strategy you use, you cannot avoid this limitation

Is this default strategy acceptable?

- With the default strategy, if a wff is a theorem, we have no clue how long the machine might take to prove it and ...
- ... *if the wff is not a theorem, the machine will run forever*
- The default strategy knows provability but it cannot determine that a wff is *not provable* and stop
- For certain formal systems, no matter what proving strategy you use, you cannot avoid this limitation
- The reason is that the logic of the system is too powerful to allow us to handle 'non-provability'

Is this default strategy acceptable?

- With the default strategy, if a wff is a theorem, we have no clue how long the machine might take to prove it and . . .
- . . . *if the wff is not a theorem, the machine will run forever*
- The default strategy knows provability but it cannot determine that a wff is *not provable* and stop
- For certain formal systems, no matter what proving strategy you use, you cannot avoid this limitation
- The reason is that the logic of the system is too powerful to allow us to handle 'non-provability'
- We say that the system is *undecidable*; we saw that DANS is a decidable system, but DANS is simple

Undecidability and software

Undecidability and software

- So there is this teeter-toter between expressiveness (i.e. power of the formal system) and its decidability

Undecidability and software

- So there is this teeter-toter between expressiveness (i.e. power of the formal system) and its decidability
- Let's say you design a very expressive language which would allow you to prove certain properties of your programs

Undecidability and software

- So there is this teeter-toter between expressiveness (i.e. power of the formal system) and its decidability
- Let's say you design a very expressive language which would allow you to prove certain properties of your programs
- Would it be OK for the compiler of this language to hang on certain programs?

Undecidability and software

- So there is this teeter-toter between expressiveness (i.e. power of the formal system) and its decidability
- Let's say you design a very expressive language which would allow you to prove certain properties of your programs
- Would it be OK for the compiler of this language to hang on certain programs?
- This sounds unacceptable, but let's look at it differently

Undecidability and software

- So there is this teeter-toter between expressiveness (i.e. power of the formal system) and its decidability
- Let's say you design a very expressive language which would allow you to prove certain properties of your programs
- Would it be OK for the compiler of this language to hang on certain programs?
- This sounds unacceptable, but let's look at it differently
  - The compiler of a logically-stronger language hangs because it cannot understand your program

Undecidability and software

- So there is this teeter-toter between expressiveness (i.e. power of the formal system) and its decidability
- Let's say you design a very expressive language which would allow you to prove certain properties of your programs
- Would it be OK for the compiler of this language to hang on certain programs?
- This sounds unacceptable, but let's look at it differently
  - The compiler of a logically-stronger language hangs because it cannot understand your program
  - The compiler passes your code in a weaker logic, but the system crashes in the field

Undecidability and software

- So there is this teeter-toter between expressiveness (i.e. power of the formal system) and its decidability
- Let's say you design a very expressive language which would allow you to prove certain properties of your programs
- Would it be OK for the compiler of this language to hang on certain programs?
- This sounds unacceptable, but let's look at it differently
  - The compiler of a logically-stronger language hangs because it cannot understand your program
  - The compiler passes your code in a weaker logic, but the system crashes in the field

Undecidability and software

- So there is this teeter-toter between expressiveness (i.e. power of the formal system) and its decidability
- Let's say you design a very expressive language which would allow you to prove certain properties of your programs
- Would it be OK for the compiler of this language to hang on certain programs?
- This sounds unacceptable, but let's look at it differently
  - The compiler of a logically-stronger language hangs because it cannot understand your program
  - The compiler passes your code in a weaker logic, but the system crashes in the field

  You will hopefully see through examples that undecidable type-checking in a stronger language is sometimes acceptable (you just press Ctrl-C to kill the compiler and rethink your code); it's better if the compiler hangs than if your code hangs in the field

Automatic provers

Automatic provers

- Automatic provers have auto strategies that replace the default strategy

Automatic provers

- Automatic provers have auto strategies that replace the default strategy
- They are based on *first-order logic* (FOL)

Automatic provers

- Automatic provers have auto strategies that replace the default strategy
- They are based on *first-order logic* (FOL)
- These auto strategies use semantics

Automatic provers

- Automatic provers have auto strategies that replace the default strategy
- They are based on *first-order logic* (FOL)
- These auto strategies use semantics
- We'll study FOL and various auto strategies in the 'Main Concepts of Logic' course

Automatic provers

- Automatic provers have auto strategies that replace the default strategy
- They are based on *first-order logic* (FOL)
- These auto strategies use semantics
- We'll study FOL and various auto strategies in the 'Main Concepts of Logic' course
- *Resolution* is the most used auto strategy

Decision procedures

Decision procedures

- Let's look again at the stronger question: is a wff a theorem *or not*?

Decision procedures

- Let's look again at the stronger question: is a wff a theorem *or not*?
- A procedure that provides that answer, when it exists, is called a *decision procedure*

Decision procedures

- Let's look again at the stronger question: is a wff a theorem *or not*?
- A procedure that provides that answer, when it exists, is called a *decision procedure*
- Mostly very special theories based on FOL admit such procedures

Decision procedures

- Let's look again at the stronger question: is a wff a theorem *or not*?
- A procedure that provides that answer, when it exists, is called a *decision procedure*
- Mostly very special theories based on FOL admit such procedures
- Decision procedures are the basis of Satisfiability Modulo Theories (SMT) solvers

Decision procedures

- Let's look again at the stronger question: is a wff a theorem *or not*?
- A procedure that provides that answer, when it exists, is called a *decision procedure*
- Mostly very special theories based on FOL admit such procedures
- Decision procedures are the basis of Satisfiability Modulo Theories (SMT) solvers
- Although SMT solvers are very powerful . . .

Decision procedures

- Let's look again at the stronger question: is a wff a theorem *or not*?
- A procedure that provides that answer, when it exists, is called a *decision procedure*
- Mostly very special theories based on FOL admit such procedures
- Decision procedures are the basis of Satisfiability Modulo Theories (SMT) solvers
- Although SMT solvers are very powerful . . .
- . . . these special theories are not sufficiently expressive to cover all the software development needs

The question of meaning

The question of meaning

- So, are formal systems just meaningless games?

The question of meaning

- So, are formal systems just meaningless games?
- Yes (meaning is a separate issue)

The question of meaning

- So, are formal systems just meaningless games?
- Yes (meaning is a separate issue)
- *But formal systems are rarely born as meaningless games*

The question of meaning

- So, are formal systems just meaningless games?
- Yes (meaning is a separate issue)
- *But formal systems are rarely born as meaningless games*
- All important formal systems were obtained by *abstracting rules from meaningful domains*

The question of meaning

- So, are formal systems just meaningless games?
- Yes (meaning is a separate issue)
- *But formal systems are rarely born as meaningless games*
- All important formal systems were obtained by *abstracting rules from meaningful domains*
- All important formal systems have an *intended interpretation/model*

The question of meaning

- So, are formal systems just meaningless games?
- Yes (meaning is a separate issue)
- *But formal systems are rarely born as meaningless games*
- All important formal systems were obtained by *abstracting rules from meaningful domains*
- All important formal systems have an *intended interpretation/model*
- Arithmetic, set theory, all sorts of geometries,...

What about DANS?

What about DANS?

- I had an intended interpretation (model) in mind

What about DANS?

- I had an intended interpretation (model) in mind
- When I was a young boy, there were always more girls than boys at a dance club (DANS is a misspelling of dance)

What about DANS?

- I had an intended interpretation (model) in mind
- When I was a young boy, there were always more girls than boys at a dance club (DANS is a misspelling of dance)
- (apparently, things have changed quite a bit)

What about DANS?

- I had an intended interpretation (model) in mind
- When I was a young boy, there were always more girls than boys at a dance club (DANS is a misspelling of dance)
- (apparently, things have changed quite a bit)
- So, * is the door to the dance floor, a is a girl, b is a boy

What about DANS?

- I had an intended interpretation (model) in mind
- When I was a young boy, there were always more girls than boys at a dance club (DANS is a misspelling of dance)
- (apparently, things have changed quite a bit)
- So, * is the door to the dance floor, a is a girl, b is a boy
- The formation rules say that boys must line up on the left, girls on the right, no mix up in the lines

What about DANS?

- I had an intended interpretation (model) in mind
- When I was a young boy, there were always more girls than boys at a dance club (DANS is a misspelling of dance)
- (apparently, things have changed quite a bit)
- So, * is the door to the dance floor, a is a girl, b is a boy
- The formation rules say that boys must line up on the left, girls on the right, no mix up in the lines
- The first rule says that an empty club is OK

What about DANS?

- I had an intended interpretation (model) in mind
- When I was a young boy, there were always more girls than boys at a dance club (DANS is a misspelling of dance)
- (apparently, things have changed quite a bit)
- So, * is the door to the dance floor, a is a girl, b is a boy
- The formation rules say that boys must line up on the left, girls on the right, no mix up in the lines
- The first rule says that an empty club is OK
- The second rule says that boys can arrive at anytime, third rules ensures that there is at least a boy waiting when a girl arrives

What about DANS?

- I had an intended interpretation (model) in mind
- When I was a young boy, there were always more girls than boys at a dance club (DANS is a misspelling of dance)
- (apparently, things have changed quite a bit)
- So, * is the door to the dance floor, a is a girl, b is a boy
- The formation rules say that boys must line up on the left, girls on the right, no mix up in the lines
- The first rule says that an empty club is OK
- The second rule says that boys can arrive at anytime, third rules ensures that there is at least a boy waiting when a girl arrives
- The fourth rule says . . . well, she accepted the invitation

The balancing act of the rules

The balancing act of the rules

- DANS is very simple, anything more complicated would have lenghtened this overview

The balancing act of the rules

- DANS is very simple, anything more complicated would have lenghtened this overview
- You should pick up some situation that interests you and try to make a formal system for it

The balancing act of the rules

- DANS is very simple, anything more complicated would have lenghtened this overview
- You should pick up some situation that interests you and try to make a formal system for it
- Don't choose only rules that make longer sentences, have some shortening rules too

The balancing act of the rules

- DANS is very simple, anything more complicated would have lenghtened this overview
- You should pick up some situation that interests you and try to make a formal system for it
- Don't choose only rules that make longer sentences, have some shortening rules too
- What makes a system interesting is the delicate balance between the two kinds of rules

The balancing act of the rules

- DANS is very simple, anything more complicated would have lenghtened this overview
- You should pick up some situation that interests you and try to make a formal system for it
- Don't choose only rules that make longer sentences, have some shortening rules too
- What makes a system interesting is the delicate balance between the two kinds of rules
- You will see how hard it is to come up with an interesting system

The balancing act of the rules

- DANS is very simple, anything more complicated would have lenghtened this overview
- You should pick up some situation that interests you and try to make a formal system for it
- Don't choose only rules that make longer sentences, have some shortening rules too
- What makes a system interesting is the delicate balance between the two kinds of rules
- You will see how hard it is to come up with an interesting system
- You will see that it is easier to cook up a system if you have an intended interpretation

The balancing act of the rules

The balancing act of the rules

- It is easier if you pick a property of the formulas that does not change as rules get applied

The balancing act of the rules

- It is easier if you pick a property of the formulas that does not change as rules get applied
- That property will give you a decision procedure

The balancing act of the rules

- It is easier if you pick a property of the formulas that does not change as rules get applied
- That property will give you a decision procedure
- Warning: games are difficult, try them only after you gain some experience

The balancing act of the rules

- It is easier if you pick a property of the formulas that does not change as rules get applied
- That property will give you a decision procedure
- Warning: games are difficult, try them only after you gain some experience
- Alternative: take DANS and make it more interesting, by adding more language and more rules

An extension of our view of a formal system

An extension of our view of a formal system

- We defined a formal system as a language and a set of rules of inference for defining a subset of sentences, the theorems

An extension of our view of a formal system

- We defined a formal system as a language and a set of rules of inference for defining a subset of sentences, the theorems
- We saw that a formal system may admit a decision procedure

An extension of our view of a formal system

- We defined a formal system as a language and a set of rules of inference for defining a subset of sentences, the theorems
- We saw that a formal system may admit a decision procedure
- A metamathematical proof that a decision procedure exists is highly non-trivial

An extension of our view of a formal system

- We defined a formal system as a language and a set of rules of inference for defining a subset of sentences, the theorems
- We saw that a formal system may admit a decision procedure
- A metamathematical proof that a decision procedure exists is highly non-trivial
- The decision procedure can decide if a sentence is provable or not *by analyzing its structure, not by actually proving it*

An extension of our view of a formal system

- We defined a formal system as a language and a set of rules of inference for defining a subset of sentences, the theorems
- We saw that a formal system may admit a decision procedure
- A metamathematical proof that a decision procedure exists is highly non-trivial
- The decision procedure can decide if a sentence is provable or not *by analyzing its structure, not by actually proving it*
- It is helpful sometimes to raise the profile of this decision procedure and have the proof calculus recede in the background

An extension of our view of a formal system

- We defined a formal system as a language and a set of rules of inference for defining a subset of sentences, the theorems
- We saw that a formal system may admit a decision procedure
- A metamathematical proof that a decision procedure exists is highly non-trivial
- The decision procedure can decide if a sentence is provable or not *by analyzing its structure, not by actually proving it*
- It is helpful sometimes to raise the profile of this decision procedure and have the proof calculus recede in the background
- So we will sometimes look at a formal system as consisting of language and a decision procedure (without a proof calculus necessarily present)

Review of form versus meaning

Review of form versus meaning

- Simply put, syntax is form and semantics is meaning

Review of form versus meaning

- Simply put, syntax is form and semantics is meaning
- Warning: more formal definitions of *syntax* and *semantics* vary, depending on where the emphasys is:

Review of form versus meaning

- Simply put, syntax is form and semantics is meaning
- Warning: more formal definitions of *syntax* and *semantics* vary, depending on where the emphasys is:
  1. Emphasys is on linguistics (the study of natural language, we don't touch this)

Review of form versus meaning

- Simply put, syntax is form and semantics is meaning
- Warning: more formal definitions of *syntax* and *semantics* vary, depending on where the emphasys is:
  1. Emphasys is on linguistics (the study of natural language, we don't touch this)
  2. Emphasys is on mathematical logic

Review of form versus meaning

- Simply put, syntax is form and semantics is meaning
- Warning: more formal definitions of *syntax* and *semantics* vary, depending on where the emphasys is:
  1. Emphasys is on linguistics (the study of natural language, we don't touch this)
  2. Emphasys is on mathematical logic
  3. Emphasys is on programming languages

Review of form versus meaning

- Simply put, syntax is form and semantics is meaning
- Warning: more formal definitions of *syntax* and *semantics* vary, depending on where the emphasys is:
  1. Emphasys is on linguistics (the study of natural language, we don't touch this)
  2. Emphasys is on mathematical logic
  3. Emphasys is on programming languages
- For us, both 2 and 3 are important, so our definitions account for this need

Review of form versus meaning

- Simply put, syntax is form and semantics is meaning
- Warning: more formal definitions of *syntax* and *semantics* vary, depending on where the emphasys is:
  1. Emphasys is on linguistics (the study of natural language, we don't touch this)
  2. Emphasys is on mathematical logic
  3. Emphasys is on programming languages
- For us, both 2 and 3 are important, so our definitions account for this need
- To begin with, for us *syntax = language, and nothing else*, i.e.:

Review of form versus meaning

- Simply put, syntax is form and semantics is meaning
- Warning: more formal definitions of *syntax* and *semantics* vary, depending on where the emphasys is:
  1. Emphasys is on linguistics (the study of natural language, we don't touch this)
  2. Emphasys is on mathematical logic
  3. Emphasys is on programming languages
- For us, both 2 and 3 are important, so our definitions account for this need
- To begin with, for us *syntax = language, and nothing else*, i.e.:
  1. A lexical structure

Review of form versus meaning

- Simply put, syntax is form and semantics is meaning
- Warning: more formal definitions of *syntax* and *semantics* vary, depending on where the emphasys is:
  1. Emphasys is on linguistics (the study of natural language, we don't touch this)
  2. Emphasys is on mathematical logic
  3. Emphasys is on programming languages
- For us, both 2 and 3 are important, so our definitions account for this need
- To begin with, for us *syntax = language, and nothing else*, i.e.:
  1. A lexical structure
  2. A grammatical structure

A formal system does not include a semantics

A formal system does not include a semantics

- As we said earlier, a formal system is a language (syntax) and a set of rules for establishing the subset of provable sentences (the theorems)

A formal system does not include a semantics

- As we said earlier, a formal system is a language (syntax) and a set of rules for establishing the subset of provable sentences (the theorems)
- A formal system does not require a specific meaning to be associated with a provable sentence

A formal system does not include a semantics

- As we said earlier, a formal system is a language (syntax) and a set of rules for establishing the subset of provable sentences (the theorems)
- A formal system does not require a specific meaning to be associated with a provable sentence
- Since specific meaning = semantics, this implies that *a formal system does not include a semantics explicitly*

A formal system does not include a semantics

- As we said earlier, a formal system is a language (syntax) and a set of rules for establishing the subset of provable sentences (the theorems)
- A formal system does not require a specific meaning to be associated with a provable sentence
- Since specific meaning = semantics, this implies that *a formal system does not include a semantics explicitly*
- The rules can be a proof calculus or a YES/NO decision procedure

Semantics is specific meaning

Semantics is specific meaning

- On the other hand, a semantics associates a specific meaning to a subset of sentences

Semantics is specific meaning

- On the other hand, a semantics associates a specific meaning to a subset of sentences
- So, just like a formal system, a semantics also separates out a subset of sentences; they are called the 'valid' sentences

Semantics is specific meaning

- On the other hand, a semantics associates a specific meaning to a subset of sentences
- So, just like a formal system, a semantics also separates out a subset of sentences; they are called the 'valid' sentences
- The question of whether the rules of inference and the semantics determine the same subset of sentences is a central one (when this is the case we say that the system is *sound* (all theorems are valid) and *complete* (all valid sentences are theorems)

Semantics is specific meaning

- On the other hand, a semantics associates a specific meaning to a subset of sentences
- So, just like a formal system, a semantics also separates out a subset of sentences; they are called the 'valid' sentences
- The question of whether the rules of inference and the semantics determine the same subset of sentences is a central one (when this is the case we say that the system is *sound* (all theorems are valid) and *complete* (all valid sentences are theorems)
- There are two major types of semantics we are interested in this program

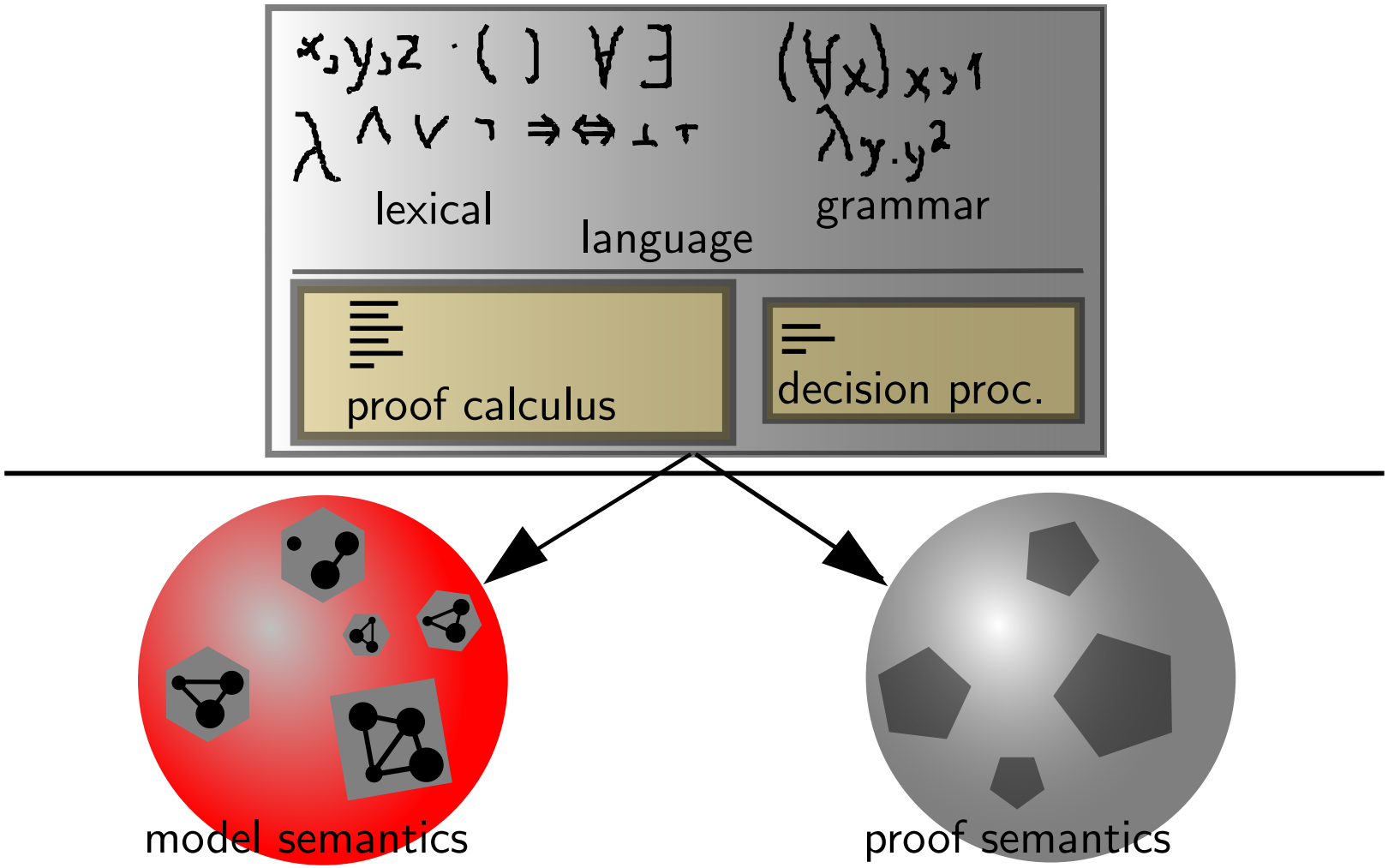
Semantics is specific meaning

- On the other hand, a semantics associates a specific meaning to a subset of sentences
- So, just like a formal system, a semantics also separates out a subset of sentences; they are called the 'valid' sentences
- The question of whether the rules of inference and the semantics determine the same subset of sentences is a central one (when this is the case we say that the system is *sound* (all theorems are valid) and *complete* (all valid sentences are theorems)
- There are two major types of semantics we are interested in this program
  - Proof semantics (meaning of language is given by the use of the rules; the meaning of a sentence is its proof; not all systems have such a semantics)

Semantics is specific meaning

- On the other hand, a semantics associates a specific meaning to a subset of sentences
- So, just like a formal system, a semantics also separates out a subset of sentences; they are called the 'valid' sentences
- The question of whether the rules of inference and the semantics determine the same subset of sentences is a central one (when this is the case we say that the system is *sound* (all theorems are valid) and *complete* (all valid sentences are theorems)
- There are two major types of semantics we are interested in this program
  - Proof semantics (meaning of language is given by the use of the rules; the meaning of a sentence is its proof; not all systems have such a semantics)
  - Model semantics (meaning of language is through the 'real' things that it represents)

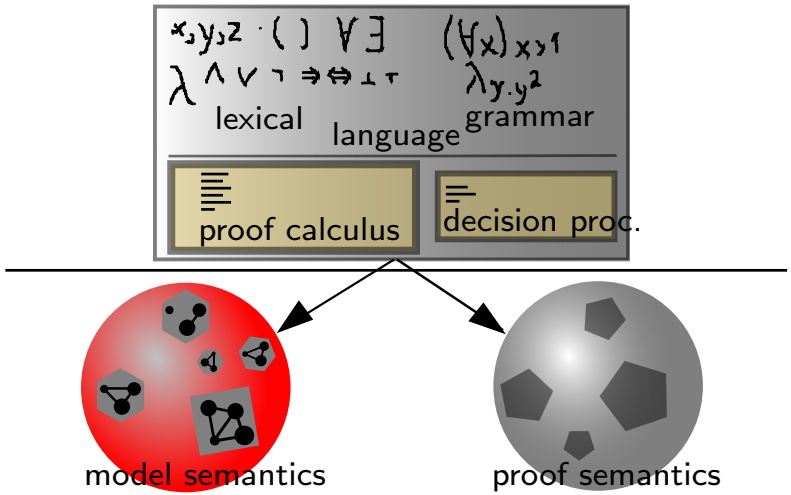
Formal systems and their semantics



Formal systems and their semantics

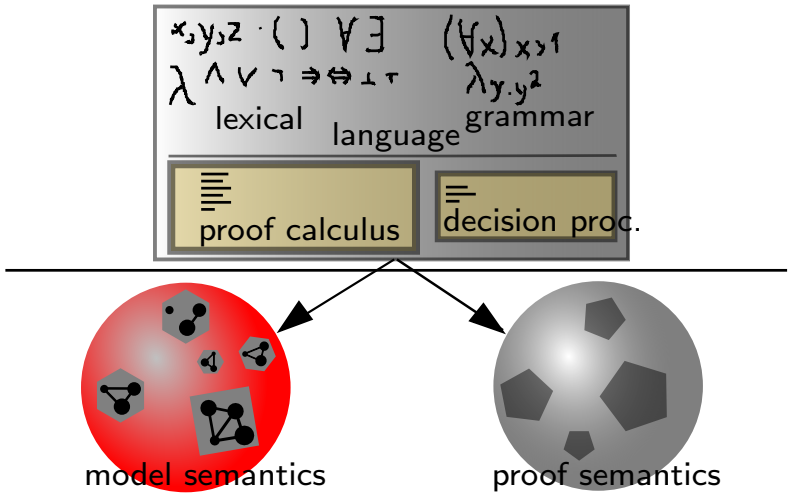
Formal systems and their semantics

- Formal system at top, its semantics at bottom



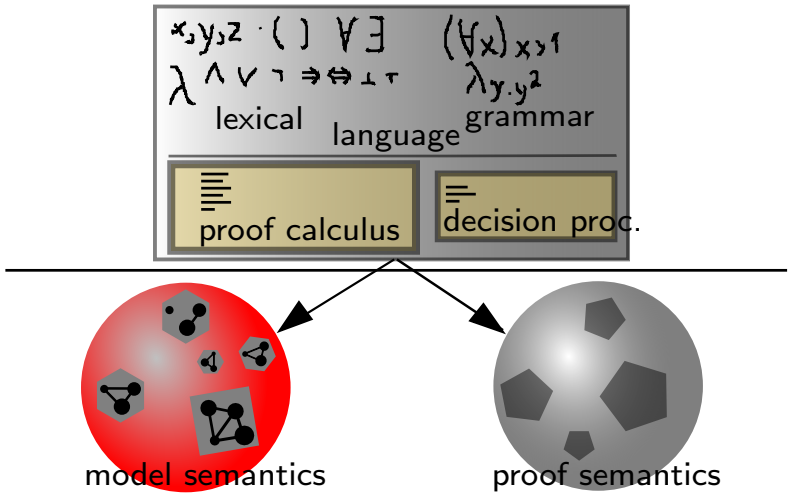
Formal systems and their semantics

- Formal system at top, its semantics at bottom
- Again, *formal system = language + proof calculus*



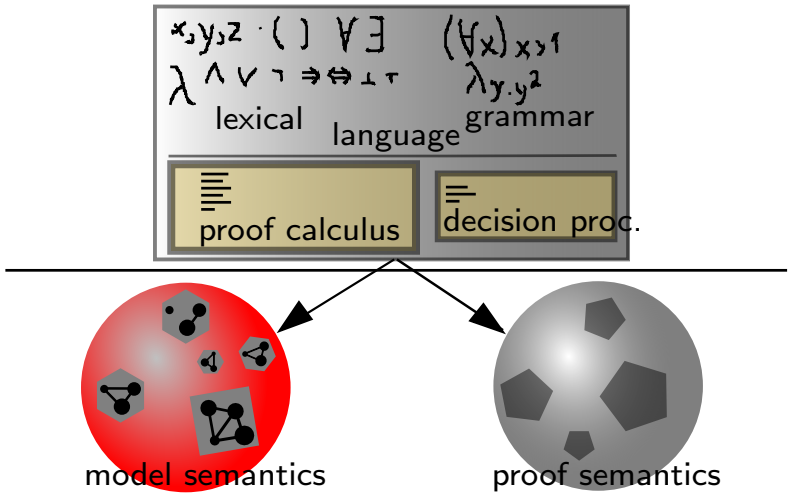
Formal systems and their semantics

- Formal system at top, its semantics at bottom
- Again, *formal system = language + proof calculus*
- Proof semantics extract meaning from the execution of the rules themselves



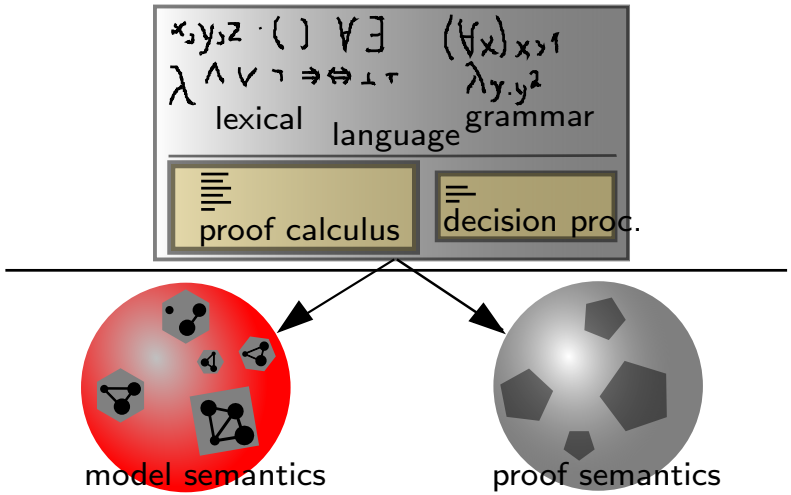
Formal systems and their semantics

- Formal system at top, its semantics at bottom
- Again, *formal system = language + proof calculus*
- Proof semantics extract meaning from the execution of the rules themselves
- *A subtler kind of meaning, takes a bit getting used to*



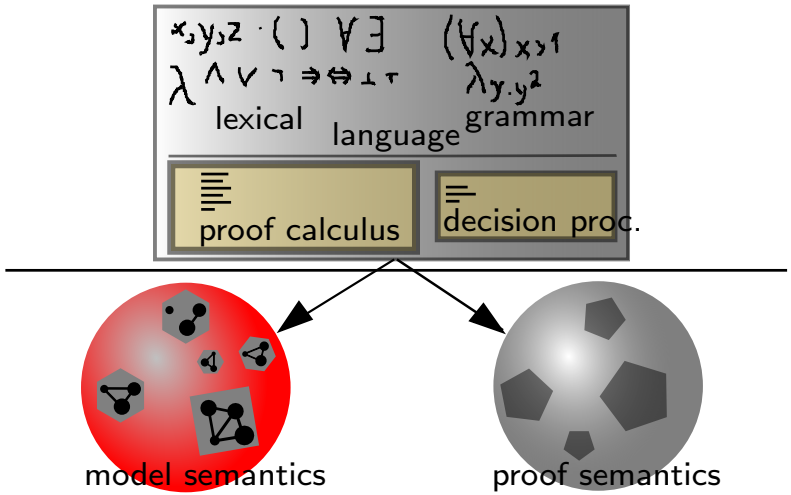
Formal systems and their semantics

- Formal system at top, its semantics at bottom
- Again, *formal system = language + proof calculus*
- Proof semantics extract meaning from the execution of the rules themselves
- *A subtler kind of meaning, takes a bit getting used to*
- Important when we look at the *semantics of programming languages*



Formal systems and their semantics

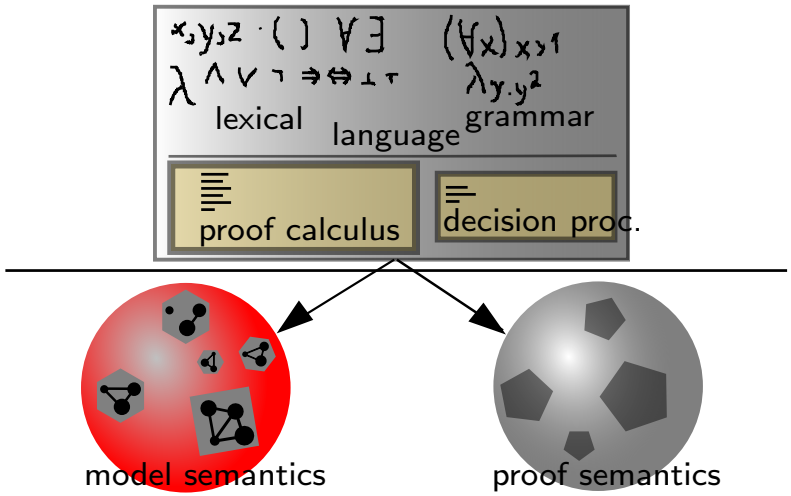
- Formal system at top, its semantics at bottom
- Again, *formal system = language + proof calculus*
- Proof semantics extract meaning from the execution of the rules themselves
- *A subtler kind of meaning, takes a bit getting used to*
- Important when we look at the *semantics of programming languages*
- An interpreter gives a 'proof semantics' to your program



Soundness and completeness

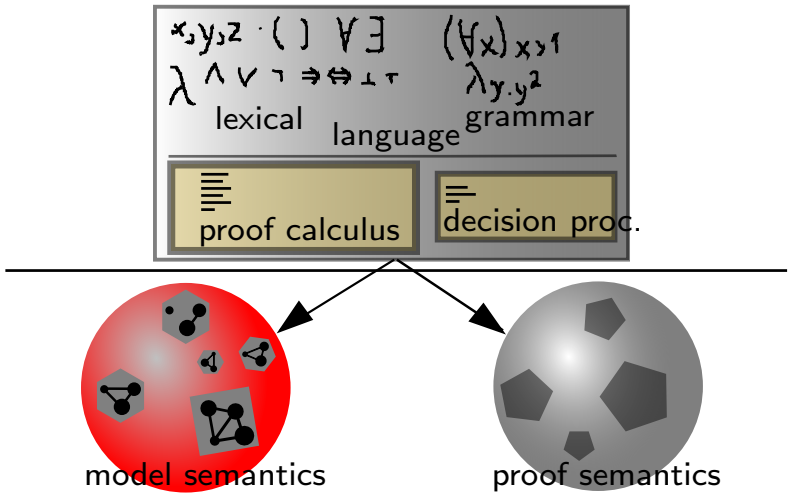
Soundness and completeness

- The essential question again: does the proof calculus describe the models *soundly and completely*?



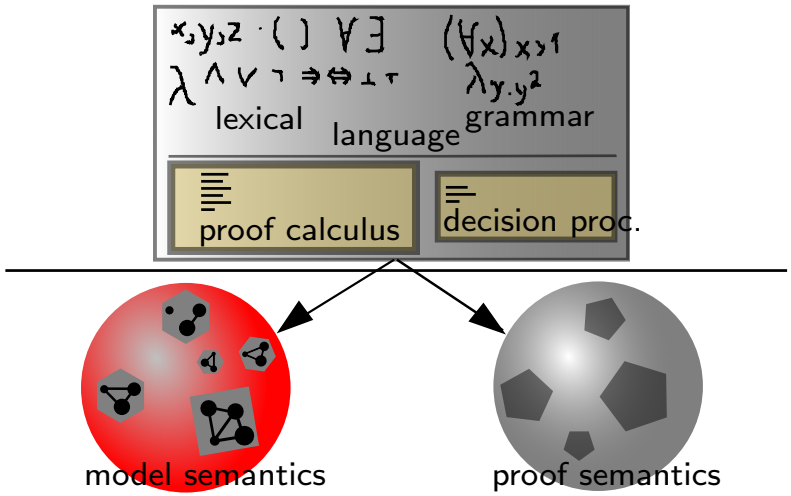
Soundness and completeness

- The essential question again: does the proof calculus describe the models *soundly and completely*?
- Sometimes it does, many times it does not



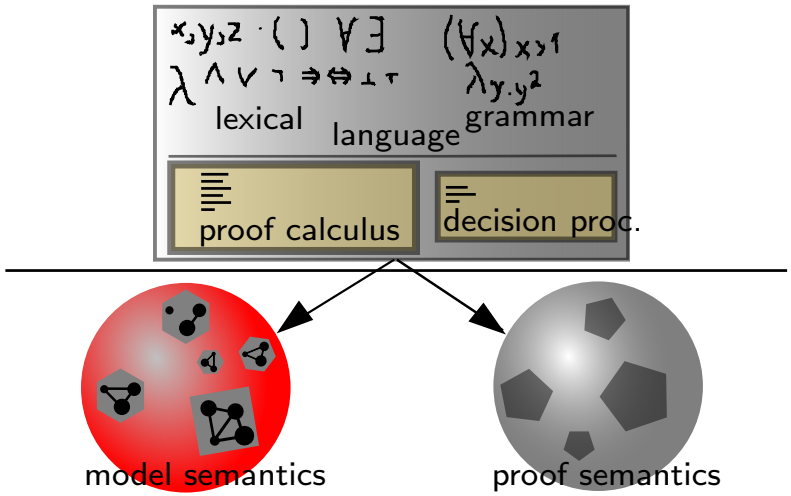
Soundness and completeness

- The essential question again: does the proof calculus describe the models *soundly and completely*?
- Sometimes it does, many times it does not
- Many times the class of models has to be enlarged beyond 'ordinary' sets to categories



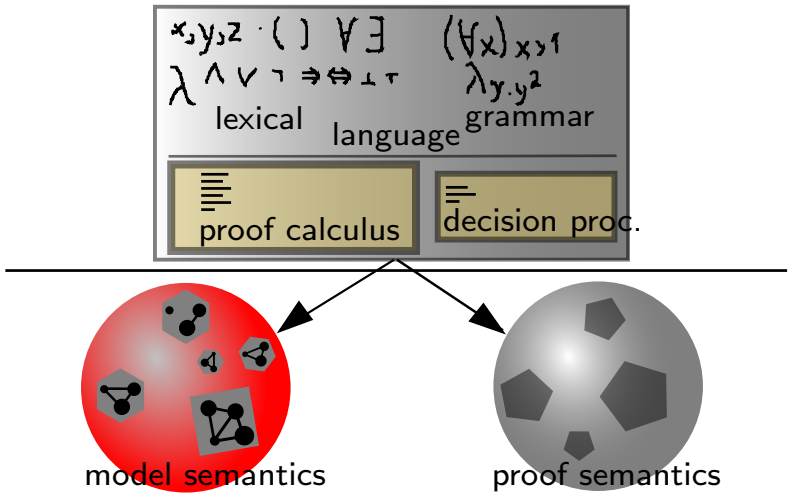
Soundness and completeness

- The essential question again: does the proof calculus describe the models *soundly and completely*?
- Sometimes it does, many times it does not
- Many times the class of models has to be enlarged beyond 'ordinary' sets to categories
- One reason why you'll hear category theory so much in this program



Soundness and completeness

- The essential question again: does the proof calculus describe the models *soundly and completely*?
- Sometimes it does, many times it does not
- Many times the class of models has to be enlarged beyond 'ordinary' sets to categories
- One reason why you'll hear category theory so much in this program
- This picture of the two basic semantics is good to keep in mind



Model Checking

Model Checking

- Model checking is a special decision procedure that is extracted from a model semantics

Model Checking

- Model checking is a special decision procedure that is extracted from a model semantics
- Model checking produces counterexamples for a formula that is not a theorem

Model Checking

- Model checking is a special decision procedure that is extracted from a model semantics
- Model checking produces counterexamples for a formula that is not a theorem
- Some formal systems have both a proof calculus and such a decision procedure

Model Checking

- Model checking is a special decision procedure that is extracted from a model semantics
- Model checking produces counterexamples for a formula that is not a theorem
- Some formal systems have both a proof calculus and such a decision procedure
- This the best of worlds, because you can switch between proof search and counterexample search

Model Checking

- Model checking is a special decision procedure that is extracted from a model semantics
- Model checking produces counterexamples for a formula that is not a theorem
- Some formal systems have both a proof calculus and such a decision procedure
- This the best of worlds, because you can switch between proof search and counterexample search
- It is the interaction between the two that makes things interesting

Model Checking

- Model checking is a special decision procedure that is extracted from a model semantics
- Model checking produces counterexamples for a formula that is not a theorem
- Some formal systems have both a proof calculus and such a decision procedure
- This the best of worlds, because you can switch between proof search and counterexample search
- It is the interaction between the two that makes things interesting
- Just like in informal mathematics, you need to acquire a feel of when to look for a proof and when to look for a counterexample

Model Checking

- Model checking is a special decision procedure that is extracted from a model semantics
- Model checking produces counterexamples for a formula that is not a theorem
- Some formal systems have both a proof calculus and such a decision procedure
- This the best of worlds, because you can switch between proof search and counterexample search
- It is the interaction between the two that makes things interesting
- Just like in informal mathematics, you need to acquire a feel of when to look for a proof and when to look for a counterexample
- Most of *formal software development is a continuous back-and-forth between theorem proving and model checking*

Model Checking

- Model checking is a special decision procedure that is extracted from a model semantics
- Model checking produces counterexamples for a formula that is not a theorem
- Some formal systems have both a proof calculus and such a decision procedure
- This the best of worlds, because you can switch between proof search and counterexample search
- It is the interaction between the two that makes things interesting
- Just like in informal mathematics, you need to acquire a feel of when to look for a proof and when to look for a counterexample
- Most of *formal software development is a continuous back-and-forth between theorem proving and model checking*
- In the program, we will develop some engineering experience allowing us to decide when to switch from one to the other

Model Checking

Model Checking

- Many times it is easier to infer properties of the language directly from the models

Model Checking

- Many times it is easier to infer properties of the language directly from the models
- Model checking is a sort of proving, but based on a terminating, deterministic set of rules

Model Checking

- Many times it is easier to infer properties of the language directly from the models
- Model checking is a sort of proving, but based on a terminating, deterministic set of rules
- 'Model calculus' versus 'proof calculus' if you wish

Model Checking

- Many times it is easier to infer properties of the language directly from the models
- Model checking is a sort of proving, but based on a terminating, deterministic set of rules
- 'Model calculus' versus 'proof calculus' if you wish
- Model checking works only for finite models, or models that can use predicate abstraction to obtain a simulating finite model

Model Checking

- Many times it is easier to infer properties of the language directly from the models
- Model checking is a sort of proving, but based on a terminating, deterministic set of rules
- 'Model calculus' versus 'proof calculus' if you wish
- Model checking works only for finite models, or models that can use predicate abstraction to obtain a simulating finite model
- So, model checking cannot handle inductive data and recursive functions

Model Checking

- Many times it is easier to infer properties of the language directly from the models
- Model checking is a sort of proving, but based on a terminating, deterministic set of rules
- 'Model calculus' versus 'proof calculus' if you wish
- Model checking works only for finite models, or models that can use predicate abstraction to obtain a simulating finite model
- So, model checking cannot handle inductive data and recursive functions
- Suffers from state space explosion

Model Checking

- Many times it is easier to infer properties of the language directly from the models
- Model checking is a sort of proving, but based on a terminating, deterministic set of rules
- 'Model calculus' versus 'proof calculus' if you wish
- Model checking works only for finite models, or models that can use predicate abstraction to obtain a simulating finite model
- So, model checking cannot handle inductive data and recursive functions
- Suffers from state space explosion
- Concurrency models are finite and model checking them is very useful

Model Checking

- Many times it is easier to infer properties of the language directly from the models
- Model checking is a sort of proving, but based on a terminating, deterministic set of rules
- 'Model calculus' versus 'proof calculus' if you wish
- Model checking works only for finite models, or models that can use predicate abstraction to obtain a simulating finite model
- So, model checking cannot handle inductive data and recursive functions
- Suffers from state space explosion
- Concurrency models are finite and model checking them is very useful
- We dedicate a course to it: 'Model Checking and Concurrency'

Nondeterminism of proof calculus

Nondeterminism of proof calculus

- Proof calculi of formal systems are nondeterministic, the rules can be applied in any order, and even the same rule can match a wff in different ways

Nondeterminism of proof calculus

- Proof calculi of formal systems are nondeterministic, the rules can be applied in any order, and even the same rule can match a wff in different ways
- aaa*bbbb can become aaaa*bbbb, or aaa*bbbbb, or aa*bbb

Nondeterminism of proof calculus

- Proof calculi of formal systems are nondeterministic, the rules can be applied in any order, and even the same rule can match a wff in different ways
- aaa*bbbb can become aaaa*bbbb, or aaa*bbbbb, or aa*bbb
- The DANS system does not demand that the pair closest to the door must go dancing

Nondeterminism of proof calculus

- Proof calculi of formal systems are nondeterministic, the rules can be applied in any order, and even the same rule can match a wff in different ways
- aaa*bbbb can become aaaa*bbbb, or aaa*bbbbb, or aa*bbb
- The DANS system does not demand that the pair closest to the door must go dancing
- The system says that they *could*, but the girl can potentially refuse all night long, while more boys and girls swell up the lines

Mathematics is founded on formal systems

Mathematics is founded on formal systems

- Are there formal systems that could potentially output all of mathematics?

Mathematics is founded on formal systems

- Are there formal systems that could potentially output all of mathematics?
- Yes, essentially all of mathematics!

Mathematics is founded on formal systems

- Are there formal systems that could potentially output all of mathematics?
- Yes, essentially all of mathematics!
- One such system, for classical mathematics, is an axiomatization of set theory called Zermelo-Fraenkel-with-Choice (ZFC)

Mathematics is founded on formal systems

- Are there formal systems that could potentially output all of mathematics?
- Yes, essentially all of mathematics!
- One such system, for classical mathematics, is an axiomatization of set theory called Zermelo-Fraenkel-with-Choice (ZFC)
- The Mizar project formalized a significant ammount of known mathematics

Mathematics is founded on formal systems

- Are there formal systems that could potentially output all of mathematics?
- Yes, essentially all of mathematics!
- One such system, for classical mathematics, is an axiomatization of set theory called Zermelo-Fraenkel-with-Choice (ZFC)
- The Mizar project formalized a significant ammount of known mathematics
- Other systems, based on type theory, can output more than classical mathematics

Software is founded on formal systems

Software is founded on formal systems

- Refining requirements analysis

Software is founded on formal systems

- Refining requirements analysis
- Building executable models

Software is founded on formal systems

- Refining requirements analysis
- Building executable models
- Building a concurrent model

Software is founded on formal systems

- Refining requirements analysis
- Building executable models
- Building a concurrent model
- Verify total correctness

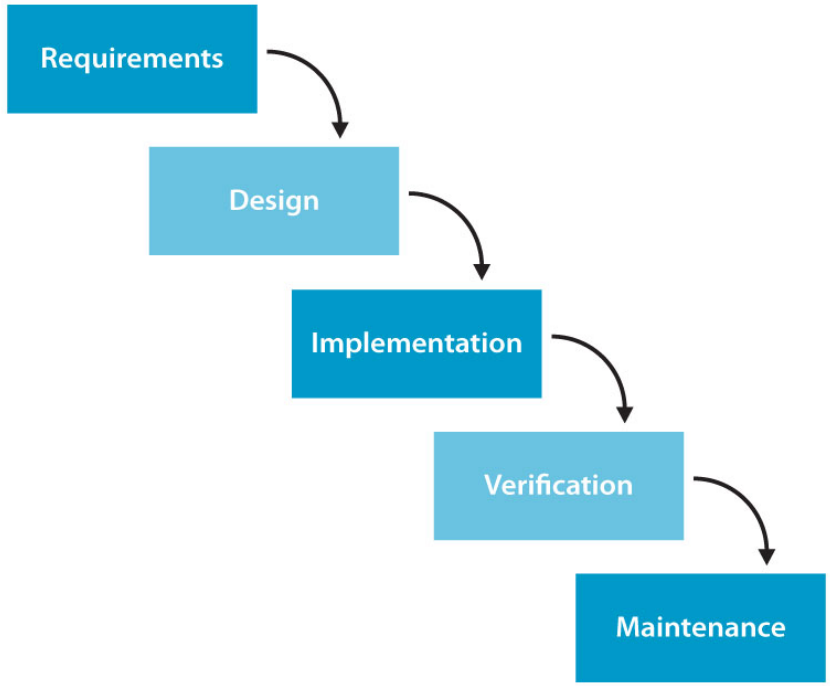
Software is founded on formal systems

- Refining requirements analysis
- Building executable models
- Building a concurrent model
- Verify total correctness
- Verify safety properties

When are proofs used in software development?

When are proofs used in software development?

Proofs may be used during all phases of the software waterfall model:



The earlier, the better

The earlier, the better

- Formalism is especially effective during requirements and design!

The earlier, the better

- Formalism is especially effective during requirements and design!
- *Many faults appear in systems that perform as specified!*; the earlier we use these methods the better (e.g. requirements analysis): we prove that we build the correct system (*validation*), not that we build the system correctly (*verification*).

The earlier, the better

- Formalism is especially effective during requirements and design!
- *Many faults appear in systems that perform as specified!*; the earlier we use these methods the better (e.g. requirements analysis): we prove that we build the correct system (*validation*), not that we build the system correctly (*verification*).
- For many projects, requirements and design are the only parts of the development that are formal (i.e. no formal *verification of the code*)

The earlier, the better

- Formalism is especially effective during requirements and design!
- *Many faults appear in systems that perform as specified!*; the earlier we use these methods the better (e.g. requirements analysis): we prove that we build the correct system (*validation*), not that we build the system correctly (*verification*).
- For many projects, requirements and design are the only parts of the development that are formal (i.e. no formal *verification of the code*)
- We can develop an early mathematical model of the system being built, with no code in sight yet

The earlier, the better

- Formalism is especially effective during requirements and design!
- *Many faults appear in systems that perform as specified!*; the earlier we use these methods the better (e.g. requirements analysis): we prove that we build the correct system (*validation*), not that we build the system correctly (*verification*).
- For many projects, requirements and design are the only parts of the development that are formal (i.e. no formal *verification of the code*)
- We can develop an early mathematical model of the system being built, with no code in sight yet
- This requirements model is usually based on axiomatic set theory, which we saw that it is a particular formal system

The earlier, the better

- Formalism is especially effective during requirements and design!
- *Many faults appear in systems that perform as specified!*; the earlier we use these methods the better (e.g. requirements analysis): we prove that we build the correct system (*validation*), not that we build the system correctly (*verification*).
- For many projects, requirements and design are the only parts of the development that are formal (i.e. no formal *verification of the code*)
- We can develop an early mathematical model of the system being built, with no code in sight yet
- This requirements model is usually based on axiomatic set theory, which we saw that it is a particular formal system
- Communications between parts of the model are specified during design; a different kind of formal system is used here: temporal logic

The earlier, the better

- Formalism is especially effective during requirements and design!
- *Many faults appear in systems that perform as specified!*; the earlier we use these methods the better (e.g. requirements analysis): we prove that we build the correct system (*validation*), not that we build the system correctly (*verification*).
- For many projects, requirements and design are the only parts of the development that are formal (i.e. no formal *verification of the code*)
- We can develop an early mathematical model of the system being built, with no code in sight yet
- This requirements model is usually based on axiomatic set theory, which we saw that it is a particular formal system
- Communications between parts of the model are specified during design; a different kind of formal system is used here: temporal logic
- The implementation and testing phases can also be treated formally, using a variety of other formal systems

Conceptual integrity

Conceptual integrity

A software project's aims are stated in a few sentences. Everyone fully understands those sentences.

Conceptual integrity

A software project's aims are stated in a few sentences. Everyone fully understands those sentences. Reality?

Conceptual integrity

A software project's aims are stated in a few sentences. Everyone fully understands those sentences. Reality?

- How many projects have you worked on where the project's goals could not be stated in a few sentences?

Conceptual integrity

A software project's aims are stated in a few sentences. Everyone fully understands those sentences. Reality?

- How many projects have you worked on where the project's goals could not be stated in a few sentences?
- How many times has the project acquired collateral goals or simply degenerated into something else?

Conceptual integrity

A software project's aims are stated in a few sentences. Everyone fully understands those sentences. Reality?

- How many projects have you worked on where the project's goals could not be stated in a few sentences?
- How many times has the project acquired collateral goals or simply degenerated into something else?
- Think about all the methodologies and the processes you learned in the past. *Have they helped with maintaining conceptual integrity?*

Formalization underlines conceptual integrity

Formalization underlines conceptual integrity

- Because it supports a gradual refinement and reevaluation of the requirements within a rigorous frame. At the end of the formalized requirements we have not only a correct spec but a clearer spec.

Formalization underlines conceptual integrity

- Because it supports a gradual refinement and reevaluation of the requirements within a rigorous frame. At the end of the formalized requirements we have not only a correct spec but a clearer spec.
- You cannot deviate from the goals because logic will simply not allow it; if you state a theorem, the proof must prove that theorem, not other theorems. Moreover, the proof will be better received if it shows a 'necessary' progression from the project axioms, axioms that everyone should be comfortable with

Formalization underlines conceptual integrity

- Because it supports a gradual refinement and reevaluation of the requirements within a rigorous frame. At the end of the formalized requirements we have not only a correct spec but a clearer spec.
- You cannot deviate from the goals because logic will simply not allow it; if you state a theorem, the proof must prove that theorem, not other theorems. Moreover, the proof will be better received if it shows a 'necessary' progression from the project axioms, axioms that everyone should be comfortable with
- Because it supports clearer advancement from requirements to design and implementation, again within rigorous frames

Formalization underlines conceptual integrity

- Because it supports a gradual refinement and reevaluation of the requirements within a rigorous frame. At the end of the formalized requirements we have not only a correct spec but a clearer spec.
- You cannot deviate from the goals because logic will simply not allow it; if you state a theorem, the proof must prove that theorem, not other theorems. Moreover, the proof will be better received if it shows a 'necessary' progression from the project axioms, axioms that everyone should be comfortable with
- Because it supports clearer advancement from requirements to design and implementation, again within rigorous frames
- Because conceptual integrity correlates to a higher level of abstraction, which formalism also supports

Formalization underlines higher quality

Formalization underlines higher quality

Formalization underlines higher quality

Imagine a different source-control system:

Formalization underlines higher quality

Imagine a different source-control system:

- Each component checked in has one or more proofs attached, preferably in different formalisms

Formalization underlines higher quality

Imagine a different source-control system:

- Each component checked in has one or more proofs attached, preferably in different formalisms
- The checkin is not accepted otherwise

Formalization underlines higher quality

Imagine a different source-control system:

- Each component checked in has one or more proofs attached, preferably in different formalisms
- The checkin is not accepted otherwise
- After check-out, you may choose to verify any one of these proofs based on your knowledge of the formalism

Formalization underlines higher security

Formalization underlines higher security

Formalization underlines higher security

Imagine using the web differently:

Formalization underlines higher security

Imagine using the web differently:

- Downloaded executable code has one or more associated proof objects (or certificates)

Formalization underlines higher security

Imagine using the web differently:

- Downloaded executable code has one or more associated proof objects (or certificates)
- Your computer reads the specification (=theorem) and accepts that specification

Formalization underlines higher security

Imagine using the web differently:

- Downloaded executable code has one or more associated proof objects (or certificates)
- Your computer reads the specification (=theorem) and accepts that specification
- Your computer verifies that the code is a proof of that specification

Formalization underlines higher security

Imagine using the web differently:

- Downloaded executable code has one or more associated proof objects (or certificates)
- Your computer reads the specification (=theorem) and accepts that specification
- Your computer verifies that the code is a proof of that specification
- If such proof objects are absent, your computer transforms the downloaded code into semantically equivalent functional snippets which are subsequently compared with a list of known threats

Formalization underlines higher security

IMAGINE USING THE WEB DIFFERENTLY:

- Downloaded executable code has one or more associated proof objects (or certificates)
- Your computer reads the specification (=theorem) and accepts that specification
- Your computer verifies that the code is a proof of that specification
- If such proof objects are absent, your computer transforms the downloaded code into semantically equivalent functional snippets which are subsequently compared with a list of known threats
- Not a database of syntactic signatures, which can easily be (and often is) circumvented

Why is it so important that we use formal methods in software engineering?

Why is it so important that we use formal methods in software engineering?

- All engineering disciplines use mathematics, but traditionally they do not use formal methods

Why is it so important that we use formal methods in software engineering?

- All engineering disciplines use mathematics, but traditionally they do not use formal methods
- So, why are formal methods so important in software engineering?

Why is it so important that we use formal methods in software engineering?

- All engineering disciplines use mathematics, but traditionally they do not use formal methods
- So, why are formal methods so important in software engineering?
- The main reason is that all the tools used in software, from the computer, to operating system, to the compiler, to other applications, are essentially formal systems, i.e. specialized languages to be used according to set rules. So *it is the very essence of software that calls for the use of formal methods.*

More engineering participation

- Technologies to achieve quality and security are already known

More engineering participation

- Technologies to achieve quality and security are already known
- Some are not mature enough, and they probably won't be until *YOU get interested*

More engineering participation

- Technologies to achieve quality and security are already known
- Some are not mature enough, and they probably won't be until *YOU get interested*
- Academia and industrial research have been interested for a long time, *but that's not sufficient*

Models versus programs

Models versus programs

- Models are developed at a higher level of abstraction

Models versus programs

- Models are developed at a higher level of abstraction
- In functional, not imperative languages

Models versus programs

- Models are developed at a higher level of abstraction
- In functional, not imperative languages
- There are logical tools available for building executable models

Models versus programs

- Models are developed at a higher level of abstraction
- In functional, not imperative languages
- There are logical tools available for building executable models
- They can prove that these models implement the system specification

Models versus programs

- Models are developed at a higher level of abstraction
- In functional, not imperative languages
- There are logical tools available for building executable models
- They can prove that these models implement the system specification
- They deepen our understanding of the problem that we want to solve

Models versus programs

- Models are developed at a higher level of abstraction
- In functional, not imperative languages
- There are logical tools available for building executable models
- They can prove that these models implement the system specification
- They deepen our understanding of the problem that we want to solve
- Many times it is not even necessary to translate from a model to an imperative program

Models versus programs

- Models are developed at a higher level of abstraction
- In functional, not imperative languages
- There are logical tools available for building executable models
- They can prove that these models implement the system specification
- They deepen our understanding of the problem that we want to solve
- Many times it is not even necessary to translate from a model to an imperative program
- More time should be spent with models than with programs

Some soft reasons why we do formal development

Some soft reasons why we do formal development

We also use formal methods because . . .

- . . . The subject is beautiful and fun

Some soft reasons why we do formal development

We also use formal methods because . . .

- . . . The subject is beautiful and fun
- . . . We feel that there is more to developing software, in particular that there should be a scientific basis to it

Some soft reasons why we do formal development

We also use formal methods because . . .

- . . . The subject is beautiful and fun
- . . . We feel that there is more to developing software, in particular that there should be a scientific basis to it
- . . . We get a clarification of many difficult subjects

Some soft reasons why we do formal development

We also use formal methods because . . .

- . . . The subject is beautiful and fun
- . . . We feel that there is more to developing software, in particular that there should be a scientific basis to it
- . . . We get a clarification of many difficult subjects
- . . . We don't need any more development methodologies and processes

Why use formal methods now?

Why use formal methods now?

- Enough success stories

Why use formal methods now?

- Enough success stories
- Momentum in academia and research centers

Why use formal methods now?

- Enough success stories
- Momentum in academia and research centers
- Cost of buggy software increasing, despite increased spending on testing

Why use formal methods now?

- Enough success stories
- Momentum in academia and research centers
- Cost of buggy software increasing, despite increased spending on testing
- Formal methods are incorporated into standards

Why use formal methods now?

- Enough success stories
- Momentum in academia and research centers
- Cost of buggy software increasing, despite increased spending on testing
- Formal methods are incorporated into standards
- Companies need to prepare for the future

Why use formal methods now?

- Enough success stories
- Momentum in academia and research centers
- Cost of buggy software increasing, despite increased spending on testing
- Formal methods are incorporated into standards
- Companies need to prepare for the future
- Many government contracts are requiring formal development

Remarkable success story: seL4

Remarkable success story: seL4

A full microkernel has been verified:

(find complete info at http://www.nicta.com.au/news/)

Remarkable success story: seL4

A full microkernel has been verified:

(find complete info at http://www.nicta.com.au/news/)

- Verified 7,500 lines of C code

Remarkable success story: seL4

A FULL MICROKERNEL HAS BEEN VERIFIED:

(find complete info at http://www.nicta.com.au/news/)

- Verified 7,500 lines of C code
- Proved over 10,000 intermediate theorems in over 200,000 lines of formal proof.

Remarkable success story: seL4

A FULL MICROKERNEL HAS BEEN VERIFIED:

(find complete info at http://www.nicta.com.au/news/)

- Verified 7,500 lines of C code
- Proved over 10,000 intermediate theorems in over 200,000 lines of formal proof.
- The proof was checked with Isabelle (we will study Isabelle in the 'Interactive Provers' course)

Remarkable success story: seL4

A full microkernel has been verified:

(find complete info at http://www.nicta.com.au/news/)

- Verified 7,500 lines of C code
- Proved over 10,000 intermediate theorems in over 200,000 lines of formal proof.
- The proof was checked with Isabelle (we will study Isabelle in the 'Interactive Provers' course)
- One of the largest machine-checked proofs ever done.

Remarkable success story: seL4

‘ Formal proofs for specific properties have been conducted for smaller kernels, but what we have done is a general, functional correctness proof which has never before been achieved for real-world, high-performance software of this complexity or size. ’

– Gerwin Klein, NICTA

‘ It is hard to comment on this achievement without resorting to clichés. Proving the correctness of 7,500 lines of C code in an operating system’s kernel is a unique achievement, which should eventually lead to software that meets currently unimaginable standards of reliability. ’

– Lawrence Paulson, Cambridge University Computer Laboratory

High complexity: the seL4 function call graph

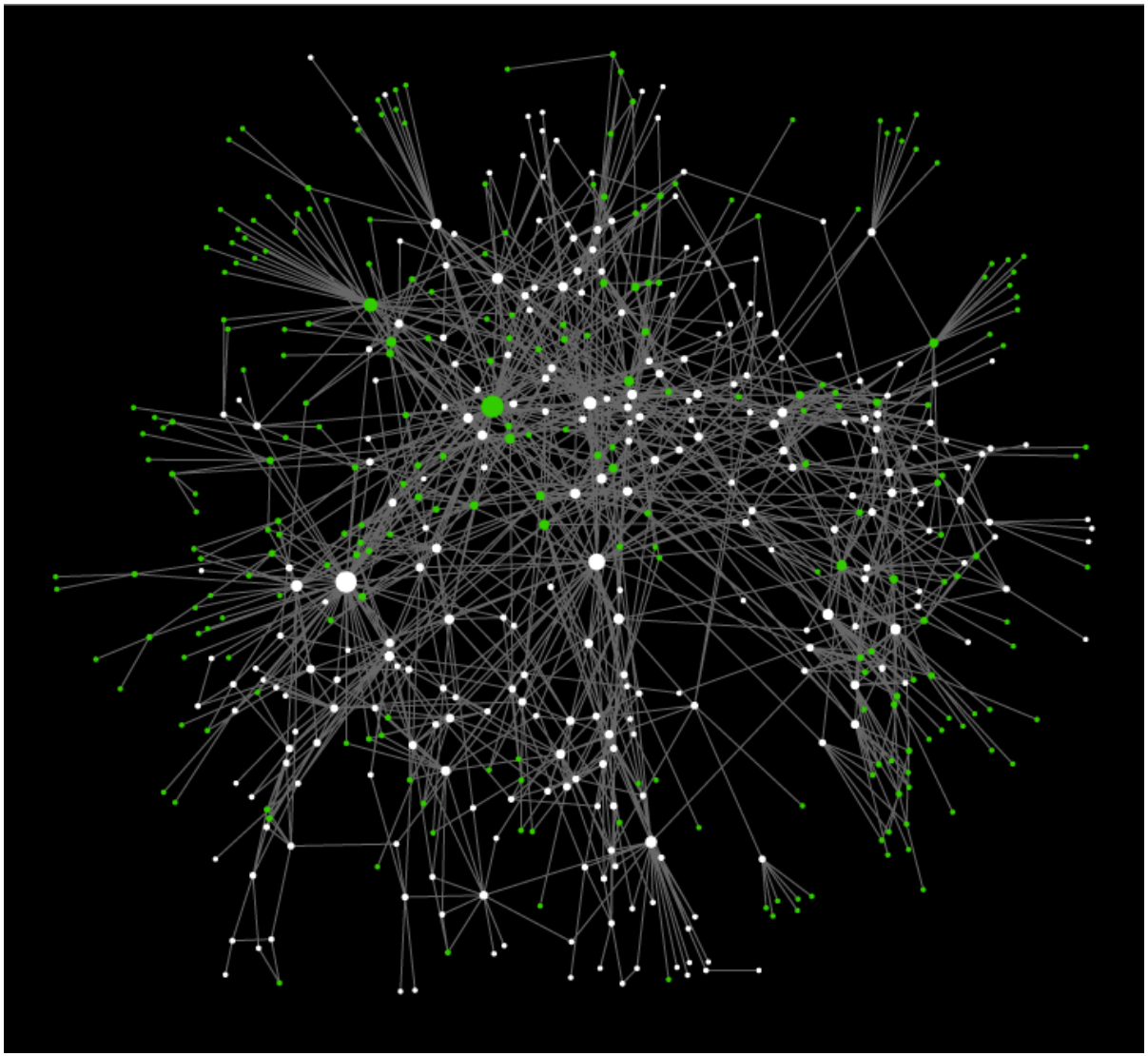
Vertices in the graph represent functions in the kernel

Edges between vertices are function calls

An abstract functional specification of seL4 is given

The C implementation is proven to satisfy this specification

http://ertos.nicta.com.au/research/



The Robbins conjecture

The Robbins conjecture

- In 1996, a most important success of computer-based theorem proving

The Robbins conjecture

- In 1996, a most important success of computer-based theorem proving
- A problem that was open for 60 years is solved by a theorem prover

The Robbins conjecture

- In 1996, a most important success of computer-based theorem proving
- A problem that was open for 60 years is solved by a theorem prover
- So the solution came from a program that was *designed to reason, in equational logic, not a program designed for this specific problem*

The Robbins conjecture

- In 1996, a most important success of computer-based theorem proving
- A problem that was open for 60 years is solved by a theorem prover
- So the solution came from a program that was *designed to reason, in equational logic, not a program designed for this specific problem*
- (it took about 8 days on an RS/6000 processor)

The Robbins conjecture

- In 1996, a most important success of computer-based theorem proving
- A problem that was open for 60 years is solved by a theorem prover
- So the solution came from a program that was *designed to reason, in equational logic, not a program designed for this specific problem*
- (it took about 8 days on an RS/6000 processor)
- Formulate in your own words an explanation of what this might mean

Example: Formal methods are incorporated into standards

Example: Formal methods are incorporated into standards

Common Criteria (an international standard since 1999) is a framework for providing assurance that the process of specification, implementation and evaluation of a computer security product has been conducted in a rigorous and standard manner. Namely:

Example: Formal methods are incorporated into standards

Common Criteria (an international standard since 1999) is a framework for providing assurance that the process of specification, implementation and evaluation of a computer security product has been conducted in a rigorous and standard manner. Namely:

- Users can specify their security functional and assurance requirements

Example: Formal methods are incorporated into standards

Common Criteria (an international standard since 1999) is a framework for providing assurance that the process of specification, implementation and evaluation of a computer security product has been conducted in a rigorous and standard manner. Namely:

- Users can specify their security functional and assurance requirements
- Vendors can implement and/or make claims about the security attributes of their products

Example: Formal methods are incorporated into standards

Common Criteria (an international standard since 1999) is a framework for providing assurance that the process of specification, implementation and evaluation of a computer security product has been conducted in a rigorous and standard manner. Namely:

- Users can specify their security functional and assurance requirements
- Vendors can implement and/or make claims about the security attributes of their products
- Testing laboratories can evaluate the products to determine if they actually meet the claims

Formal methods are incorporated into standards (cont'd)

Formal methods are incorporated into standards (cont'd)

- Assurance levels specified by Common Criteria

Formal methods are incorporated into standards (cont'd)

- Assurance levels specified by Common Criteria
  - EAL1: Functionally Tested

Formal methods are incorporated into standards (cont'd)

- Assurance levels specified by Common Criteria
  - EAL1: Functionally Tested
  - EAL2: Structurally Tested

Formal methods are incorporated into standards (cont'd)

- Assurance levels specified by Common Criteria
  - EAL1: Functionally Tested
  - EAL2: Structurally Tested
  - EAL3: Methodically Tested and Checked

Formal methods are incorporated into standards (cont'd)

- Assurance levels specified by Common Criteria
  - EAL1: Functionally Tested
  - EAL2: Structurally Tested
  - EAL3: Methodically Tested and Checked
  - EAL4: Methodically Designed, Tested, and Reviewed

Formal methods are incorporated into standards (cont'd)

- Assurance levels specified by Common Criteria
  - EAL1: Functionally Tested
  - EAL2: Structurally Tested
  - EAL3: Methodically Tested and Checked
  - EAL4: Methodically Designed, Tested, and Reviewed
  - EAL5: Semi formally Designed and Tested

Formal methods are incorporated into standards (cont'd)

- Assurance levels specified by Common Criteria
  - EAL1: Functionally Tested
  - EAL2: Structurally Tested
  - EAL3: Methodically Tested and Checked
  - EAL4: Methodically Designed, Tested, and Reviewed
  - EAL5: Semi formally Designed and Tested
  - EAL6: Semi formally Verified Design and Tested

Formal methods are incorporated into standards (cont'd)

- Assurance levels specified by Common Criteria
  - EAL1: Functionally Tested
  - EAL2: Structurally Tested
  - EAL3: Methodically Tested and Checked
  - EAL4: Methodically Designed, Tested, and Reviewed
  - EAL5: Semi formally Designed and Tested
  - EAL6: Semi formally Verified Design and Tested
  - EAL7: Formally Verified Design and Tested

Formal methods are incorporated into standards (cont'd)

- Assurance levels specified by Common Criteria
  - EAL1: Functionally Tested
  - EAL2: Structurally Tested
  - EAL3: Methodically Tested and Checked
  - EAL4: Methodically Designed, Tested, and Reviewed
  - EAL5: Semi formally Designed and Tested
  - EAL6: Semi formally Verified Design and Tested
  - EAL7: Formally Verified Design and Tested
- *Formal requirements are present in levels 5 to 7*

Formal methods are incorporated into standards (cont'd)

- Assurance levels specified by Common Criteria
  - EAL1: Functionally Tested
  - EAL2: Structurally Tested
  - EAL3: Methodically Tested and Checked
  - EAL4: Methodically Designed, Tested, and Reviewed
  - EAL5: Semi formally Designed and Tested
  - EAL6: Semi formally Verified Design and Tested
  - EAL7: Formally Verified Design and Tested
- *Formal requirements are present in levels 5 to 7*
- This (formal methods in security apps) is just an example of *international awareness of the need for formal methods*

Formal methods are incorporated into standards (cont'd)

- Assurance levels specified by Common Criteria
  - EAL1: Functionally Tested
  - EAL2: Structurally Tested
  - EAL3: Methodically Tested and Checked
  - EAL4: Methodically Designed, Tested, and Reviewed
  - EAL5: Semi formally Designed and Tested
  - EAL6: Semi formally Verified Design and Tested
  - EAL7: Formally Verified Design and Tested
- *Formal requirements are present in levels 5 to 7*
- This (formal methods in security apps) is just an example of *international awareness of the need for formal methods*
- If this example interests you, see `http://www.commoncriteriaportal.org/cc/`

Assurance levels in the world

Assurance levels in the world

- Shlumberger and Trusted Logic took 3 years to develop a completely formalized model of the execution environment for the JavaCard language and VM

Assurance levels in the world

- Shlumberger and Trusted Logic took 3 years to develop a completely formalized model of the execution environment for the JavaCard language and VM
- Regarded as a major work in security and the first I believe to receive EAL7 certification for some parts

Assurance levels in the world

- Shlumberger and Trusted Logic took 3 years to develop a completely formalized model of the execution environment for the JavaCard language and VM
- Regarded as a major work in security and the first I believe to receive EAL7 certification for some parts
- Just to realize the scope of this achievement of formal methods in the security area, the JavaCard development was done in 121000 lines of Coq spread over 278 modules (we study Coq in the 'Interactive Provers' course).

Assurance levels in the world

- Shlumberger and Trusted Logic took 3 years to develop a completely formalized model of the execution environment for the JavaCard language and VM
- Regarded as a major work in security and the first I believe to receive EAL7 certification for some parts
- Just to realize the scope of this achievement of formal methods in the security area, the JavaCard development was done in 121000 lines of Coq spread over 278 modules (we study Coq in the 'Interactive Provers' course).
- More on JavaCard formalization at http://www.commoncriteriaportal.org/iccc/9iccc/pdf/B2404.pdf

Assurance levels in the world

- Shlumberger and Trusted Logic took 3 years to develop a completely formalized model of the execution environment for the JavaCard language and VM
- Regarded as a major work in security and the first I believe to receive EAL7 certification for some parts
- Just to realize the scope of this achievement of formal methods in the security area, the JavaCard development was done in 121000 lines of Coq spread over 278 modules (we study Coq in the 'Interactive Provers' course).
- More on JavaCard formalization at http://www.commoncriteriaportal.org/iccc/9iccc/pdf/B2404.pdf
- Australia, august 2009, NICTA's Secure Embedded L4 (seL4) microkernel was certified EAL7

Assurance levels in US

Assurance levels in US

- November 2008, National Security Agency (NSA) certified Green Hills Software's Integrity operating system at EAL6+. First deployed in the B1B bomber in 1997, today runs in military and commercial aircraft,

Assurance levels in US

- November 2008, National Security Agency (NSA) certified Green Hills Software's Integrity operating system at EAL6+. First deployed in the B1B bomber in 1997, today runs in military and commercial aircraft,
  - including the Airbus 380 and Boeing 787 airplanes

Assurance levels in US

- November 2008, National Security Agency (NSA) certified Green Hills Software's Integrity operating system at EAL6+. First deployed in the B1B bomber in 1997, today runs in military and commercial aircraft,
  - including the Airbus 380 and Boeing 787 airplanes
  - and the F-16, F-22, and F-35 military jets

Assurance levels in US

- November 2008, National Security Agency (NSA) certified Green Hills Software's Integrity operating system at EAL6+. First deployed in the B1B bomber in 1997, today runs in military and commercial aircraft,
  - including the Airbus 380 and Boeing 787 airplanes
  - and the F-16, F-22, and F-35 military jets
- Read more at http://www.integrityglobalsecurity.com/

Assurance levels in US

- November 2008, National Security Agency (NSA) certified Green Hills Software's Integrity operating system at EAL6+. First deployed in the B1B bomber in 1997, today runs in military and commercial aircraft,
  - including the Airbus 380 and Boeing 787 airplanes
  - and the F-16, F-22, and F-35 military jets
- Read more at http://www.integrityglobalsecurity.com/
- Windows and Linux can be run under Integrity, virtualized as guest OSs

Assurance levels in US

- November 2008, National Security Agency (NSA) certified Green Hills Software's Integrity operating system at EAL6+. First deployed in the B1B bomber in 1997, today runs in military and commercial aircraft,
  - including the Airbus 380 and Boeing 787 airplanes
  - and the F-16, F-22, and F-35 military jets
- Read more at http://www.integrityglobalsecurity.com/
- Windows and Linux can be run under Integrity, virtualized as guest OSs
- Windows and Linux are EAL 4+

Why do this program at the UC Extension?

Why do this program at the UC Extension?

- Your experience as active professionals is essential

Why do this program at the UC Extension?

- Your experience as active professionals is essential
- Your frustration with ever buggy software is a motivating factor

Why do this program at the UC Extension?

- Your experience as active professionals is essential
- Your frustration with ever buggy software is a motivating factor
- Your computational intuition is needed, especially for concurrent systems

Why do this program at the UC Extension?

- Your experience as active professionals is essential
- Your frustration with ever buggy software is a motivating factor
- Your computational intuition is needed, especially for concurrent systems
- We need to accumulate more experience on how these logical methods fit into an industrial setting

Why do this program at the UC Extension?

- Your experience as active professionals is essential
- Your frustration with ever buggy software is a motivating factor
- Your computational intuition is needed, especially for concurrent systems
- We need to accumulate more experience on how these logical methods fit into an industrial setting
- CS departments teach formal methods in bits and pieces, they have plenty of other material to worry about

Why do this program at the UC Extension?

- Your experience as active professionals is essential
- Your frustration with ever buggy software is a motivating factor
- Your computational intuition is needed, especially for concurrent systems
- We need to accumulate more experience on how these logical methods fit into an industrial setting
- CS departments teach formal methods in bits and pieces, they have plenty of other material to worry about
- Specialized conferences cater to specialists/implementers, not much to software engineers

Why do this program at the UC Extension?

- Your experience as active professionals is essential
- Your frustration with ever buggy software is a motivating factor
- Your computational intuition is needed, especially for concurrent systems
- We need to accumulate more experience on how these logical methods fit into an industrial setting
- CS departments teach formal methods in bits and pieces, they have plenty of other material to worry about
- Specialized conferences cater to specialists/implementers, not much to software engineers
- Not everyone has NASA's budget to afford training of their staff

Why do this program at the UC Extension?

- Your experience as active professionals is essential
- Your frustration with ever buggy software is a motivating factor
- Your computational intuition is needed, especially for concurrent systems
- We need to accumulate more experience on how these logical methods fit into an industrial setting
- CS departments teach formal methods in bits and pieces, they have plenty of other material to worry about
- Specialized conferences cater to specialists/implementers, not much to software engineers
- Not everyone has NASA's budget to afford training of their staff
- We might get your organization interested in customizing/improving these (mostly open-source) tools

Some controversy about applicability

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Some controversy about applicability

- The use of formal methods in software development has been controversial

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Some controversy about applicability

- The use of formal methods in software development has been controversial
- When foundations were laid out in the 60's and 70's, industry reacted with enthusiasm

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Some controversy about applicability

- The use of formal methods in software development has been controversial
- When foundations were laid out in the 60's and 70's, industry reacted with enthusiasm
- In the 80's and 90's, although important advances were made, few of them saw practical applications, and the enthusiasm in the industry went down

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Some controversy about applicability

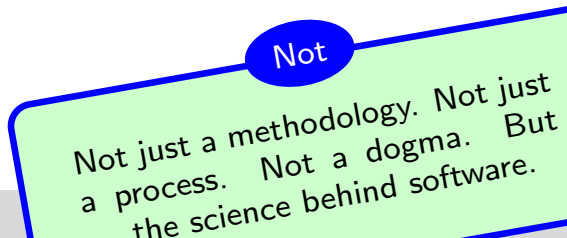
- The use of formal methods in software development has been controversial
- When foundations were laid out in the 60's and 70's, industry reacted with enthusiasm
- In the 80's and 90's, although important advances were made, few of them saw practical applications, and the enthusiasm in the industry went down
- Computer-generated proofs were viewed suspiciously both by mathematicians and by software engineers, for different reasons:

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

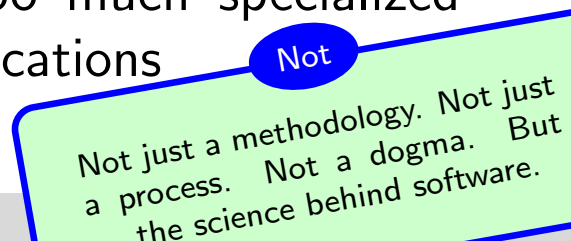
Some controversy about applicability

- The use of formal methods in software development has been controversial
- When foundations were laid out in the 60's and 70's, industry reacted with enthusiasm
- In the 80's and 90's, although important advances were made, few of them saw practical applications, and the enthusiasm in the industry went down
- Computer-generated proofs were viewed suspiciously both by mathematicians and by software engineers, for different reasons:
  - By mathematicians: obstacles were attributed to the theoretical limitations imposed by the incompleteness theorems of mathematical logic



Some controversy about applicability

- The use of formal methods in software development has been controversial
- When foundations were laid out in the 60's and 70's, industry reacted with enthusiasm
- In the 80's and 90's, although important advances were made, few of them saw practical applications, and the enthusiasm in the industry went down
- Computer-generated proofs were viewed suspiciously both by mathematicians and by software engineers, for different reasons:
  - By mathematicians: obstacles were attributed to the theoretical limitations imposed by the incompleteness theorems of mathematical logic
  - By software engineers: the tools required too much specialized knowledge and were only useful for small applications



Nevertheless, formal methods kept making steady progress

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Nevertheless, formal methods kept making steady progress

- During the 80's and 90's perceived downtime, many people *ignored the lack of enthusiasm for formalism and developed efficient algorithms* for high-complexity problems, that performed remarkably well in practical situations

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Nevertheless, formal methods kept making steady progress

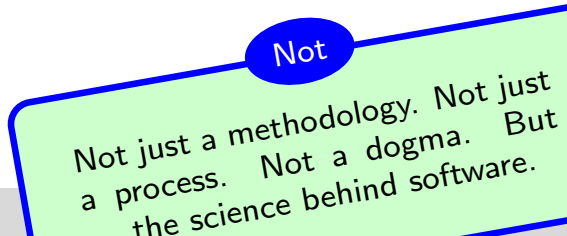
- During the 80's and 90's perceived downtime, many people *ignored the lack of enthusiasm for formalism and developed efficient algorithms* for high-complexity problems, that performed remarkably well in practical situations
- Many of the most successful logical frameworks were born in this period

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

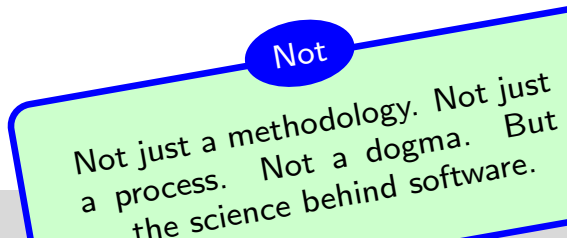
Nevertheless, formal methods kept making steady progress

- During the 80's and 90's perceived downtime, many people *ignored the lack of enthusiasm for formalism and developed efficient algorithms* for high-complexity problems, that performed remarkably well in practical situations
- Many of the most successful logical frameworks were born in this period
- This not-so-visible successes led to the current revival of formal methods in software development



Nevertheless, formal methods kept making steady progress

- During the 80's and 90's perceived downtime, many people *ignored the lack of enthusiasm for formalism and developed efficient algorithms* for high-complexity problems, that performed remarkably well in practical situations
- Many of the most successful logical frameworks were born in this period
- This not-so-visible successes led to the current revival of formal methods in software development
- We are at the beginning of a forceful and active period of formal software development



Engineering or science?

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Engineering or science?

- It's been said that 'Software engineering is not computer science'

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Engineering or science?

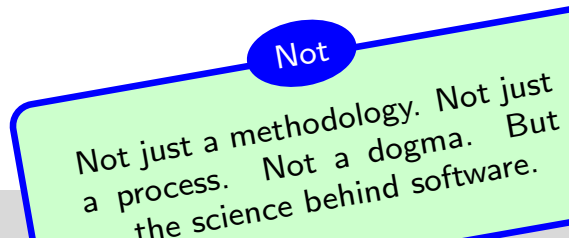
- It's been said that 'Software engineering is not computer science'
- Fair enough: engineering, period . . . is not science, period

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Engineering or science?

- It's been said that 'Software engineering is not computer science'
- Fair enough: engineering, period . . . is not science, period
- That does not imply that there must not be *a scientific basis to software*



Engineering or science?

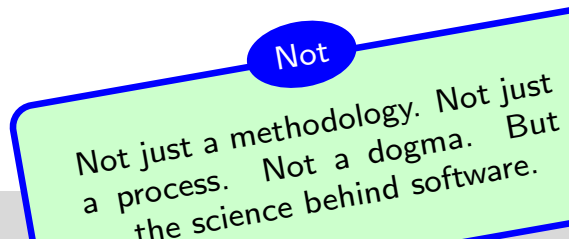
- It's been said that 'Software engineering is not computer science'
- Fair enough: engineering, period . . . is not science, period
- That does not imply that there must not be *a scientific basis to software*
- Chemical engineers need to know a science called chemistry

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

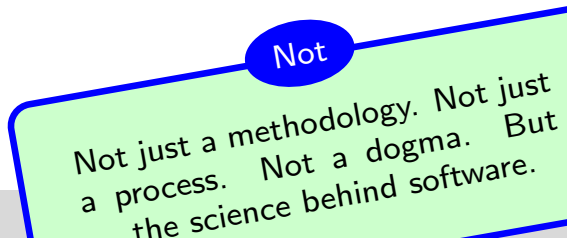
Engineering or science?

- It's been said that 'Software engineering is not computer science'
- Fair enough: engineering, period . . . is not science, period
- That does not imply that there must not be *a scientific basis to software*
- Chemical engineers need to know a science called chemistry
- Nuclear engineers need to know a science called physics



Engineering or science?

- It's been said that 'Software engineering is not computer science'
- Fair enough: engineering, period . . . is not science, period
- That does not imply that there must not be *a scientific basis to software*
- Chemical engineers need to know a science called chemistry
- Nuclear engineers need to know a science called physics
- *What is the science that software engineers need to know?*



Engineering AND science!

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Engineering AND science!

- Software engineering made huge advances, without waiting for the science

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Engineering AND science!

- Software engineering made huge advances, without waiting for the science
- It's been one of the most astonishing engineering disciplines of our time

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Engineering AND science!

- Software engineering made huge advances, without waiting for the science
- It's been one of the most astonishing engineering disciplines of our time
- We can't say 'Stop, you have to learn formal systems before you write one more line of C code'

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Engineering AND science!

- Software engineering made huge advances, without waiting for the science
- It's been one of the most astonishing engineering disciplines of our time
- We can't say 'Stop, you have to learn formal systems before you write one more line of C code'
- What we do say is that some training in formal systems is becoming necessary

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Engineering AND science!

- Software engineering made huge advances, without waiting for the science
- It's been one of the most astonishing engineering disciplines of our time
- We can't say 'Stop, you have to learn formal systems before you write one more line of C code'
- What we do say is that some training in formal systems is becoming necessary
- For critical software, this has been clear for some time

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Engineering AND science!

- Software engineering made huge advances, without waiting for the science
- It's been one of the most astonishing engineering disciplines of our time
- We can't say 'Stop, you have to learn formal systems before you write one more line of C code'
- What we do say is that some training in formal systems is becoming necessary
- For critical software, this has been clear for some time
- But the benefits of formal systems are that *they lead to a different way of thinking about software, any software, not just critical software*

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Are formal methods providing certainty?

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Are formal methods providing certainty?

- *Formal software development is many times wrongly associated with an absolute 'certainty that there are no faults'*

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Are formal methods providing certainty?

- *Formal software development is many times wrongly associated with an absolute 'certainty that there are no faults'*
- This is not so and cannot be so

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Are formal methods providing certainty?

- *Formal software development is many times wrongly associated with an absolute 'certainty that there are no faults'*
- This is not so and cannot be so
- We can never be certain of our formal systems themselves; this is actually a theorem of metamathematics, not an ideological position

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Are formal methods providing certainty?

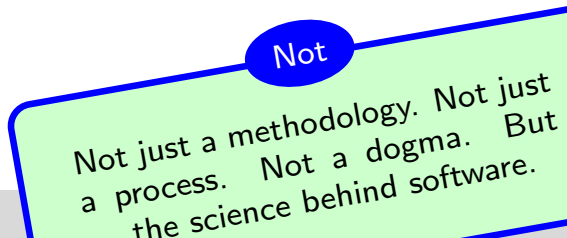
- *Formal software development is many times wrongly associated with an absolute 'certainty that there are no faults'*
- This is not so and cannot be so
- We can never be certain of our formal systems themselves; this is actually a theorem of metamathematics, not an ideological position
- We cannot prove that ZFC is consistent without using principles that are outside of ZFC, we just have enough evidence that it is

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

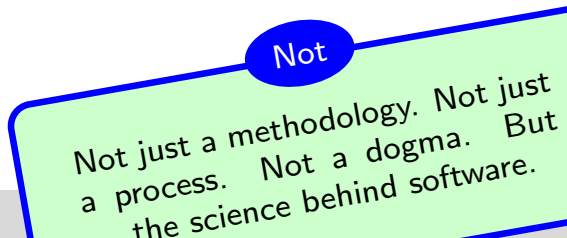
Are formal methods providing certainty?

- *Formal software development is many times wrongly associated with an absolute 'certainty that there are no faults'*
- This is not so and cannot be so
- We can never be certain of our formal systems themselves; this is actually a theorem of metamathematics, not an ideological position
- We cannot prove that ZFC is consistent without using principles that are outside of ZFC, we just have enough evidence that it is
- We can prove ZFC consistency in a more powerful system than ZFC (which by the way is not finitary anymore), and then this system needs a proof of consistency in a still more powerful system, in a never ending chain.



Are formal methods providing certainty?

- *Formal software development is many times wrongly associated with an absolute 'certainty that there are no faults'*
- This is not so and cannot be so
- We can never be certain of our formal systems themselves; this is actually a theorem of metamathematics, not an ideological position
- We cannot prove that ZFC is consistent without using principles that are outside of ZFC, we just have enough evidence that it is
- We can prove ZFC consistency in a more powerful system than ZFC (which by the way is not finitary anymore), and then this system needs a proof of consistency in a still more powerful system, in a never ending chain.
- We cannot bootstrap this process.



If we cannot do that, what can we do?

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

If we cannot do that, what can we do?

What we can and do achieve with formalism is proving correctness relative to a given formal system, i.e. relative to a set of rules, and under a set of assumptions that we need to fully spell out.

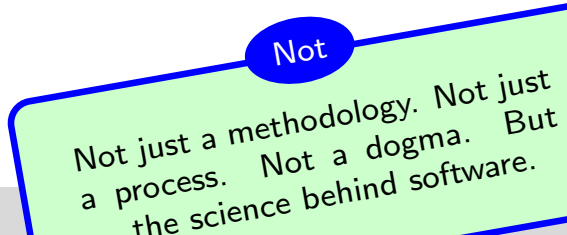
Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

If we cannot do that, what can we do?

What we can and do achieve with formalism is proving correctness relative to a given formal system, i.e. relative to a set of rules, and under a set of assumptions that we need to fully spell out.

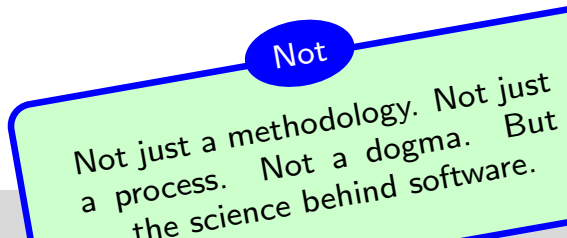
That's all, and that's all we'll ever get.



If we cannot do that, what can we do?

What we can and do achieve with formalism is proving correctness relative to a given formal system, i.e. relative to a set of rules, and under a set of assumptions that we need to fully spell out.

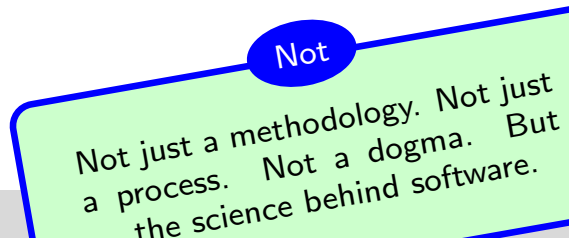
That's all, and that's all we'll ever get. But this is also all that we need to get!



If we cannot do that, what can we do?

What we can and do achieve with formalism is proving correctness relative to a given formal system, i.e. relative to a set of rules, and under a set of assumptions that we need to fully spell out.

That's all, and that's all we'll ever get. But this is also all that we need to get! This qualified claim of correctness is based on a chain of trust, shown in the following slide.



Relative correctness

The chain of trust

Assuming that:

- the hardware and the OS work properly
- the compiler of your logical tool works properly
- the logic behind the tool is consistent
- the tool implements the logic correctly
- your theory is consistent with the logic
- your proof is checked by the logical tool

then

- your proof is correct.

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Strongest claim we can make

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Strongest claim we can make

- This relative correctness is the strongest claim we can make

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Strongest claim we can make

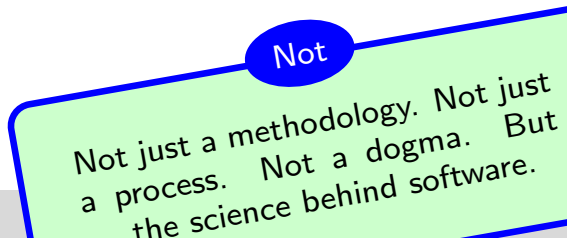
- This relative correctness is the strongest claim we can make
- No other approach to software development can give stronger assurances

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

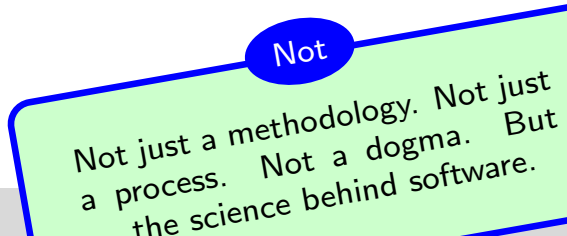
Strongest claim we can make

- This relative correctness is the strongest claim we can make
- No other approach to software development can give stronger assurances
- *The most likely failure would be that your theory is not consistent with the prover's logic*



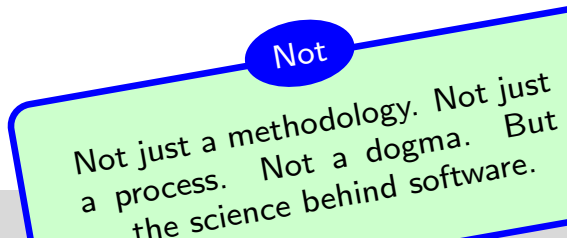
Strongest claim we can make

- This relative correctness is the strongest claim we can make
- No other approach to software development can give stronger assurances
- *The most likely failure would be that your theory is not consistent with the prover's logic*
- If the hardware fails, all hell breaks loose



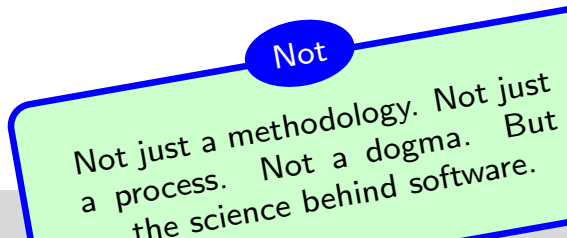
Strongest claim we can make

- This relative correctness is the strongest claim we can make
- No other approach to software development can give stronger assurances
- *The most likely failure would be that your theory is not consistent with the prover's logic*
- If the hardware fails, all hell breaks loose
- (formal hardware verification uses some of the same provers we study)



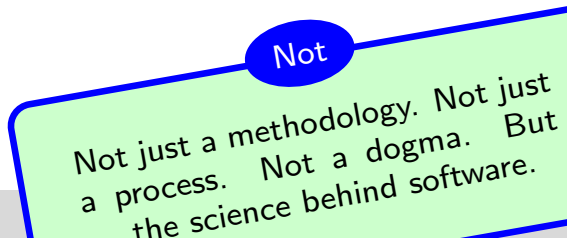
Strongest claim we can make

- This relative correctness is the strongest claim we can make
- No other approach to software development can give stronger assurances
- *The most likely failure would be that your theory is not consistent with the prover's logic*
- If the hardware fails, all hell breaks loose
- (formal hardware verification uses some of the same provers we study)
- If the compiler or the logical tool fail, the failure will likely show up in more obvious places



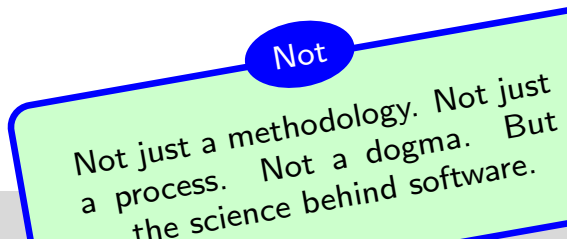
Strongest claim we can make

- This relative correctness is the strongest claim we can make
- No other approach to software development can give stronger assurances
- *The most likely failure would be that your theory is not consistent with the prover's logic*
- If the hardware fails, all hell breaks loose
- (formal hardware verification uses some of the same provers we study)
- If the compiler or the logical tool fail, the failure will likely show up in more obvious places
- (much work is done in formal compiler and tool verification)



Strongest claim we can make

- This relative correctness is the strongest claim we can make
- No other approach to software development can give stronger assurances
- *The most likely failure would be that your theory is not consistent with the prover's logic*
- If the hardware fails, all hell breaks loose
- (formal hardware verification uses some of the same provers we study)
- If the compiler or the logical tool fail, the failure will likely show up in more obvious places
- (much work is done in formal compiler and tool verification)
- The correctness of a tool's implementation is a subject of much research



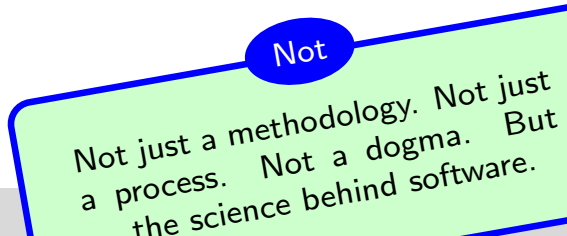
Not just another methodology or process

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

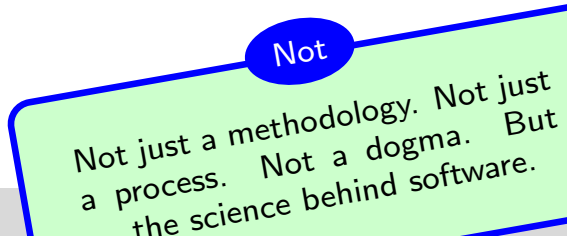
Not just another methodology or process

- We have had many methodologies: structured, object-oriented, extreme, test-driven, ...



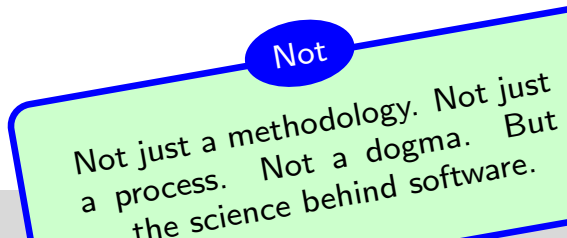
Not just another methodology or process

- We have had many methodologies: structured, object-oriented, extreme, test-driven, ...
- Formal development is not just another methodology!



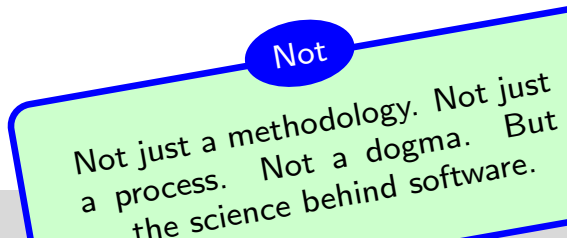
Not just another methodology or process

- We have had many methodologies: structured, object-oriented, extreme, test-driven, ...
- Formal development is not just another methodology!
- (chemistry is not just a methodology to be used by chemical engineers)



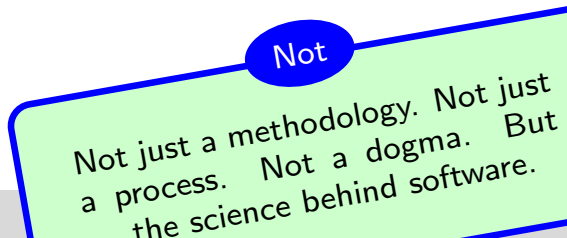
Not just another methodology or process

- We have had many methodologies: structured, object-oriented, extreme, test-driven, ...
- Formal development is not just another methodology!
- (chemistry is not just a methodology to be used by chemical engineers)
- There is of course some methodology and process associated with formal methods



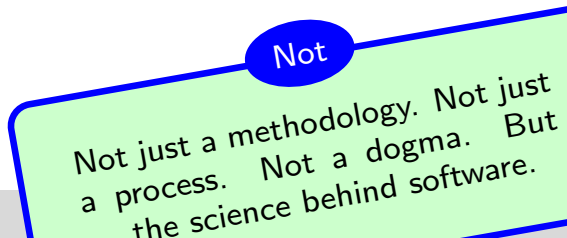
Not just another methodology or process

- We have had many methodologies: structured, object-oriented, extreme, test-driven, ...
- Formal development is not just another methodology!
- (chemistry is not just a methodology to be used by chemical engineers)
- There is of course some methodology and process associated with formal methods
- We will actually spend some time looking at formal methodology and process



Not just another methodology or process

- We have had many methodologies: structured, object-oriented, extreme, test-driven, ...
- Formal development is not just another methodology!
- (chemistry is not just a methodology to be used by chemical engineers)
- There is of course some methodology and process associated with formal methods
- We will actually spend some time looking at formal methodology and process
- But the main point is that formal systems are the scientific basis of software



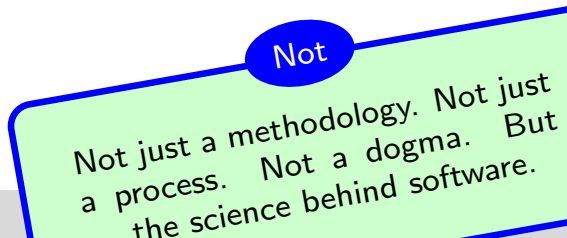
Are there two kinds of software?

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Are there two kinds of software?

- It is said that formal methods are too expensive



Are there two kinds of software?

- It is said that formal methods are too expensive
- . . . and are only used in ‘critical software’

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Are there two kinds of software?

- It is said that formal methods are too expensive
- . . . and are only used in 'critical software'
- In other words, we have 2 kinds of software

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Are there two kinds of software?

- It is said that formal methods are too expensive
- . . . and are only used in 'critical software'
- In other words, we have 2 kinds of software
  - The kind that is not OK to fail

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Are there two kinds of software?

- It is said that formal methods are too expensive
- . . . and are only used in 'critical software'
- In other words, we have 2 kinds of software
  - The kind that is not OK to fail
  - The kind that is OK to fail

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Are there two kinds of software?

- It is said that formal methods are too expensive
- . . . and are only used in 'critical software'
- In other words, we have 2 kinds of software
  - The kind that is not OK to fail
  - The kind that is OK to fail

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Are there two kinds of software?

- It is said that formal methods are too expensive
- . . . and are only used in 'critical software'
- In other words, we have 2 kinds of software
  - The kind that is not OK to fail
  - The kind that is OK to fail

It this true?

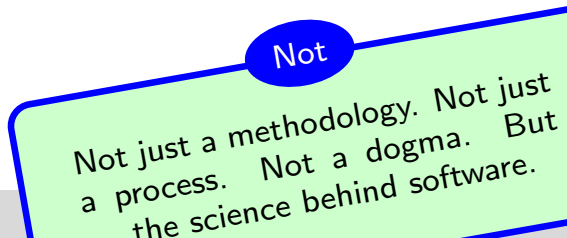
Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Are there two kinds of software?

- It is said that formal methods are too expensive
- . . . and are only used in 'critical software'
- In other words, we have 2 kinds of software
  - The kind that is not OK to fail
  - The kind that is OK to fail

It this true? Of course it is true, and if we base our value judgements on testing, software will always fall into the second category, regardless of when the testing is done or how good it is. Quite often, decision makers are not even aware that there is an alternative.

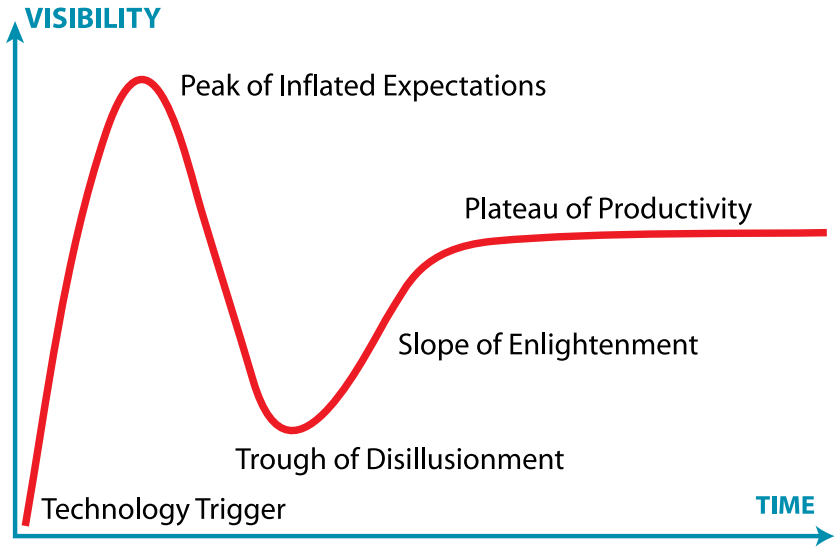


The hype issue

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

The hype issue

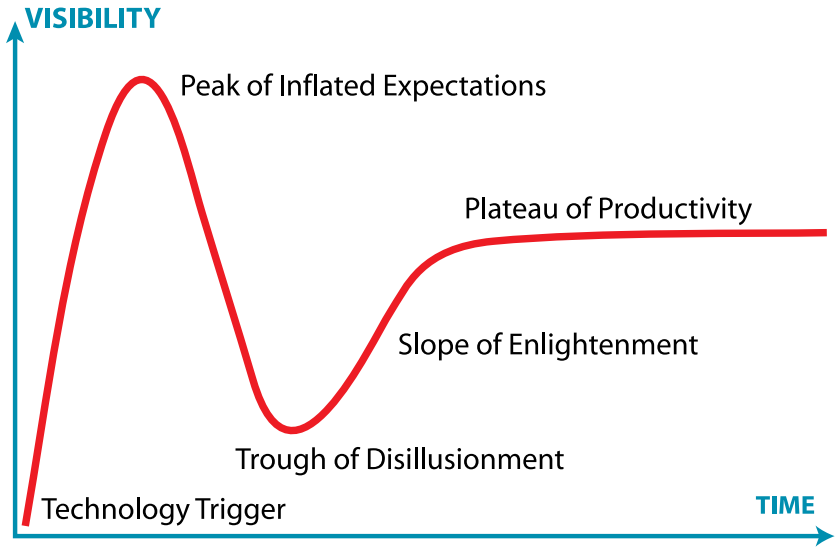


On Wikipedia, the 'hype curve' is defined as a 'graphic representation of the maturity, adoption and social application of specific technologies'. It matches quite well with the trajectory of many technologies that were used for software development.

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

The hype issue



On Wikipedia, the 'hype curve' is defined as a 'graphic representation of the maturity, adoption and social application of specific technologies'. It matches quite well with the trajectory of many technologies that were used for software development.

It has also been mentioned in the context of formal methods. *But formal methods are not just a technology. Their scientific component has continuously advanced upward, as any science does.*

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

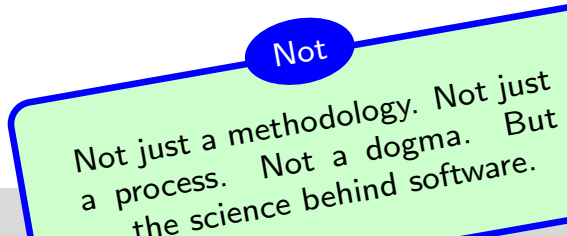
Not just software with math but also math with software

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

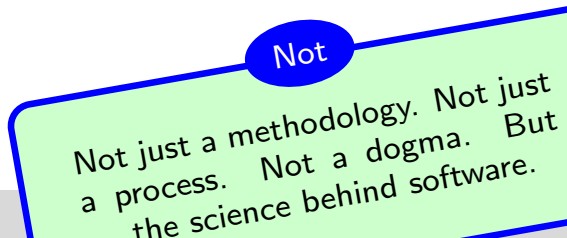
Not just software with math but also math with software

- Our definition shows a one-way dependency of formal software development on mathematics



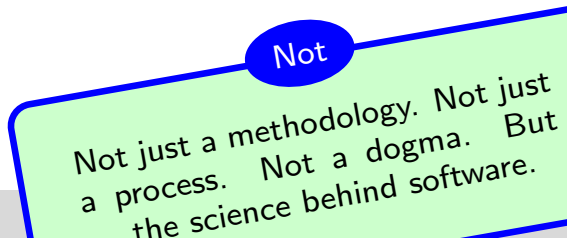
Not just software with math but also math with software

- Our definition shows a one-way dependency of formal software development on mathematics
- (this fits our purpose)



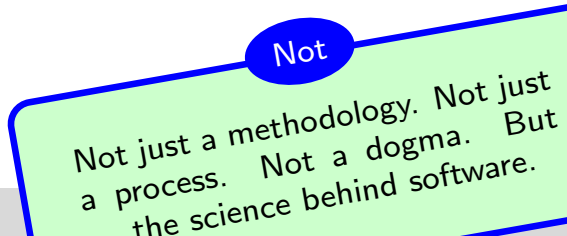
Not just software with math but also math with software

- Our definition shows a one-way dependency of formal software development on mathematics
- (this fits our purpose)
- However, software engineering also has an increasing impact on mathematics



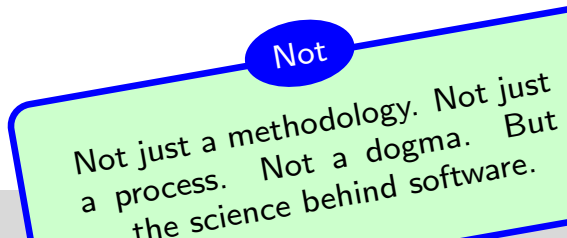
Not just software with math but also math with software

- Our definition shows a one-way dependency of formal software development on mathematics
- (this fits our purpose)
- However, software engineering also has an increasing impact on mathematics
  - Some important math has been done and is being done with software systems



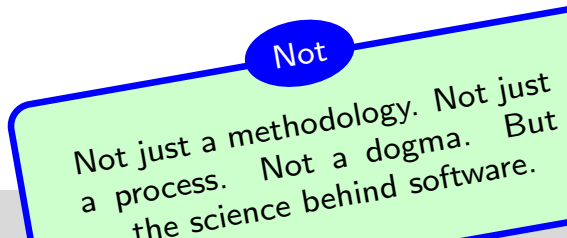
Not just software with math but also math with software

- Our definition shows a one-way dependency of formal software development on mathematics
- (this fits our purpose)
- However, software engineering also has an increasing impact on mathematics
  - Some important math has been done and is being done with software systems
  - Despite a barrage of limiting results from mathematics in the 1930's, efficient algorithms and structures have been developed for very hard problems



Not just software with math but also math with software

- Our definition shows a one-way dependency of formal software development on mathematics
- (this fits our purpose)
- However, software engineering also has an increasing impact on mathematics
  - Some important math has been done and is being done with software systems
  - Despite a barrage of limiting results from mathematics in the 1930's, efficient algorithms and structures have been developed for very hard problems
  - The extreme cases that are behind those limiting results of mathematics rarely pop up in practical situations (the kind we encounter in software development)



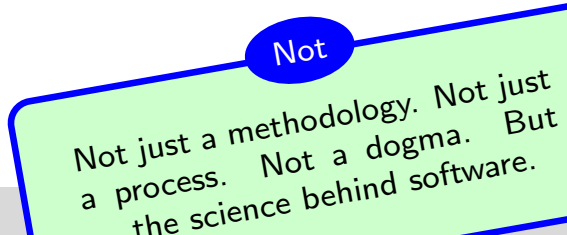
Proofs do not replace engineering

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

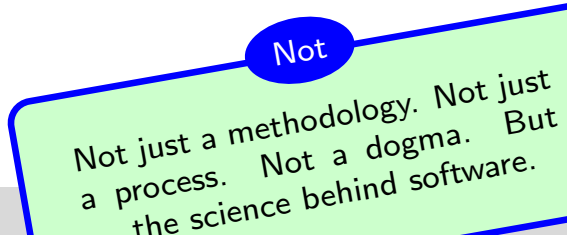
Proofs do not replace engineering

- As we have said already, proofs are *relative* to a specified formal system



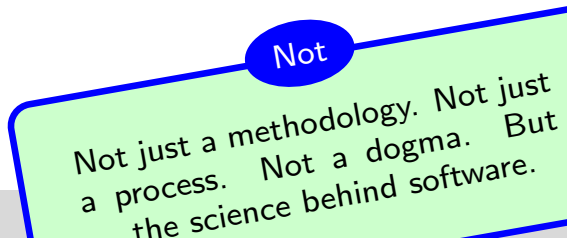
Proofs do not replace engineering

- As we have said already, proofs are *relative* to a specified formal system
- They are not a reflection of reality



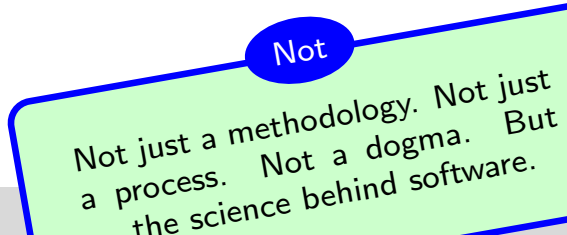
Proofs do not replace engineering

- As we have said already, proofs are *relative* to a specified formal system
- They are not a reflection of reality
- Whether the formal system captures the form of an intended reality is a different issue



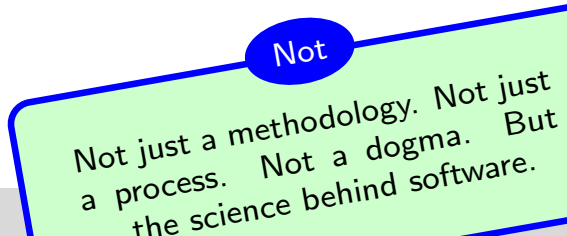
Proofs do not replace engineering

- As we have said already, proofs are *relative* to a specified formal system
- They are not a reflection of reality
- Whether the formal system captures the form of an intended reality is a different issue
- That issue is important though, and *only our engineering expertize can address it*



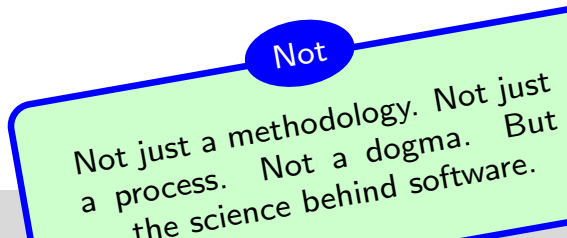
Proofs do not replace engineering

- As we have said already, proofs are *relative* to a specified formal system
- They are not a reflection of reality
- Whether the formal system captures the form of an intended reality is a different issue
- That issue is important though, and *only our engineering expertize can address it*
- So, engineering expertize is essential for the effective applicability of formal methods



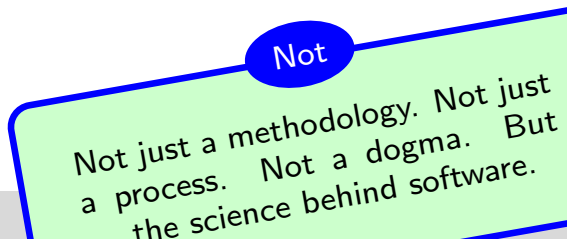
Proofs do not replace engineering

- As we have said already, proofs are *relative* to a specified formal system
- They are not a reflection of reality
- Whether the formal system captures the form of an intended reality is a different issue
- That issue is important though, and *only our engineering expertize can address it*
- So, engineering expertize is essential for the effective applicability of formal methods
- Very likely that engineering expertize could also help with the effort to produce a more usable formalization of mathematics itself



Proofs do not replace engineering

- As we have said already, proofs are *relative* to a specified formal system
- They are not a reflection of reality
- Whether the formal system captures the form of an intended reality is a different issue
- That issue is important though, and *only our engineering expertize can address it*
- So, engineering expertize is essential for the effective applicability of formal methods
- Very likely that engineering expertize could also help with the effort to produce a more usable formalization of mathematics itself
- The same way that the engineering of the Large Hadron Collider is helping physicists do their work



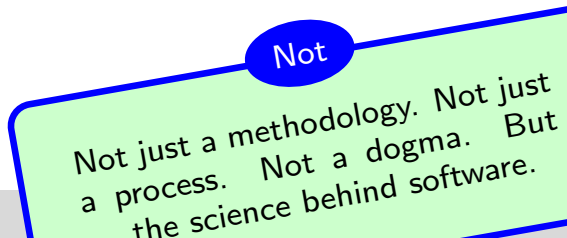
Do formal methods eliminate the need for informal testing?

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

Do formal methods eliminate the need for informal testing?

- Absolutely not!



Do formal methods eliminate the need for informal testing?

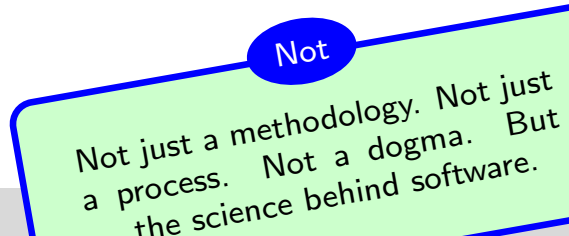
- Absolutely not!
- Any link in the chain of trust can fail!

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

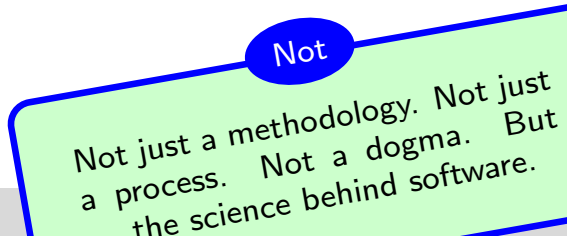
Do formal methods eliminate the need for informal testing?

- Absolutely not!
- Any link in the chain of trust can fail!
- Informal testing needs to exclude common-sense faults . . .



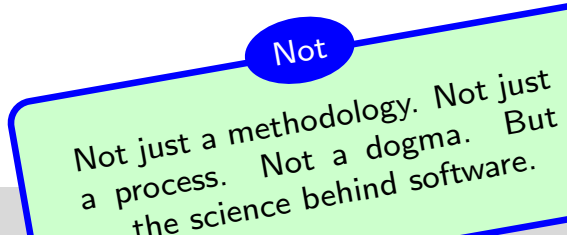
Do formal methods eliminate the need for informal testing?

- Absolutely not!
- Any link in the chain of trust can fail!
- Informal testing needs to exclude common-sense faults ...
- There are various estimates for the radius of the universe. One of the original formulas for such a radius was a beautiful one and had a beautiful proof. It took years before someone bothered to plug in all the numbers and do the calculation.



Do formal methods eliminate the need for informal testing?

- Absolutely not!
- Any link in the chain of trust can fail!
- Informal testing needs to exclude common-sense faults ...
- There are various estimates for the radius of the universe. One of the original formulas for such a radius was a beautiful one and had a beautiful proof. It took years before someone bothered to plug in all the numbers and do the calculation. *It came to about 4 inches.*



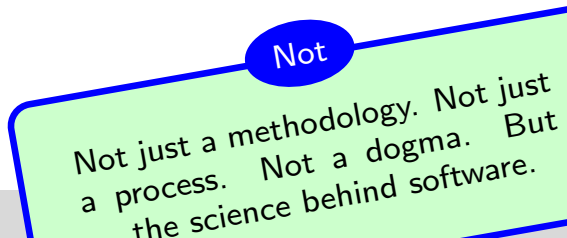
Should everyone use formal methods?

Not

Not just a methodology. Not just a process. Not a dogma. But the science behind software.

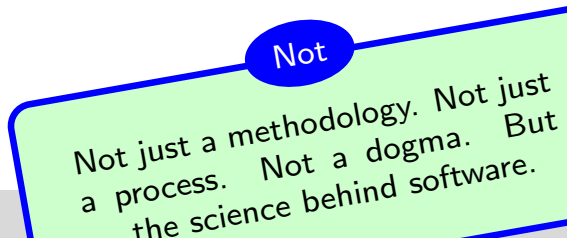
Should everyone use formal methods?

- Every organization would benefit from a gradual introduction of formal methods



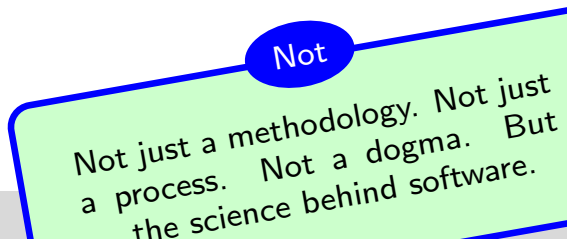
Should everyone use formal methods?

- Every organization would benefit from a gradual introduction of formal methods
- What about the good software that has been developed without any formalism? Quick example: STL and Boost C++ libraries



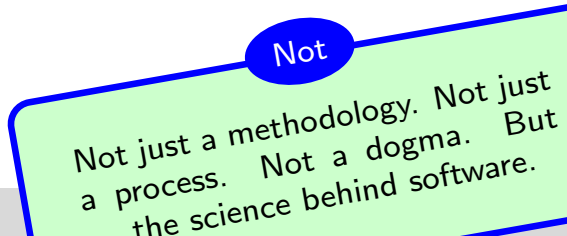
Should everyone use formal methods?

- Every organization would benefit from a gradual introduction of formal methods
- What about the good software that has been developed without any formalism? Quick example: STL and Boost C++ libraries
- These libraries have some mathematical background, giving guarantees of performance for the algorithms



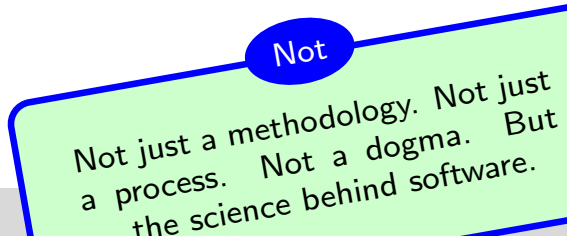
Should everyone use formal methods?

- Every organization would benefit from a gradual introduction of formal methods
- What about the good software that has been developed without any formalism? Quick example: STL and Boost C++ libraries
- These libraries have some mathematical background, giving guarantees of performance for the algorithms
- *These guarantees are not formal though*, there is no proof



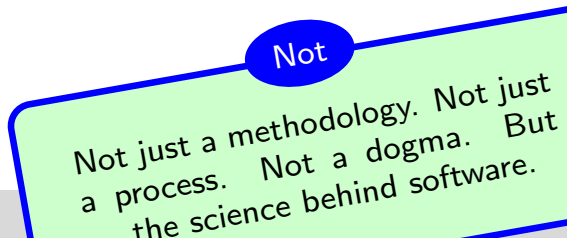
Should everyone use formal methods?

- Every organization would benefit from a gradual introduction of formal methods
- What about the good software that has been developed without any formalism? Quick example: STL and Boost C++ libraries
- These libraries have some mathematical background, giving guarantees of performance for the algorithms
- *These guarantees are not formal though*, there is no proof
- Imagine libraries that can give formal guarantees, with proofs that can be independently verified



Should everyone use formal methods?

- Every organization would benefit from a gradual introduction of formal methods
- What about the good software that has been developed without any formalism? Quick example: STL and Boost C++ libraries
- These libraries have some mathematical background, giving guarantees of performance for the algorithms
- *These guarantees are not formal though*, there is no proof
- Imagine libraries that can give formal guarantees, with proofs that can be independently verified
- Once more, the benefits of formal systems are that *they lead to a different way of thinking about software, any software, not just critical software*



Formal verification of programs

Formal verification of programs

- In a sense narrower than formal software development, *formal software verification* is the use of mathematical techniques to *prove properties of programs*: correctness, termination, deadlock-freedom, etc ...

Formal verification of programs

- In a sense narrower than formal software development, *formal software verification* is the use of mathematical techniques to *prove properties of programs*: correctness, termination, deadlock-freedom, etc ...
- Very often, when people talk about formal software, *they mean formal verification of programs*

Formal verification of programs

- In a sense narrower than formal software development, *formal software verification* is the use of mathematical techniques to *prove properties of programs*: correctness, termination, deadlock-freedom, etc ...
- Very often, when people talk about formal software, *they mean formal verification of programs*
- We repeat that, for many projects, requirements and specifications are the only parts of the development that are formal, because faulty specifications are more expensive than faulty programs

Formal verification of programs

- In a sense narrower than formal software development, *formal software verification* is the use of mathematical techniques to *prove properties of programs*: correctness, termination, deadlock-freedom, etc ...
- Very often, when people talk about formal software, *they mean formal verification of programs*
- We repeat that, for many projects, requirements and specifications are the only parts of the development that are formal, because faulty specifications are more expensive than faulty programs
- So, although the topic of this section is formal program verification, we always *keep the entire formal development cycle in mind*

Verification of functional and imperative programs

Verification of functional and imperative programs

- Functional languages begin high on the abstraction level, staying close to our thinking, and our mathematics; they emphasize expressiveness

Verification of functional and imperative programs

- Functional languages begin high on the abstraction level, staying close to our thinking, and our mathematics; they emphasize expressiveness
- Imperative languages begin low on the abstraction level, staying close to the machine; they emphasize efficiency

Verification of functional and imperative programs

- Functional languages begin high on the abstraction level, staying close to our thinking, and our mathematics; they emphasize expressiveness
- Imperative languages begin low on the abstraction level, staying close to the machine; they emphasize efficiency
- The two paradigms are *complementary*

Verification of functional and imperative programs

- Functional languages begin high on the abstraction level, staying close to our thinking, and our mathematics; they emphasize expressiveness
- Imperative languages begin low on the abstraction level, staying close to the machine; they emphasize efficiency
- The two paradigms are *complementary*
- We need to understand how to verify both types of programs

Verification of functional and imperative programs

- Functional languages begin high on the abstraction level, staying close to our thinking, and our mathematics; they emphasize expressiveness
- Imperative languages begin low on the abstraction level, staying close to the machine; they emphasize efficiency
- The two paradigms are *complementary*
- We need to understand how to verify both types of programs
- We begin by looking at functional programs first

Simple functional programs are proofs

Simple functional programs are proofs

Simple functional models of computation are structurally indistinguishable from certain deductive systems of logic. *This correspondence, known as the Curry-Howard correspondence, is a pillar of our program.*

Simple functional programs are proofs

Simple functional models of computation are structurally indistinguishable from certain deductive systems of logic. *This correspondence, known as the Curry-Howard correspondence, is a pillar of our program.*

The correspondence has an equational content, an application of *category theory*. We say more about this need for category theory later, for now let's note that the correspondence gives us a dictionary from functional programming practice to theory:

Simple functional programs are proofs

Simple functional models of computation are structurally indistinguishable from certain deductive systems of logic. *This correspondence, known as the Curry-Howard correspondence, is a pillar of our program.*

The correspondence has an equational content, an application of *category theory*. We say more about this need for category theory later, for now let's note that the correspondence gives us a dictionary from functional programming practice to theory:

- Program specifications (types) are theorems (formulas)

Simple functional programs are proofs

Simple functional models of computation are structurally indistinguishable from certain deductive systems of logic. *This correspondence, known as the Curry-Howard correspondence, is a pillar of our program.*

The correspondence has an equational content, an application of *category theory*. We say more about this need for category theory later, for now let's note that the correspondence gives us a dictionary from functional programming practice to theory:

- Program specifications (types) are theorems (formulas)
- Programs are proofs

Simple functional programs are proofs

Simple functional models of computation are structurally indistinguishable from certain deductive systems of logic. *This correspondence, known as the Curry-Howard correspondence, is a pillar of our program.*

The correspondence has an equational content, an application of *category theory*. We say more about this need for category theory later, for now let's note that the correspondence gives us a dictionary from functional programming practice to theory:

- Program specifications (types) are theorems (formulas)
- Programs are proofs
- Functions between types are logical implications between formulas

Proving properties of imperative programs

Proving properties of imperative programs

- For us imperative programming means programming in C/Java/C#.

Proving properties of imperative programs

- For us imperative programming means programming in C/Java/C#.
- We saw that simple functional programs are already proofs (Curry-Howard correspondence)

Proving properties of imperative programs

- For us imperative programming means programming in C/Java/C#.
- We saw that simple functional programs are already proofs (Curry-Howard correspondence)
- *It is much harder to prove properties of imperative programs.*

Proving properties of imperative programs

- For us imperative programming means programming in C/Java/C#.
- We saw that simple functional programs are already proofs (Curry-Howard correspondence)
- *It is much harder to prove properties of imperative programs.*
- We study a few approaches to the verification of imperative programs; all approaches transform the program or parts of it to semantically equivalent functional programs, relative to a fixed logical framework:

Proving properties of imperative programs

- For us imperative programming means programming in C/Java/C#.
- We saw that simple functional programs are already proofs (Curry-Howard correspondence)
- *It is much harder to prove properties of imperative programs.*
- We study a few approaches to the verification of imperative programs; all approaches transform the program or parts of it to semantically equivalent functional programs, relative to a fixed logical framework:
  1. Using the axiomatic semantics of the programming language (probably the most intuitive approach)

Proving properties of imperative programs

- For us imperative programming means programming in C/Java/C#.
- We saw that simple functional programs are already proofs (Curry-Howard correspondence)
- *It is much harder to prove properties of imperative programs.*
- We study a few approaches to the verification of imperative programs; all approaches transform the program or parts of it to semantically equivalent functional programs, relative to a fixed logical framework:
  1. Using the axiomatic semantics of the programming language (probably the most intuitive approach)
  2. Using other semantics

Proving properties of imperative programs

- For us imperative programming means programming in C/Java/C#.
- We saw that simple functional programs are already proofs (Curry-Howard correspondence)
- *It is much harder to prove properties of imperative programs.*
- We study a few approaches to the verification of imperative programs; all approaches transform the program or parts of it to semantically equivalent functional programs, relative to a fixed logical framework:
  1. Using the axiomatic semantics of the programming language (probably the most intuitive approach)
  2. Using other semantics
  3. Using effect types

Proving properties of imperative programs

- For us imperative programming means programming in C/Java/C#.
- We saw that simple functional programs are already proofs (Curry-Howard correspondence)
- *It is much harder to prove properties of imperative programs.*
- We study a few approaches to the verification of imperative programs; all approaches transform the program or parts of it to semantically equivalent functional programs, relative to a fixed logical framework:
  1. Using the axiomatic semantics of the programming language (probably the most intuitive approach)
  2. Using other semantics
  3. Using effect types
  4. Using monads (another reason for us to study category theory, this being the more general approach; the other methods can be described through monads)

Why don't we just switch to functional languages?

Why don't we just switch to functional languages?

- Because the hardware matches imperative languages better

Why don't we just switch to functional languages?

- Because the hardware matches imperative languages better
- There are yet no Intel or NVIDIA chips based on lambda-calculus; they would need specialized stacks, very fast and very large

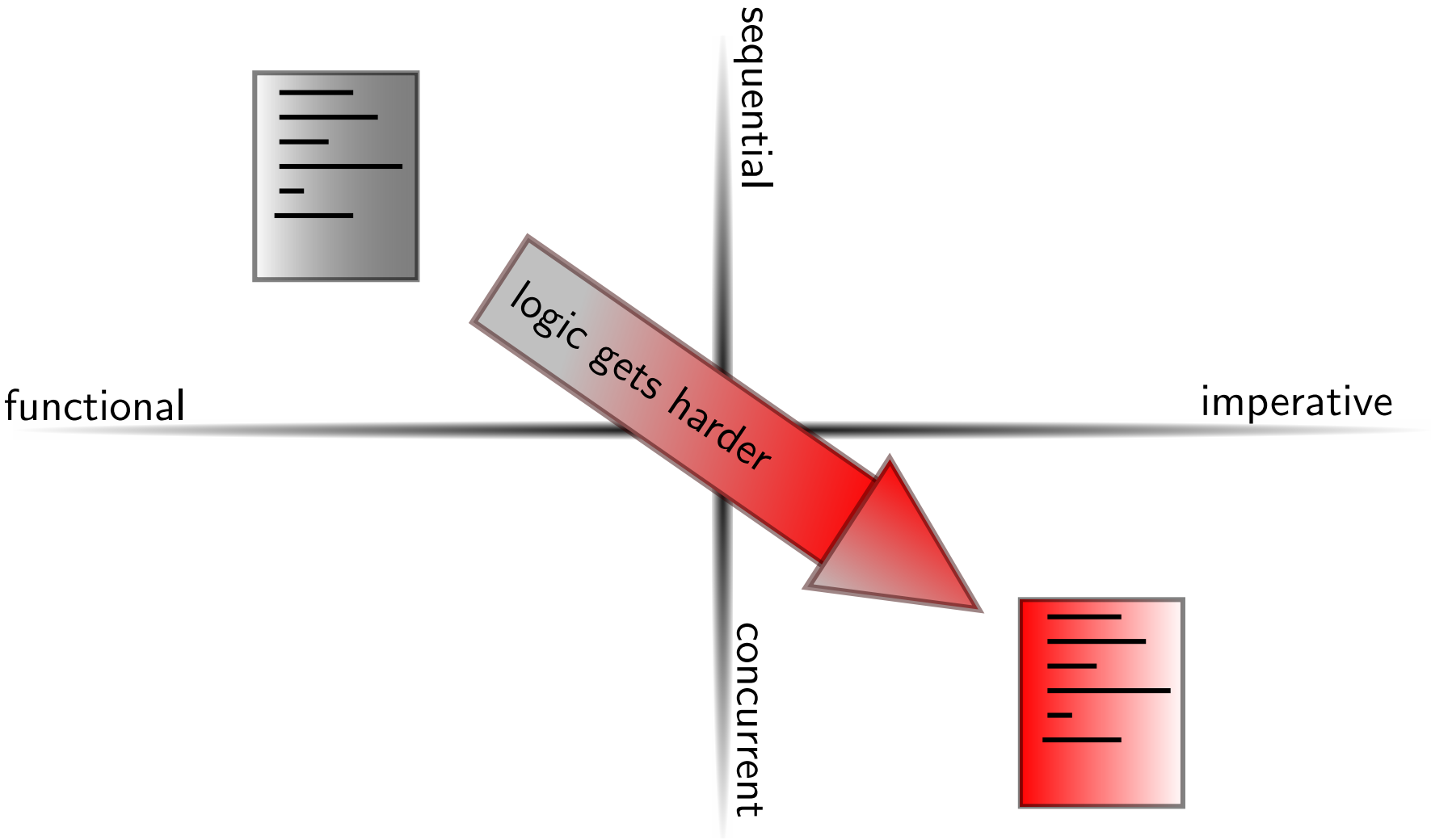
Why don't we just switch to functional languages?

- Because the hardware matches imperative languages better
- There are yet no Intel or NVIDIA chips based on lambda-calculus; they would need specialized stacks, very fast and very large
- And functional languages do not support concurrency well; the fact that referential transparency leads to the *possibility of concurrency* is not really useful. We still need to support explicit concurrency, in the hands of the programmer.

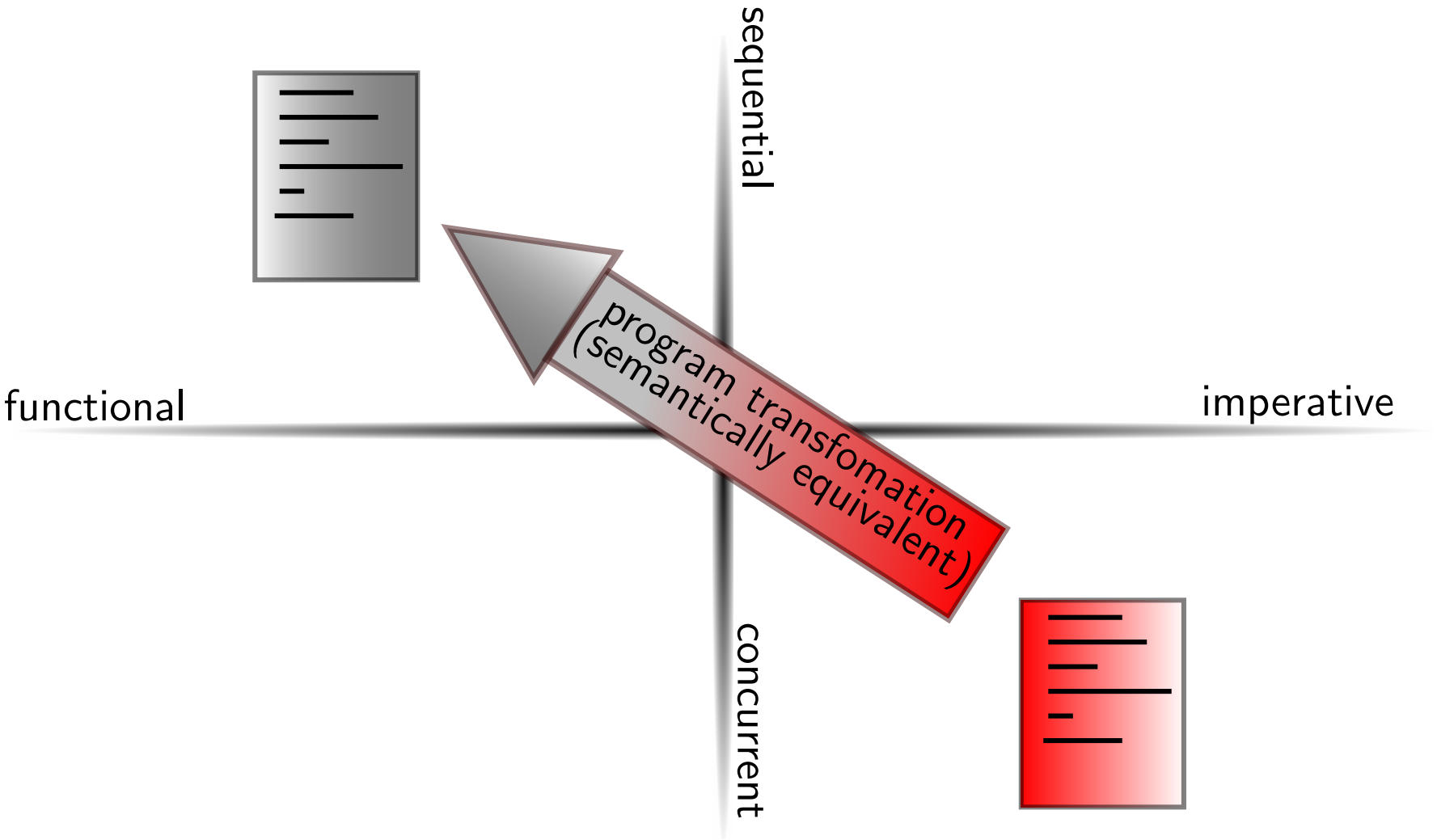
Why don't we just switch to functional languages?

- Because the hardware matches imperative languages better
- There are yet no Intel or NVIDIA chips based on lambda-calculus; they would need specialized stacks, very fast and very large
- And functional languages do not support concurrency well; the fact that referential transparency leads to the *possibility of concurrency* is not really useful. We still need to support explicit concurrency, in the hands of the programmer.
- Gödel himself was not convinced that his theory of recursive functions or Church's lambda calculus captured the concept of computability until Turing proposed his machine; imperative models are usually more convincing.

Impact of concurrency



Solution: program transformations



Correctness properties

Correctness properties

What properties do we verify?

Correctness properties

What properties do we verify? At the high end sits the *total correctness* of the program relative to a specification.

Correctness properties

What properties do we verify? At the high end sits the *total correctness* of the program relative to a specification.

We will study ways to produce programs that are correct-by-construction, meaning that the code and the proof of total correctness are developed at the same time. This correctness-by-construction is usually accomplished for functional programs only.

Correctness properties

What properties do we verify? At the high end sits the *total correctness* of the program relative to a specification.

We will study ways to produce programs that are correct-by-construction, meaning that the code and the proof of total correctness are developed at the same time. This correctness-by-construction is usually accomplished for functional programs only.

Total correctness is composed of two properties that can be proven separately:

Correctness properties

What properties do we verify? At the high end sits the *total correctness* of the program relative to a specification.

We will study ways to produce programs that are correct-by-construction, meaning that the code and the proof of total correctness are developed at the same time. This correctness-by-construction is usually accomplished for functional programs only.

Total correctness is composed of two properties that can be proven separately:

- *Partial correctness* (*if* delivered, results are correct)

Correctness properties

What properties do we verify? At the high end sits the *total correctness* of the program relative to a specification.

We will study ways to produce programs that are correct-by-construction, meaning that the code and the proof of total correctness are developed at the same time. This correctness-by-construction is usually accomplished for functional programs only.

Total correctness is composed of two properties that can be proven separately:

- *Partial correctness* (*if* delivered, results are correct)
- *Termination* (results *are* delivered)

Safety and liveness properties

Safety and liveness properties

Partial correctness relative to a specification and termination are not the only important properties. We will see that many other desirable properties of programs can be formally verified:

Safety and liveness properties

Partial correctness relative to a specification and termination are not the only important properties. We will see that many other desirable properties of programs can be formally verified:

- *General safety*: no arithmetic overflow, no buffer overflow, no null pointer dereferencing, no division by zero

Safety and liveness properties

Partial correctness relative to a specification and termination are not the only important properties. We will see that many other desirable properties of programs can be formally verified:

- *General safety*: no arithmetic overflow, no buffer overflow, no null pointer dereferencing, no division by zero
- *Heap safety*: no dereferencing of deallocated heap, no memory leaks

Safety and liveness properties

Partial correctness relative to a specification and termination are not the only important properties. We will see that many other desirable properties of programs can be formally verified:

- *General safety*: no arithmetic overflow, no buffer overflow, no null pointer dereferencing, no division by zero
- *Heap safety*: no dereferencing of deallocated heap, no memory leaks
- *Liveness* (program *does not* terminate)

Concurrency properties

Concurrency properties

Proving concurrency properties of imperative programs is very laborious. We will attack such properties only after you have accumulated enough experience with proofs of sequential properties. In order of difficulty, we will look at proofs of:

Concurrency properties

Proving concurrency properties of imperative programs is very laborious. We will attack such properties only after you have accumulated enough experience with proofs of sequential properties. In order of difficulty, we will look at proofs of:

- *Interference freedom* (needed if the program uses shared variables)

Concurrency properties

Proving concurrency properties of imperative programs is very laborious. We will attack such properties only after you have accumulated enough experience with proofs of sequential properties. In order of difficulty, we will look at proofs of:

- *Interference freedom* (needed if the program uses shared variables)
- *Deadlock freedom* (needed if the program uses synchronization objects)

Concurrency properties

Proving concurrency properties of imperative programs is very laborious. We will attack such properties only after you have accumulated enough experience with proofs of sequential properties. In order of difficulty, we will look at proofs of:

- *Interference freedom* (needed if the program uses shared variables)
- *Deadlock freedom* (needed if the program uses synchronization objects)
- *Fairness* (needed to ensure that all components that are ready for execution do indeed get a chance to execute)

Debug assertions

Debug assertions

We saw that imperative programs are harder to reason about. Assertions are one of the more intuitive ways to reason about imperative programs.

Debug assertions

We saw that imperative programs are harder to reason about. Assertions are one of the more intuitive ways to reason about imperative programs.

- Assertions are not exactly new to software engineers

Debug assertions

We saw that imperative programs are harder to reason about. Assertions are one of the more intuitive ways to reason about imperative programs.

- Assertions are not exactly new to software engineers
- We use debug assertions to validate our understanding of what the running state should be at some control point

Debug assertions

We saw that imperative programs are harder to reason about. Assertions are one of the more intuitive ways to reason about imperative programs.

- Assertions are not exactly new to software engineers
- We use debug assertions to validate our understanding of what the running state should be at some control point
- We set debugger breakpoints at control points to confirm a certain scenario. These scenarios are in fact implicit assertions, many times including logical connectives, assertions which the debugger may or may not confirm; the scenarios are mostly based on informal API documentation.

Assertions moved to compile time

Assertions moved to compile time

Essentially what we do when formally verifying imperative programs is *move assertions from run time to compile time*.

Assertions moved to compile time

Essentially what we do when formally verifying imperative programs is *move assertions from run time to compile time*.

- Now we must make the assertions explicit and attach them to critical control points in the code with the understanding that they must be valid for all the states that may reach that control point.

Assertions moved to compile time

Essentially what we do when formally verifying imperative programs is *move assertions from run time to compile time*.

- Now we must make the assertions explicit and attach them to critical control points in the code with the understanding that they must be valid for all the states that may reach that control point.
- Some of these assertions are pre-conditions and post-conditions that we attach to all the function definitions in our code.

Assertions moved to compile time

Essentially what we do when formally verifying imperative programs is *move assertions from run time to compile time*.

- Now we must make the assertions explicit and attach them to critical control points in the code with the understanding that they must be valid for all the states that may reach that control point.
- Some of these assertions are pre-conditions and post-conditions that we attach to all the function definitions in our code.
- The verifying compiler (a smarter compiler, backed by a logical tool) combines these assertions with the formal semantics of the programming language being used and obtains formulas (i.e. theorems) which its logical back-end must subsequently prove.

Axiomatic semantics

Axiomatic semantics

One of the most intuitive and most used formal semantics for imperative programming languages. Your program s is a composition of various commands written in the language. Then its behavior can be viewed as a state (or predicate) transformer:

$$\{P\}\ s\ \{Q\}$$

where P is a pre-condition (a predicate on the state of the system) and Q is the post-condition (another predicate on the state of the system).

Axiomatic semantics

One of the most intuitive and most used formal semantics for imperative programming languages. Your program s is a composition of various commands written in the language. Then its behavior can be viewed as a state (or predicate) transformer:

$$\{P\}\ s\ \{Q\}$$

where P is a pre-condition (a predicate on the state of the system) and Q is the post-condition (another predicate on the state of the system).

- Need to have the semantics of each statement in the language

Axiomatic semantics

One of the most intuitive and most used formal semantics for imperative programming languages. Your program s is a composition of various commands written in the language. Then its behavior can be viewed as a state (or predicate) transformer:

$$\{P\}\ s\ \{Q\}$$

where P is a pre-condition (a predicate on the state of the system) and Q is the post-condition (another predicate on the state of the system).

- Need to have the semantics of each statement in the language
- Each such semantics is given by a rule

Axiomatic semantics

One of the most intuitive and most used formal semantics for imperative programming languages. Your program s is a composition of various commands written in the language. Then its behavior can be viewed as a state (or predicate) transformer:

$$\{P\}\ s\ \{Q\}$$

where P is a pre-condition (a predicate on the state of the system) and Q is the post-condition (another predicate on the state of the system).

- Need to have the semantics of each statement in the language
- Each such semantics is given by a rule
- Then use *induction on the structure of the program* s to prove that if the state of the system satisfies P and s is executed, then upon termination of s, the state of the system satisfies Q. This is partial correctness, because we still need to separately prove that s does indeed terminate.

Assertions for verification, assertions for development

Assertions for verification, assertions for development

- Verification means attaching assertions to existing code

Assertions for verification, assertions for development

- Verification means attaching assertions to existing code
- But assertions could be attached *while writing the code*: JML, Spec#

Assertions for verification, assertions for development

- Verification means attaching assertions to existing code
- But assertions could be attached *while writing the code*: JML, Spec#
- When used this way, assertions are called code contracts; it started with Eiffel

Assertions for verification, assertions for development

- Verification means attaching assertions to existing code
- But assertions could be attached *while writing the code*: JML, Spec#
- When used this way, assertions are called code contracts; it started with Eiffel
- *Having to attach these assertions to code means that we have to understand the code at the time the code is written, not later in the debugger*

Assertions for verification, assertions for development

- Verification means attaching assertions to existing code
- But assertions could be attached *while writing the code*: JML, Spec#
- When used this way, assertions are called code contracts; it started with Eiffel
- *Having to attach these assertions to code means that we have to understand the code at the time the code is written, not later in the debugger*
- So assertions force the writing of better code

Assertions as a form of literate programming

Assertions as a form of literate programming

You may view assertions as a natural progression of the idea that comments come first (because code is written for people just as much as it it written for machines):

Assertions as a form of literate programming

You may view assertions as a natural progression of the idea that comments come first (because code is written for people just as much as it it written for machines):

1. Kernighan & Ritchie: write good comments, i.e. comments that explain the intent of the code

Assertions as a form of literate programming

You may view assertions as a natural progression of the idea that comments come first (because code is written for people just as much as it it written for machines):

1. Kernighan & Ritchie: write good comments, i.e. comments that explain the intent of the code
2. Knuth's literate programming: reverse the order of importance, comments come first, code is the comment

Assertions as a form of literate programming

You may view assertions as a natural progression of the idea that comments come first (because code is written for people just as much as it it written for machines):

1. Kernighan & Ritchie: write good comments, i.e. comments that explain the intent of the code
2. Knuth's literate programming: reverse the order of importance, comments come first, code is the comment
3. Assertions are the best comments, they represent the *logic* behind the program

What about assertions attached to API functions?

What about assertions attached to API functions?

This is clearly an important issue since it may very well be that your code calls more operating-system or platform (Java/.NET) API functions than functions that are defined in your program (and to which you can add as many assertions as you need). These user-mode APIs have complicated semantics. So, are such pre-conditions, post-conditions and invariants being developed for operating-system or platform APIs?

What about assertions attached to API functions?

This is clearly an important issue since it may very well be that your code calls more operating-system or platform (Java/.NET) API functions than functions that are defined in your program (and to which you can add as many assertions as you need). These user-mode APIs have complicated semantics. So, are such pre-conditions, post-conditions and invariants being developed for operating-system or platform APIs?

1. Yes (but not enough), we'll look at some solutions for Linux and Java

What about assertions attached to API functions?

This is clearly an important issue since it may very well be that your code calls more operating-system or platform (Java/.NET) API functions than functions that are defined in your program (and to which you can add as many assertions as you need). These user-mode APIs have complicated semantics. So, are such pre-conditions, post-conditions and invariants being developed for operating-system or platform APIs?

1. Yes (but not enough), we'll look at some solutions for Linux and Java
2. Yes (but not enough), we'll look at some solutions for Windows and .NET

User-mode APIs have complicated semantics

UNIX/LINUX EXAMPLE:

(This example uses traditional System V semaphores, a Posix semaphore would have a similarly complicated semantics.)

NAME int semget(key_t key, int nsems, int semflg);

DESCRIPTION This function returns the semaphore set identifier associated with the argument key. A new set of nsems semaphores is created if key has the value IPC_PRIVATE or if no existing semaphore set is associated to key and IPC_CREAT is asserted in semflg (i.e. semflg & IPC_CREAT isn't zero). The presence in semflg of the fields IPC_CREAT and IPC_EXCL plays the same role, with respect to the existence of the semaphore set, as the presence of O_CREAT and O_EXCL in the mode argument of the open(2) system call: i.e. the semget function fails if semflg asserts both IPC_CREAT and IPC_EXCL and a semaphore set already exists for key. Upon creation, the low-order 9 bits of the argument semflg define the access permissions (for owner, group and others) for the semaphore set. These bits have the same format, and the same meaning, as the mode argument in the open(2) or creat(2) system calls (though the execute permissions are not meaningful for semaphores, and write permissions mean permission to alter semaphore values).

When creating a new semaphore set, semget initializes the semaphore set's associated data structure semid_ds as follows:

sem_perm.cuid and sem_perm.uid are set to the effective user-ID of the calling process. sem_perm.cgid and sem_perm.gid are set to the effective group-ID of the calling process. The low-order 9 bits of sem_perm.mode are set to the low-order 9 bits of semflg. sem_nsems is set to the value of nsems. sem_otime is set to 0. sem_ctime is set to the current time. The argument nsems can be 0 (a don't care) when a semaphore set is not being created. Otherwise nsems must be greater than 0 and less than or equal to the maximum number of semaphores per semaphore set (SEMMSL).

If the semaphore set already exists, the access permissions are verified.

User-mode APIs have complicated semantics

RETURN VALUE If successful, the return value will be the semaphore set identifier (a nonnegative integer), otherwise -1 is returned, with errno indicating the error. ERRORS On failure errno will be set to one of the following: EACCES A semaphore set exists for key, but the calling process does not have permission to access the set. EEXIST A semaphore set exists for key and semflg was asserting both IPC_CREAT and IPC_EXCL. ENOENT No semaphore set exists for key and semflg wasn't asserting IPC_CREAT. EINVAL nsems is less than 0 or greater than the limit on the number of semaphores per semaphore set (SEMMSL), or a semaphore set corresponding to key already exists, and nsems is larger than the number of semaphores in that set. ENOMEM A semaphore set has to be created but the system has not enough memory for the new data structure. ENOSPC A semaphore set has to be created but the system limit for the maximum number of semaphore sets (SEMMNI), or the system wide maximum number of semaphores (SEMMNS), would be exceeded.

User-mode APIs have complicated semantics

Windows example:

Syntax HANDLE WINAPI CreateSemaphore(__in_opt LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, __in LONG lInitialCount, __in LONG lMaximumCount, __in_opt LPCTSTR lpName);

Parameters lpSemaphoreAttributes [in, optional] A pointer to a SECURITY_ATTRIBUTES structure. If this parameter is NULL, the handle cannot be inherited by child processes.

The lpSecurityDescriptor member of the structure specifies a security descriptor for the new semaphore. If this parameter is NULL, the semaphore gets a default security descriptor. The ACLs in the default security descriptor for a semaphore come from the primary or impersonation token of the creator.

lInitialCount [in] The initial count for the semaphore object. This value must be greater than or equal to zero and less than or equal to lMaximumCount. The state of a semaphore is signaled when its count is greater than zero and nonsignaled when it is zero. The count is decreased by one whenever a wait function releases a thread that was waiting for the semaphore. The count is increased by a specified amount by calling the ReleaseSemaphore function.

lMaximumCount [in] The maximum count for the semaphore object. This value must be greater than zero.

lpName [in, optional] The name of the semaphore object. The name is limited to MAX_PATH characters. Name comparison is case sensitive.

If lpName matches the name of an existing named semaphore object, this function requests the SEMAPHORE_ALL_ACCESS access right. In this case, the lInitialCount and lMaximum-Count parameters are ignored because they have already been set by the creating process. If the lpSemaphoreAttributes parameter is not NULL, it determines whether the handle can be inherited, but its security-descriptor member is ignored.

If lpName is NULL, the semaphore object is created without a name.

...

User-mode APIs have complicated semantics

Return Value If the function succeeds, the return value is a handle to the semaphore object. If the named semaphore object existed before the function call, the function returns a handle to the existing object and GetLastError returns ERROR_ALREADY_EXISTS.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks The handle returned by CreateSemaphore has the SEMAPHORE_ALL_ACCESS access right; it can be used in any function that requires a handle to a semaphore object, provided that the caller has been granted access. If an semaphore is created from a service or a thread that is impersonating a different user, you can either apply a security descriptor to the semaphore when you create it, or change the default security descriptor for the creating process by changing its default DACL. For more information, see Synchronization Object Security and Access Rights.

Any thread of the calling process can specify the semaphore-object handle in a call to one of the wait functions. The single-object wait functions return when the state of the specified object is signaled. The multiple-object wait functions can be instructed to return either when any one or when all of the specified objects are signaled. When a wait function returns, the waiting thread is released to continue its execution.

The state of a semaphore object is signaled when its count is greater than zero, and nonsignaled when its count is equal to zero. The lInitialCount parameter specifies the initial count. Each time a waiting thread is released because of the semaphore's signaled state, the count of the semaphore is decreased by one. Use the ReleaseSemaphore function to increment a semaphore's count by a specified amount. The count can never be less than zero or greater than the value specified in the lMaximumCount parameter.

Multiple processes can have handles of the same semaphore object, enabling use of the object for interprocess synchronization. The following object-sharing mechanisms are available:

A child process created by the CreateProcess function can inherit a handle to a semaphore object if the lpSemaphoreAttributes parameter of CreateSemaphore enabled inheritance. A process can specify the semaphore-object handle in a call to the DuplicateHandle function to create a duplicate handle that can be used by another process.

...

Microkernels and hypervisors

Microkernels and hypervisors

Verification of microkernels and hypervisors is far more doable, the semantics of their internal APIs is simpler and is available to the team doing the verification.

Microkernels and hypervisors

Verification of microkernels and hypervisors is far more doable, the semantics of their internal APIs is simpler and is available to the team doing the verification.

1. the microkernel seL4 has about 7,500 lines of C code; proof obligations discharged with Isabelle

Microkernels and hypervisors

Verification of microkernels and hypervisors is far more doable, the semantics of their internal APIs is simpler and is available to the team doing the verification.

1. the microkernel seL4 has about 7,500 lines of C code; proof obligations discharged with Isabelle
2. the hypervisor Hyper-V has 60,000 lines of C; a lot of work using VCC, proof obligations discharged mostly with Z3

Microkernels and hypervisors

Verification of microkernels and hypervisors is far more doable, the semantics of their internal APIs is simpler and is available to the team doing the verification.

1. the microkernel seL4 has about 7,500 lines of C code; proof obligations discharged with Isabelle
2. the hypervisor Hyper-V has 60,000 lines of C; a lot of work using VCC, proof obligations discharged mostly with Z3
3. we'll learn in the program the VCC tool used for the verification of Hyper-V

Static code analysis

Static code analysis

- There are other useful ways to 'reason' about programs, besides theorem proving and model checking

Static code analysis

- There are other useful ways to 'reason' about programs, besides theorem proving and model checking
- By 'reason' we mean a wider form of arguments; practically, a formal method should present the engineer with evidence that a program or a property of a program is either correct or is an error:

Static code analysis

- There are other useful ways to 'reason' about programs, besides theorem proving and model checking
- By 'reason' we mean a wider form of arguments; practically, a formal method should present the engineer with evidence that a program or a property of a program is either correct or is an error:
- Rice's theorem, applied to programming languages: *every interesting property of programs (in a Turing-complete PL, which most PLs are) is undecidable* (interesting means that there are programs that have the property and others that don't have it)

Static code analysis

- There are other useful ways to 'reason' about programs, besides theorem proving and model checking
- By 'reason' we mean a wider form of arguments; practically, a formal method should present the engineer with evidence that a program or a property of a program is either correct or is an error:
- Rice's theorem, applied to programming languages: *every interesting property of programs (in a Turing-complete PL, which most PLs are) is undecidable* (interesting means that there are programs that have the property and others that don't have it)
- So, all we can do is build *decidable approximations* of solutions to interesting problems

Static code analysis

- There are other useful ways to 'reason' about programs, besides theorem proving and model checking
- By 'reason' we mean a wider form of arguments; practically, a formal method should present the engineer with evidence that a program or a property of a program is either correct or is an error:
- Rice's theorem, applied to programming languages: *every interesting property of programs (in a Turing-complete PL, which most PLs are) is undecidable* (interesting means that there are programs that have the property and others that don't have it)
- So, all we can do is build *decidable approximations* of solutions to interesting problems
  - There are both theoretical and engineering challenges

Static code analysis

- There are other useful ways to 'reason' about programs, besides theorem proving and model checking
- By 'reason' we mean a wider form of arguments; practically, a formal method should present the engineer with evidence that a program or a property of a program is either correct or is an error:
- Rice's theorem, applied to programming languages: *every interesting property of programs (in a Turing-complete PL, which most PLs are) is undecidable* (interesting means that there are programs that have the property and others that don't have it)
- So, all we can do is build *decidable approximations* of solutions to interesting problems
  - There are both theoretical and engineering challenges
  - Good approximation algorithms give useful answers often enough (and false positives rare enough)

Static code analysis

- There are other useful ways to 'reason' about programs, besides theorem proving and model checking
- By 'reason' we mean a wider form of arguments; practically, a formal method should present the engineer with evidence that a program or a property of a program is either correct or is an error:
- Rice's theorem, applied to programming languages: *every interesting property of programs (in a Turing-complete PL, which most PLs are) is undecidable* (interesting means that there are programs that have the property and others that don't have it)
- So, all we can do is build *decidable approximations* of solutions to interesting problems
  - There are both theoretical and engineering challenges
  - Good approximation algorithms give useful answers often enough (and false positives rare enough)
- Our program is obviously *more focussed on theorem proving and model checking than static analysis*

Static code analysis

Static code analysis

- Quite a large and heterogeneous collection of theory and tools: we study some of them briefly in the two verification courses

Static code analysis

- Quite a large and heterogeneous collection of theory and tools: we study some of them briefly in the two verification courses
- Some lightweight static code analysis is already woven into the compiler

Static code analysis

- Quite a large and heterogeneous collection of theory and tools: we study some of them briefly in the two verification courses
- Some lightweight static code analysis is already woven into the compiler
- Static code analysis is a formal method though, it uses some mathematical techniques (lattice theory, lambda calculus, ...) in all forms of analysis (look these terms up on Wikipedia, we have no space here):

Static code analysis

- Quite a large and heterogeneous collection of theory and tools: we study some of them briefly in the two verification courses
- Some lightweight static code analysis is already woven into the compiler
- Static code analysis is a formal method though, it uses some mathematical techniques (lattice theory, lambda calculus, ...) in all forms of analysis (look these terms up on Wikipedia, we have no space here):
  - Data flow and control flow analysis

Static code analysis

- Quite a large and heterogeneous collection of theory and tools: we study some of them briefly in the two verification courses
- Some lightweight static code analysis is already woven into the compiler
- Static code analysis is a formal method though, it uses some mathematical techniques (lattice theory, lambda calculus, ...) in all forms of analysis (look these terms up on Wikipedia, we have no space here):
  - Data flow and control flow analysis
  - Interprocedural analysis

Static code analysis

- Quite a large and heterogeneous collection of theory and tools: we study some of them briefly in the two verification courses
- Some lightweight static code analysis is already woven into the compiler
- Static code analysis is a formal method though, it uses some mathematical techniques (lattice theory, lambda calculus, ...) in all forms of analysis (look these terms up on Wikipedia, we have no space here):
  - Data flow and control flow analysis
  - Interprocedural analysis
  - Shape and pointer analysis

Static code analysis

- Quite a large and heterogeneous collection of theory and tools: we study some of them briefly in the two verification courses
- Some lightweight static code analysis is already woven into the compiler
- Static code analysis is a formal method though, it uses some mathematical techniques (lattice theory, lambda calculus, ...) in all forms of analysis (look these terms up on Wikipedia, we have no space here):
  - Data flow and control flow analysis
  - Interprocedural analysis
  - Shape and pointer analysis
  - Type analysis

Static code analysis

- Quite a large and heterogeneous collection of theory and tools: we study some of them briefly in the two verification courses
- Some lightweight static code analysis is already woven into the compiler
- Static code analysis is a formal method though, it uses some mathematical techniques (lattice theory, lambda calculus, ...) in all forms of analysis (look these terms up on Wikipedia, we have no space here):
  - Data flow and control flow analysis
  - Interprocedural analysis
  - Shape and pointer analysis
  - Type analysis
  - Widening and narrowing

Static code analysis

- Quite a large and heterogeneous collection of theory and tools: we study some of them briefly in the two verification courses
- Some lightweight static code analysis is already woven into the compiler
- Static code analysis is a formal method though, it uses some mathematical techniques (lattice theory, lambda calculus, ...) in all forms of analysis (look these terms up on Wikipedia, we have no space here):
  - Data flow and control flow analysis
  - Interprocedural analysis
  - Shape and pointer analysis
  - Type analysis
  - Widening and narrowing
- There is a large step when moving from static code analysis to theorem proving/model checking

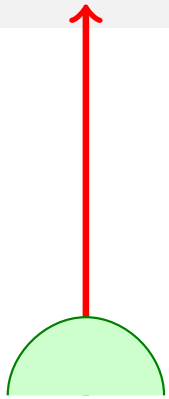
The evolution of mathematics

The evolution of mathematics

- It continuously grows upwards, drawing on existing body of knowledge

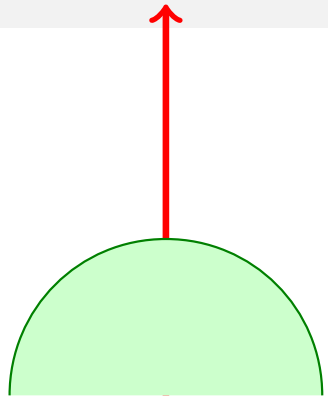
The evolution of mathematics

- It continuously grows upwards, drawing on existing body of knowledge



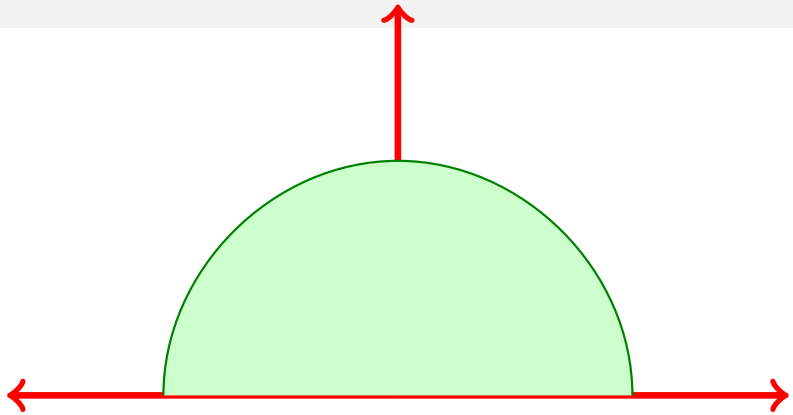
The evolution of mathematics

- It continuously grows upwards, drawing on existing body of knowledge



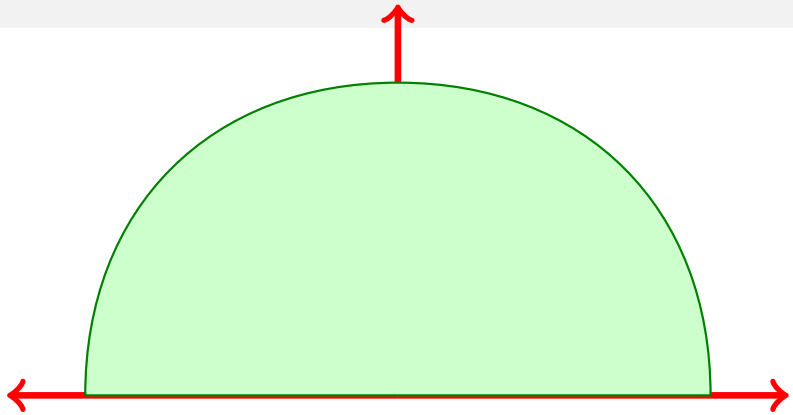
The evolution of mathematics

- It continuously grows upwards, drawing on existing body of knowledge
- It continuously grows side-ways, finding more and more fields of application

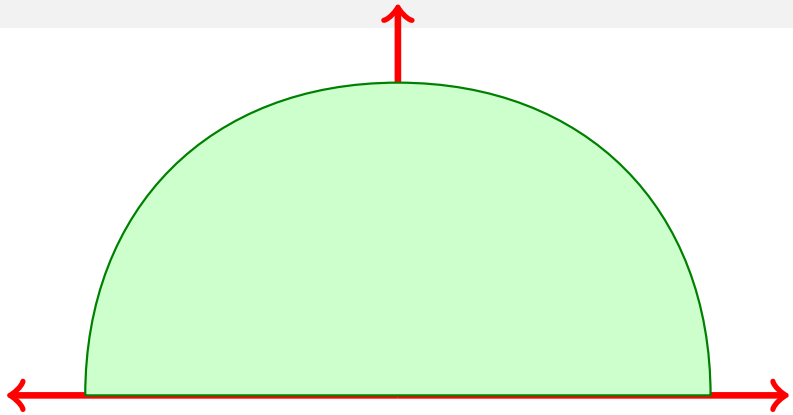


The evolution of mathematics

- It continuously grows upwards, drawing on existing body of knowledge
- It continuously grows side-ways, finding more and more fields of application



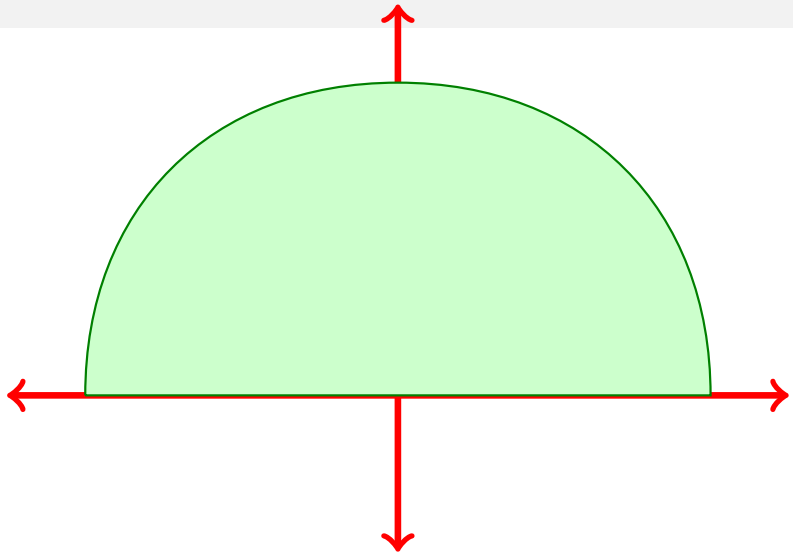
The evolution of mathematics



- It continuously grows upwards, drawing on existing body of knowledge
- It continuously grows side-ways, finding more and more fields of application
- Around 1900: Oops, there are difficulties! We now have to go deeper!

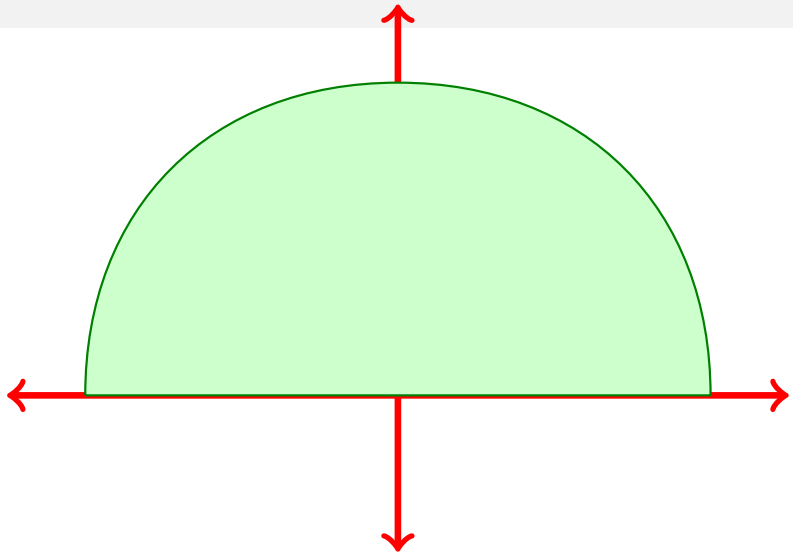
The evolution of mathematics

- It continuously grows upwards, drawing on existing body of knowledge
- It continuously grows side-ways, finding more and more fields of application



- Around 1900: Oops, there are difficulties! We now have to go deeper!

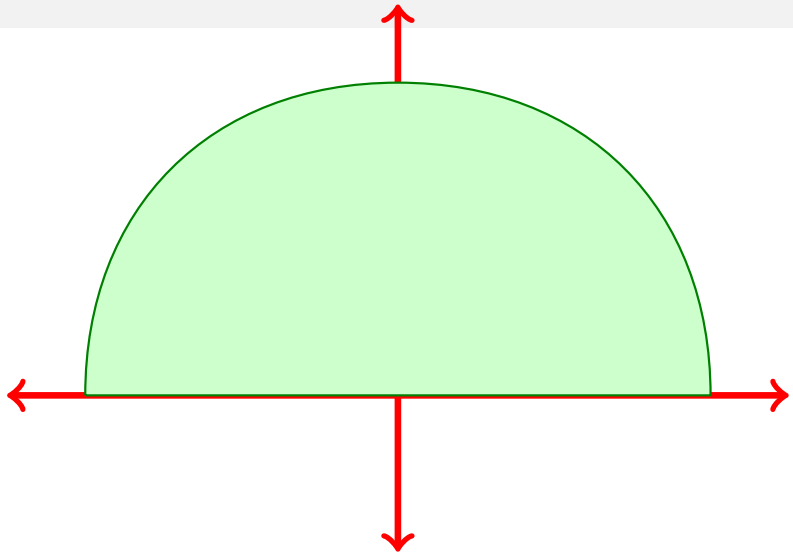
The evolution of mathematics



- It continuously grows upwards, drawing on existing body of knowledge
- It continuously grows side-ways, finding more and more fields of application

- Around 1900: Oops, there are difficulties! We now have to go deeper!
- . . . and what do we find?

The evolution of mathematics



- It continuously grows upwards, drawing on existing body of knowledge
- It continuously grows side-ways, finding more and more fields of application
- Around 1900: Oops, there are difficulties! We now have to go deeper!
- . . . and what do we find?

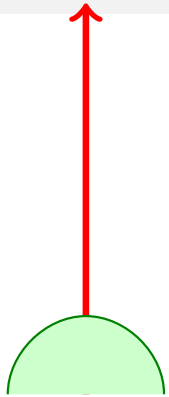
The evolution of software

The evolution of software

- It continuously grows upwards, drawing on existing body of knowledge

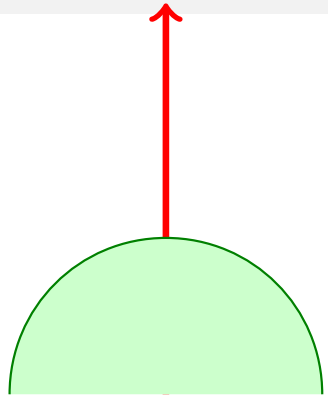
The evolution of software

- It continuously grows upwards, drawing on existing body of knowledge



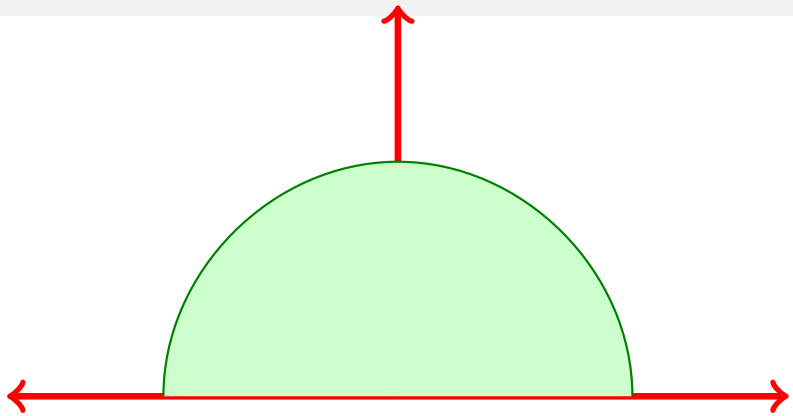
The evolution of software

- It continuously grows upwards, drawing on existing body of knowledge



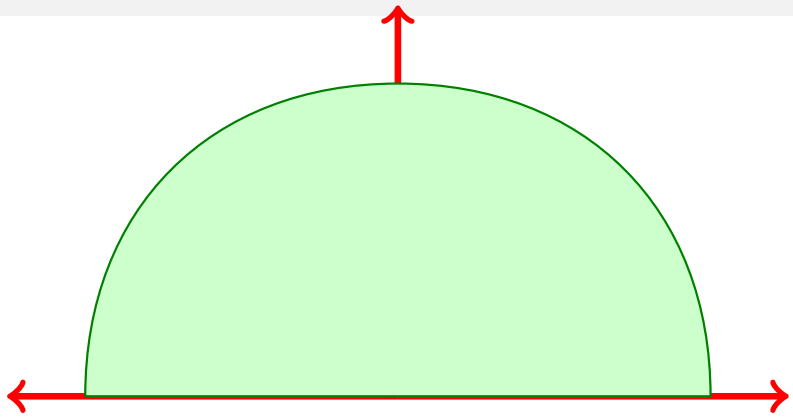
The evolution of software

- It continuously grows upwards, drawing on existing body of knowledge
- It continuously grows side-ways, finding more and more fields of application

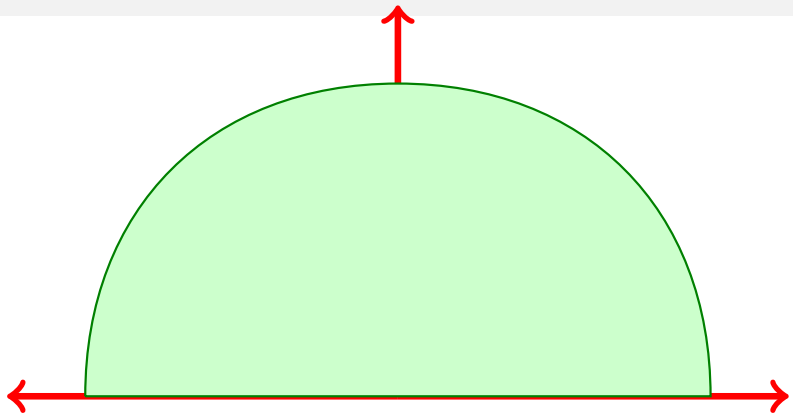


The evolution of software

- It continuously grows upwards, drawing on existing body of knowledge
- It continuously grows side-ways, finding more and more fields of application



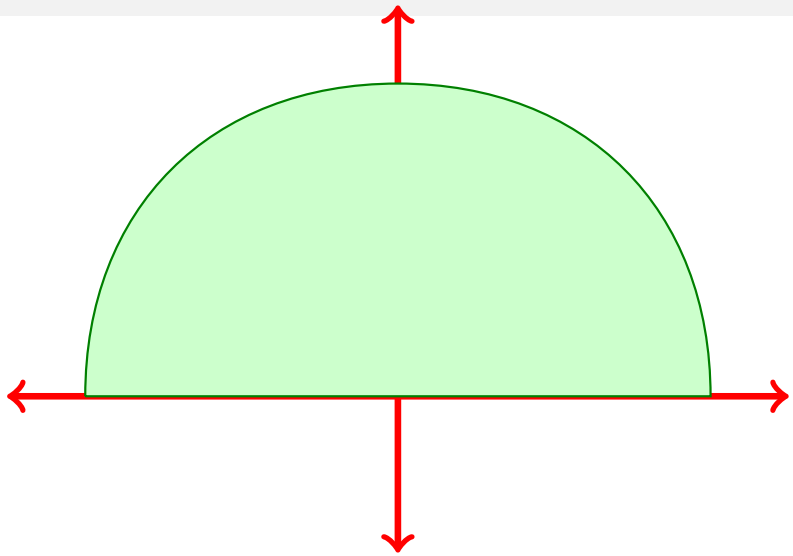
The evolution of software



- It continuously grows upwards, drawing on existing body of knowledge
- It continuously grows side-ways, finding more and more fields of application
- Around 1960: Oops, there are difficulties! We now have to go deeper!

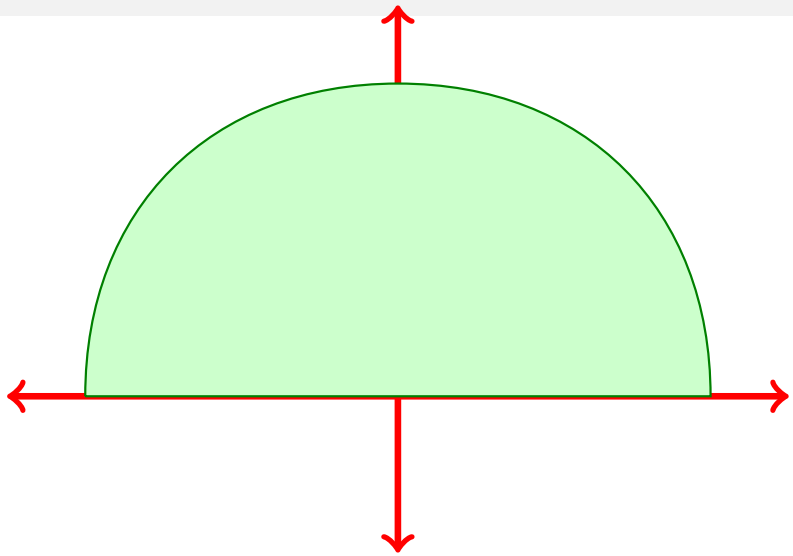
The evolution of software

- It continuously grows upwards, drawing on existing body of knowledge
- It continuously grows side-ways, finding more and more fields of application



- Around 1960: Oops, there are difficulties! We now have to go deeper!

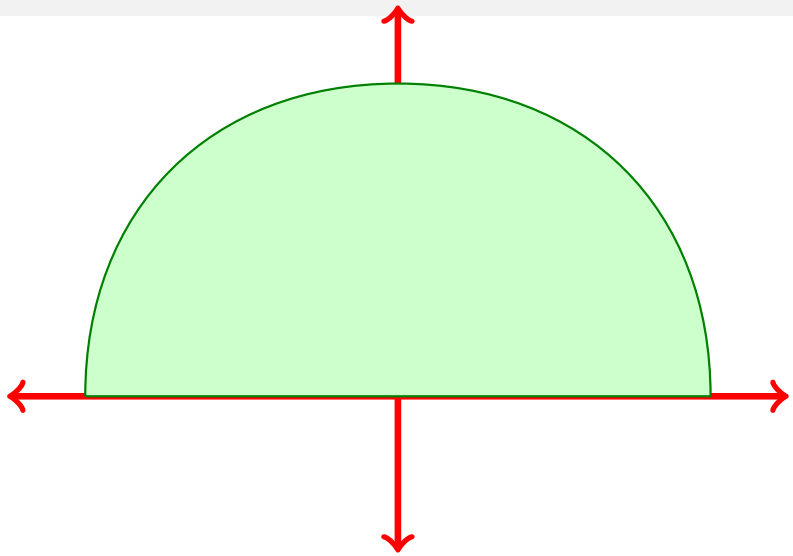
The evolution of software



- It continuously grows upwards, drawing on existing body of knowledge
- It continuously grows side-ways, finding more and more fields of application

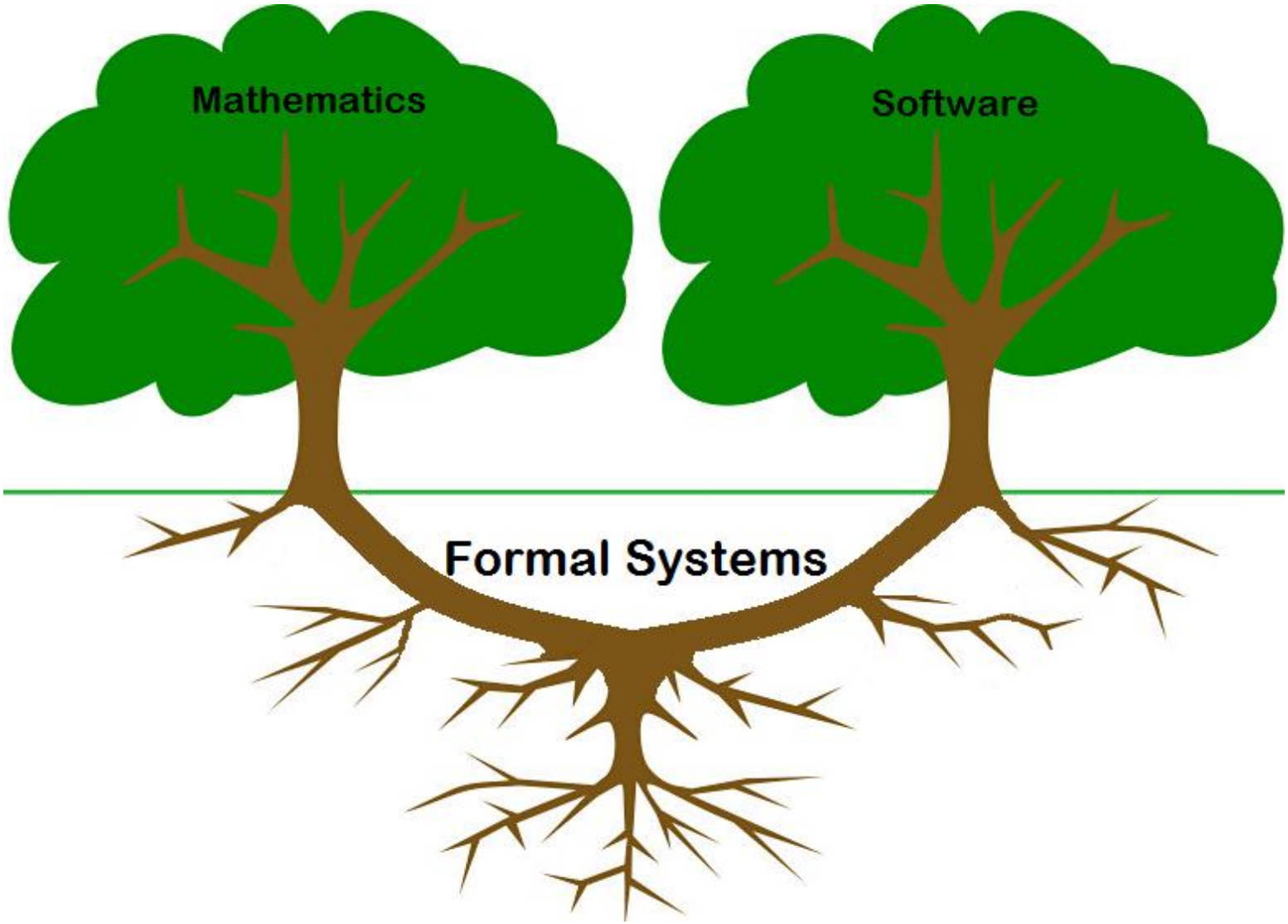
- Around 1960: Oops, there are difficulties! We now have to go deeper!
- . . . and what do we find?

The evolution of software

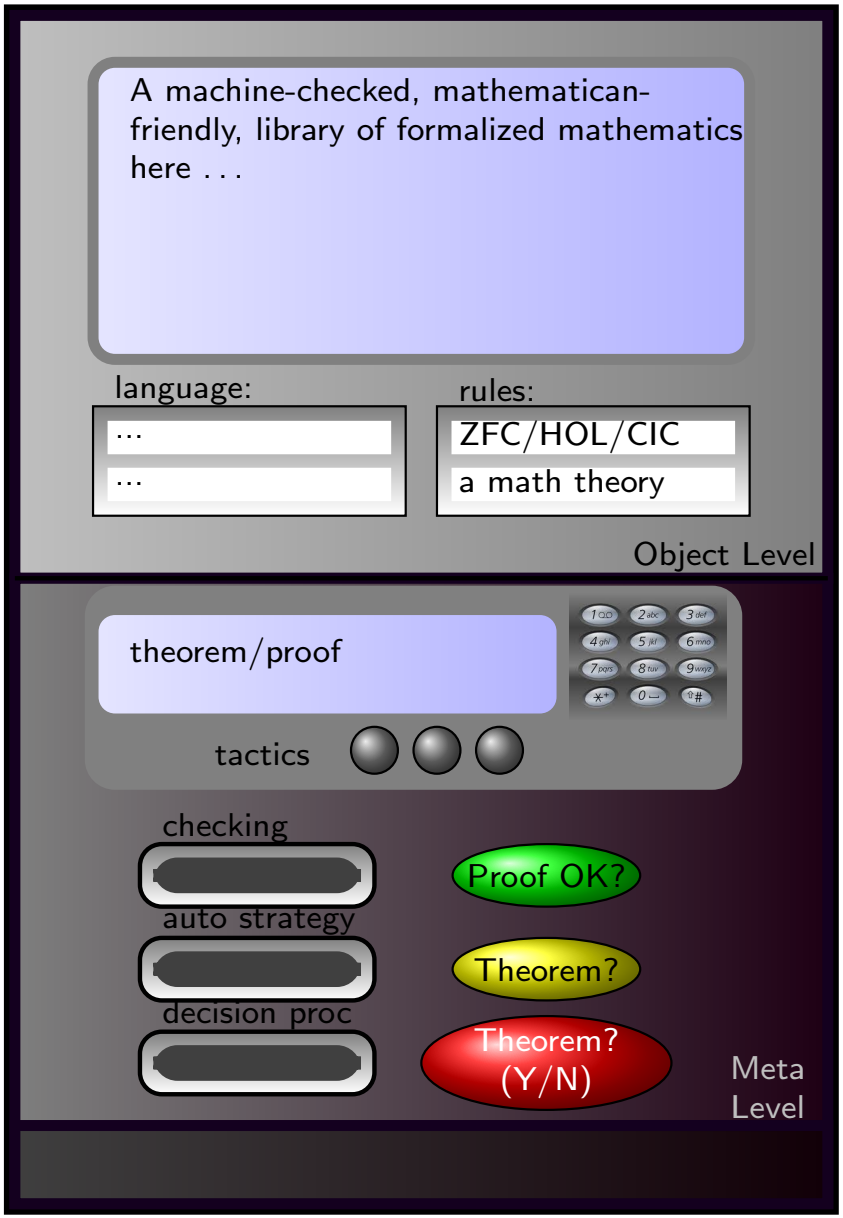


- It continuously grows upwards, drawing on existing body of knowledge
- It continuously grows side-ways, finding more and more fields of application
- Around 1960: Oops, there are difficulties! We now have to go deeper!
- . . . and what do we find?

Common roots



Common future



A quote of a quote

‘ But this self-examination (my note: of mathematics by mathematical means) did have wonderful and totally unexpected consequences in an area far removed from its original goals. It played a big role in the development of the most successful technology of our age, the computer, which after all is just a mathematical machine, a machine for doing mathematics. As E.T. Bell put it, the attempt to soar above mathematics ended in the bowels of a computer! ’

– Gregory Chaitin, The Unknowable, 1999.

Mathematicians and software engineers

Since we refer to these professions often, and ignore all others, we need some clarification. It's easier to define what a mathematician does: mathematics. With software the situation is more complicated, as there are: software engineers, computer systems analysts, developers, programmers, QA engineers, software managers, database designers, etc . . . All of them do software, and with formal methods in mind, these distinctions get in the way.

Mathematicians and software engineers

Since we refer to these professions often, and ignore all others, we need some clarification. It's easier to define what a mathematician does: mathematics. With software the situation is more complicated, as there are: software engineers, computer systems analysts, developers, programmers, QA engineers, software managers, database designers, etc . . . All of them do software, and with formal methods in mind, these distinctions get in the way.

So when we refer to 'software engineers' we include all these software professionals. On the other hand, computer scientists and actuaries are included in the mathematicians group. There are many common characteristics between the two professions, some of which are the reasons why they have been ranked in the top 10 best jobs in the past few years.

Mathematicians and software engineers

Since we refer to these professions often, and ignore all others, we need some clarification. It's easier to define what a mathematician does: mathematics. With software the situation is more complicated, as there are: software engineers, computer systems analysts, developers, programmers, QA engineers, software managers, database designers, etc . . . All of them do software, and with formal methods in mind, these distinctions get in the way.

So when we refer to 'software engineers' we include all these software professionals. On the other hand, computer scientists and actuaries are included in the mathematicians group. There are many common characteristics between the two professions, some of which are the reasons why they have been ranked in the top 10 best jobs in the past few years.

But the most interesting common factor is their foundation.

Common roots, but different perspectives

Common roots, but different perspectives

Valid argument coming from the mathematical side: Why don't engineers use the logical tools produced in academia/industrial research centers?

Common roots, but different perspectives

Valid argument coming from the mathematical side: Why don't engineers use the logical tools produced in academia/industrial research centers?

Valid argument coming from the engineering side: Why are the logical tools so unfriendly, unpolished, and under-performing? *Why should we trust a tool that proves concurrency theorems when the tool itself does not use concurrency?*

Common roots, but different perspectives

Valid argument coming from the mathematical side: Why don't engineers use the logical tools produced in academia/industrial research centers?

Valid argument coming from the engineering side: Why are the logical tools so unfriendly, unpolished, and under-performing? *Why should we trust a tool that proves concurrency theorems when the tool itself does not use concurrency?*

- When mathematicians look at the output of a logical tool, they gasp: It looks like code!

Common roots, but different perspectives

Valid argument coming from the mathematical side: Why don't engineers use the logical tools produced in academia/industrial research centers?

Valid argument coming from the engineering side: Why are the logical tools so unfriendly, unpolished, and under-performing? *Why should we trust a tool that proves concurrency theorems when the tool itself does not use concurrency?*

- When mathematicians look at the output of a logical tool, they gasp: It looks like code!
- When engineers look at the output of a logical tool, they gasp: It looks like math!

How do we bridge the gap?

How do we bridge the gap?

- Let's look at numbers, estimating roughly

How do we bridge the gap?

- Let's look at numbers, estimating roughly
- Estimated number of software engineers: 30 million worldwide

How do we bridge the gap?

- Let's look at numbers, estimating roughly
- Estimated number of software engineers: 30 million worldwide
- Estimated number of mathematicians: 300,000 worldwide.

How do we bridge the gap?

- Let's look at numbers, estimating roughly
- Estimated number of software engineers: 30 million worldwide
- Estimated number of mathematicians: 300,000 worldwide.
- (only very small percentages on either side are working with formalized knowledge, but let's say that the percentages are the same)

How do we bridge the gap?

- Let's look at numbers, estimating roughly
- Estimated number of software engineers: 30 million worldwide
- Estimated number of mathematicians: 300,000 worldwide.
- (only very small percentages on either side are working with formalized knowledge, but let's say that the percentages are the same)
- The estimated ratio would be 1:100

How do we bridge the gap?

- Let's look at numbers, estimating roughly
- Estimated number of software engineers: 30 million worldwide
- Estimated number of mathematicians: 300,000 worldwide.
- (only very small percentages on either side are working with formalized knowledge, but let's say that the percentages are the same)
- The estimated ratio would be 1:100
- If we are to make significant progress, *we must make the effort to bridge this gap from the engineering side*

How do we bridge the gap?

- Let's look at numbers, estimating roughly
- Estimated number of software engineers: 30 million worldwide
- Estimated number of mathematicians: 300,000 worldwide.
- (only very small percentages on either side are working with formalized knowledge, but let's say that the percentages are the same)
- The estimated ratio would be 1:100
- If we are to make significant progress, *we must make the effort to bridge this gap from the engineering side*
- There just isn't enough population on the mathematical side

Formalization is far more important for software

Formalization is far more important for software

It is unlikely that if we do not formalize mathematics,

Formalization is far more important for software

It is unlikely that if we do not formalize mathematics,

- Airplanes will flip upside down when they cross the equator

Formalization is far more important for software

It is unlikely that if we do not formalize mathematics,

- Airplanes will flip upside down when they cross the equator
- Wrong doses of radiation will be administered to cancer patients

Formalization is far more important for software

It is unlikely that if we do not formalize mathematics,

- Airplanes will flip upside down when they cross the equator
- Wrong doses of radiation will be administered to cancer patients
- Expensive space exploration missions will fail

Formalization is far more important for software

It is unlikely that if we do not formalize mathematics,

- Airplanes will flip upside down when they cross the equator
- Wrong doses of radiation will be administered to cancer patients
- Expensive space exploration missions will fail
- Servers or entire networks will go down under targeted attacks

Four views on the gap situation

Four views on the gap situation

There are four views about the state of software engineers' mathematical knowledge:

Four views on the gap situation

There are four views about the state of software engineers' mathematical knowledge:

1. Hopeless

Four views on the gap situation

There are four views about the state of software engineers' mathematical knowledge:

1. Hopeless
2. Bad, but it *could* be fixed

Four views on the gap situation

There are four views about the state of software engineers' mathematical knowledge:

1. Hopeless
2. Bad, but it *could* be fixed
3. Bad, but it *must* be fixed

Four views on the gap situation

There are four views about the state of software engineers' mathematical knowledge:

1. Hopeless
2. Bad, but it *could* be fixed
3. Bad, but it *must* be fixed
4. Not bad

Four views on the gap situation

There are four views about the state of software engineers' mathematical knowledge:

1. Hopeless
2. Bad, but it *could* be fixed
3. Bad, but it *must* be fixed
4. Not bad

Four views on the gap situation

There are four views about the state of software engineers' mathematical knowledge:

1. Hopeless
2. Bad, but it *could* be fixed
3. Bad, but it *must* be fixed
4. Not bad

My personal experience agrees with item 4. But regardless of my views, the point is that we cannot conceivably accept either 1 or 2; even if they were true, we have to turn things around and move 1 and 2 towards 3 and then on towards 4. The stakes are too high. This program is a small step.

What lessons have we learned so far

What lessons have we learned so far

- Engineers like tools

What lessons have we learned so far

- Engineers like tools
- Engineers like lab work

What lessons have we learned so far

- Engineers like tools
- Engineers like lab work
- Engineers do not care for long-winded theory; we'll have to approach theory differently

What lessons have we learned so far

- Engineers like tools
- Engineers like lab work
- Engineers do not care for long-winded theory; we'll have to approach theory differently
- Engineers do not learn by listening and imitating; they learn by doing

What lessons have we learned so far

- Engineers like tools
- Engineers like lab work
- Engineers do not care for long-winded theory; we'll have to approach theory differently
- Engineers do not learn by listening and imitating; they learn by doing
- Relevance to the real world and impact on careers is important

What lessons have we learned so far

- Engineers like tools
- Engineers like lab work
- Engineers do not care for long-winded theory; we'll have to approach theory differently
- Engineers do not learn by listening and imitating; they learn by doing
- Relevance to the real world and impact on careers is important
- Big problem: many Computer Science/Computer Engineeering degrees do not include math requirements for admission and many CS/CE depts do not make up for this during undergraduate studies. The emphasis has been traditionally on the continuous and not on the discrete (because the objective was physics).

There is a way: learn math via logic and computation

There is a way: learn math via logic and computation

- Maybe math is not properly taught

There is a way: learn math via logic and computation

- Maybe math is not properly taught
- Students still have a hard time with proofs, because proofs are given informally; they look arbitrary no matter how much the $\varepsilon - \delta$ definition of limits is being drilled

There is a way: learn math via logic and computation

- Maybe math is not properly taught
- Students still have a hard time with proofs, because proofs are given informally; they look arbitrary no matter how much the $\varepsilon - \delta$ definition of limits is being drilled
- Situation has a known remedy, but it's not consistently applied

There is a way: learn math via logic and computation

- Maybe math is not properly taught
- Students still have a hard time with proofs, because proofs are given informally; they look arbitrary no matter how much the $\varepsilon - \delta$ definition of limits is being drilled
- Situation has a known remedy, but it's not consistently applied
- *Define proofs for what they really are, i.e. deductions in a formal system, and perception changes dramatically*

There is a way: learn math via logic and computation

- Maybe math is not properly taught
- Students still have a hard time with proofs, because proofs are given informally; they look arbitrary no matter how much the $\varepsilon - \delta$ definition of limits is being drilled
- Situation has a known remedy, but it's not consistently applied
- *Define proofs for what they really are, i.e. deductions in a formal system, and perception changes dramatically*
- Software engineers have strong computational intuitions, so proofs done as mechanical computations feel more natural

There is a way: learn math via logic and computation

- Maybe math is not properly taught
- Students still have a hard time with proofs, because proofs are given informally; they look arbitrary no matter how much the $\varepsilon - \delta$ definition of limits is being drilled
- Situation has a known remedy, but it's not consistently applied
- *Define proofs for what they really are, i.e. deductions in a formal system, and perception changes dramatically*
- Software engineers have strong computational intuitions, so proofs done as mechanical computations feel more natural
- This has already been done with great success in CS courses (nothing new here): Knuth, Dijkstra, Gries, ...

There is a way: learn math via logic and computation

- Maybe math is not properly taught
- Students still have a hard time with proofs, because proofs are given informally; they look arbitrary no matter how much the $\varepsilon - \delta$ definition of limits is being drilled
- Situation has a known remedy, but it's not consistently applied
- *Define proofs for what they really are, i.e. deductions in a formal system, and perception changes dramatically*
- Software engineers have strong computational intuitions, so proofs done as mechanical computations feel more natural
- This has already been done with great success in CS courses (nothing new here): Knuth, Dijkstra, Gries, ...
- Following their lead, the program includes a course in 'Discrete Mathematics via Equational Logic'

And use a computer algebra system

And use a computer algebra system

- Secondly, we will combine theorem proving with a computer algebra system; this must be necessarily a patchy job since no available system does both well, yet

And use a computer algebra system

- Secondly, we will combine theorem proving with a computer algebra system; this must be necessarily a patchy job since no available system does both well, yet
- With theorem proving and computer algebra combined, you can see a dazzling display of cooperation between math and software

And use a computer algebra system

- Secondly, we will combine theorem proving with a computer algebra system; this must be necessarily a patchy job since no available system does both well, yet
- With theorem proving and computer algebra combined, you can see a dazzling display of cooperation between math and software
- We will do this in the 'Computer Algebra Systems' course

We need them, they need us

We need them, they need us

- As we said before, the dependency between software and mathematics is not one way

We need them, they need us

- As we said before, the dependency between software and mathematics is not one way
- Projects that attempted the formalization of mathematics moved slowly

We need them, they need us

- As we said before, the dependency between software and mathematics is not one way
- Projects that attempted the formalization of mathematics moved slowly
- Clearly engineering experience would have been useful

We need them, they need us

- As we said before, the dependency between software and mathematics is not one way
- Projects that attempted the formalization of mathematics moved slowly
- Clearly engineering experience would have been useful
- Especially when it came to library design, backward compatibility, ...

We need them, they need us

- As we said before, the dependency between software and mathematics is not one way
- Projects that attempted the formalization of mathematics moved slowly
- Clearly engineering experience would have been useful
- Especially when it came to library design, backward compatibility, ...
- Tools are still not friendly towards mathematicians' view of proofs (remember: 'looks like code!')

We need them, they need us

- As we said before, the dependency between software and mathematics is not one way
- Projects that attempted the formalization of mathematics moved slowly
- Clearly engineering experience would have been useful
- Especially when it came to library design, backward compatibility, ...
- Tools are still not friendly towards mathematicians' view of proofs (remember: 'looks like code!')
- User interfaces could use expertize from people who specialize in usability

We need them, they need us

- As we said before, the dependency between software and mathematics is not one way
- Projects that attempted the formalization of mathematics moved slowly
- Clearly engineering experience would have been useful
- Especially when it came to library design, backward compatibility, ...
- Tools are still not friendly towards mathematicians' view of proofs (remember: 'looks like code!')
- User interfaces could use expertize from people who specialize in usability
- If the gap is closed, mathematics and software engineering can together lead to better and more powerful tools, which in turn can support more powerful theories, either in mathematics or in software development

In this program, should we dial down the level of mathematics?

In this program, should we dial down the level of mathematics?

- Let's look at an example

In this program, should we dial down the level of mathematics?

- Let's look at an example
- One of the most most important concepts to emerge in computation has been the concept of a *monad*

In this program, should we dial down the level of mathematics?

- Let's look at an example
- One of the most most important concepts to emerge in computation has been the concept of a *monad*
- Now a monad is a mathematical object, right at *the core of category theory*

In this program, should we dial down the level of mathematics?

- Let's look at an example
- One of the most most important concepts to emerge in computation has been the concept of a *monad*
- Now a monad is a mathematical object, right at *the core of category theory*
- The first practical language that introduced them was Haskell. Many tutorials explain by different means what a monad is (a container, a computation, etc...) because the thinking is that monads are too mathematical and students won't get them.

In this program, should we dial down the level of mathematics?

- Let's look at an example
- One of the most most important concepts to emerge in computation has been the concept of a *monad*
- Now a monad is a mathematical object, right at *the core of category theory*
- The first practical language that introduced them was Haskell. Many tutorials explain by different means what a monad is (a container, a computation, etc...) because the thinking is that monads are too mathematical and students won't get them.
- This is not something to be quickly dismissed. *Twisting an abstract mathematical object into something easier to use in engineering is a good intention.*

We cannot dial down the level of mathematics

We cannot dial down the level of mathematics

- We cannot do that in this program, we have to learn mathematics as it is

We cannot dial down the level of mathematics

- We cannot do that in this program, we have to learn mathematics as it is
- Why? First of all, recall the definition of formal software development, it's the use of mathematics

We cannot dial down the level of mathematics

- We cannot do that in this program, we have to learn mathematics as it is
- Why? First of all, recall the definition of formal software development, it's the use of mathematics
- More specifically, *concepts like monads are at the core of what we do*; one way of handling imperative programs is to transform them into semantically equivalent functional programs, which logical tools can handle easier; monads allow this transformation.

We cannot dial down the level of mathematics

- We cannot do that in this program, we have to learn mathematics as it is
- Why? First of all, recall the definition of formal software development, it's the use of mathematics
- More specifically, *concepts like monads are at the core of what we do*; one way of handling imperative programs is to transform them into semantically equivalent functional programs, which logical tools can handle easier; monads allow this transformation.

We cannot dial down the level of mathematics

- We cannot do that in this program, we have to learn mathematics as it is
- Why? First of all, recall the definition of formal software development, it's the use of mathematics
- More specifically, *concepts like monads are at the core of what we do*; one way of handling imperative programs is to transform them into semantically equivalent functional programs, which logical tools can handle easier; monads allow this transformation.

We not only cannot avoid categories, *we have to embrace them*. We do that in the advanced sequence of the program, but the basic definitions and examples will be introduced very early. Following is a set of slides that explain this need to become comfortable with categories. They are intended for people who have some familiarity with abstract algebra. Others can safely skip these slides, nothing else in the overview depends on them.

The equational content of formal systems

may be skipped

The equational content of formal systems

may be skipped

First of all, categories are special formal systems in which only the equational aspects matter, i.e.

The equational content of formal systems

may be skipped

First of all, categories are special formal systems in which only the equational aspects matter, i.e.

- the proofs of the same theorem are indistinguishable

The equational content of formal systems

may be skipped

First of all, categories are special formal systems in which only the equational aspects matter, i.e.

- the proofs of the same theorem are indistinguishable
- only the fact that there exists such a proof matters

The equational content of formal systems

may be skipped

First of all, categories are special formal systems in which only the equational aspects matter, i.e.

- the proofs of the same theorem are indistinguishable
- only the fact that there exists such a proof matters

The equational content of formal systems

may be skipped

First of all, categories are special formal systems in which only the equational aspects matter, i.e.

- the proofs of the same theorem are indistinguishable
- only the fact that there exists such a proof matters

Removing the operational aspects of a formal system leaves us with an algebraic content that can be studied independently and leads to useful classifications of such systems. Even though these classifications may not give us deep insights into a particular formal system, they give us a very effective way to organize our knowledge so we don't have to keep repeating the same constructions over and over. You may think of categories as the *ultimate pattern recognition tool*.

Higher level of abstraction

may be skipped

Higher level of abstraction

may be skipped

- Categorial arguments raise the abstraction level

Higher level of abstraction

may be skipped

- Categorial arguments raise the abstraction level
- So category theory should feel *natural* to software engineers ...

Higher level of abstraction

may be skipped

- Categorial arguments raise the abstraction level
- So category theory should feel *natural* to software engineers ...
- ... *because of its high power of abstraction*, not despite of it

Higher level of abstraction

may be skipped

- Categorial arguments raise the abstraction level
- So category theory should feel *natural* to software engineers ...
- ... *because of its high power of abstraction*, not despite of it
- Half-jokingly, the following is referred to as the 'fundamental theorem of software engineering': when things get difficult, raise the abstraction level (term attributed to Koenig)

More reasons why we need categories

may be skipped

More reasons why we need categories

may be skipped

1. We have already mentioned the Curry-Howard-Lambek correspondence

More reasons why we need categories

may be skipped

1. We have already mentioned the Curry-Howard-Lambek correspondence
2. Can this correspondence be extended to reason about imperative programs? Not directly, but with the use of monads.

More reasons why we need categories

may be skipped

1. We have already mentioned the Curry-Howard-Lambek correspondence
2. Can this correspondence be extended to reason about imperative programs? Not directly, but with the use of monads.
   - *Monads* encapsulate imperative features inside functional languages

More reasons why we need categories

may be skipped

1. We have already mentioned the Curry-Howard-Lambek correspondence
2. Can this correspondence be extended to reason about imperative programs? Not directly, but with the use of monads.
   - *Monads* encapsulate imperative features inside functional languages
3. We need to understand the *syntax/semantics pair* from an algebraic viewpoint

More reasons why we need categories

may be skipped

1. We have already mentioned the Curry-Howard-Lambek correspondence
2. Can this correspondence be extended to reason about imperative programs? Not directly, but with the use of monads.
   - *Monads* encapsulate imperative features inside functional languages
3. We need to understand the *syntax/semantics pair* from an algebraic viewpoint
4. To understand the *algebraic semantics of programming languages*, we need to understand *initiality, induction and recursion*

More reasons why we need categories

may be skipped

1. We have already mentioned the Curry-Howard-Lambek correspondence
2. Can this correspondence be extended to reason about imperative programs? Not directly, but with the use of monads.
   - *Monads* encapsulate imperative features inside functional languages
3. We need to understand the *syntax/semantics pair* from an algebraic viewpoint
4. To understand the *algebraic semantics of programming languages*, we need to understand *initiality, induction and recursion*
5. We need to understand the difficulties surrounding the logical quantifiers and *how to handle quantifiers algebraically*

More reasons why we need categories

may be skipped

1. We have already mentioned the Curry-Howard-Lambek correspondence
2. Can this correspondence be extended to reason about imperative programs? Not directly, but with the use of monads.
   - *Monads* encapsulate imperative features inside functional languages
3. We need to understand the *syntax/semantics pair* from an algebraic viewpoint
4. To understand the *algebraic semantics of programming languages*, we need to understand *initiality, induction and recursion*
5. We need to understand the difficulties surrounding the logical quantifiers and *how to handle quantifiers algebraically*

More reasons why we need categories

may be skipped

1. We have already mentioned the Curry-Howard-Lambek correspondence
2. Can this correspondence be extended to reason about imperative programs? Not directly, but with the use of monads.
   - *Monads* encapsulate imperative features inside functional languages
3. We need to understand the *syntax/semantics pair* from an algebraic viewpoint
4. To understand the *algebraic semantics of programming languages*, we need to understand *initiality, induction and recursion*
5. We need to understand the difficulties surrounding the logical quantifiers and *how to handle quantifiers algebraically*

In this overview we can only pick one of these items and give some motivation. We pick 'initiality, induction and recursion' because we bump into this in one of the early core courses: 'Formal Semantics of Programming Languages'.

Graphs

may be skipped

Graphs

may be skipped

Categories are special proof calculi and proof calculi are special directed graphs. Directed graphs should be more familiar (and proof calculi, after reading so far) .

Graphs

may be skipped

Categories are special proof calculi and proof calculi are special directed graphs. Directed graphs should be more familiar (and proof calculi, after reading so far) .

graph

A directed graph is a set of vertices $A, B, C, \ldots$ *and a set of edges* $f, g, h, \ldots$, *with two functions from edges to vertices, called source and target. If* A *is* $source(f)$ *and* B *is* $target(f)$, *we denote the edge as*

$$f : A \to B$$

Graphs

may be skipped

Categories are special proof calculi and proof calculi are special directed graphs. Directed graphs should be more familiar (and proof calculi, after reading so far) .

graph

A directed graph is a set of vertices $A, B, C, \ldots$ *and a set of edges* $f, g, h, \ldots$, *with two functions from edges to vertices, called source and target. If* A *is* $source(f)$ *and* B *is* $target(f)$, *we denote the edge as*

$$f : A \to B$$

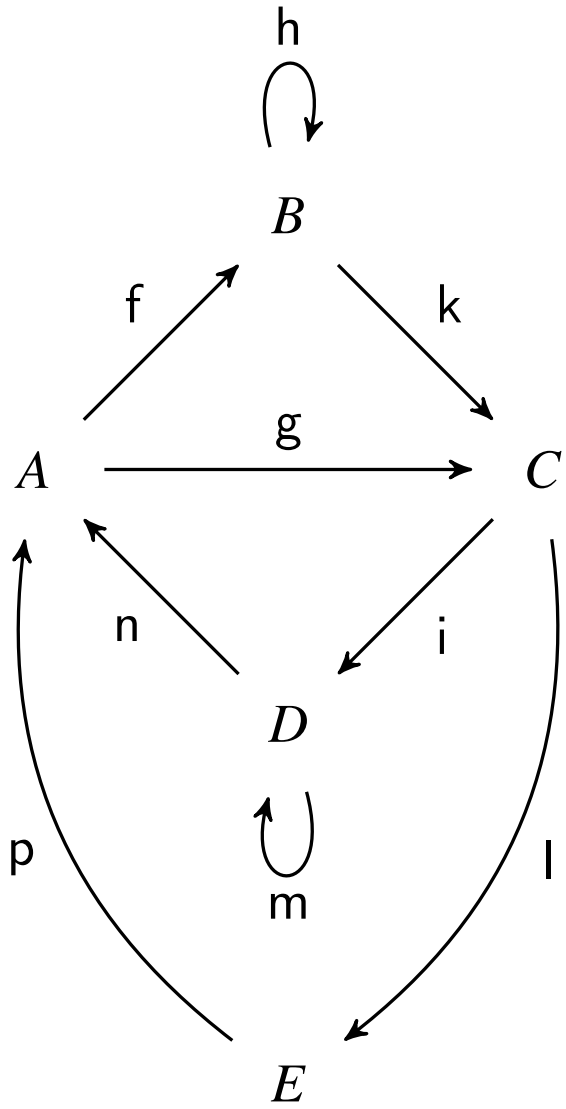
Source is also called domain, and target codomain. An example of a directed graph:

An example of a directed graph

may be skipped

An example of a directed graph

may be skipped



Proof calculi are special graphs

may be skipped

Proof calculi are special graphs

may be skipped

With proof calculi, the terminology changes, vertices are formulas and edges are proofs.

proof calculus

A proof calculus is a graph satisfying two conditions

- *for each formula* A *there is a special proof* $1_A : A \to A$
- *any two proofs* $f : A \to B$ *and* $g : B \to C$ *can be combined into a proof* $g \circ f : A \to C$

Rules of inference

may be skipped

Rules of inference

may be skipped

These conditions can be written as rules of inference (this explains why they are called 'proof calculi'):

- $\dfrac{}{1_A : A \to A}$ *rlIdentity*
- $\dfrac{f : A \to B \qquad g : B \to C}{g \circ f : A \to C}$ *rlModusPonens*

Categories are special proof calculi

may be skipped

Categories are special proof calculi

may be skipped

With categories the terminology changes again, formulas are objects proofs are morphisms.

Categories are special proof calculi

may be skipped

With categories the terminology changes again, formulas are objects proofs are morphisms.

category

A category is a proof calculus where composition of morphisms satisfies two laws

- *For all* $f : A \to B, g : B \to C, h : C \to D$

$$h \circ (g \circ f) = (h \circ g) \circ f \qquad \textit{rlAssociativity}$$

- *For all* $f : A \to B$

$$f \circ 1_A = 1_B \circ f = f \qquad \textit{rlUnit}$$

Categories are special proof calculi

may be skipped

With categories the terminology changes again, formulas are objects proofs are morphisms.

category

A category is a proof calculus where composition of morphisms satisfies two laws

- *For all* $f : A \to B, g : B \to C, h : C \to D$

$$h \circ (g \circ f) = (h \circ g) \circ f \qquad \textit{rlAssociativity}$$

- *For all* $f : A \to B$

$$f \circ 1_A = 1_B \circ f = f \qquad \textit{rlUnit}$$

With categories the perspective also changes, and the emphasis is now on the equations introduced by the associativity and the unit rules. The only example of a category that we need here: **Sets**, with sets as objects and functions as morphisms (verify that all the rules apply).

Commutative diagrams

may be skipped

Commutative diagrams

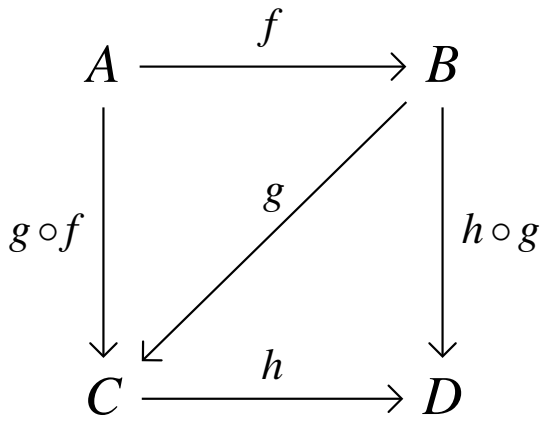
may be skipped

In categories, equations are represented pictorially as commutative diagrams. A diagram is *commutative* if for any two objects in the diagram, all paths leading from one object to the other are equal. For example, the associativity and unit equations are represented by the commutativity of the following two diagrams:

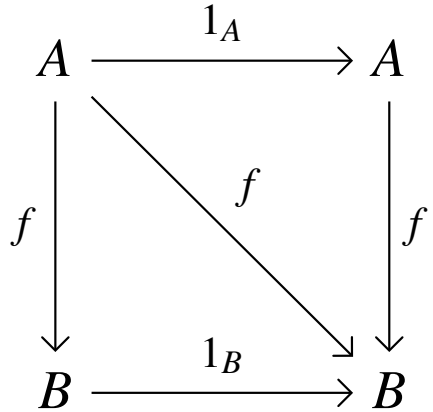
Commutative diagrams

may be skipped

In categories, equations are represented pictorially as commutative diagrams. A diagram is *commutative* if for any two objects in the diagram, all paths leading from one object to the other are equal. For example, the associativity and unit equations are represented by the commutativity of the following two diagrams:



rlAssociativity



rlUnit

Functors

may be skipped

Functors

may be skipped

Since the perspective now is algebraic, we need to define what sort of maps between categories work well with their algebraic structure.

Functors

may be skipped

Since the perspective now is algebraic, we need to define what sort of maps between categories work well with their algebraic structure.

functor

A functor F between categories **C** *and* **D** *is a map $F : \mathbf{C} \to \mathbf{C}$ taking objects of* **C** *to objects of* **D** *and morphisms $f : A \to B$ of* **C** *to morphisms $Ff : FA \to FB$ of* **D**, *such that*

$$F(g \circ f) = Fg \circ Ff$$

$$F(1_A) = 1_{FA}$$

A functor from a category to itself is called an *endofunctor*.

Polynomial endofunctors

may be skipped

Polynomial endofunctors

may be skipped

For two sets A and B, their product $A \times B$ is defined as the set of pairs (x, y) with $x \in X$ and $y \in Y$. Their sum $A + B$ is defined as the disjoint union $\{(x, 1) | x \in A\} \cup \{(y, 2) | y \in B\}$.

Polynomial endofunctors

may be skipped

For two sets A and B, their product $A \times B$ is defined as the set of pairs (x, y) with $x \in X$ and $y \in Y$. Their sum $A + B$ is defined as the disjoint union $\{(x, 1) | x \in A\} \cup \{(y, 2) | y \in B\}$.

The $\times$ and $+$ operations can be defined more generally for objects A and B of any category through certain 'universal properties' which do not concern us here, except that these universal properties give us two functors: $X \to A \times X$ and $X \to A + X$ for some fixed A; the values on morphisms is given correspondingly. This allows us to build *polynomial endofunctors* on the category **Sets**:

Polynomial endofunctors

may be skipped

For two sets A and B, their product $A \times B$ is defined as the set of pairs (x, y) with $x \in X$ and $y \in Y$. Their sum $A + B$ is defined as the disjoint union $\{(x,1)|x \in A\} \cup \{(y,2)|y \in B\}$.

The $\times$ and $+$ operations can be defined more generally for objects A and B of any category through certain 'universal properties' which do not concern us here, except that these universal properties give us two functors: $X \to A \times X$ and $X \to A + X$ for some fixed A; the values on morphisms is given correspondingly. This allows us to build *polynomial endofunctors* on the category **Sets**:

$$F(X) = C_0 + C_1 \times X + C_2 \times X^2 + \ldots, \text{where } C_n \text{ are fixed sets}$$

Algebraic structures as polynomial endofunctors

may be skipped

Algebraic structures as polynomial endofunctors

may be skipped

Let's look at monoids, sets with a binary associative multiplication and an identity element. We focus on the signature of the monoid structure, ignoring the monoid axioms. The signature consists of *one constant*, the unit, and *one binary function*, the multiplication, which, if X is a set with a monoid structure, can be written as

Algebraic structures as polynomial endofunctors

may be skipped

Let's look at monoids, sets with a binary associative multiplication and an identity element. We focus on the signature of the monoid structure, ignoring the monoid axioms. The signature consists of *one constant*, the unit, and *one binary function*, the multiplication, which, if X is a set with a monoid structure, can be written as

$$1 \xrightarrow{u} X$$

$$X \times X \xrightarrow{m} X$$

Algebraic structures as polynomial endofunctors

may be skipped

Let's look at monoids, sets with a binary associative multiplication and an identity element. We focus on the signature of the monoid structure, ignoring the monoid axioms. The signature consists of *one constant*, the unit, and *one binary function*, the multiplication, which, if X is a set with a monoid structure, can be written as

$$1 \xrightarrow{u} X$$

$$X \times X \xrightarrow{m} X$$

In the category **Sets**, the object 1 stands for a one-element set, $\{*\}$. (By the way, monoids are themselves categories, and the collection of all monoids is also a category, a sign of how broad the concept of category is)

Algebraic structures as polynomial endofunctors

may be skipped

Algebraic structures as polynomial endofunctors

may be skipped

Instead of looking at these functions individually, we collect them all together into one morphism $1 + X^2 \to X$ in the category **Sets** and therefore obtain that the monoid structure is captured by the polynomial functor:

$$F_{monoid}(X) = 1 + X^2$$

Algebraic structures as polynomial endofunctors

may be skipped

Instead of looking at these functions individually, we collect them all together into one morphism $1 + X^2 \to X$ in the category **Sets** and therefore obtain that the monoid structure is captured by the polynomial functor:

$$F_{monoid}(X) = 1 + X^2$$

Similarly, the following structures are given by their corresponding functors:

Algebraic structures as polynomial endofunctors

may be skipped

Instead of looking at these functions individually, we collect them all together into one morphism $1 + X^2 \to X$ in the category **Sets** and therefore obtain that the monoid structure is captured by the polynomial functor:

$$F_{monoid}(X) = 1 + X^2$$

Similarly, the following structures are given by their corresponding functors:

algebraic structure	polynomial endofunctor
semigroup	$F_{semigroup}(X) = X^2$
monoid	$F_{monoid}(X) = 1 + X^2$
group	$F_{group}(X) = 1 + X + X^2$
ring	$F_{ring}(X) = 2 + X + 2 \times X^2$

F-algebras

may be skipped

F-algebras

may be skipped

F-algebra

If F is an endofunctor on a category $\mathbf{C}$*, an F-algebra is a pair* (A, α) *where* A *is an object of* $\mathbf{C}$ *and* α *is a morphism*

$$\alpha : FA \to A$$

F-algebras

may be skipped

F-algebra

If F is an endofunctor on a category $\mathbf{C}$, an F-algebra is a pair (A, α) where A is an object of $\mathbf{C}$ and α is a morphism

$$\alpha : FA \to A$$

A morphism $h : (A, \alpha) \to (B, \beta)$ of F-algebras is a morphism $h : A \to B$ in $\mathbf{C}$ making the following diagram commute

F-algebras

may be skipped

F-algebra

If F is an endofunctor on a category $\mathbf{C}$, an F-algebra is a pair (A, α) where A is an object of $\mathbf{C}$ and α is a morphism

$$\alpha : FA \to A$$

A morphism $h : (A, \alpha) \to (B, \beta)$ of F-algebras is a morphism $h : A \to B$ in $\mathbf{C}$ making the following diagram commute

$$\begin{array}{ccc} FA & \xrightarrow{Fh} & FB \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \beta} \\ A & \xrightarrow{h} & B \end{array}$$

F-algebras

may be skipped

F-algebras

may be skipped

It is easy to see that F-algebras form a category of their own. Once you defined an F-algebra this way (and we now *know* the motivation for defining it this way), it is very natural to see what the morphisms should be; remember that everything in category theory is defined by commutative diagrams, just put two F-algebras side by side and the commutative diagram will suggest itself quite naturally:

F-algebras

may be skipped

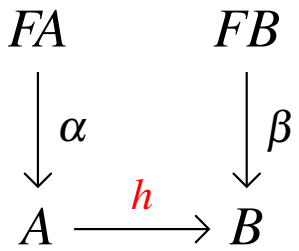
It is easy to see that F-algebras form a category of their own. Once you defined an F-algebra this way (and we now *know* the motivation for defining it this way), it is very natural to see what the morphisms should be; remember that everything in category theory is defined by commutative diagrams, just put two F-algebras side by side and the commutative diagram will suggest itself quite naturally:

$$\begin{array}{ccc} FA & & FB \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \beta} \\ A & & B \end{array}$$

F-algebras

may be skipped

It is easy to see that F-algebras form a category of their own. Once you defined an F-algebra this way (and we now *know* the motivation for defining it this way), it is very natural to see what the morphisms should be; remember that everything in category theory is defined by commutative diagrams, just put two F-algebras side by side and the commutative diagram will suggest itself quite naturally:



F-algebras

may be skipped

It is easy to see that F-algebras form a category of their own. Once you defined an F-algebra this way (and we now *know* the motivation for defining it this way), it is very natural to see what the morphisms should be; remember that everything in category theory is defined by commutative diagrams, just put two F-algebras side by side and the commutative diagram will suggest itself quite naturally:

$$\begin{array}{ccc} FA & \xrightarrow{Fh} & FB \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \beta} \\ A & \xrightarrow{h} & B \end{array}$$

Initial and final objects in a category

may be skipped

Initial and final objects in a category

may be skipped

initial and final objects

An object I of a category **C** *is called initial if for any object A of* **C** *there exists a unique morphism $I \to A$. An object T is final if for any object A of* **C** *there exists a unique morphism $A \to T$.*

Initial and final objects in a category

may be skipped

initial and final objects

An object I of a category **C** *is called initial if for any object A of* **C** *there exists a unique morphism $I \to A$. An object T is final if for any object A of* **C** *there exists a unique morphism $A \to T$.*

- A category may have neither an initial object nor a final one

Initial and final objects in a category

may be skipped

initial and final objects

An object I of a category **C** *is called initial if for any object A of* **C** *there exists a unique morphism $I \to A$. An object T is final if for any object A of* **C** *there exists a unique morphism $A \to T$.*

- A category may have neither an initial object nor a final one
- Any two initial objects, when they exist, are isomorphic and any two final objects are isomorphic (easy to check)

Initial and final objects in a category

may be skipped

initial and final objects

An object I of a category **C** *is called initial if for any object A of* **C** *there exists a unique morphism $I \to A$. An object T is final if for any object A of* **C** *there exists a unique morphism $A \to T$.*

- A category may have neither an initial object nor a final one
- Any two initial objects, when they exist, are isomorphic and any two final objects are isomorphic (easy to check)
- Examples: In **Sets**, the empty set is an initial object and any set consisting of just one element is a final object

Initial and final objects in a category

may be skipped

initial and final objects

An object I of a category **C** *is called initial if for any object A of* **C** *there exists a unique morphism $I \to A$. An object T is final if for any object A of* **C** *there exists a unique morphism $A \to T$.*

- A category may have neither an initial object nor a final one
- Any two initial objects, when they exist, are isomorphic and any two final objects are isomorphic (easy to check)
- Examples: In **Sets**, the empty set is an initial object and any set consisting of just one element is a final object
- **Most important data types have semantics as initial or final objects in the appropriate category**

Why are initial algebras important?

may be skipped

Why are initial algebras important?

may be skipped

Recursive definitions of functions and inductive proofs of their properties are two of the most important concepts in the formalization of software.

What is the relationship between the two? What do they really mean? *Initiality, a categorial concept, gives us the answer*. This is a typical example of why we lean on categories: our program is large and we would not have the time to motivate recursion and induction many times over. We do it once, at the most abstract, conceptual level.

Specifically, the fact that a data type admits an initial algebra semantics means exactly that we can define functions on it *recursively* and we can prove things about these functions *inductively*. Let's look at the simplest case, the data type of natural numbers.

Example: natural numbers as a data type

may be skipped

Example: natural numbers as a data type

may be skipped

The data type of natural numbers, Nat, is defined as follows (we use Haskell syntax, but any other functional language would do):

$$data\ Nat\ =\ Zero \mid S\ Nat$$

Example: natural numbers as a data type

may be skipped

The data type of natural numbers, Nat, is defined as follows (we use Haskell syntax, but any other functional language would do):

$$data\ Nat\ =\ Zero \mid S\ Nat$$

The type has a signature consisting of the constant $Zero$ and the unary operation S. This structure is embodied by the endofunctor $F_{nat}(X) = 1 + X$.

Example: natural numbers as a data type

may be skipped

The data type of natural numbers, Nat, is defined as follows (we use Haskell syntax, but any other functional language would do):

$$data\ Nat\ =\ Zero \mid S\ Nat$$

The type has a signature consisting of the constant $Zero$ and the unary operation S. This structure is embodied by the endofunctor $F_{nat}(X) = 1 + X$. Let's look at the set of natural numbers $\mathbb{N}$, with the constant $0 \in \mathbb{N}$ and function $s : \mathbb{N} \to \mathbb{N}$ given by $s(n) = n + 1$, in other words we look at the F_{nat}-algebra $(\mathbb{N}, (0, s))$.

Example: natural numbers as a data type

may be skipped

The data type of natural numbers, Nat, is defined as follows (we use Haskell syntax, but any other functional language would do):

$$data\ Nat\ =\ Zero \mid S\ Nat$$

The type has a signature consisting of the constant $Zero$ and the unary operation S. This structure is embodied by the endofunctor $F_{nat}(X) = 1 + X$. Let's look at the set of natural numbers $\mathbb{N}$, with the constant $0 \in \mathbb{N}$ and function $s : \mathbb{N} \to \mathbb{N}$ given by $s(n) = n + 1$, in other words we look at the F_{nat}-algebra $(\mathbb{N}, (0, s))$.

Say $f : X \to X$ is a function on an arbitrary set X. Fix an $a \in X$ which is the same thing as fixing a unary function $u : 1 \to X$ where $u(*) = a$. In other words, we have a pair $(X, (u, f) : 1 + X \to X)$, i.e. another F_{nat}-algebra.

Example: natural numbers as a data type

may be skipped

The data type of natural numbers, Nat, is defined as follows (we use Haskell syntax, but any other functional language would do):

$$data\ Nat\ =\ Zero \mid S\ Nat$$

The type has a signature consisting of the constant $Zero$ and the unary operation S. This structure is embodied by the endofunctor $F_{nat}(X) = 1 + X$. Let's look at the set of natural numbers $\mathbb{N}$, with the constant $0 \in \mathbb{N}$ and function $s : \mathbb{N} \to \mathbb{N}$ given by $s(n) = n + 1$, in other words we look at the F_{nat}-algebra $(\mathbb{N}, (0, s))$.

Say $f : X \to X$ is a function on an arbitrary set X. Fix an $a \in X$ which is the same thing as fixing a unary function $u : 1 \to X$ where $u(*) = a$. In other words, we have a pair $(X, (u, f) : 1 + X \to X)$, i.e. another F_{nat}-algebra. Let's see what it means that $(\mathbb{N}, (0, s))$ *is an initial* F_{nat}*-algebra*.

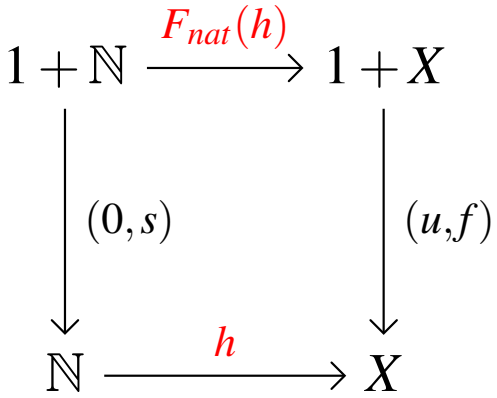
Example: the initial algebra for natural numbers

may be skipped

Example: the initial algebra for natural numbers

may be skipped

To say that $(\mathbb{N}, (0, s))$ is the initial object in the category of F_{nat}-algebras means, according to the definitions of F-algebras and initiality, that there exists a unique $h : \mathbb{N} \to X$ that makes the following diagram commute:



Example: natural numbers

may be skipped

Example: natural numbers

may be skipped

By looking at the previous diagram, we see that initiality of $(\mathbb{N}, (0, s))$ means two things:

Example: natural numbers

may be skipped

By looking at the previous diagram, we see that initiality of $(\mathbb{N}, (0, s))$ means two things:

- (Existence) There exists a function $h : \mathbb{N} \to X$ such that $h(0) = a$ and $h(S(n)) = f(h(n))$

Example: natural numbers

may be skipped

By looking at the previous diagram, we see that initiality of $(\mathbb{N}, (0, s))$ means two things:

- (Existence) There exists a function $h : \mathbb{N} \to X$ such that $h(0) = a$ and $h(S(n)) = f(h(n))$
- (Uniqueness) If a function $g : \mathbb{N} \to X$ is such that $g(0) = a$ and $g(S(n)) = f(g(n))$, then this function must be h

Example: natural numbers

may be skipped

By looking at the previous diagram, we see that initiality of $(\mathbb{N}, (0, s))$ means two things:

- (Existence) There exists a function $h : \mathbb{N} \to X$ such that $h(0) = a$ and $h(S(n)) = f(h(n))$
- (Uniqueness) If a function $g : \mathbb{N} \to X$ is such that $g(0) = a$ and $g(S(n)) = f(g(n))$, then this function must be h

Example: natural numbers

may be skipped

By looking at the previous diagram, we see that initiality of $(\mathbb{N}, (0, s))$ means two things:

- (Existence) There exists a function $h : \mathbb{N} \to X$ such that $h(0) = a$ and $h(S(n)) = f(h(n))$
- (Uniqueness) If a function $g : \mathbb{N} \to X$ is such that $g(0) = a$ and $g(S(n)) = f(g(n))$, then this function must be h

So, the *existence part of initiality expresses definition by recursion* while the *uniqueness part of initiality expresses proof by induction*.

Example: natural numbers

may be skipped

By looking at the previous diagram, we see that initiality of $(\mathbb{N}, (0, s))$ means two things:

- (Existence) There exists a function $h : \mathbb{N} \to X$ such that $h(0) = a$ and $h(S(n)) = f(h(n))$
- (Uniqueness) If a function $g : \mathbb{N} \to X$ is such that $g(0) = a$ and $g(S(n)) = f(g(n))$, then this function must be h

So, the *existence part of initiality expresses definition by recursion* while the *uniqueness part of initiality expresses proof by induction*. There is no better way to understand induction and recursion than through initiality.

Example: natural numbers

may be skipped

By looking at the previous diagram, we see that initiality of $(\mathbb{N}, (0, s))$ means two things:

- (Existence) There exists a function $h : \mathbb{N} \to X$ such that $h(0) = a$ and $h(S(n)) = f(h(n))$
- (Uniqueness) If a function $g : \mathbb{N} \to X$ is such that $g(0) = a$ and $g(S(n)) = f(g(n))$, then this function must be h

So, the *existence part of initiality expresses definition by recursion* while the *uniqueness part of initiality expresses proof by induction*. There is no better way to understand induction and recursion than through initiality.

Having said that, note that the terms induction and recursion are often interchanged: data can be called inductive or recursive, functions can be called inductive or recursive, etc...

Algebraic data types have initial algebra semantics

may be skipped

Algebraic data types have initial algebra semantics

may be skipped

- The type List A with signature consisting of constant [], and unary cons, has initial algebra semantics, with $F(X) = 1 + A \times X$.

Algebraic data types have initial algebra semantics

may be skipped

- The type List A with signature consisting of constant [], and unary cons, has initial algebra semantics, with $F(X) = 1 + A \times X$.
  - *We have a very pleasant explanation of why lists are so important in software, their functor is the linear function!*

Algebraic data types have initial algebra semantics

may be skipped

- The type List A with signature consisting of constant [], and unary cons, has initial algebra semantics, with $F(X) = 1 + A \times X$.
  - *We have a very pleasant explanation of why lists are so important in software, their functor is the linear function!*
  - If we take A to be 1, we recover the fact that the natural numbers are lists over the set with one element: 0=[]; 1=[*]; 2=[**] ...

Algebraic data types have initial algebra semantics

may be skipped

- The type List A with signature consisting of constant [], and unary cons, has initial algebra semantics, with $F(X) = 1 + A \times X$.
  - *We have a very pleasant explanation of why lists are so important in software, their functor is the linear function!*
  - If we take A to be 1, we recover the fact that the natural numbers are lists over the set with one element: 0=[]; 1=[*]; 2=[**] ...
- The type Tree (unlabeled binary tree) with signature consisting of constant Leaf, and binary Branch, has initial algebra semantics, with $F(X) = 1 + X^2$.

Algebraic data types have initial algebra semantics

may be skipped

- The type List A with signature consisting of constant [], and unary cons, has initial algebra semantics, with $F(X) = 1 + A \times X$.
  - *We have a very pleasant explanation of why lists are so important in software, their functor is the linear function!*
  - If we take A to be 1, we recover the fact that the natural numbers are lists over the set with one element: 0=[]; 1=[*]; 2=[**] ...
- The type Tree (unlabeled binary tree) with signature consisting of constant Leaf, and binary Branch, has initial algebra semantics, with $F(X) = 1 + X^2$.
- The type Tree A (labeled binary tree with leaves of type A) with signature consisting of constant Leaf A, and binary Branch, has initial algebra semantics, with $F(X) = A + A \times X^2$.

Algebraic data types have initial algebra semantics

may be skipped

Algebraic data types have initial algebra semantics

may be skipped

Let's note the difference between a data type and its semantics, i.e. the type of natural numbers, Nat, and its model $(\mathbb{N}, 0, s)$. One belongs to a programming language, the other is a mathematical object. *Part of our work on formalizing software is to make these distinctions clear*.

Algebraic data types have initial algebra semantics

may be skipped

Let's note the difference between a data type and its semantics, i.e. the type of natural numbers, Nat, and its model $(\mathbb{N}, 0, s)$. One belongs to a programming language, the other is a mathematical object. *Part of our work on formalizing software is to make these distinctions clear*.

We can regard Nat as a theory in the logic of the compiler/interpreter. Let's do this in Maude, since its semantics are formal (do not worry about 'sort' for now):

Algebraic data types have initial algebra semantics

may be skipped

Let's note the difference between a data type and its semantics, i.e. the type of natural numbers, Nat, and its model $(\mathbb{N}, 0, s)$. One belongs to a programming language, the other is a mathematical object. *Part of our work on formalizing software is to make these distinctions clear*.

We can regard Nat as a theory in the logic of the compiler/interpreter. Let's do this in Maude, since its semantics are formal (do not worry about 'sort' for now):

$$\begin{aligned} &sort\ Nat\ . \\ &op\ zero\ :\ \rightarrow Nat\ . \\ &op\ s\_\ :\ Nat\ \rightarrow Nat\ . \end{aligned}$$

Algebraic data types have initial algebra semantics

may be skipped

Algebraic data types have initial algebra semantics

may be skipped

We can extend the Nat theory by adding addition and multiplication to arrive at (the Peano) arithmetic. So we can express arithmetic as a theory in Maude. We can add other functions using recursion. Then we can prove certain properties of these theories using induction.

Algebraic data types have initial algebra semantics

may be skipped

We can extend the Nat theory by adding addition and multiplication to arrive at (the Peano) arithmetic. So we can express arithmetic as a theory in Maude. We can add other functions using recursion. Then we can prove certain properties of these theories using induction.

So data together with function definitions also form a theory, expressed in the language of Maude. The Maude interpreter calculates (reduces) a program (term) that is expressed in such a theory (which is itself an extension of Maude's equational logic). This view of data and functions as a theory is very useful, not just in Maude but in general.

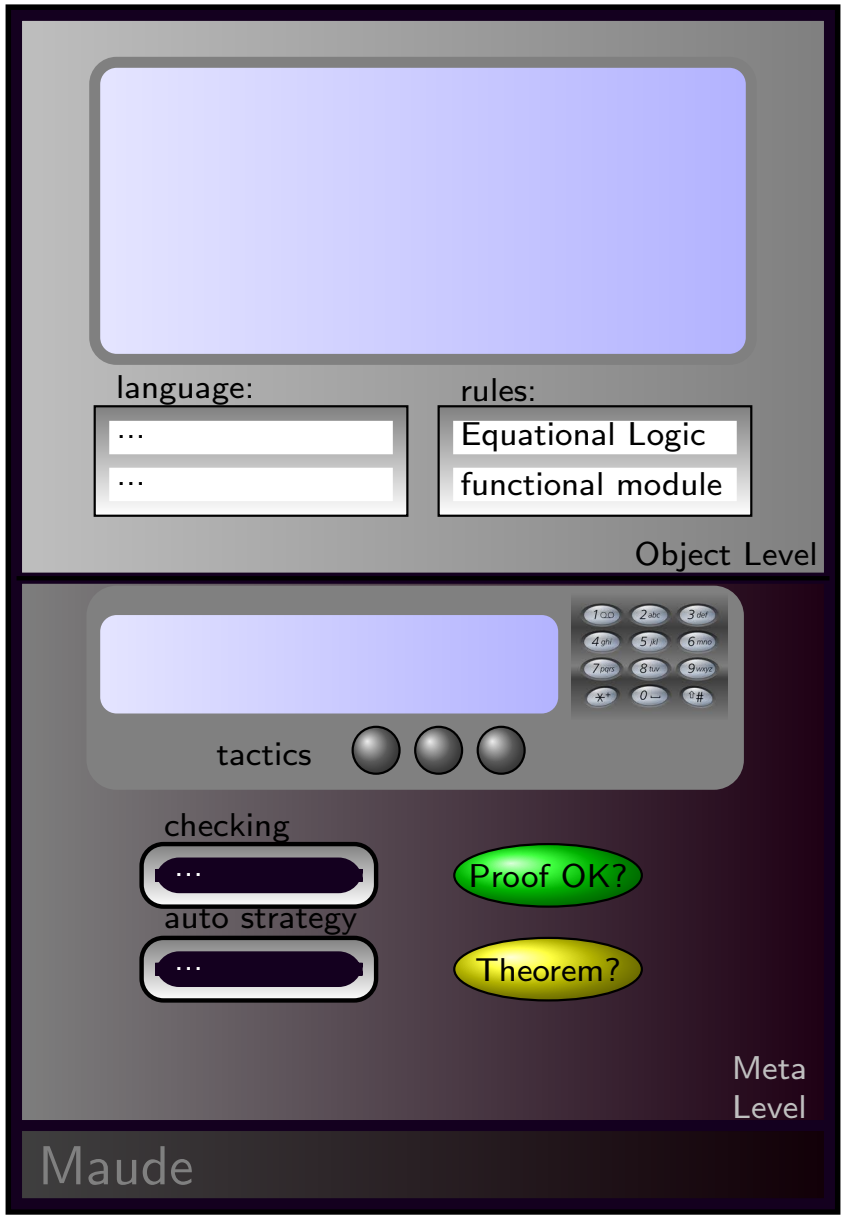
Example: initial algebra semantics in Maude

may be skipped

Example: initial algebra semantics in Maude

may be skipped

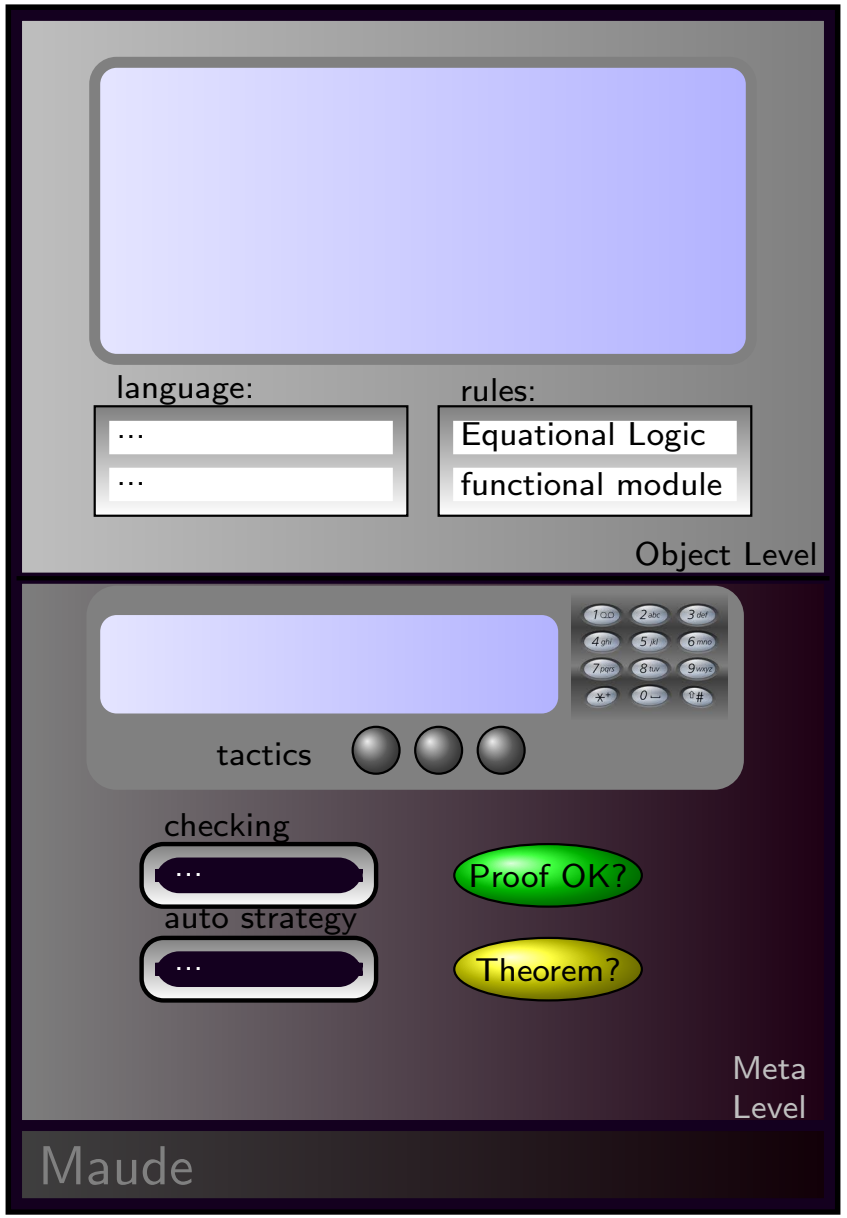
- A Maude functional module defines a theory T in equational logic



Example: initial algebra semantics in Maude

may be skipped

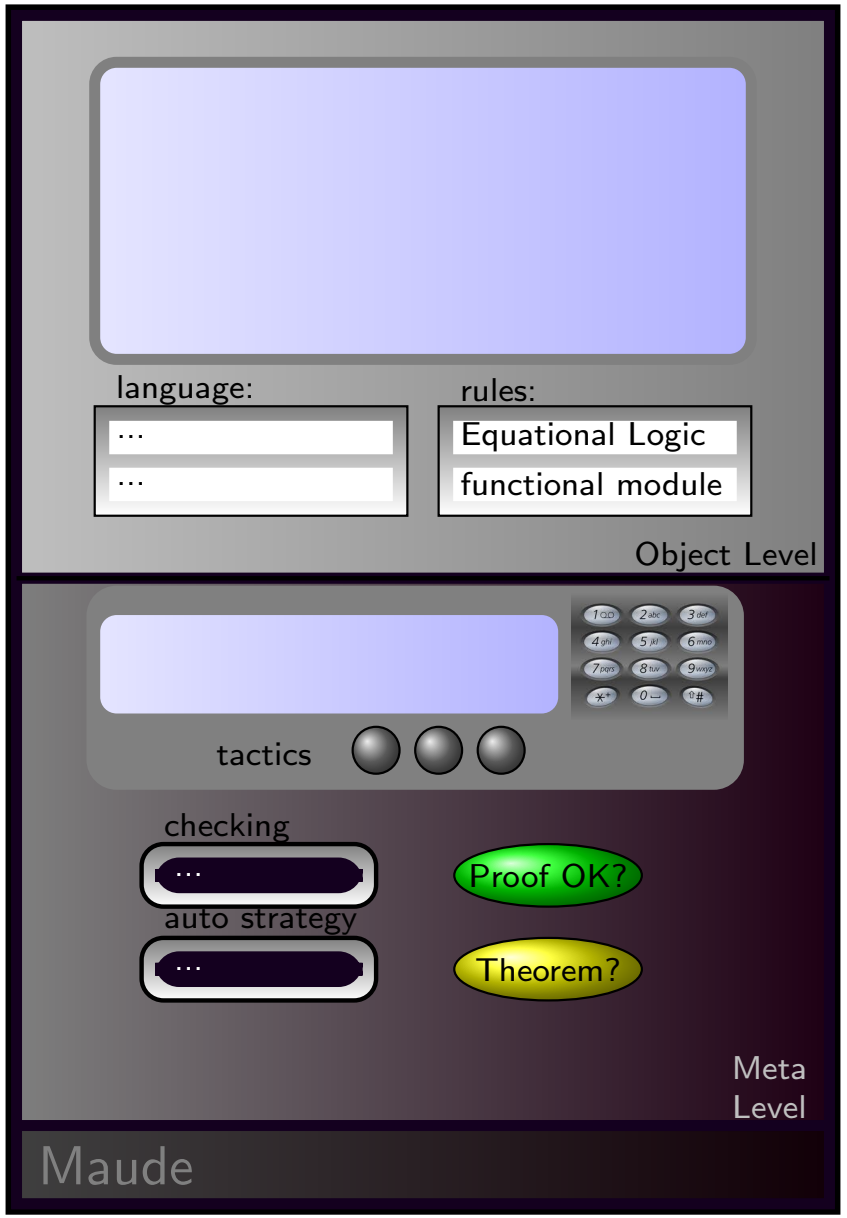
- A Maude functional module defines a theory T in equational logic
- Such a theory admits an initial semantic algebra; it is formed by taking the equivalence classes of terms provably equal in the theory



Example: initial algebra semantics in Maude

may be skipped

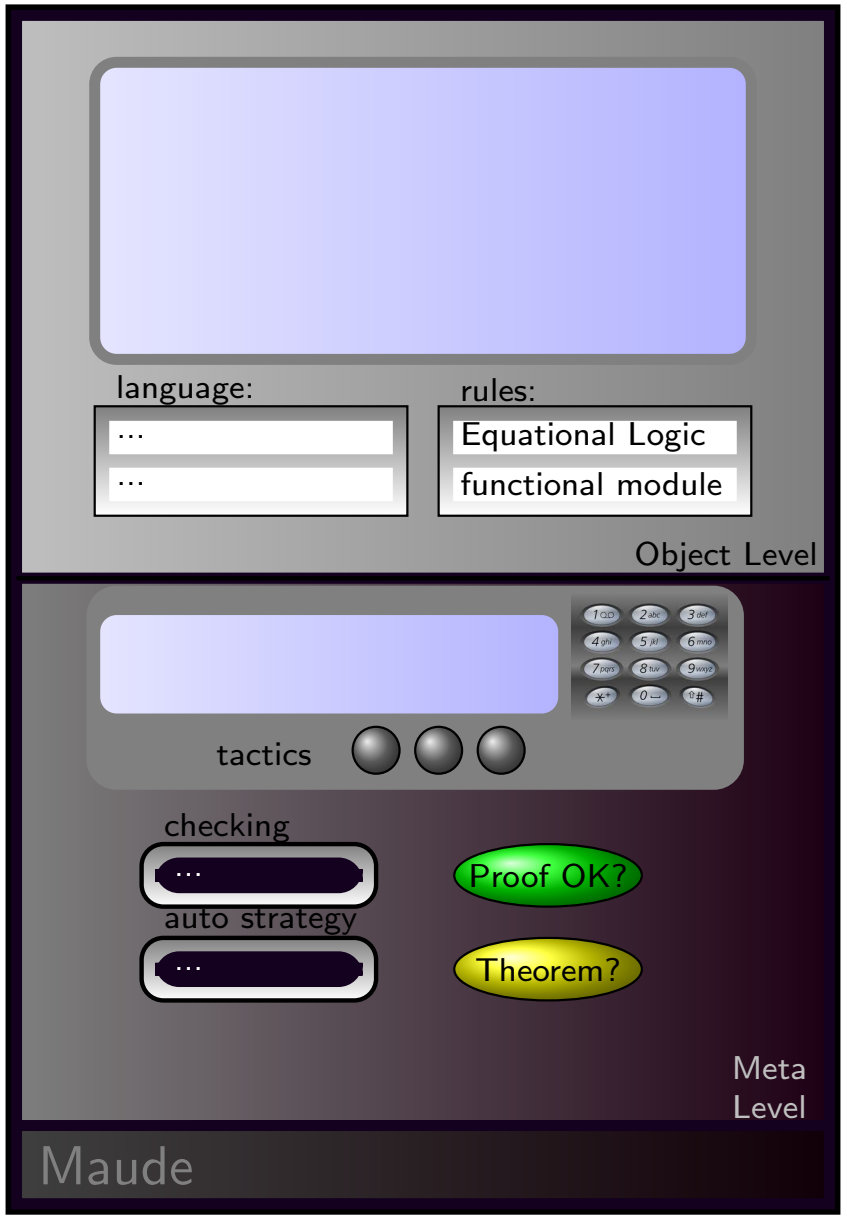
- A Maude functional module defines a theory T in equational logic
- Such a theory admits an initial semantic algebra; it is formed by taking the equivalence classes of terms provably equal in the theory
- This construction of the initial algebra of a theory is encountered many times in our program



Example: initial algebra semantics in Maude

may be skipped

- A Maude functional module defines a theory T in equational logic
- Such a theory admits an initial semantic algebra; it is formed by taking the equivalence classes of terms provably equal in the theory
- This construction of the initial algebra of a theory is encountered many times in our program
- (The initial algebra model of propositional calculus is the two-valued Boolean algebra)



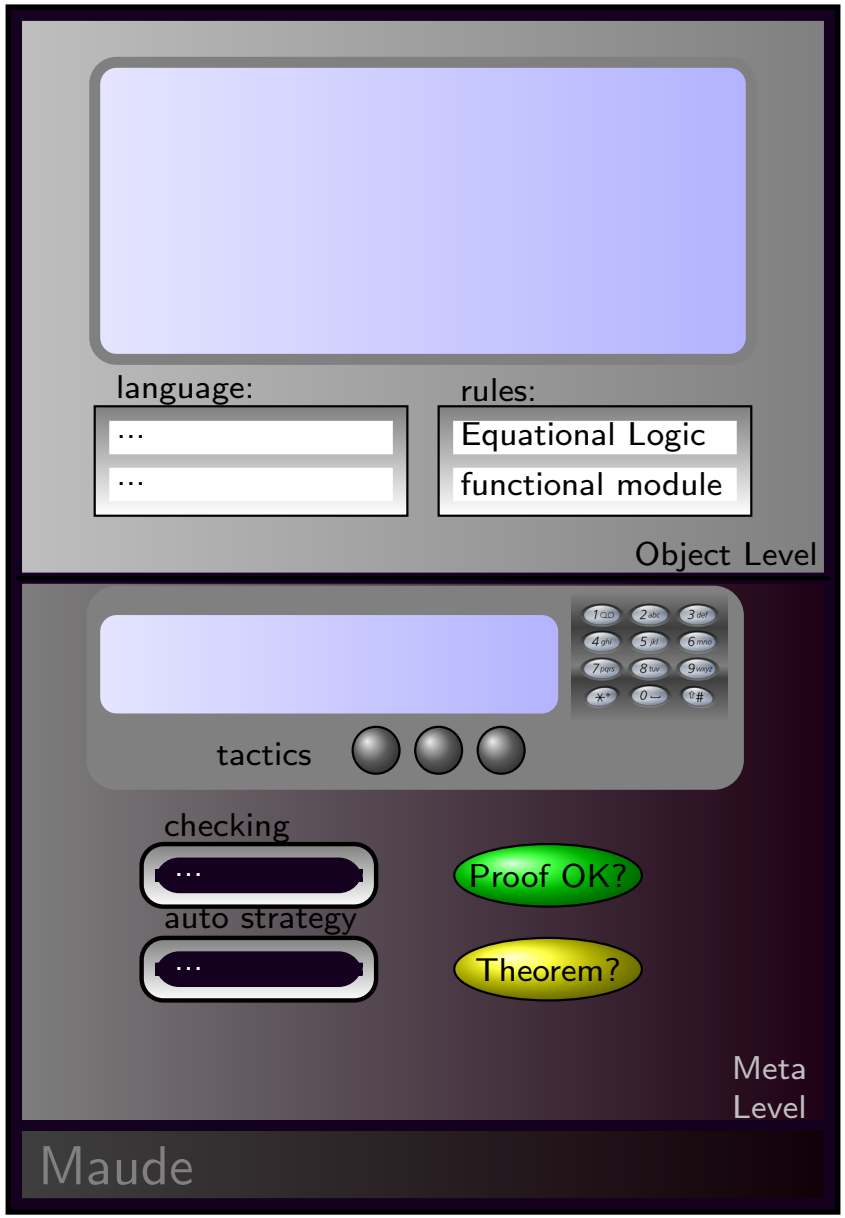
Algebraic semantics and operational semantics agree

may be skipped

Algebraic semantics and operational semantics agree

may be skipped

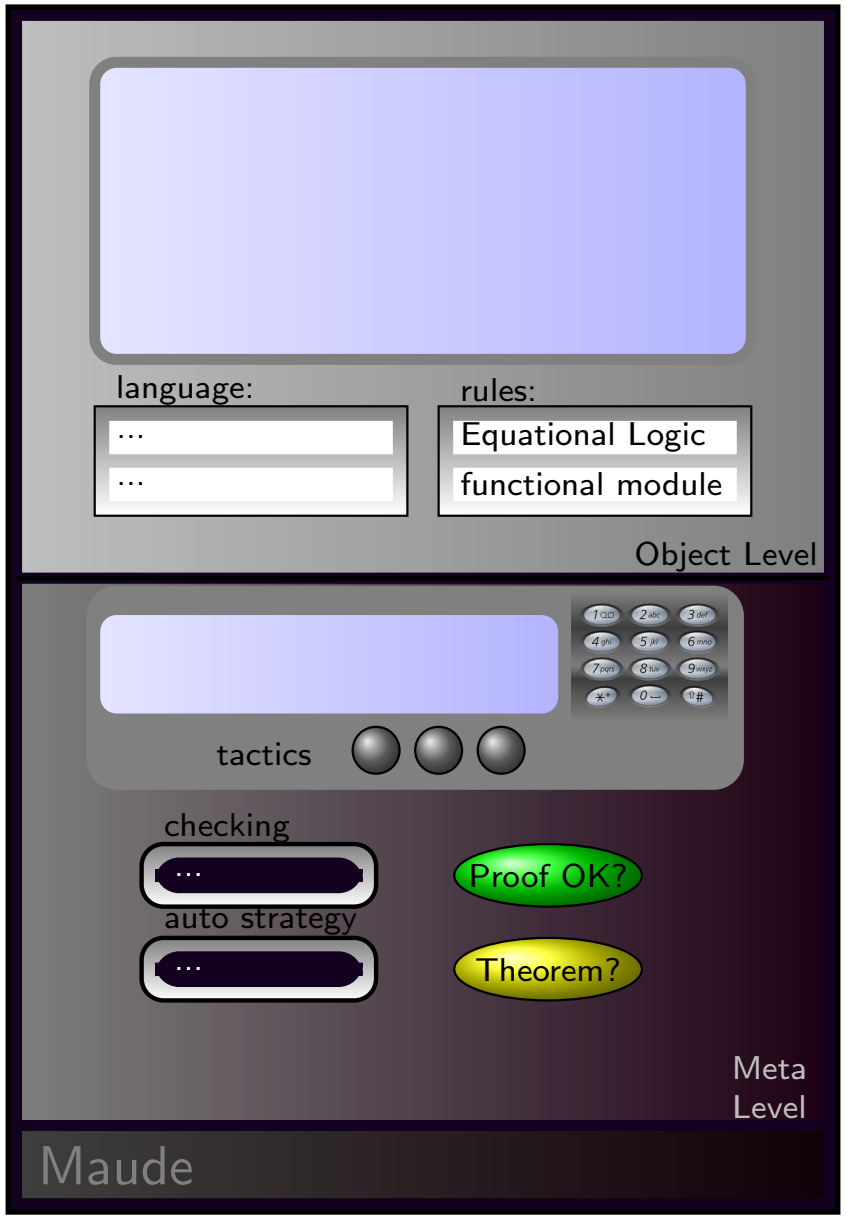
- Those terms that cannot be simplified further through the use of the theory are called *canonical terms*.



Algebraic semantics and operational semantics agree

may be skipped

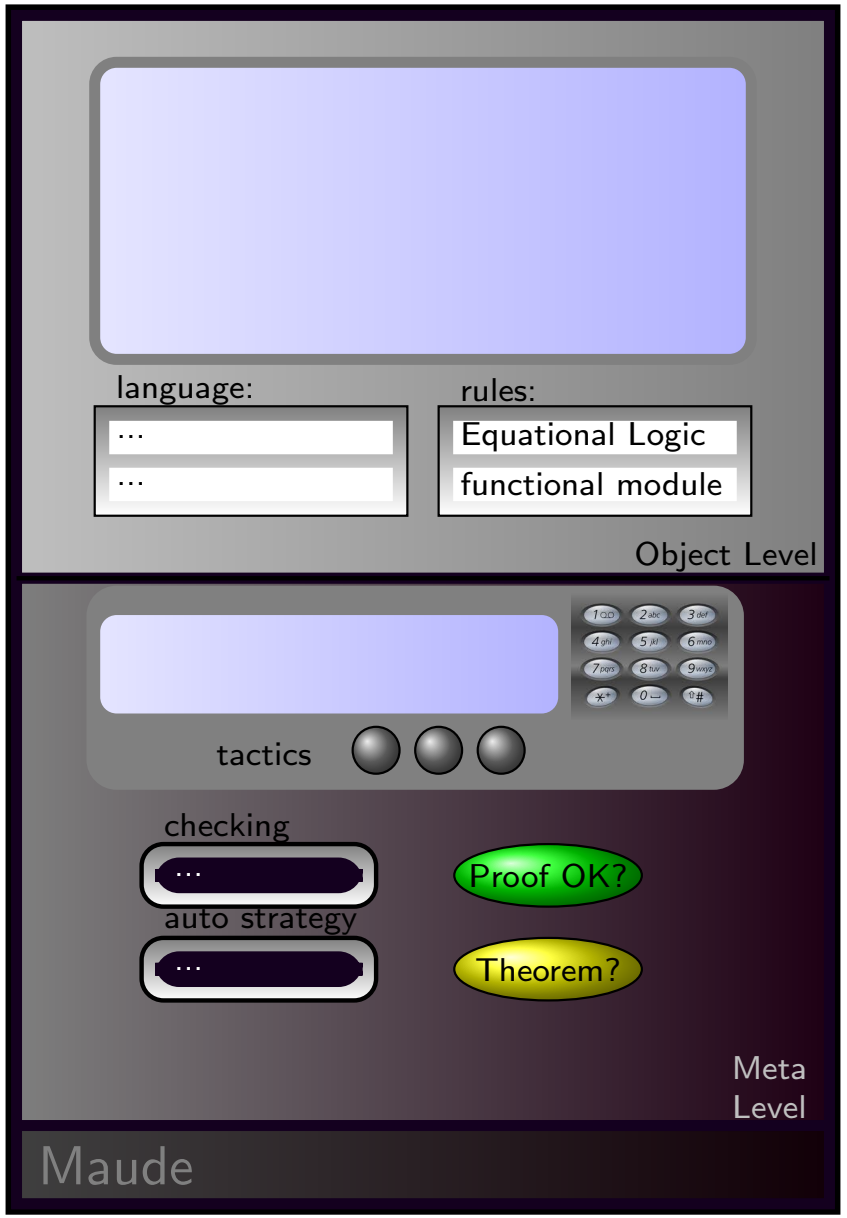
- Those terms that cannot be simplified further through the use of the theory are called *canonical terms*.
- Under certain technical conditions on the theory, these terms exist (their collection forms the canonical term algebra)



Algebraic semantics and operational semantics agree

may be skipped

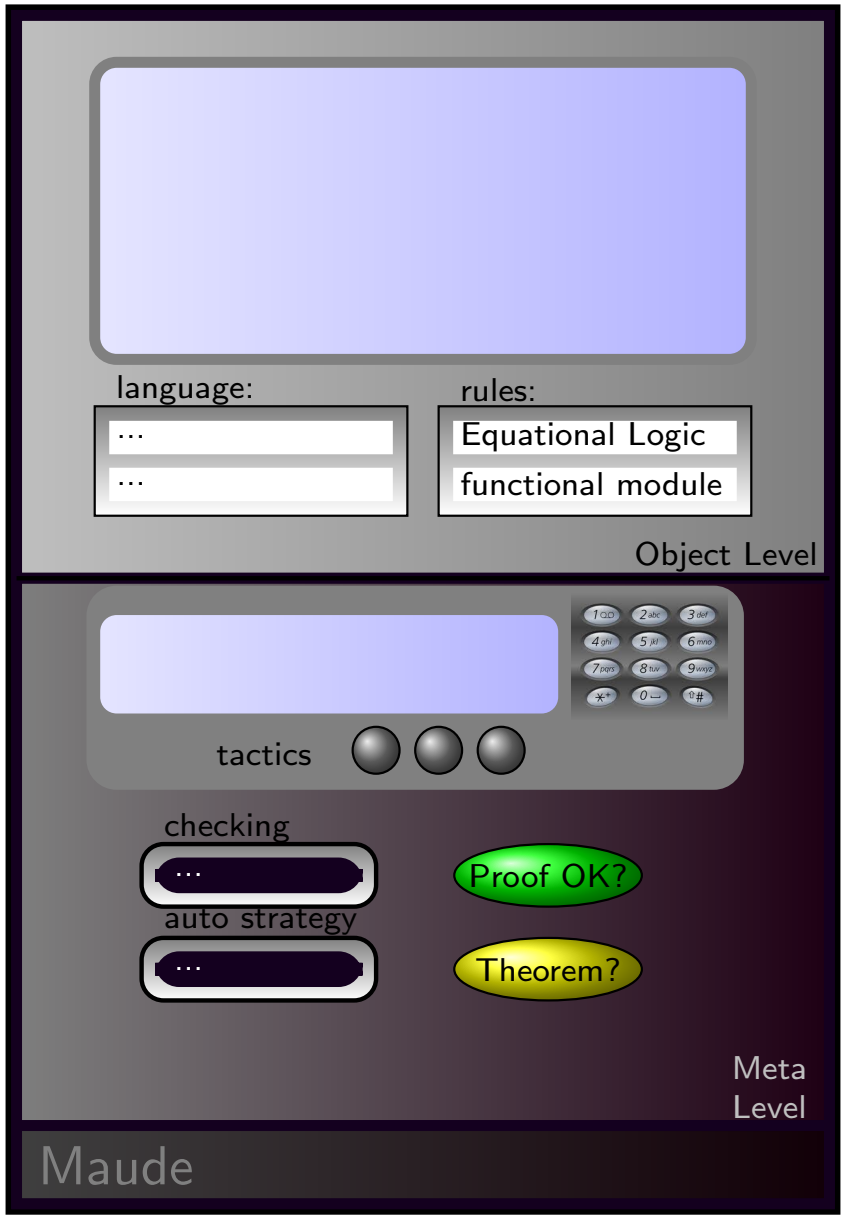
- Those terms that cannot be simplified further through the use of the theory are called *canonical terms*.
- Under certain technical conditions on the theory, these terms exist (their collection forms the canonical term algebra)
- Maude reduces a term to its canonical form, this is proof-theoretic semantics



Algebraic semantics and operational semantics agree

may be skipped

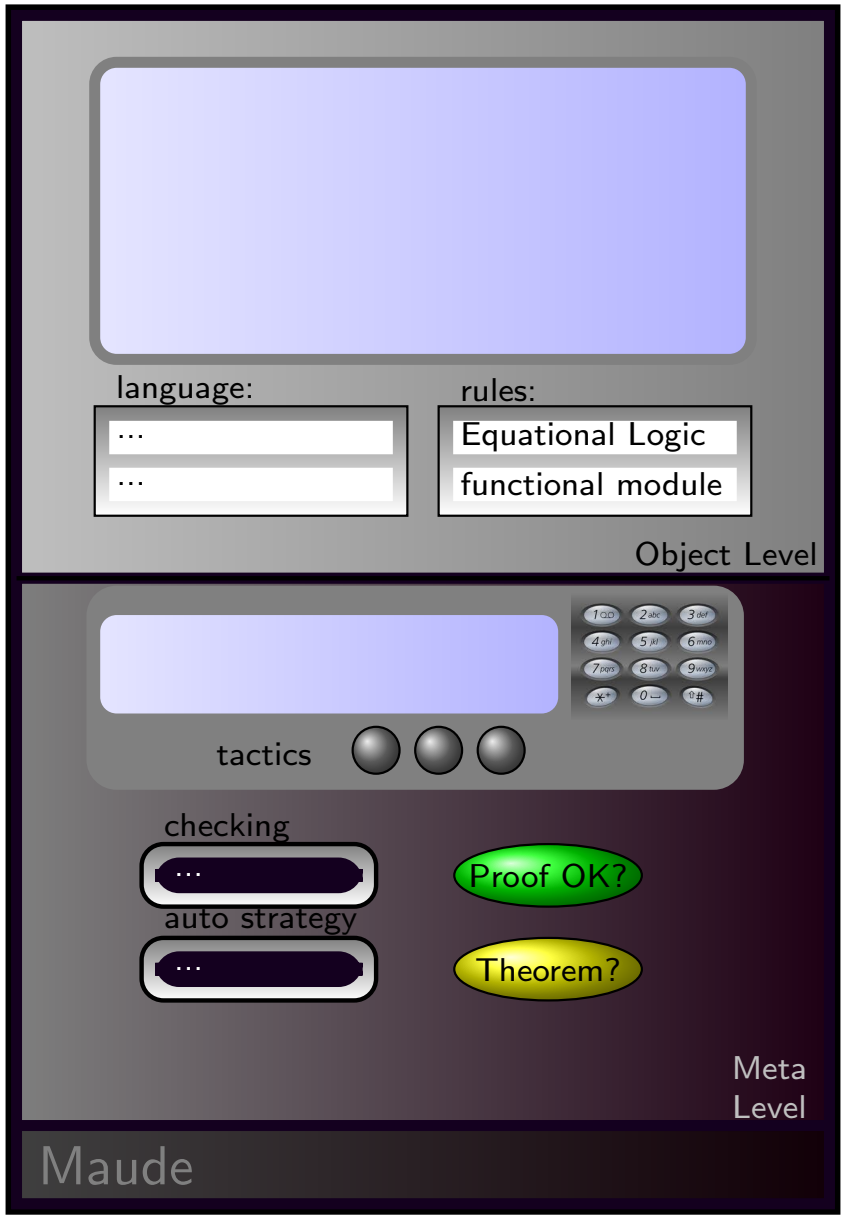
- Those terms that cannot be simplified further through the use of the theory are called *canonical terms*.
- Under certain technical conditions on the theory, these terms exist (their collection forms the canonical term algebra)
- Maude reduces a term to its canonical form, this is proof-theoretic semantics
- These two semantics agree, i.e. the two algebras are isomorphic



Algebraic semantics and operational semantics agree

may be skipped

- Those terms that cannot be simplified further through the use of the theory are called *canonical terms*.
- Under certain technical conditions on the theory, these terms exist (their collection forms the canonical term algebra)
- Maude reduces a term to its canonical form, this is proof-theoretic semantics
- These two semantics agree, i.e. the two algebras are isomorphic
- These two kinds of semantics are basic to everything we do in the program



State transitions with category theory

may be skipped

State transitions with category theory

may be skipped

So far everything has been about inductively defined data and operations on this data. In other words, functional programming. The main characteristic of imperative programs (one that functional programs do not have) is the notion of state. State transitions of systems can be also be described with category theory. Endofunctors F are used here, but in a dual fashion. We do not have the space to introduce duality, suffice it to say that $F-coalgebras$ are the dual notion to $F-algebras$, 'finality' is dual to 'initiality', and the semantics of state transitions can be given with final $F-coalgebras$. So, with duality taken for granted:

State transitions with category theory

may be skipped

So far everything has been about inductively defined data and operations on this data. In other words, functional programming. The main characteristic of imperative programs (one that functional programs do not have) is the notion of state. State transitions of systems can be also be described with category theory. Endofunctors F are used here, but in a dual fashion. We do not have the space to introduce duality, suffice it to say that $F-coalgebras$ are the dual notion to $F-algebras$, 'finality' is dual to 'initiality', and the semantics of state transitions can be given with final $F-coalgebras$. So, with duality taken for granted:

- data and operations on data have initial algebra semantics

State transitions with category theory

may be skipped

So far everything has been about inductively defined data and operations on this data. In other words, functional programming. The main characteristic of imperative programs (one that functional programs do not have) is the notion of state. State transitions of systems can be also be described with category theory. Endofunctors F are used here, but in a dual fashion. We do not have the space to introduce duality, suffice it to say that $F-coalgebras$ are the dual notion to $F-algebras$, 'finality' is dual to 'initiality', and the semantics of state transitions can be given with final $F-coalgebras$. So, with duality taken for granted:

- data and operations on data have initial algebra semantics
- systems and their state transitions have final coalgebra semantics

State transitions with category theory

may be skipped

So far everything has been about inductively defined data and operations on this data. In other words, functional programming. The main characteristic of imperative programs (one that functional programs do not have) is the notion of state. State transitions of systems can be also be described with category theory. Endofunctors F are used here, but in a dual fashion. We do not have the space to introduce duality, suffice it to say that $F-coalgebras$ are the dual notion to $F-algebras$, 'finality' is dual to 'initiality', and the semantics of state transitions can be given with final $F-coalgebras$. So, with duality taken for granted:

- data and operations on data have initial algebra semantics
- systems and their state transitions have final coalgebra semantics

State transitions with category theory

may be skipped

So far everything has been about inductively defined data and operations on this data. In other words, functional programming. The main characteristic of imperative programs (one that functional programs do not have) is the notion of state. State transitions of systems can be also be described with category theory. Endofunctors F are used here, but in a dual fashion. We do not have the space to introduce duality, suffice it to say that $F-coalgebras$ are the dual notion to $F-algebras$, 'finality' is dual to 'initiality', and the semantics of state transitions can be given with final $F-coalgebras$. So, with duality taken for granted:

- data and operations on data have initial algebra semantics
- systems and their state transitions have final coalgebra semantics

The algebraic study of software will only increase. One major piece that is still missing is a convincing algebraic description of the notion of *concurrency*.

Construction versus observation

may be skipped

Construction versus observation

may be skipped

It is easier if you look at algebraic types as data, but look at coalgebraic types as systems. So a stream would be a system. We can construct data but we cannot construct systems, we just *observe* systems. We cannot construct a stream, we can just observe it (by observing the head of the stream). We can reason inductively about data but have to reason coinductively about systems. The table below shows this duality.

data and functions	systems and transitions
algebraic type	coalgebraic type
(=inductive type)	(=coinductive type)
(=recursive type)	(=corecursive type)
definition by recursion	definition by corecursion
proof by induction	*proof by coinduction*
type is constructed	type is observed

Why all this algebra language?

may be skipped

Why all this algebra language?

may be skipped

This mention of algebra, coalgebra, categories, etc . . . should not give you the impression that, in the advanced sequence of the program, we'll chase unnecessarily abstract mathematics at the expense of practical applications. Quite the contrary, we need to find a way to speed up towards engineering goals. The truth is that while these applications of category theory are not particularly deep, they do establish a very effective vocabulary for us to use.

Why all this algebra language?

may be skipped

This mention of algebra, coalgebra, categories, etc . . . should not give you the impression that, in the advanced sequence of the program, we'll chase unnecessarily abstract mathematics at the expense of practical applications. Quite the contrary, we need to find a way to speed up towards engineering goals. The truth is that while these applications of category theory are not particularly deep, they do establish a very effective vocabulary for us to use.

But there is a stronger reason to use categories. Objects of categories can be thought of as 'variable sets'. The mathematics you learned in high-school used set theory and because of this familiarity, it would be nice if we could base everything on ordinary (=constant) sets. It turns out that *many formal systems that are needed in software cannot be modeled with ordinary sets*: e.g. lambda calculus, polymorphism, higher order logic, . . . As it was mentioned earlier, our working (but untested) assumption is that using a higher level of abstraction would eventually feel natural to software engineers.

My setup for lectures

Guest machine: Ubuntu 10.4 64-bit

- Proof General (needs GNU Emacs)
- Maude (logical platform)
- Coq (logical platform)
- Isabelle (logical platform)
- SPIN (model checker)
- LAMP (Apache, MySql and PHP)

Host machine: Windows 7 64-bit

- SWI-Prolog, GHC (Glasgow Haskell Compiler)
- Rodin (platform for Event-B modeling)
- Visual Studio 10

About labs

During the lectures, when you see the following picture . . .

. . . it means that we are branching out of the slide presentation and into a computer session on either Linux or Windows. We call these branch-outs 'labs'.

What follows here is a set of very quick labs (less than 5 minutes each) to give you an idea of the sort of tools that we use in this program and why we use them.

Labs: Examples of formal software development

- *Refining requirements with Event-B*

Labs: Examples of formal software development

- *Refining requirements with Event-B*
- *Building a model of a concurrent system with SPIN*

Labs: Examples of formal software development

- *Refining requirements with Event-B*
- *Building a model of a concurrent system with SPIN*
- *Building an executable model with Maude*

Labs: Examples of formal software development

- *Refining requirements with Event-B*
- *Building a model of a concurrent system with SPIN*
- *Building an executable model with Maude*
- *Building a certified program with Coq*

Labs: Examples of formal software development

- *Refining requirements with Event-B*
- *Building a model of a concurrent system with SPIN*
- *Building an executable model with Maude*
- *Building a certified program with Coq*
- *Solve logical problems with Prolog*

Labs: Examples of formal software development

- *Verify C programs with Why/Frama-C*

Labs: Examples of formal software development

- *Verify C programs with Why/Frama-C*
- *Verify Java programs with Why/Frama-C*

Labs: Examples of formal software development

- *Verify C programs with Why/Frama-C*
- *Verify Java programs with Why/Frama-C*
- *Program/Verify C# programs with Spec#/Boogie/Z3 tool chain*

Labs: Examples of formal software development

- *Verify C programs with Why/Frama-C*
- *Verify Java programs with Why/Frama-C*
- *Program/Verify C# programs with Spec#/Boogie/Z3 tool chain*
- *Verify concurrent C programs with VCC/Boogie/Z3 tool chain*

Labs: Examples of formal software development

- *Verify C programs with Why/Frama-C*
- *Verify Java programs with Why/Frama-C*
- *Program/Verify C# programs with Spec#/Boogie/Z3 tool chain*
- *Verify concurrent C programs with VCC/Boogie/Z3 tool chain*
- *Static Analysis of Java programs with ESC*

Labs: Examples of formal software development

- *Verify C programs with Why/Frama-C*
- *Verify Java programs with Why/Frama-C*
- *Program/Verify C# programs with Spec#/Boogie/Z3 tool chain*
- *Verify concurrent C programs with VCC/Boogie/Z3 tool chain*
- *Static Analysis of Java programs with ESC*
- *Static Analysis of C programs with PREfast*

Labs: Examples of formal software development

- *Verify C programs with Why/Frama-C*
- *Verify Java programs with Why/Frama-C*
- *Program/Verify C# programs with Spec#/Boogie/Z3 tool chain*
- *Verify concurrent C programs with VCC/Boogie/Z3 tool chain*
- *Static Analysis of Java programs with ESC*
- *Static Analysis of C programs with PREfast*
- *Model checking Linux drivers with BLAST*

Labs: Examples of formal software development

- *Verify C programs with Why/Frama-C*
- *Verify Java programs with Why/Frama-C*
- *Program/Verify C# programs with Spec#/Boogie/Z3 tool chain*
- *Verify concurrent C programs with VCC/Boogie/Z3 tool chain*
- *Static Analysis of Java programs with ESC*
- *Static Analysis of C programs with PREfast*
- *Model checking Linux drivers with BLAST*
- *Model checking Windows drivers with SLAM*

Understand the main concepts of formal software

One goal is for you to understand most of the concepts of formal software development in about 3 years.

Understand the main concepts of formal software

One goal is for you to understand most of the concepts of formal software development in about 3 years.

After that period, you should be able to:

Understand the main concepts of formal software

One goal is for you to understand most of the concepts of formal software development in about 3 years.

After that period, you should be able to:

- Understand where a certain problem belongs

Understand the main concepts of formal software

One goal is for you to understand most of the concepts of formal software development in about 3 years.

After that period, you should be able to:

- Understand where a certain problem belongs
- Read research papers

Understand the main concepts of formal software

One goal is for you to understand most of the concepts of formal software development in about 3 years.

After that period, you should be able to:

- Understand where a certain problem belongs
- Read research papers
- Use the latest versions of the tools

Understand the main concepts of formal software

One goal is for you to understand most of the concepts of formal software development in about 3 years.

After that period, you should be able to:

- Understand where a certain problem belongs
- Read research papers
- Use the latest versions of the tools
- Understand the trends

Understand the main concepts of formal software

One goal is for you to understand most of the concepts of formal software development in about 3 years.

After that period, you should be able to:

- Understand where a certain problem belongs
- Read research papers
- Use the latest versions of the tools
- Understand the trends
- Develop working relationships with teams around the globe

Understand the main concepts of formal software

One goal is for you to understand most of the concepts of formal software development in about 3 years.

After that period, you should be able to:

- Understand where a certain problem belongs
- Read research papers
- Use the latest versions of the tools
- Understand the trends
- Develop working relationships with teams around the globe
- Become an active participant in the improvement of various tools

Understand the main concepts of formal software

One goal is for you to understand most of the concepts of formal software development in about 3 years.

After that period, you should be able to:

- Understand where a certain problem belongs
- Read research papers
- Use the latest versions of the tools
- Understand the trends
- Develop working relationships with teams around the globe
- Become an active participant in the improvement of various tools
- Introduce formal thinking (and pilot projects) in your organization

Judge the effectiveness of various tools

Judge the effectiveness of various tools

Many of the tools we study are complex systems that have at least a syntactical front-end and a logical back-end implementing a certain logic. You will learn to judge the effectiveness of these tools, their tradeoffs and their suitability for certain practical applications. There are many questions to ask when contemplating formal software development with a specific tool.

Judge the effectiveness of various tools

Many of the tools we study are complex systems that have at least a syntactical front-end and a logical back-end implementing a certain logic. You will learn to judge the effectiveness of these tools, their tradeoffs and their suitability for certain practical applications. There are many questions to ask when contemplating formal software development with a specific tool.

General questions:

Judge the effectiveness of various tools

Many of the tools we study are complex systems that have at least a syntactical front-end and a logical back-end implementing a certain logic. You will learn to judge the effectiveness of these tools, their tradeoffs and their suitability for certain practical applications. There are many questions to ask when contemplating formal software development with a specific tool.

General questions:

- What other industrial-sized applications were developed with the tool?

Judge the effectiveness of various tools

Many of the tools we study are complex systems that have at least a syntactical front-end and a logical back-end implementing a certain logic. You will learn to judge the effectiveness of these tools, their tradeoffs and their suitability for certain practical applications. There are many questions to ask when contemplating formal software development with a specific tool.

General questions:

- What other industrial-sized applications were developed with the tool?
- How user-friendly is its interface?

Judge the effectiveness of various tools

Many of the tools we study are complex systems that have at least a syntactical front-end and a logical back-end implementing a certain logic. You will learn to judge the effectiveness of these tools, their tradeoffs and their suitability for certain practical applications. There are many questions to ask when contemplating formal software development with a specific tool.

General questions:

- What other industrial-sized applications were developed with the tool?
- How user-friendly is its interface?
- How configurable is the tool as a whole (front-end and back-end)?

Judge the effectiveness of various tools

Many of the tools we study are complex systems that have at least a syntactical front-end and a logical back-end implementing a certain logic. You will learn to judge the effectiveness of these tools, their tradeoffs and their suitability for certain practical applications. There are many questions to ask when contemplating formal software development with a specific tool.

General questions:

- What other industrial-sized applications were developed with the tool?
- How user-friendly is its interface?
- How configurable is the tool as a whole (front-end and back-end)?
- Can the tool support the development of a model during requirements analysis and design?

Judge the effectiveness of various tools

Judge the effectiveness of various tools

Questions specific to the logical back-end:

Judge the effectiveness of various tools

Questions specific to the logical back-end:

- How automatic is the logical back-end, i.e. what is the percentage of proofs it discharges automatically?

Judge the effectiveness of various tools

Questions specific to the logical back-end:

- How automatic is the logical back-end, i.e. what is the percentage of proofs it discharges automatically?
- How powerful is the logical back-end (how big and how complex are the formulas it can prove)?

Judge the effectiveness of various tools

Questions specific to the logical back-end:

- How automatic is the logical back-end, i.e. what is the percentage of proofs it discharges automatically?
- How powerful is the logical back-end (how big and how complex are the formulas it can prove)?
- Can it prove concurrency properties, for example interference freedom, deadlock freedom or fairness?

Judge the effectiveness of various tools

Questions specific to the logical back-end:

- How automatic is the logical back-end, i.e. what is the percentage of proofs it discharges automatically?
- How powerful is the logical back-end (how big and how complex are the formulas it can prove)?
- Can it prove concurrency properties, for example interference freedom, deadlock freedom or fairness?
- Can it prove liveness or termination?

Judge the effectiveness of various tools

Questions specific to the logical back-end:

- How automatic is the logical back-end, i.e. what is the percentage of proofs it discharges automatically?
- How powerful is the logical back-end (how big and how complex are the formulas it can prove)?
- Can it prove concurrency properties, for example interference freedom, deadlock freedom or fairness?
- Can it prove liveness or termination?
- Can it find powerful invariants on its own?

Judge the effectiveness of various tools

Questions specific to the logical back-end:

- How automatic is the logical back-end, i.e. what is the percentage of proofs it discharges automatically?
- How powerful is the logical back-end (how big and how complex are the formulas it can prove)?
- Can it prove concurrency properties, for example interference freedom, deadlock freedom or fairness?
- Can it prove liveness or termination?
- Can it find powerful invariants on its own?
- If a proof cannot be found automatically, does the system have a fallback proof assistant?

Judge the effectiveness of various tools

Questions specific to the logical back-end:

- How automatic is the logical back-end, i.e. what is the percentage of proofs it discharges automatically?
- How powerful is the logical back-end (how big and how complex are the formulas it can prove)?
- Can it prove concurrency properties, for example interference freedom, deadlock freedom or fairness?
- Can it prove liveness or termination?
- Can it find powerful invariants on its own?
- If a proof cannot be found automatically, does the system have a fallback proof assistant?
- How complicated are the logical foundations of this proof assistant?

Expected classroom participation

Expected classroom participation

Your work experience will play an essential role in understanding some of the engineering aspects of formal software development. In this process, it is the feedback you will provide during and after classes that will contribute to the program's strength.

Expected classroom participation

Your work experience will play an essential role in understanding some of the engineering aspects of formal software development. In this process, it is the feedback you will provide during and after classes that will contribute to the program's strength.

Consequently, you are expected to:

Expected classroom participation

Your work experience will play an essential role in understanding some of the engineering aspects of formal software development. In this process, it is the feedback you will provide during and after classes that will contribute to the program's strength.

Consequently, you are expected to:

- Participate in group discussions

Expected classroom participation

Your work experience will play an essential role in understanding some of the engineering aspects of formal software development. In this process, it is the feedback you will provide during and after classes that will contribute to the program's strength.

Consequently, you are expected to:

- Participate in group discussions
- Install and evaluate programming systems and logical tools between classes

Expected classroom participation

Your work experience will play an essential role in understanding some of the engineering aspects of formal software development. In this process, it is the feedback you will provide during and after classes that will contribute to the program's strength.

Consequently, you are expected to:

- Participate in group discussions
- Install and evaluate programming systems and logical tools between classes
- Read additional papers before class

Expected classroom participation

Your work experience will play an essential role in understanding some of the engineering aspects of formal software development. In this process, it is the feedback you will provide during and after classes that will contribute to the program's strength.

Consequently, you are expected to:

- Participate in group discussions
- Install and evaluate programming systems and logical tools between classes
- Read additional papers before class
- Occasionally make group presentations on selected topics

Learn to estimate the effort to formalize

Learn to estimate the effort to formalize

Learn to estimate the effort to program in JML or Spec#, rather than Java or C#. Learn to estimate the effort to add verification of safety properties (buffer overflow, arithmetic overflow, null pointer dereference, division by zero) to a C program. Learn to estimate the effort to design a concurrent model of the system that is being built. A example of such estimate would be:

$$N = \text{lines of code}$$
$$\text{theorems} = 2 \times N$$
$$\text{proof lines} = 20 \times \text{theorems}$$

Build a wide formal platform

Build a wide formal platform

Build a formal platform, an ongoing project:

Build a wide formal platform

Build a formal platform, an ongoing project:

- This platform will include all of the tools we study

Build a wide formal platform

Build a formal platform, an ongoing project:

- This platform will include all of the tools we study
- You will learn to mix and match various tools

Build a wide formal platform

Build a formal platform, an ongoing project:

- This platform will include all of the tools we study
- You will learn to mix and match various tools
- You will learn to transform the output of a tool so that it can be verified by another

Build a wide formal platform

Build a formal platform, an ongoing project:

- This platform will include all of the tools we study
- You will learn to mix and match various tools
- You will learn to transform the output of a tool so that it can be verified by another
- One goal of the program is to encourage you to participate in the development of this platform

Build a wide formal platform

Build a formal platform, an ongoing project:

- This platform will include all of the tools we study
- You will learn to mix and match various tools
- You will learn to transform the output of a tool so that it can be verified by another
- One goal of the program is to encourage you to participate in the development of this platform
- In terms of program verification, the platform will use multiple front-ends for C/Java/C#

Build a wide formal platform

Build a formal platform, an ongoing project:

- This platform will include all of the tools we study
- You will learn to mix and match various tools
- You will learn to transform the output of a tool so that it can be verified by another
- One goal of the program is to encourage you to participate in the development of this platform
- In terms of program verification, the platform will use multiple front-ends for C/Java/C#
- The platform will include multiple model checkers

Program Structure

Program Structure

Program is divided into four sequences:

Program Structure

Program is divided into four sequences:

- Foundation sequence (courses contain basic material that is needed throughout the program)

Program Structure

Program is divided into four sequences:

- Foundation sequence (courses contain basic material that is needed throughout the program)
- Core sequence (courses that cover the essential material needed for doing formal software development)

Program Structure

Program is divided into four sequences:

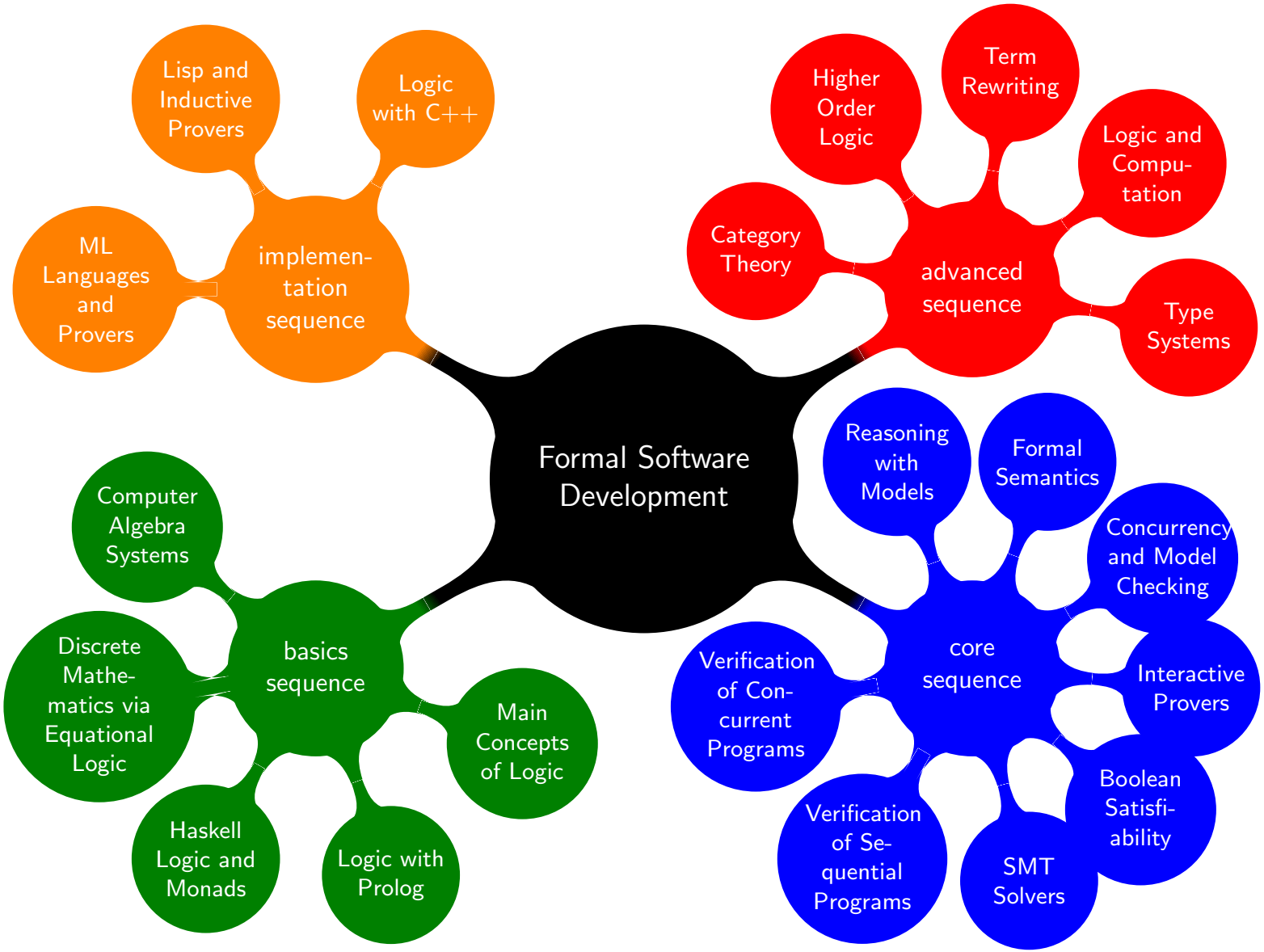
- Foundation sequence (courses contain basic material that is needed throughout the program)
- Core sequence (courses that cover the essential material needed for doing formal software development)
- Implementation sequence (how the logical tools are built)

Program Structure

Program is divided into four sequences:

- Foundation sequence (courses contain basic material that is needed throughout the program)
- Core sequence (courses that cover the essential material needed for doing formal software development)
- Implementation sequence (how the logical tools are built)
- Advanced sequence (a deeper understanding of the theory)

Program Structure



Recommended books I

Jean-Raymond Abrial

Modeling in Event-B System and Software Engineering

Cambridge University Press, 2010.

Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, Carolyn Talcott

All About Maude - A High-Performance Logical Framework

Springer-Verlag, 2007.

Yves Bertot, Pierre Castéran

Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions

Springer-Verlag, 2010.

Recommended books II


John Harrison

Handbook Of Practical Logic And Automated Reasoning

Springer-Verlag, 2009.


Gerard J. Holzmann

The SPIN Model Checker: Primer and Reference Manual

Addison-Wesley, 2004.


Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel

Isabelle/HOL: A Proof Assistant for Higher-Order Logic

Springer-Verlag, 2002.


Krzysztof R. Apt, Frank S. Boer, Ernst-Rüdiger Olderog

Verification of Sequential and Concurrent Programs

Springer-Verlag, 2009.

A big investment on your part

A big investment on your part

- Formal software development is a huge subject

A big investment on your part

- Formal software development is a huge subject
- It requires a considerable commitment

A big investment on your part

- Formal software development is a huge subject
- It requires a considerable commitment
- These are exciting times, a beehive of activity

A big investment on your part

- Formal software development is a huge subject
- It requires a considerable commitment
- These are exciting times, a beehive of activity
- Everywhere: US, Europe, Japan, China, India, ...

A big investment on your part

- Formal software development is a huge subject
- It requires a considerable commitment
- These are exciting times, a beehive of activity
- Everywhere: US, Europe, Japan, China, India, ...
- Things are falling into place, as mathematicians and software engineers work from both ends

A big investment on your part

- Formal software development is a huge subject
- It requires a considerable commitment
- These are exciting times, a beehive of activity
- Everywhere: US, Europe, Japan, China, India, ...
- Things are falling into place, as mathematicians and software engineers work from both ends
- *It is up to you how quickly they fall into place!*

A big investment on your part

- Formal software development is a huge subject
- It requires a considerable commitment
- These are exciting times, a beehive of activity
- Everywhere: US, Europe, Japan, China, India, ...
- Things are falling into place, as mathematicians and software engineers work from both ends
- *It is up to you how quickly they fall into place!*
- Great time to produce machine-proven software

A big investment on your part

- Formal software development is a huge subject
- It requires a considerable commitment
- These are exciting times, a beehive of activity
- Everywhere: US, Europe, Japan, China, India, ...
- Things are falling into place, as mathematicians and software engineers work from both ends
- *It is up to you how quickly they fall into place!*
- Great time to produce machine-proven software
  - ...and machine-proven mathematics!

Conclusion

The goals of our program are:

- To begin thinking of software in formal terms
- To deepen our understanding of logic and computation
- To know and use powerful logic tools
- To actively participate in their improvement

Software is . . . stating the right theorems and proving them