COMP 4161

Data61 Advanced Course

Advanced Topics in Software Verification

Gerwin Kein,
Johannes Åman Pohjola,
Christine Rizkallah
Miki Tanaka,

Binary Search (java.util.Arrays)


```
1:     public static int binarySearch(int[] a, int key) {
2:         int low = 0;
3:         int high = a.length - 1;
4:
5:         while (low <= high) {
6:             int mid = (low + high) / 2;
7:             int midVal = a[mid];
8:
9:             if (midVal < key)
10:                 low = mid + 1
11:             else if (midVal > key)
12:                 high = mid - 1;
13:             else
14:                 return mid; // key found
15:         }
16:         return -(low + 1);  // key not found.
17:     }
```


Binary Search (java.util.Arrays)


```
1:     public static int binarySearch(int[] a, int key) {
2:         int low = 0;
3:         int high = a.length - 1;
4:
5:         while (low <= high) {
6:             int mid = (low + high) / 2;
7:             int midVal = a[mid];
8:
9:             if (midVal < key)
10:                 low = mid + 1
11:             else if (midVal > key)
12:                 high = mid - 1;
13:             else
14:                 return mid; // key found
15:         }
16:         return -(low + 1);  // key not found.
17:     }
```

6: `int mid = (low + high) / 2;`

http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html

Organisatorials

When	Tue	11:00 – 13:00
	Wed	16:00 – 18:00

`http://www.cse.unsw.edu.au/~cs4161/`

About us

The trustworthy systems verification team

- ➜ Functional correctness and security of the seL4 microkernel
  Security↔ Isabelle/HOL model ↔ Haskell model ↔C code ↔Binary
- ➜ 10 000 LOC / 500 000 lines of proof; about 25 person years of effort
- ➜ Cogent code/proof co-generation; CakeML verified compiler; etc.

Open Source

`http://sel4.systems`

`https://ts.data61.csiro.au/projects/TS/cogent.pml`

`https://cakeml.org`

We are always embarking on exciting new projects.
We offer

- ➜ summer student scholarship projects
- ➜ honours and PhD theses
- ➜ research assistant and verification engineer positions

What you will learn

- ➜ how to use a theorem prover
- ➜ background, how it works
- ➜ how to prove and specify
- ➜ how to reason about programs

What you will learn

- ➜ how to use a theorem prover
- ➜ background, how it works
- ➜ how to prove and specify
- ➜ how to reason about programs

Health Warning

Theorem Proving is addictive

Prerequisites

This is an advanced course. It assumes knowledge in

- ➜ Functional programming
- ➜ First-order formal logic

Prerequisites

This is an advanced course. It assumes knowledge in

- ➜ Functional programming
- ➜ First-order formal logic

The following program should make sense to you:

```
map f []     =  []
map f (x:xs) =  f x : map f xs
```


Prerequisites

This is an advanced course. It assumes knowledge in

- ➜ Functional programming
- ➜ First-order formal logic

The following program should make sense to you:

```
map f []     =  []
map f (x:xs) =  f x : map f xs
```

You should be able to read and understand this formula:

$$\exists x.\ (P(x) \longrightarrow \forall x.\ P(x))$$

Content — Using Theorem Provers

Content — Using Theorem Provers

➜ Foundations & Principles
- Intro, Lambda calculus, natural deduction
- Higher Order Logic, Isar (part 1)
- Term rewriting

Content — Using Theorem Provers

- ➜ Foundations & Principles
  - Intro, Lambda calculus, natural deduction
  - Higher Order Logic, Isar (part 1)
  - Term rewriting
- ➜ Proof & Specification Techniques
  - Inductively defined sets, rule induction, datatype induction, primitive recursion
  - General recursive functions, termination proofs
  - Proof automation, Hoare logic, proofs about programs, invariants
  - C verification
  - Practice, questions, examp prep

Content — Using Theorem Provers

Rough timeline

- ➜ Foundations & Principles
  - Intro, Lambda calculus, natural deduction [1,2]
  - Higher Order Logic, Isar (part 1) [2,3[a]]
  - Term rewriting [3,4]
- ➜ Proof & Specification Techniques
  - Inductively defined sets, rule induction, datatype induction, primitive recursion [4,5]
  - General recursive functions, termination proofs [7[b]]
  - Proof automation, Hoare logic, proofs about programs, invariants [8]
  - C verification [9,10]
  - Practice, questions, examp prep [10[c]]

[a] a1 due; [b] a2 due; [c] a3 due

What you should do to have a chance at succeeding

What you should do to have a chance at succeeding

- ➜ attend lectures

What you should do to have a chance at succeeding

- ➜ attend lectures
- ➜ try Isabelle early

What you should do to have a chance at succeeding

- ➜ attend lectures
- ➜ try Isabelle early
- ➜ redo all the demos alone

What you should do to have a chance at succeeding

- ➜ attend lectures
- ➜ try Isabelle early
- ➜ redo all the demos alone
- ➜ try the exercises/homework we give, when we do give some

What you should do to have a chance at succeeding

- ➜ attend lectures
- ➜ try Isabelle early
- ➜ redo all the demos alone
- ➜ try the exercises/homework we give, when we do give some
- ➜ **DO NOT CHEAT**
  - Assignments and exams are take-home. This does NOT mean you can work in groups. Each submission is personal.
  - For more info, see Plagiarism Policy[a]

[a] `https://student.unsw.edu.au/plagiarism`

Credits

some material (in using-theorem-provers part) shamelessly stolen from



Tobias Nipkow, Larry Paulson, Markus Wenzel


David Basin, Burkhardt Wolff

Don't blame them, errors are ours

What is a formal proof?

A derivation in a formal calculus

What is a formal proof?

A derivation in a formal calculus

Example: $A \wedge B \longrightarrow B \wedge A$ derivable in the following system

Rules:

$$\frac{X \in S}{S \vdash X}\ \text{(assumption)} \qquad \frac{S \cup \{X\} \vdash Y}{S \vdash X \longrightarrow Y}\ \text{(impl)}$$

$$\frac{S \vdash X \quad S \vdash Y}{S \vdash X \wedge Y}\ \text{(conjI)} \qquad \frac{S \cup \{X, Y\} \vdash Z}{S \cup \{X \wedge Y\} \vdash Z}\ \text{(conjE)}$$

What is a formal proof?

A derivation in a formal calculus

Example: $A \wedge B \longrightarrow B \wedge A$ derivable in the following system

Rules:

$$\frac{X \in S}{S \vdash X}\ \text{(assumption)} \qquad \frac{S \cup \{X\} \vdash Y}{S \vdash X \longrightarrow Y}\ \text{(impl)}$$

$$\frac{S \vdash X \quad S \vdash Y}{S \vdash X \wedge Y}\ \text{(conjI)} \qquad \frac{S \cup \{X, Y\} \vdash Z}{S \cup \{X \wedge Y\} \vdash Z}\ \text{(conjE)}$$

Proof:

1.	$\{A, B\} \vdash B$	(by assumption)
2.	$\{A, B\} \vdash A$	(by assumption)
3.	$\{A, B\} \vdash B \wedge A$	(by conjI with 1 and 2)
4.	$\{A \wedge B\} \vdash B \wedge A$	(by conjE with 3)
5.	$\{\} \vdash A \wedge B \longrightarrow B \wedge A$	(by impl with 4)

What is a theorem prover?

Implementation of a formal logic on a computer.

- ➜ fully automated (propositional logic)
- ➜ automated, but not necessarily terminating (first order logic)
- ➜ with automation, but mainly interactive (higher order logic)

What is a theorem prover?

Implementation of a formal logic on a computer.

- ➜ fully automated (propositional logic)
- ➜ automated, but not necessarily terminating (first order logic)
- ➜ with automation, but mainly interactive (higher order logic)

- ➜ based on rules and axioms
- ➜ can deliver proofs

What is a theorem prover?

Implementation of a formal logic on a computer.

- ➜ fully automated (propositional logic)
- ➜ automated, but not necessarily terminating (first order logic)
- ➜ with automation, but mainly interactive (higher order logic)

- ➜ based on rules and axioms
- ➜ can deliver proofs

There are other (algorithmic) verification tools:

- ➜ model checking, static analysis, ...
- ➜ usually do not deliver proofs
- ➜ See COMP3153: Algorithmic Verification

Why theorem proving?

➜ Analysing systems/programs thoroughly

Why theorem proving?

- ➜ Analysing systems/programs thoroughly
- ➜ Finding design and specification errors early

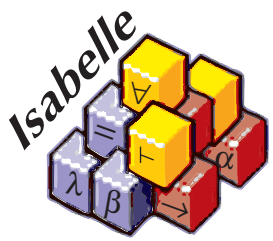
Why theorem proving?

- ➜ Analysing systems/programs thoroughly
- ➜ Finding design and specification errors early
- ➜ High assurance (mathematical, machine checked proof)

Why theorem proving?

- ➜ Analysing systems/programs thoroughly
- ➜ Finding design and specification errors early
- ➜ High assurance (mathematical, machine checked proof)
- ➜ it's not always easy
- ➜ it's fun

Main theorem proving system for this course



Isabelle

➜ used here for applications, learning how to prove

What is Isabelle?

A generic interactive proof assistant

What is Isabelle?

A generic interactive proof assistant

➜ **generic:**
not specialised to one particular logic
(two large developments: HOL and ZF, will mainly use HOL)

What is Isabelle?

A generic interactive proof assistant

- ➜ **generic:**
  not specialised to one particular logic
  (two large developments: HOL and ZF, will mainly use HOL)
- ➜ **interactive:**
  more than just yes/no, you can interactively guide the system

What is Isabelle?

A generic interactive proof assistant

- ➜ **generic:**
  not specialised to one particular logic
  (two large developments: HOL and ZF, will mainly use HOL)
- ➜ **interactive:**
  more than just yes/no, you can interactively guide the system
- ➜ **proof assistant:**
  helps to explore, find, and maintain proofs

If I prove it on the computer, it is correct, right?

If I prove it on the computer, it is correct, right?

No, because:

If I prove it on the computer, it is correct, right?

No, because:

① hardware could be faulty

If I prove it on the computer, it is correct, right?

No, because:

① hardware could be faulty
② operating system could be faulty

If I prove it on the computer, it is correct, right?

No, because:

① hardware could be faulty
② operating system could be faulty
③ implementation runtime system could be faulty

If I prove it on the computer, it is correct, right?

No, because:

① hardware could be faulty
② operating system could be faulty
③ implementation runtime system could be faulty
④ compiler could be faulty

If I prove it on the computer, it is correct, right?

No, because:

① hardware could be faulty
② operating system could be faulty
③ implementation runtime system could be faulty
④ compiler could be faulty
⑤ implementation could be faulty

If I prove it on the computer, it is correct, right?

No, because:

① hardware could be faulty
② operating system could be faulty
③ implementation runtime system could be faulty
④ compiler could be faulty
⑤ implementation could be faulty
⑥ logic could be inconsistent

If I prove it on the computer, it is correct, right?

No, because:

① hardware could be faulty
② operating system could be faulty
③ implementation runtime system could be faulty
④ compiler could be faulty
⑤ implementation could be faulty
⑥ logic could be inconsistent
⑦ theorem could mean something else

If I prove it on the computer, it is correct, right?

No, but:

If I prove it on the computer, it is correct, right?

No, but:
probability for

➜ OS and H/W issues reduced by using different systems

If I prove it on the computer, it is correct, right?

No, but:
probability for

- ➜ OS and H/W issues reduced by using different systems
- ➜ runtime/compiler bugs reduced by using different compilers

If I prove it on the computer, it is correct, right?

No, but:
probability for

- ➜ OS and H/W issues reduced by using different systems
- ➜ runtime/compiler bugs reduced by using different compilers
- ➜ faulty implementation reduced by having the right prover architecture

If I prove it on the computer, it is correct, right?

No, but:
probability for

- ➜ OS and H/W issues reduced by using different systems
- ➜ runtime/compiler bugs reduced by using different compilers
- ➜ faulty implementation reduced by having the right prover architecture
- ➜ inconsistent logic reduced by implementing and analysing it

If I prove it on the computer, it is correct, right?

No, but:
probability for

- ➜ OS and H/W issues reduced by using different systems
- ➜ runtime/compiler bugs reduced by using different compilers
- ➜ faulty implementation reduced by having the right prover architecture
- ➜ inconsistent logic reduced by implementing and analysing it
- ➜ wrong theorem reduced by expressive/intuitive logics

If I prove it on the computer, it is correct, right?

No, but:
probability for

- ➜ OS and H/W issues reduced by using different systems
- ➜ runtime/compiler bugs reduced by using different compilers
- ➜ faulty implementation reduced by having the right prover architecture
- ➜ inconsistent logic reduced by implementing and analysing it
- ➜ wrong theorem reduced by expressive/intuitive logics

No guarantees, but assurance immensly higher than manual proof

If I prove it on the computer, it is correct, right?

Soundness architectures
careful implementation PVS

If I prove it on the computer, it is correct, right?

Soundness architectures

careful implementation	PVS
LCF approach, small proof kernel	HOL4 Isabelle

If I prove it on the computer, it is correct, right?

Soundness architectures

careful implementation	PVS
LCF approach, small proof kernel	HOL4 Isabelle
explicit proofs + proof checker	Coq Twelf Isabelle HOL4

Meta Logic

Meta language:
The language used to talk about another language.

Meta Logic

Meta language:
The language used to talk about another language.

Examples:
English in a Spanish class, English in an English class

Meta Logic

Meta language:
The language used to talk about another language.

Examples:
English in a Spanish class, English in an English class

Meta logic:
The logic used to formalize another logic

Example:
Mathematics used to formalize derivations in formal logic

Meta Logic – Example

Syntax:

Formulae: $F ::= V \mid F \longrightarrow F \mid F \wedge F \mid \mathit{False}$

$V ::= [A - Z]$

Derivable: $S \vdash X$ X a formula, S a set of formulae

Meta Logic – Example

Syntax:

Formulae: $F ::= V \mid F \longrightarrow F \mid F \wedge F \mid \mathit{False}$

$V ::= [A - Z]$

Derivable: $S \vdash X$ X a formula, S a set of formulae

logic / meta logic

$$\frac{X \in S}{S \vdash X}$$

$$\frac{S \cup \{X\} \vdash Y}{S \vdash X \longrightarrow Y} \text{ ========}$$

$$\frac{S \vdash X \quad S \vdash Y}{S \vdash X \wedge Y} \text{ ¿¿¿¿¿¿ other}$$

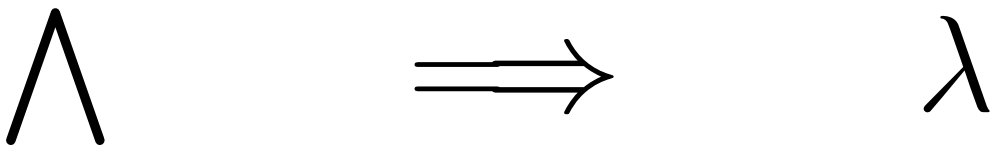
$$\frac{S \cup \{X, Y\} \vdash Z}{S \cup \{X \wedge Y\} \vdash Z} \text{ ¡¡¡¡¡¡¡ local}$$

$$\frac{S \cup \{X$$

¡¡¡¡¡¡¡ local $S \vdash X$

========

Isabelle's Meta Logic



$\bigwedge$

Syntax: $\bigwedge x.\ F$ (F another meta level formula)

in ASCII: `!!x. F`

Syntax: $\bigwedge x.\ F$ (F another meta level formula)
in ASCII: `!!x. F`

- ➜ universal quantifier on the meta level
- ➜ used to denote parameters
- ➜ example and more later

$\Longrightarrow$

Syntax: $A \Longrightarrow B$ (A, B other meta level formulae)
in ASCII: `A ==> B`

$\Longrightarrow$

Syntax: $A \Longrightarrow B$ (A, B other meta level formulae)
in ASCII: `A ==> B`

Binds to the right:

$$A \Longrightarrow B \Longrightarrow C \quad = \quad A \Longrightarrow (B \Longrightarrow C)$$

Abbreviation:

$$[\![A; B]\!] \Longrightarrow C \quad = \quad A \Longrightarrow B \Longrightarrow C$$

- ➜ read: A and B implies C
- ➜ used to write down rules, theorems, and proof states

Example: a theorem

mathematics: if $x < 0$ and $y < 0$, then $x + y < 0$

Example: a theorem

mathematics: if $x < 0$ and $y < 0$, then $x + y < 0$

formal logic: $\vdash\ x < 0 \wedge y < 0 \longrightarrow x + y < 0$

variation: $x < 0; y < 0\ \vdash\ x + y < 0$

Example: a theorem

mathematics: if $x < 0$ and $y < 0$, then $x + y < 0$

formal logic: $\vdash\ x < 0 \wedge y < 0 \longrightarrow x + y < 0$
variation: $x < 0; y < 0 \vdash\ x + y < 0$

Isabelle: **lemma** "$x < 0 \wedge y < 0 \longrightarrow x + y < 0$"
variation: **lemma** "$[\![x < 0; y < 0]\!] \Longrightarrow x + y < 0$"

Example: a theorem

mathematics: if $x < 0$ and $y < 0$, then $x + y < 0$

formal logic: $\vdash\ x < 0 \wedge y < 0 \longrightarrow x + y < 0$
variation: $x < 0; y < 0 \vdash\ x + y < 0$

Isabelle: **lemma** "$x < 0 \wedge y < 0 \longrightarrow x + y < 0$"
variation: **lemma** "$[\![x < 0; y < 0]\!] \Longrightarrow x + y < 0$"
variation: **lemma**
assumes "$x < 0$" and "$y < 0$" shows "$x + y < 0$"

Example: a rule

logic:

$$\frac{X \quad Y}{X \wedge Y}$$

Example: a rule

logic:
$$\frac{X \quad Y}{X \wedge Y}$$

variation:
$$\frac{S \vdash X \quad S \vdash Y}{S \vdash X \wedge Y}$$

Example: a rule

logic: $\dfrac{X \quad Y}{X \wedge Y}$

variation: $\dfrac{S \vdash X \quad S \vdash Y}{S \vdash X \wedge Y}$

Isabelle: $[\![X; Y]\!] \Longrightarrow X \wedge Y$

Example: a rule with nested implication

logic:

$$\frac{X \vee Y \qquad \begin{array}{c}[X]\\ \vdots \\ Z\end{array} \qquad \begin{array}{c}[Y]\\ \vdots \\ Z\end{array}}{Z}$$

Example: a rule with nested implication

logic:

$$\dfrac{X \vee Y \quad \begin{array}{c}[X]\\ \vdots \\ Z\end{array} \quad \begin{array}{c}[Y]\\ \vdots \\ Z\end{array}}{Z}$$

variation:

$$\dfrac{S \cup \{X\} \vdash Z \quad S \cup \{Y\} \vdash Z}{S \cup \{X \vee Y\} \vdash Z}$$

Example: a rule with nested implication

logic:

$$\frac{X \vee Y \quad \begin{array}{c}[X]\\ \vdots \\ Z\end{array} \quad \begin{array}{c}[Y]\\ \vdots \\ Z\end{array}}{Z}$$

variation:

$$\frac{S \cup \{X\} \vdash Z \quad S \cup \{Y\} \vdash Z}{S \cup \{X \vee Y\} \vdash Z}$$

Isabelle:

$$[\![X \vee Y; X \Longrightarrow Z; Y \Longrightarrow Z]\!] \Longrightarrow Z$$

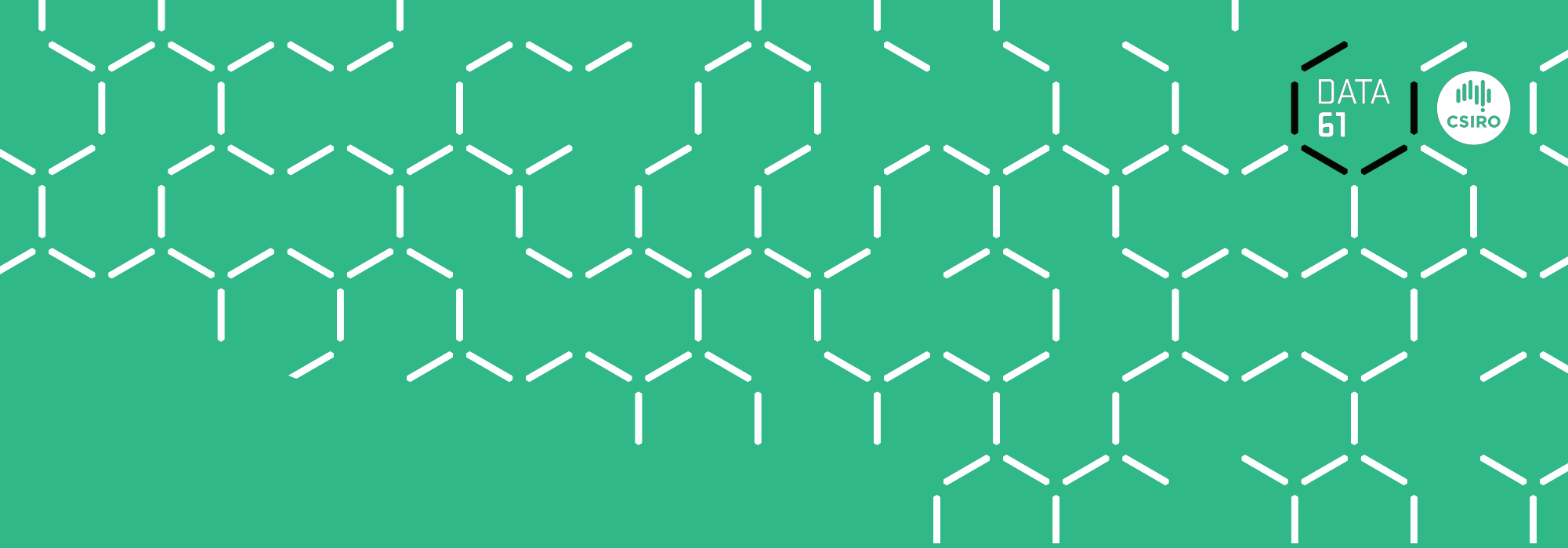
λ

Syntax: $\lambda x.\ F$ (F another meta level formula)
in ASCII: `%x. F`

λ

Syntax: $\lambda x.\ F$ (F another meta level formula)
in ASCII: `%x. F`

- ➜ lambda abstraction
- ➜ used for functions in object logics
- ➜ used to encode bound variables in object logics
- ➜ more about this in the next lecture



Enough Theory!

Getting started with Isabelle

System Architecture

Isabelle – generic, interactive theorem prover

System Architecture

Isabelle – generic, interactive theorem prover

Standard ML – logic implemented as ADT

System Architecture

HOL, ZF – object-logics

Isabelle – generic, interactive theorem prover

Standard ML – logic implemented as ADT

System Architecture

Prover IDE (jEdit) – user interface

HOL, ZF – object-logics

Isabelle – generic, interactive theorem prover

Standard ML – logic implemented as ADT

System Architecture

Prover IDE (jEdit) – user interface

HOL, ZF – object-logics

Isabelle – generic, interactive theorem prover

Standard ML – logic implemented as ADT

User can access all layers!

System Requirements

- **Linux**, **Windows**, or **MacOS X (10.8 +)**
- **Standard ML** (PolyML implementation)
- **Java** (for jEdit)

Premade packages for Linux, Mac, and Windows + info on:
`http://mirror.cse.unsw.edu.au/pub/isabelle/`

Documentation

Available from `http://isabelle.in.tum.de`

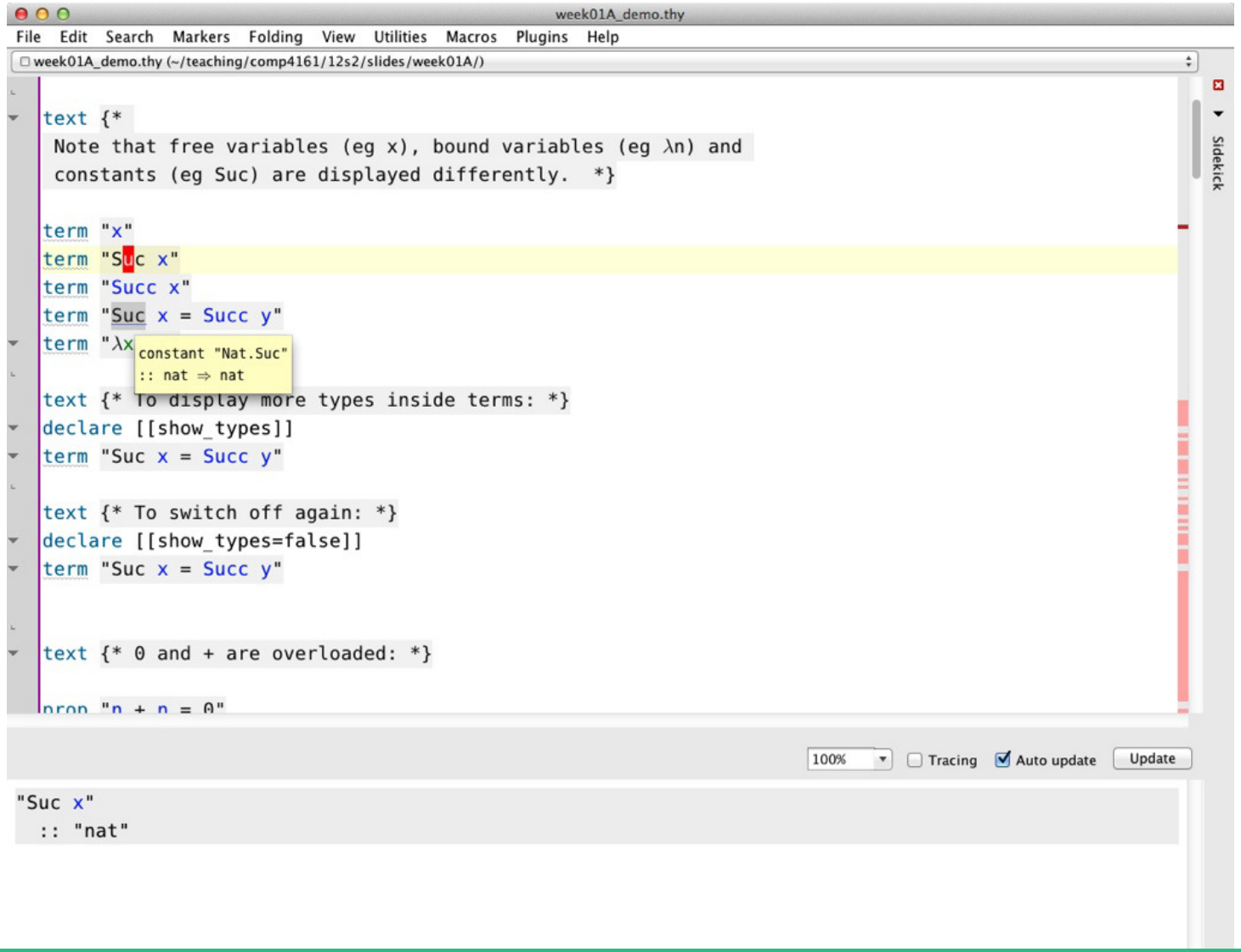
➜ Learning Isabelle
- Concrete Semantics Book
- Tutorial on Isabelle/HOL (LNCS 2283)
- Tutorial on Isar
- Tutorial on Locales

➜ Reference Manuals
- Isabelle/Isar Reference Manual
- Isabelle Reference Manual
- Isabelle System Manual

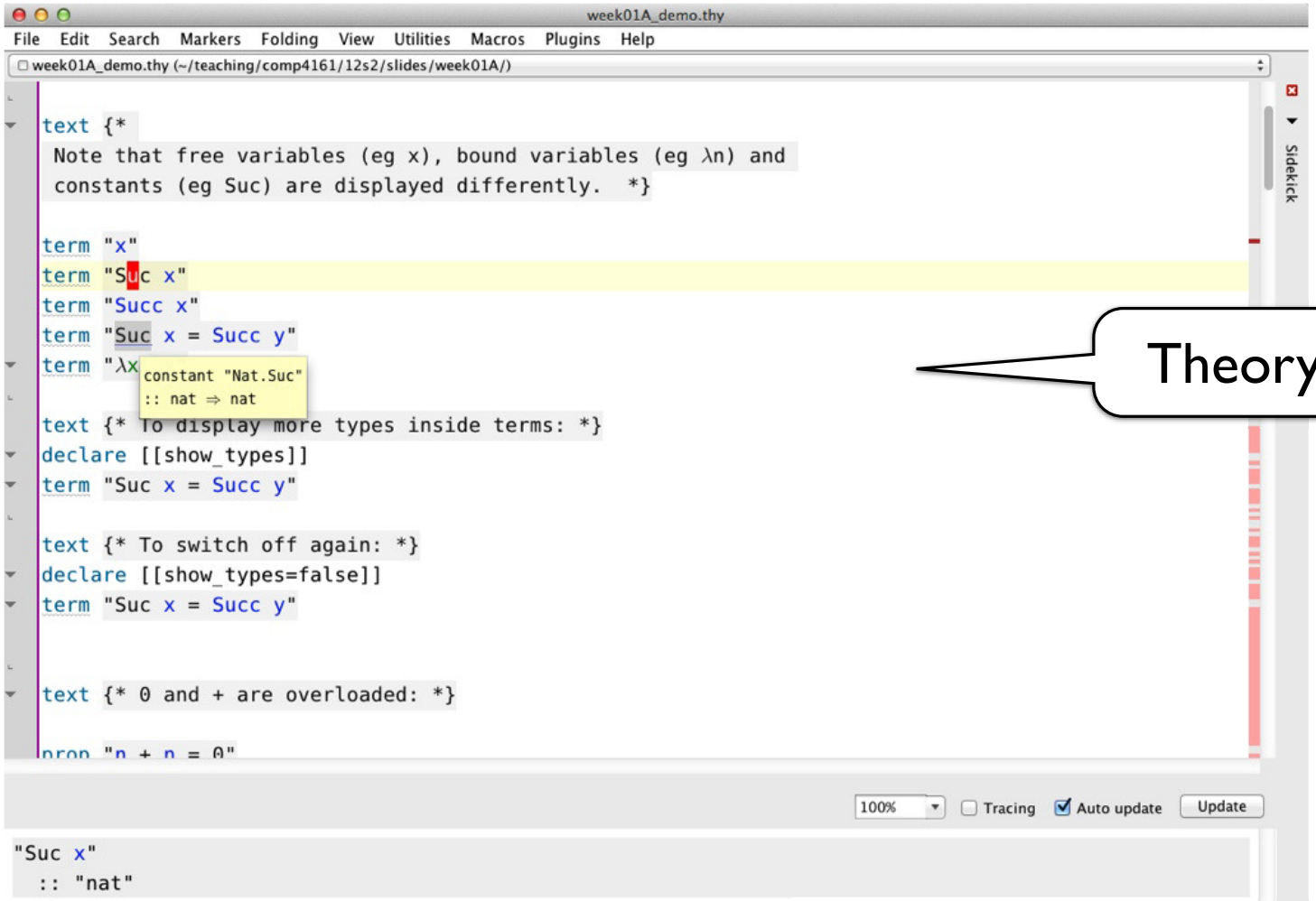
➜ Reference Manuals for Object-Logics

Demo

jEdit/PIDE



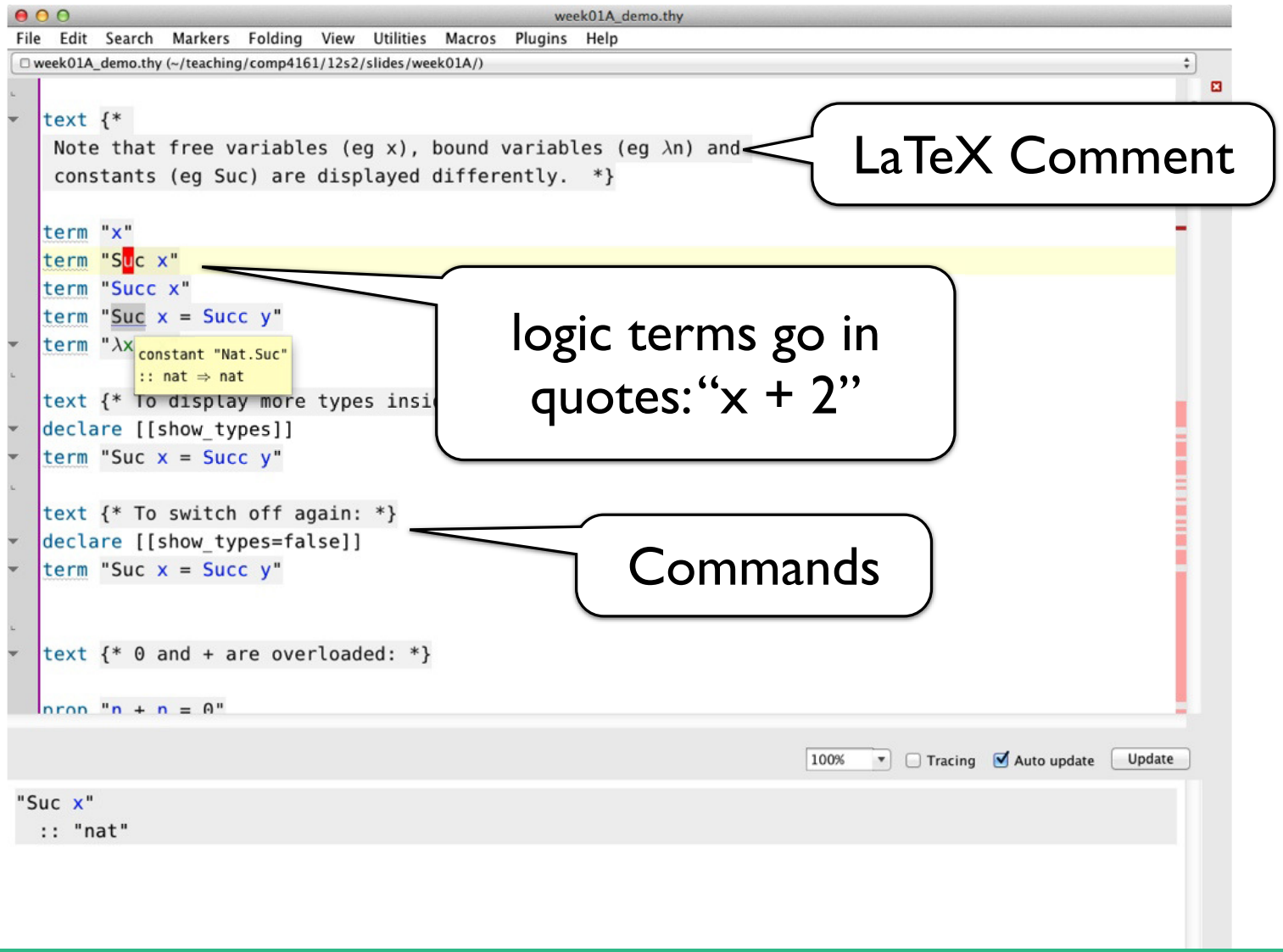
jEdit/PIDE



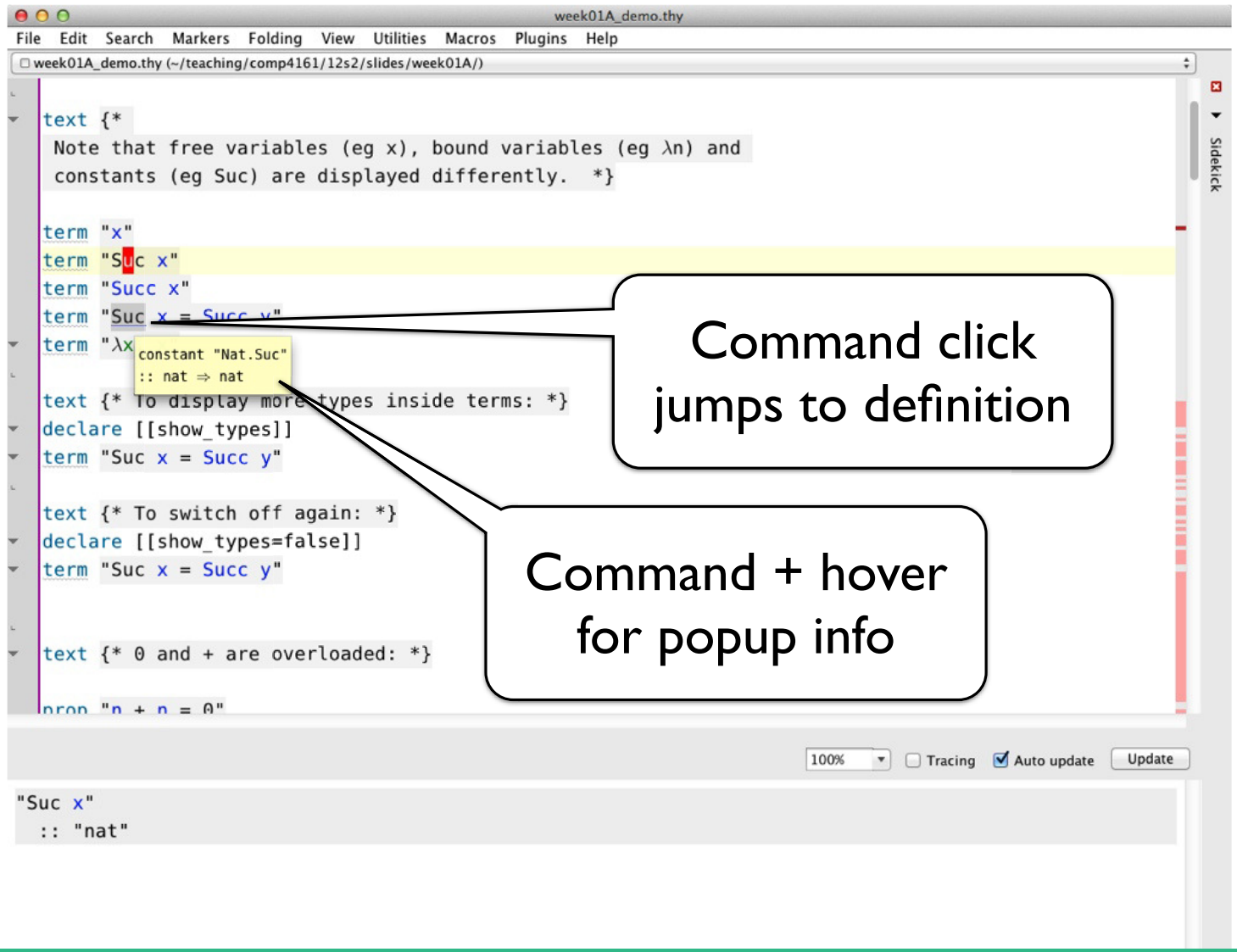
Theory File

Isabelle Output

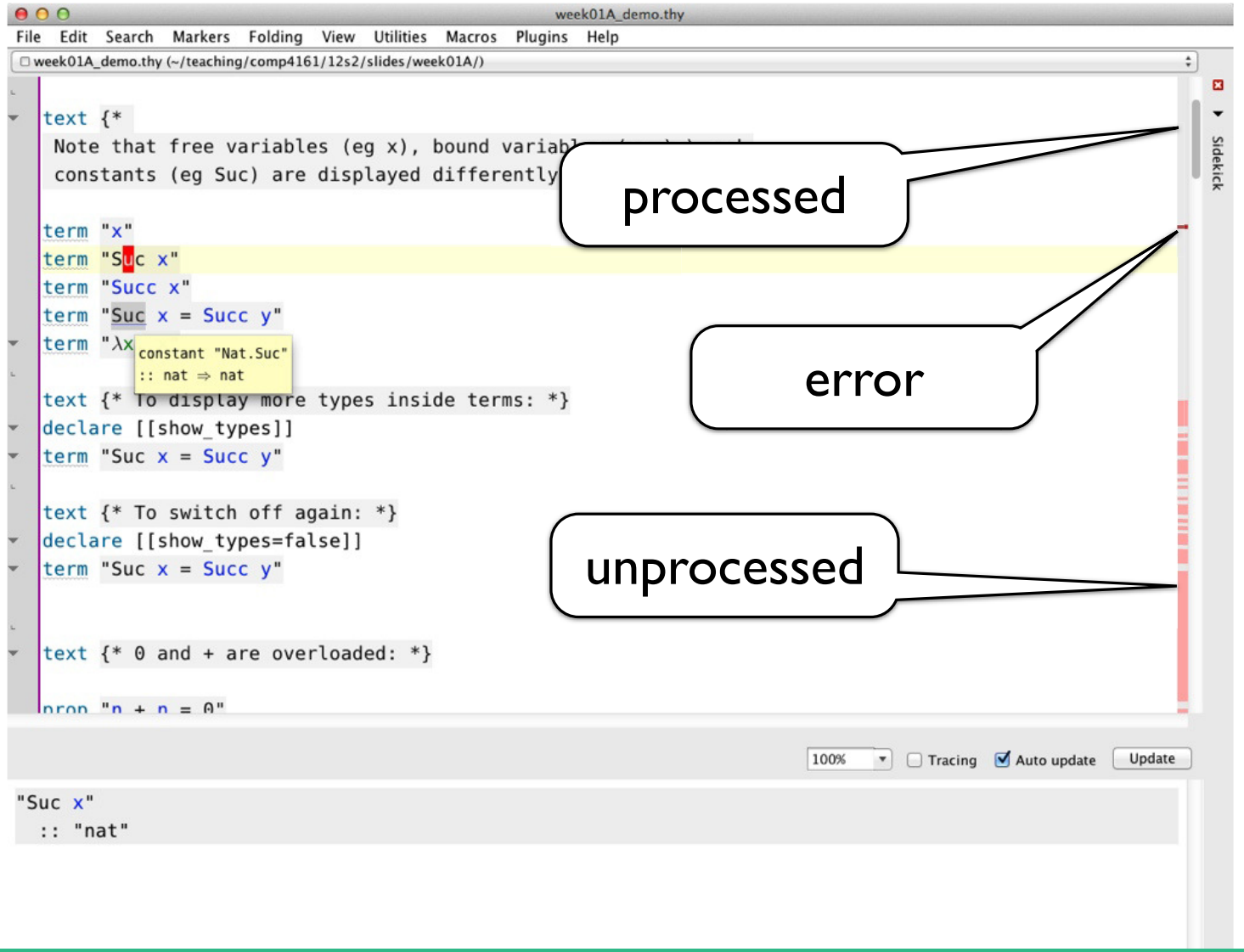
jEdit/PIDE



jEdit/PIDE

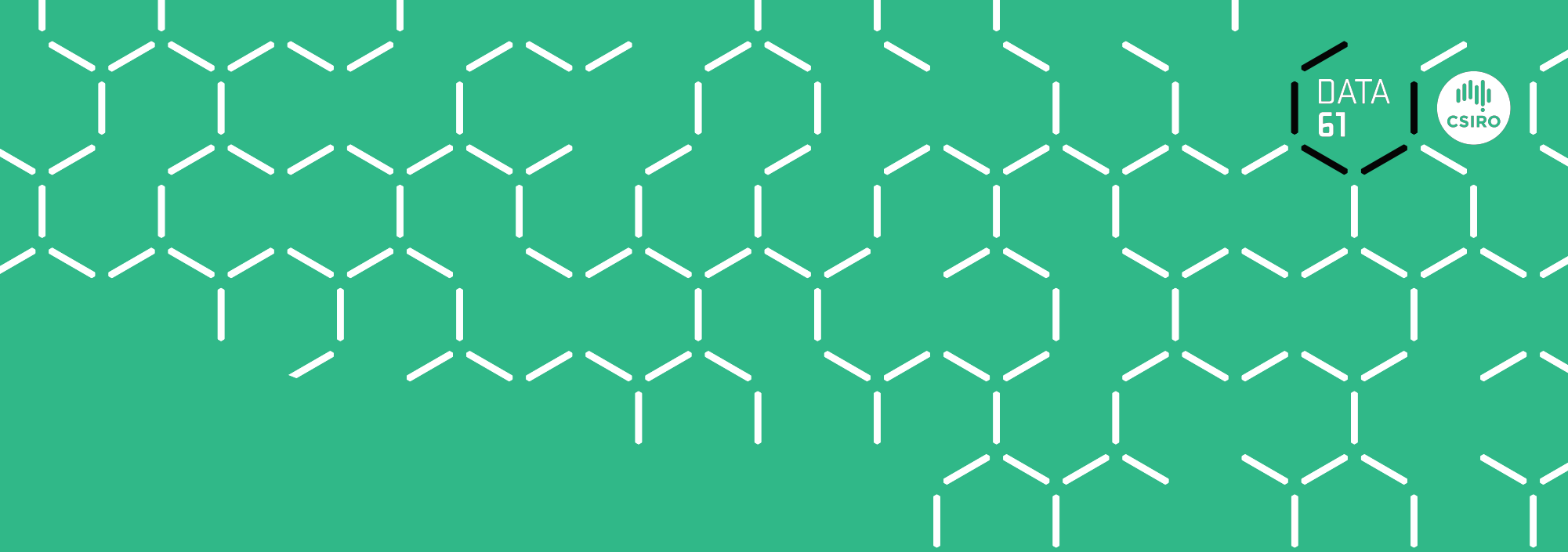


jEdit/PIDE



Exercises

- Download and install Isabelle from `http://mirror.cse.unsw.edu.au/pub/isabelle/`
- Step through the demo files from the lecture web page
- Write your own theory file, look at some theorems in the library, try 'find_theorems'
- How many theorems can help you if you need to prove something containing the term "Suc(Suc x)"?
- What is the name of the theorem for associativity of addition of natural numbers in the library?



λ-Calculus

Content

➜ Foundations & Principles

- Intro, Lambda calculus, natural deduction [1,2]
- Higher Order Logic, Isar (part 1) [2,3[a]]
- Term rewriting [3,4]

➜ Proof & Specification Techniques

- Inductively defined sets, rule induction, datatype induction, primitive recursion [4,5]
- General recursive functions, termination proofs [7[b]]
- Proof automation, Hoare logic, proofs about programs, invariants [8]
- C verification [9,10]
- Practice, questions, examp prep [10[c]]

---

[a] a1 due; [b] a2 due; [c] a3 due

λ-calculus

Alonzo Church

- ➜ lived 1903–1995
- ➜ supervised people like Alan Turing, Stephen Kleene
- ➜ famous for Church-Turing thesis, lambda calculus, first undecidability results
- ➜ invented λ calculus in 1930's

λ-calculus

Alonzo Church

- ➜ lived 1903–1995
- ➜ supervised people like Alan Turing, Stephen Kleene
- ➜ famous for Church-Turing thesis, lambda calculus, first undecidability results
- ➜ invented λ calculus in 1930's

λ-calculus

- ➜ originally meant as foundation of mathematics
- ➜ important applications in theoretical computer science
- ➜ foundation of computability and functional programming

untyped λ-calculus

- ➜ turing complete model of computation
- ➜ a simple way of writing down functions

untyped λ-calculus

- ➜ turing complete model of computation
- ➜ a simple way of writing down functions

Basic intuition:

instead of	$f(x) = x + 5$
write	$f = \lambda x.\ x + 5$

untyped λ-calculus

- ➜ turing complete model of computation
- ➜ a simple way of writing down functions

Basic intuition:

instead of	$f(x) = x + 5$
write	$f = \lambda x.\ x + 5$

$\lambda x.\ x + 5$

- ➜ a term

untyped λ-calculus

- ➜ turing complete model of computation
- ➜ a simple way of writing down functions

Basic intuition:

instead of	$f(x) = x + 5$
write	$f = \lambda x.\ x + 5$

$\lambda x.\ x + 5$

- ➜ a term
- ➜ a nameless function

untyped λ-calculus

- ➜ turing complete model of computation
- ➜ a simple way of writing down functions

Basic intuition:

instead of	$f(x) = x + 5$
write	$f = \lambda x.\ x + 5$

$\lambda x.\ x + 5$

- ➜ a term
- ➜ a nameless function
- ➜ that adds 5 to its parameter

Function Application

For applying arguments to functions

instead of	$f(a)$
write	$f\ a$

Function Application

For applying arguments to functions

instead of	$f(a)$
write	$f\ a$

Example: $(\lambda x.\ x + 5)\ a$

Function Application

For applying arguments to functions

instead of	$f(a)$
write	$f\ a$

Example: $(\lambda x.\ x + 5)\ a$

Evaluating: in $(\lambda x.\ t)\ a$ replace x by a in t
(computation!)

Function Application

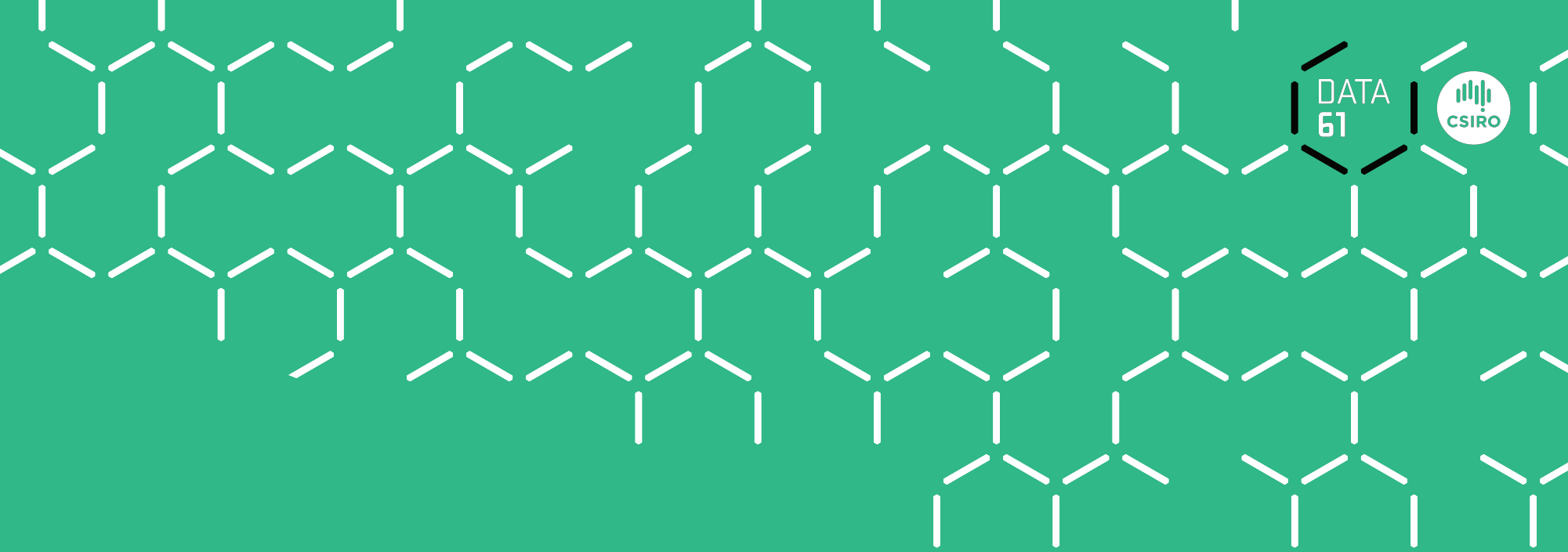
For applying arguments to functions

instead of	$f(a)$
write	$f\ a$

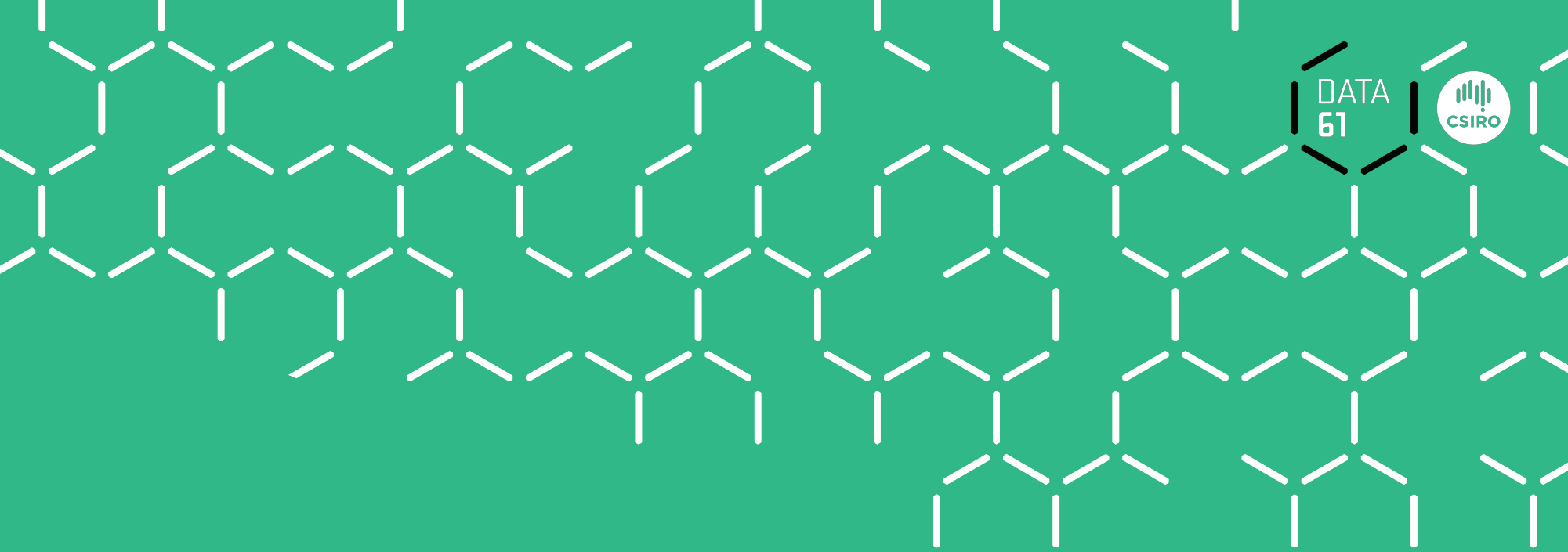
Example: $(\lambda x.\ x + 5)\ a$

Evaluating: in $(\lambda x.\ t)\ a$ replace x by a in t
(computation!)

Example: $(\lambda x.\ x + 5)\ (a + b)$ evaluates to $(a + b) + 5$



That's it!



Now Formal

Syntax

Terms: $t ::= v \mid c \mid (t\ t) \mid (\lambda x.\ t)$

$v, x \in V, \quad c \in C, \quad V, C$ sets of names

Syntax

Terms: $t \quad ::= \quad v \mid c \mid (t\ t) \mid (\lambda x.\ t)$

$v, x \in V, \quad c \in C, \quad V, C$ sets of names

- v, x variables
- c constants
- $(t\ t)$ application
- $(\lambda x.\ t)$ abstraction

Conventions

- ➜ leave out parentheses where possible
- ➜ list variables instead of multiple λ

Example: instead of $(\lambda y.\ (\lambda x.\ (x\ y)))$ write $\lambda y\ x.\ x\ y$

Conventions

- ➜ leave out parentheses where possible
- ➜ list variables instead of multiple λ

Example: instead of $(\lambda y.\ (\lambda x.\ (x\ y)))$ write $\lambda y\ x.\ x\ y$

Rules:

- ➜ list variables: $\lambda x.\ (\lambda y.\ t) = \lambda x\ y.\ t$
- ➜ application binds to the left: $x\ y\ z = (x\ y)\ z \neq x\ (y\ z)$
- ➜ abstraction binds to the right: $\lambda x.\ x\ y = \lambda x.\ (x\ y) \neq (\lambda x.\ x)\ y$
- ➜ leave out outermost parentheses

Getting used to the Syntax

Example:

$\lambda x\ y\ z.\ x\ z\ (y\ z) =$

Getting used to the Syntax

Example:

$\lambda x\ y\ z.\ x\ z\ (y\ z) =$

$\lambda x\ y\ z.\ (x\ z)\ (y\ z) =$

Getting used to the Syntax

Example:

$\lambda x\ y\ z.\ x\ z\ (y\ z) =$

$\lambda x\ y\ z.\ (x\ z)\ (y\ z) =$

$\lambda x\ y\ z.\ ((x\ z)\ (y\ z)) =$

Getting used to the Syntax

Example:

$\lambda x\ y\ z.\ x\ z\ (y\ z) =$

$\lambda x\ y\ z.\ (x\ z)\ (y\ z) =$

$\lambda x\ y\ z.\ ((x\ z)\ (y\ z)) =$

$\lambda x.\ \lambda y.\ \lambda z.\ ((x\ z)\ (y\ z)) =$

Getting used to the Syntax

Example:

$\lambda x\ y\ z.\ x\ z\ (y\ z) =$

$\lambda x\ y\ z.\ (x\ z)\ (y\ z) =$

$\lambda x\ y\ z.\ ((x\ z)\ (y\ z)) =$

$\lambda x.\ \lambda y.\ \lambda z.\ ((x\ z)\ (y\ z)) =$

$(\lambda x.\ (\lambda y.\ (\lambda z.\ ((x\ z)\ (y\ z)))))$

Computation

Intuition: replace parameter by argument
this is called β-reduction

Example

$$(\lambda x\ y.\ f\ (y\ x))\ \ 5\ \ (\lambda x.\ x) \longrightarrow_\beta$$

Computation

Intuition: replace parameter by argument
this is called β-reduction

Example

$$(\lambda x\ y.\ f\ (y\ x))\ \ 5\ \ (\lambda x.\ x) \longrightarrow_\beta$$
$$(\lambda y.\ f\ (y\ 5))\ \ (\lambda x.\ x) \longrightarrow_\beta$$

Computation

Intuition: replace parameter by argument
this is called β-reduction

Example

$$(\lambda x\ y.\ f\ (y\ x))\ \ 5\ \ (\lambda x.\ x) \longrightarrow_\beta$$
$$(\lambda y.\ f\ (y\ 5))\ \ (\lambda x.\ x) \longrightarrow_\beta$$
$$f\ ((\lambda x.\ x)\ 5) \longrightarrow_\beta$$

Computation

Intuition: replace parameter by argument
this is called β-reduction

Example

$$(\lambda x\ y.\ f\ (y\ x))\ \ 5\ \ (\lambda x.\ x) \longrightarrow_\beta$$
$$(\lambda y.\ f\ (y\ 5))\ \ (\lambda x.\ x) \longrightarrow_\beta$$
$$f\ ((\lambda x.\ x)\ 5) \longrightarrow_\beta$$
$$f\ 5$$

Defining Computation

β reduction:

$$
\begin{array}{rcl}
(\lambda x.\ s)\ t & \longrightarrow_\beta & s[x \leftarrow t] \\
s \longrightarrow_\beta s' \implies (s\ t) & \longrightarrow_\beta & (s'\ t) \\
t \longrightarrow_\beta t' \implies (s\ t) & \longrightarrow_\beta & (s\ t') \\
s \longrightarrow_\beta s' \implies (\lambda x.\ s) & \longrightarrow_\beta & (\lambda x.\ s')
\end{array}
$$

Defining Computation

β **reduction:**

$$
\begin{array}{lllll}
 & & & (\lambda x.\ s)\ t & \longrightarrow_\beta\ s[x \leftarrow t] \\
s \longrightarrow_\beta s' & \Longrightarrow & (s\ t) & \longrightarrow_\beta & (s'\ t) \\
t \longrightarrow_\beta t' & \Longrightarrow & (s\ t) & \longrightarrow_\beta & (s\ t') \\
s \longrightarrow_\beta s' & \Longrightarrow & (\lambda x.\ s) & \longrightarrow_\beta & (\lambda x.\ s')
\end{array}
$$

Still to do: define $s[x \leftarrow t]$

Defining Substitution

Easy concept. Small problem: variable capture.

Example: $(\lambda x.\ x\ z)[z \leftarrow x]$

Defining Substitution

Easy concept. Small problem: variable capture.
Example: $(\lambda x.\ x\ z)[z \leftarrow x]$

We do **not** want: $(\lambda x.\ x\ x)$ as result.

What do we want?

Defining Substitution

Easy concept. Small problem: variable capture.
Example: $(\lambda x.\ x\ z)[z \leftarrow x]$

We do **not** want: $(\lambda x.\ x\ x)$ as result.

What do we want?

In $(\lambda y.\ y\ z)\ [z \leftarrow x] = (\lambda y.\ y\ x)$ there would be no problem.

So, solution is: rename bound variables.

Free Variables

Bound variables: in $(\lambda x.\ t)$, x is a bound variable.

Free Variables

Bound variables: in $(\lambda x.\ t)$, x is a bound variable.

Free variables FV of a term:

$$
\begin{aligned}
FV\ (x) &= \{x\} \\
FV\ (c) &= \{\} \\
FV\ (s\ t) &= FV(s) \cup FV(t) \\
FV\ (\lambda x.\ t) &= FV(t) \setminus \{x\}
\end{aligned}
$$

Example: $FV(\quad \lambda x.\ (\lambda y.\ (\lambda x.\ x)\ y)\ y\ x \quad)$

Free Variables

Bound variables: in $(\lambda x.\ t)$, x is a bound variable.

Free variables FV of a term:

$$
\begin{aligned}
FV\ (x) &= \{x\} \\
FV\ (c) &= \{\} \\
FV\ (s\ t) &= FV(s) \cup FV(t) \\
FV\ (\lambda x.\ t) &= FV(t) \setminus \{x\}
\end{aligned}
$$

Example: $FV(\quad \lambda x.\ (\lambda y.\ (\lambda x.\ x)\ y)\ y\ x \quad) = \{y\}$

Free Variables

Bound variables: in $(\lambda x.\ t)$, x is a bound variable.

Free variables FV of a term:

$$\begin{aligned} FV\ (x) &= \{x\} \\ FV\ (c) &= \{\} \\ FV\ (s\ t) &= FV(s) \cup FV(t) \\ FV\ (\lambda x.\ t) &= FV(t) \setminus \{x\} \end{aligned}$$

Example: $FV(\quad \lambda x.\ (\lambda y.\ (\lambda x.\ x)\ y)\ y\ x \quad) = \{y\}$

Term t is called **closed** if $FV(t) = \{\}$

Free Variables

Bound variables: in $(\lambda x.\ t)$, x is a bound variable.

Free variables FV of a term:

$$\begin{aligned} FV\ (x) &= \{x\} \\ FV\ (c) &= \{\} \\ FV\ (s\ t) &= FV(s) \cup FV(t) \\ FV\ (\lambda x.\ t) &= FV(t) \setminus \{x\} \end{aligned}$$

Example: $FV(\quad \lambda x.\ (\lambda y.\ (\lambda x.\ x)\ y)\ y\ x \quad) = \{y\}$

Term t is called **closed** if $FV(t) = \{\}$

The substitution example, $(\lambda x.\ x\ z)[z \leftarrow x]$, is problematic because the bound variable x is a free variable of the replacement term "x".

Substitution

$$
\begin{aligned}
x\ [x \leftarrow t] &= t \\
y\ [x \leftarrow t] &= y \qquad \text{if } x \neq y \\
c\ [x \leftarrow t] &= c
\end{aligned}
$$

$$
(s_1\ s_2)\ [x \leftarrow t] =
$$

Substitution

$$
\begin{aligned}
x\ [x \leftarrow t] &= t \\
y\ [x \leftarrow t] &= y \qquad \text{if } x \neq y \\
c\ [x \leftarrow t] &= c
\end{aligned}
$$

$$(s_1\ s_2)\ [x \leftarrow t] = (s_1[x \leftarrow t]\ s_2[x \leftarrow t])$$

$$(\lambda x.\ s)\ [x \leftarrow t] =$$

Substitution

$$
\begin{aligned}
x\ [x \leftarrow t] &= t \\
y\ [x \leftarrow t] &= y \qquad \text{if } x \neq y \\
c\ [x \leftarrow t] &= c
\end{aligned}
$$

$$
(s_1\ s_2)\ [x \leftarrow t] = (s_1[x \leftarrow t]\ s_2[x \leftarrow t])
$$

$$
\begin{aligned}
(\lambda x.\ s)\ [x \leftarrow t] &= (\lambda x.\ s) \\
(\lambda y.\ s)\ [x \leftarrow t] &=
\end{aligned}
$$

Substitution

$$
\begin{array}{lll}
x\ [x \leftarrow t] & = t & \\
y\ [x \leftarrow t] & = y & \text{if } x \neq y \\
c\ [x \leftarrow t] & = c & \\
 & & \\
(s_1\ s_2)\ [x \leftarrow t] & = (s_1[x \leftarrow t]\ s_2[x \leftarrow t]) & \\
 & & \\
(\lambda x.\ s)\ [x \leftarrow t] & = (\lambda x.\ s) & \\
(\lambda y.\ s)\ [x \leftarrow t] & = (\lambda y.\ s[x \leftarrow t]) & \text{if } x \neq y \text{ and } y \notin FV(t) \\
(\lambda y.\ s)\ [x \leftarrow t] & = &
\end{array}
$$

Substitution

$$
\begin{array}{lll}
x\ [x \leftarrow t] & = t & \\
y\ [x \leftarrow t] & = y & \text{if } x \neq y \\
c\ [x \leftarrow t] & = c & \\
 & & \\
(s_1\ s_2)\ [x \leftarrow t] & = (s_1[x \leftarrow t]\ s_2[x \leftarrow t]) & \\
 & & \\
(\lambda x.\ s)\ [x \leftarrow t] & = (\lambda x.\ s) & \\
(\lambda y.\ s)\ [x \leftarrow t] & = (\lambda y.\ s[x \leftarrow t]) & \text{if } x \neq y \text{ and } y \notin FV(t) \\
(\lambda y.\ s)\ [x \leftarrow t] & = (\lambda z.\ s[y \leftarrow z][x \leftarrow t]) & \text{if } x \neq y \\
 & & \text{and } z \notin FV(t) \cup FV(s)
\end{array}
$$

Substitution Example

$$(x \;\; (\lambda x.\; x) \;\; (\lambda y.\; z\; x))[x \leftarrow y]$$

Substitution Example

$$
\begin{array}{rl}
 & (x \;\; (\lambda x.\; x) \;\; (\lambda y.\; z\; x))[x \leftarrow y] \\
= & (x[x \leftarrow y]) \;\; ((\lambda x.\; x)[x \leftarrow y]) \;\; ((\lambda y.\; z\; x)[x \leftarrow y])
\end{array}
$$

Substitution Example

$$
\begin{array}{rl}
 & (x\ \ (\lambda x.\ x)\ \ (\lambda y.\ z\ x))[x \leftarrow y] \\
= & (x[x \leftarrow y])\ \ ((\lambda x.\ x)[x \leftarrow y])\ \ ((\lambda y.\ z\ x)[x \leftarrow y]) \\
= & y\ (\lambda x.\ x)\ (\lambda y'.\ z\ y)
\end{array}
$$

α Conversion

Bound names are irrelevant:
$\lambda x.\ x$ and $\lambda y.\ y$ denote the same function.

α conversion:
$s =_\alpha t$ means $s = t$ up to renaming of bound variables.

α Conversion

Bound names are irrelevant:

$\lambda x.\ x$ and $\lambda y.\ y$ denote the same function.

α conversion:

$s =_\alpha t$ means $s = t$ up to renaming of bound variables.

Formally:

$$
\begin{array}{rcllcll}
 & & & & (\lambda x.\ t) & \longrightarrow_\alpha & (\lambda y.\ t[x \leftarrow y]) \text{ if } y \notin FV(t) \\
s & \longrightarrow_\alpha & s' & \Longrightarrow & (s\ t) & \longrightarrow_\alpha & (s'\ t) \\
t & \longrightarrow_\alpha & t' & \Longrightarrow & (s\ t) & \longrightarrow_\alpha & (s\ t') \\
s & \longrightarrow_\alpha & s' & \Longrightarrow & (\lambda x.\ s) & \longrightarrow_\alpha & (\lambda x.\ s')
\end{array}
$$

α Conversion

Bound names are irrelevant:
$\lambda x.\ x$ and $\lambda y.\ y$ denote the same function.

α conversion:
$s =_\alpha t$ means $s = t$ up to renaming of bound variables.

Formally:

$$
\begin{array}{lllllll}
 & & & & (\lambda x.\ t) & \longrightarrow_\alpha & (\lambda y.\ t[x \leftarrow y]) \ \text{ if } y \notin FV(t) \\
s & \longrightarrow_\alpha & s' & \Longrightarrow & (s\ t) & \longrightarrow_\alpha & (s'\ t) \\
t & \longrightarrow_\alpha & t' & \Longrightarrow & (s\ t) & \longrightarrow_\alpha & (s\ t') \\
s & \longrightarrow_\alpha & s' & \Longrightarrow & (\lambda x.\ s) & \longrightarrow_\alpha & (\lambda x.\ s')
\end{array}
$$

$$s =_\alpha t \quad \text{iff} \quad s \longrightarrow_\alpha^* t$$

$(\longrightarrow_\alpha^* = \text{transitive, reflexive closure of } \longrightarrow_\alpha = \text{multiple steps})$

α Conversion

Equality in Isabelle is equality modulo α conversion:

if $s =_\alpha t$ then s and t are syntactically equal.

Examples:

$$x\ (\lambda x\ y.\ x\ y)$$

α Conversion

Equality in Isabelle is equality modulo α conversion:

if $s =_\alpha t$ then s and t are syntactically equal.

Examples:

$$\begin{array}{rl} & x\ (\lambda x\ y.\ x\ y) \\ =_\alpha & x\ (\lambda y\ x.\ y\ x) \end{array}$$

α Conversion

Equality in Isabelle is equality modulo α conversion:

if $s =_\alpha t$ then s and t are syntactically equal.

Examples:

$$\begin{aligned} & x\ (\lambda x\ y.\ x\ y) \\ =_\alpha\ & x\ (\lambda y\ x.\ y\ x) \\ =_\alpha\ & x\ (\lambda z\ y.\ z\ y) \end{aligned}$$

α Conversion

Equality in Isabelle is equality modulo α conversion:

if $s =_\alpha t$ then s and t are syntactically equal.

Examples:

$$\begin{array}{ll} & x\ (\lambda x\ y.\ x\ y) \\ =_\alpha & x\ (\lambda y\ x.\ y\ x) \\ =_\alpha & x\ (\lambda z\ y.\ z\ y) \\ \neq_\alpha & z\ (\lambda z\ y.\ z\ y) \end{array}$$

α Conversion

Equality in Isabelle is equality modulo α conversion:

if $s =_\alpha t$ then s and t are syntactically equal.

Examples:

$$
\begin{array}{ll}
 & x\ (\lambda x\ y.\ x\ y) \\
=_\alpha & x\ (\lambda y\ x.\ y\ x) \\
=_\alpha & x\ (\lambda z\ y.\ z\ y) \\
\neq_\alpha & z\ (\lambda z\ y.\ z\ y) \\
\neq_\alpha & x\ (\lambda x\ x.\ x\ x)
\end{array}
$$

Back to β

We have defined β reduction: $\longrightarrow_\beta$
Some notation and concepts:

- ➜ β **conversion:** $s =_\beta t$ iff $\exists n.\ s \longrightarrow_\beta^* n \wedge t \longrightarrow_\beta^* n$

Back to β

We have defined β reduction: $\longrightarrow_\beta$
Some notation and concepts:

- β **conversion:** $s =_\beta t$ iff $\exists n.\ s \longrightarrow_\beta^* n \wedge t \longrightarrow_\beta^* n$
- t is **reducible** if there is an s such that $t \longrightarrow_\beta s$

Back to β

We have defined β reduction: $\longrightarrow_\beta$
Some notation and concepts:

- β **conversion:** $s =_\beta t$ iff $\exists n.\ s \longrightarrow_\beta^* n \wedge t \longrightarrow_\beta^* n$
- t is **reducible** if there is an s such that $t \longrightarrow_\beta s$
- $(\lambda x.\ s)\ t$ is called a **redex** (reducible expression)

Back to β

We have defined β reduction: $\longrightarrow_\beta$
Some notation and concepts:

- β **conversion:** $s =_\beta t$ iff $\exists n.\ s \longrightarrow_\beta^* n \wedge t \longrightarrow_\beta^* n$
- t is **reducible** if there is an s such that $t \longrightarrow_\beta s$
- $(\lambda x.\ s)\ t$ is called a **redex** (reducible expression)
- t is reducible iff it contains a redex

Back to β

We have defined β reduction: $\longrightarrow_\beta$
Some notation and concepts:

- β **conversion:** $s =_\beta t$ iff $\exists n.\ s \longrightarrow_\beta^* n \wedge t \longrightarrow_\beta^* n$
- t is **reducible** if there is an s such that $t \longrightarrow_\beta s$
- $(\lambda x.\ s)\ t$ is called a **redex** (reducible expression)
- t is reducible iff it contains a redex
- if it is not reducible, t is in **normal form**

Does every λ term have a normal form?

Example:

$$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)\ \longrightarrow_\beta$$

Does every λ term have a normal form?

Example:

$$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \longrightarrow_\beta$$
$$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \longrightarrow_\beta$$

Does every λ term have a normal form?

No!

Example:

$$
\begin{aligned}
(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) &\longrightarrow_\beta \\
(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) &\longrightarrow_\beta \\
(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) &\longrightarrow_\beta \ldots
\end{aligned}
$$

Does every λ term have a normal form?

No!

Example:

$$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \longrightarrow_\beta$$
$$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \longrightarrow_\beta$$
$$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \longrightarrow_\beta \ldots$$

(but: $(\lambda x\ y.\ y)\ ((\lambda x.\ x\ x)\ (\lambda x.\ x\ x)) \longrightarrow_\beta \lambda y.\ y$)

Does every λ term have a normal form?

No!

Example:

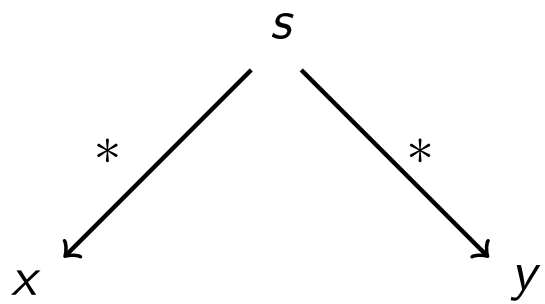
$$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \longrightarrow_\beta$$
$$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \longrightarrow_\beta$$
$$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \longrightarrow_\beta \ldots$$

(but: $(\lambda x\ y.\ y)\ ((\lambda x.\ x\ x)\ (\lambda x.\ x\ x)) \longrightarrow_\beta \lambda y.\ y$)

λ calculus is not terminating

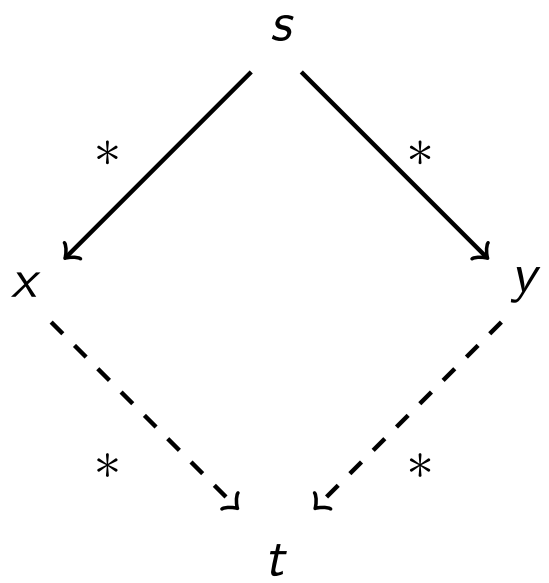
β reduction is confluent

Confluence: $s \longrightarrow_\beta^* x \wedge s \longrightarrow_\beta^* y \Longrightarrow \exists t.\ x \longrightarrow_\beta^* t \wedge y \longrightarrow_\beta^* t$



β reduction is confluent

Confluence: $s \longrightarrow_\beta^* x \wedge s \longrightarrow_\beta^* y \Longrightarrow \exists t.\ x \longrightarrow_\beta^* t \wedge y \longrightarrow_\beta^* t$



Order of reduction does not matter for result
Normal forms in λ calculus are unique

β reduction is confluent

Example:

$(\lambda x\ y.\ y)\ ((\lambda x.\ x\ x)\ a)$

$(\lambda x\ y.\ y)\ ((\lambda x.\ x\ x)\ a)$

β reduction is confluent

Example:

$$(\lambda x\ y.\ y)\ ((\lambda x.\ x\ x)\ a) \longrightarrow_\beta (\lambda x\ y.\ y)\ (a\ a)$$

$$(\lambda x\ y.\ y)\ ((\lambda x.\ x\ x)\ a) \longrightarrow_\beta \lambda y.\ y$$

β reduction is confluent

Example:

$(\lambda x\ y.\ y)\ ((\lambda x.\ x\ x)\ a) \longrightarrow_\beta (\lambda x\ y.\ y)\ (a\ a) \longrightarrow_\beta \lambda y.\ y$

$(\lambda x\ y.\ y)\ ((\lambda x.\ x\ x)\ a) \longrightarrow_\beta \lambda y.\ y$

η Conversion

Another case of trivially equal functions: $t = (\lambda x.\ t\ x)$

η Conversion

Another case of trivially equal functions: $t = (\lambda x.\ t\ x)$

Definition:

$$
\begin{array}{llllrll}
 & & & & (\lambda x.\ t\ x) & \longrightarrow_\eta & t \qquad \text{if } x \notin FV(t) \\
s & \longrightarrow_\eta & s' & \Longrightarrow & (s\ t) & \longrightarrow_\eta & (s'\ t) \\
t & \longrightarrow_\eta & t' & \Longrightarrow & (s\ t) & \longrightarrow_\eta & (s\ t') \\
s & \longrightarrow_\eta & s' & \Longrightarrow & (\lambda x.\ s) & \longrightarrow_\eta & (\lambda x.\ s')
\end{array}
$$

$$s =_\eta t \quad \text{iff} \quad \exists n.\ s \longrightarrow_\eta^* n \wedge t \longrightarrow_\eta^* n$$

Example: $(\lambda x.\ f\ x)\ (\lambda y.\ g\ y) \longrightarrow_\eta$

η Conversion

Another case of trivially equal functions: $t = (\lambda x.\ t\ x)$

Definition:

$$\begin{array}{llllll} & & (\lambda x.\ t\ x) & \longrightarrow_\eta & t & \text{if } x \notin FV(t) \\ s \longrightarrow_\eta s' & \Longrightarrow & (s\ t) & \longrightarrow_\eta & (s'\ t) & \\ t \longrightarrow_\eta t' & \Longrightarrow & (s\ t) & \longrightarrow_\eta & (s\ t') & \\ s \longrightarrow_\eta s' & \Longrightarrow & (\lambda x.\ s) & \longrightarrow_\eta & (\lambda x.\ s') & \end{array}$$

$$s =_\eta t \quad \text{iff} \quad \exists n.\ s \longrightarrow_\eta^* n \wedge t \longrightarrow_\eta^* n$$

Example: $(\lambda x.\ f\ x)\ (\lambda y.\ g\ y) \longrightarrow_\eta (\lambda x.\ f\ x)\ g \longrightarrow_\eta$

η Conversion

Another case of trivially equal functions: $t = (\lambda x.\ t\ x)$

Definition:

$$
\begin{array}{lllrll}
 & & & (\lambda x.\ t\ x) & \longrightarrow_\eta & t \qquad \text{if } x \notin FV(t) \\
s & \longrightarrow_\eta & s' & \Longrightarrow\ (s\ t) & \longrightarrow_\eta & (s'\ t) \\
t & \longrightarrow_\eta & t' & \Longrightarrow\ (s\ t) & \longrightarrow_\eta & (s\ t') \\
s & \longrightarrow_\eta & s' & \Longrightarrow\ (\lambda x.\ s) & \longrightarrow_\eta & (\lambda x.\ s')
\end{array}
$$

$$s =_\eta t \quad \text{iff} \quad \exists n.\ s \longrightarrow_\eta^* n \wedge t \longrightarrow_\eta^* n$$

Example: $(\lambda x.\ f\ x)\ (\lambda y.\ g\ y) \longrightarrow_\eta (\lambda x.\ f\ x)\ g \longrightarrow_\eta f\ g$

η Conversion

Another case of trivially equal functions: $t = (\lambda x.\ t\ x)$

Definition:

$$(\lambda x.\ t\ x) \longrightarrow_\eta t \quad \text{if } x \notin FV(t)$$

$$\begin{array}{lllll} s \longrightarrow_\eta s' & \Longrightarrow & (s\ t) & \longrightarrow_\eta & (s'\ t) \\ t \longrightarrow_\eta t' & \Longrightarrow & (s\ t) & \longrightarrow_\eta & (s\ t') \\ s \longrightarrow_\eta s' & \Longrightarrow & (\lambda x.\ s) & \longrightarrow_\eta & (\lambda x.\ s') \end{array}$$

$$s =_\eta t \quad \text{iff} \quad \exists n.\ s \longrightarrow_\eta^* n \wedge t \longrightarrow_\eta^* n$$

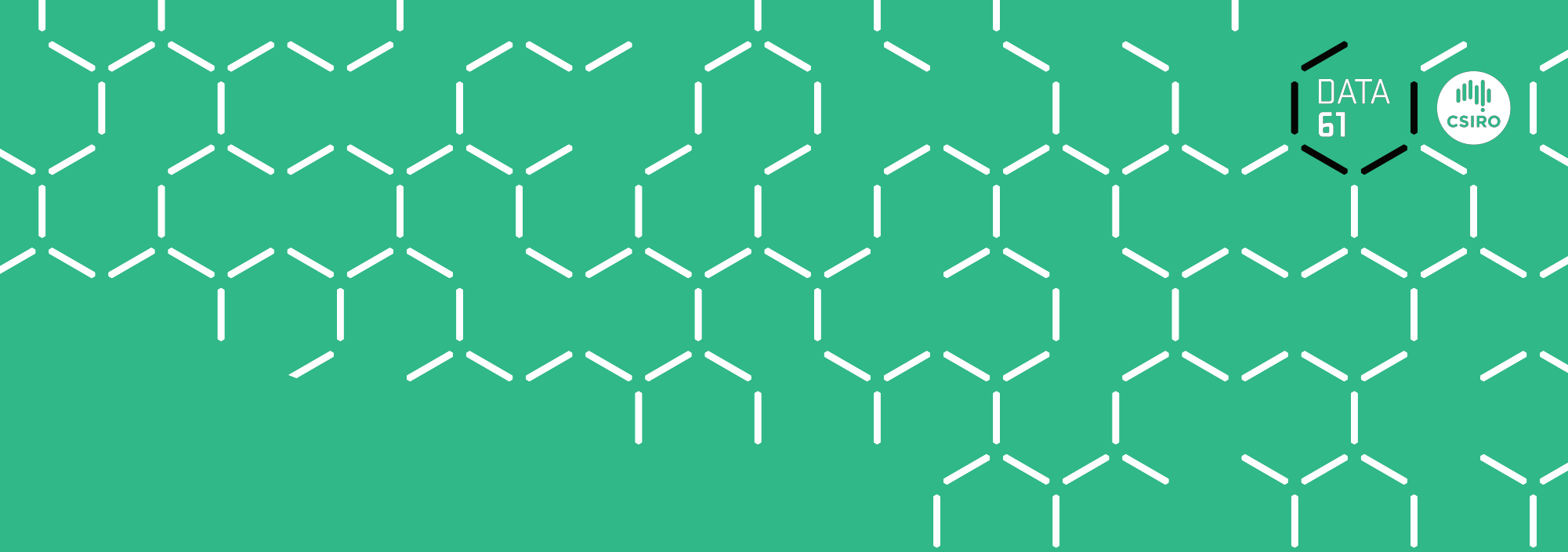
Example: $(\lambda x.\ f\ x)\ (\lambda y.\ g\ y) \longrightarrow_\eta (\lambda x.\ f\ x)\ g \longrightarrow_\eta f\ g$

- ➜ η reduction is confluent and terminating.
- ➜ $\longrightarrow_{\beta\eta}$ is confluent.
  $\longrightarrow_{\beta\eta}$ means $\longrightarrow_\beta$ and $\longrightarrow_\eta$ steps are both allowed.
- ➜ **Equality in Isabelle is also modulo η conversion.**

In fact ...

Equality in Isabelle is modulo α, β, and η conversion.

We will see later why that is possible.



Isabelle Demo

So, what can you do with λ calculus?

λ calculus is very expressive, you can encode:

- ➜ logic, set theory
- ➜ turing machines, functional programs, etc.

Examples:

So, what can you do with λ calculus?

λ calculus is very expressive, you can encode:

- ➜ logic, set theory
- ➜ turing machines, functional programs, etc.

Examples:

$$\texttt{true} \equiv \lambda x\ y.\ x$$
$$\texttt{false} \equiv \lambda x\ y.\ y$$
$$\texttt{if} \equiv \lambda z\ x\ y.\ z\ x\ y$$

So, what can you do with λ calculus?

λ calculus is very expressive, you can encode:

- ➜ logic, set theory
- ➜ turing machines, functional programs, etc.

Examples:

$$\texttt{true} \equiv \lambda x\ y.\ x \qquad \texttt{if true}\ x\ y \longrightarrow^*_\beta x$$
$$\texttt{false} \equiv \lambda x\ y.\ y \qquad \texttt{if false}\ x\ y \longrightarrow^*_\beta y$$
$$\texttt{if} \equiv \lambda z\ x\ y.\ z\ x\ y$$

So, what can you do with λ calculus?

λ calculus is very expressive, you can encode:

- ➜ logic, set theory
- ➜ turing machines, functional programs, etc.

Examples:

$$\texttt{true} \equiv \lambda x\ y.\ x \qquad \texttt{if true } x\ y \longrightarrow_\beta^* x$$
$$\texttt{false} \equiv \lambda x\ y.\ y \qquad \texttt{if false } x\ y \longrightarrow_\beta^* y$$
$$\texttt{if} \equiv \lambda z\ x\ y.\ z\ x\ y$$

Now, `not`, `and`, `or`, etc is easy:

So, what can you do with λ calculus?

λ calculus is very expressive, you can encode:

- ➜ logic, set theory
- ➜ turing machines, functional programs, etc.

Examples:

$$\texttt{true} \equiv \lambda x\ y.\ x \qquad \texttt{if true } x\ y \longrightarrow_\beta^* x$$
$$\texttt{false} \equiv \lambda x\ y.\ y \qquad \texttt{if false } x\ y \longrightarrow_\beta^* y$$
$$\texttt{if} \equiv \lambda z\ x\ y.\ z\ x\ y$$

Now, `not`, `and`, `or`, etc is easy:

$$\texttt{not} \equiv \lambda x.\ \texttt{if } x \texttt{ false true}$$
$$\texttt{and} \equiv \lambda x\ y.\ \texttt{if } x\ y \texttt{ false}$$
$$\texttt{or} \equiv \lambda x\ y.\ \texttt{if } x \texttt{ true } y$$

More Examples

Encoding natural numbers (Church Numerals)

$$
\begin{aligned}
0 &\equiv \lambda f\ x.\ x \\
1 &\equiv \lambda f\ x.\ f\ x \\
2 &\equiv \lambda f\ x.\ f\ (f\ x) \\
3 &\equiv \lambda f\ x.\ f\ (f\ (f\ x)) \\
&\ldots
\end{aligned}
$$

Numeral n takes arguments f and x, applies f n-times to x.

More Examples

Encoding natural numbers (Church Numerals)

$$
\begin{aligned}
0 &\equiv \lambda f\ x.\ x \\
1 &\equiv \lambda f\ x.\ f\ x \\
2 &\equiv \lambda f\ x.\ f\ (f\ x) \\
3 &\equiv \lambda f\ x.\ f\ (f\ (f\ x)) \\
&\ldots
\end{aligned}
$$

Numeral n takes arguments f and x, applies f n-times to x.

$\texttt{iszero} \equiv \lambda n.\ n\ (\lambda x.\ \texttt{false})\ \texttt{true}$

More Examples

Encoding natural numbers (Church Numerals)

$$
\begin{aligned}
0 &\equiv \lambda f\ x.\ x \\
1 &\equiv \lambda f\ x.\ f\ x \\
2 &\equiv \lambda f\ x.\ f\ (f\ x) \\
3 &\equiv \lambda f\ x.\ f\ (f\ (f\ x)) \\
&\ldots
\end{aligned}
$$

Numeral n takes arguments f and x, applies f n-times to x.

$$
\begin{aligned}
\texttt{iszero} &\equiv \lambda n.\ n\ (\lambda x.\ \texttt{false})\ \texttt{true} \\
\texttt{succ} &\equiv \lambda n\ f\ x.\ f\ (n\ f\ x)
\end{aligned}
$$

More Examples

Encoding natural numbers (Church Numerals)

$$
\begin{aligned}
0 &\equiv \lambda f\ x.\ x \\
1 &\equiv \lambda f\ x.\ f\ x \\
2 &\equiv \lambda f\ x.\ f\ (f\ x) \\
3 &\equiv \lambda f\ x.\ f\ (f\ (f\ x)) \\
&\ldots
\end{aligned}
$$

Numeral n takes arguments f and x, applies f n-times to x.

$$
\begin{aligned}
\texttt{iszero} &\equiv \lambda n.\ n\ (\lambda x.\ \texttt{false})\ \texttt{true} \\
\texttt{succ} &\equiv \lambda n\ f\ x.\ f\ (n\ f\ x) \\
\texttt{add} &\equiv \lambda m\ n.\ \lambda f\ x.\ m\ f\ (n\ f\ x)
\end{aligned}
$$

Fix Points

$$(\lambda x\ f.\ f\ (x\ x\ f))\ \ (\lambda x\ f.\ f\ (x\ x\ f))\ \ t \longrightarrow_\beta$$

Fix Points

$$(\lambda x\ f.\ f\ (x\ x\ f))\ \ (\lambda x\ f.\ f\ (x\ x\ f))\ \ t \longrightarrow_\beta$$
$$(\lambda f.\ f\ ((\lambda x\ f.\ f\ (x\ x\ f))\ \ (\lambda x\ f.\ f\ (x\ x\ f))\ f))\ \ t \longrightarrow_\beta$$

Fix Points

$$
\begin{aligned}
&(\lambda x\ f.\ f\ (x\ x\ f))\ \ (\lambda x\ f.\ f\ (x\ x\ f))\ \ t \longrightarrow_\beta \\
&(\lambda f.\ f\ ((\lambda x\ f.\ f\ (x\ x\ f))\ \ (\lambda x\ f.\ f\ (x\ x\ f))\ f))\ \ t \longrightarrow_\beta \\
&t\ \ ((\lambda x\ f.\ f\ (x\ x\ f))\ (\lambda x\ f.\ f\ (x\ x\ f))\ t)
\end{aligned}
$$

Fix Points

$$(\lambda x\ f.\ f\ (x\ x\ f))\ \ (\lambda x\ f.\ f\ (x\ x\ f))\ \ t \longrightarrow_\beta$$
$$(\lambda f.\ f\ ((\lambda x\ f.\ f\ (x\ x\ f))\ \ (\lambda x\ f.\ f\ (x\ x\ f))\ f))\ \ t \longrightarrow_\beta$$
$$t\ \ ((\lambda x\ f.\ f\ (x\ x\ f))\ (\lambda x\ f.\ f\ (x\ x\ f))\ t)$$

$$\mu = (\lambda x\ f.\ f\ (x\ x\ f))\ (\lambda x f.\ f\ (x\ x\ f))$$
$$\mu\ t \longrightarrow_\beta t\ (\mu\ t) \longrightarrow_\beta t\ (t\ (\mu\ t)) \longrightarrow_\beta t\ (t\ (t\ (\mu\ t))) \longrightarrow_\beta \ldots$$

Fix Points

$(\lambda x\ f.\ f\ (x\ x\ f))\ \ (\lambda x\ f.\ f\ (x\ x\ f))\ \ t \longrightarrow_\beta$
$(\lambda f.\ f\ ((\lambda x\ f.\ f\ (x\ x\ f))\ \ (\lambda x\ f.\ f\ (x\ x\ f))\ f))\ \ t \longrightarrow_\beta$
$t\ \ ((\lambda x\ f.\ f\ (x\ x\ f))\ (\lambda x\ f.\ f\ (x\ x\ f))\ t)$

$\mu = (\lambda x\ f.\ f\ (x\ x\ f))\ (\lambda x f.\ f\ (x\ x\ f))$
$\mu\ t \longrightarrow_\beta t\ (\mu\ t) \longrightarrow_\beta t\ (t\ (\mu\ t)) \longrightarrow_\beta t\ (t\ (t\ (\mu\ t))) \longrightarrow_\beta \ldots$

$(\lambda x f.\ f\ (x\ x\ f))\ (\lambda x f.\ f\ (x\ x\ f))$ is Turing's fix point operator

Nice, but ...

As a mathematical foundation, λ does not work. **It resulted in an inconsistent logic.**

Nice, but ...

As a mathematical foundation, λ does not work. **It resulted in an inconsistent logic.**

- ➜ **Frege** (Predicate Logic, $\sim$ 1879): allows arbitrary quantification over predicates
- ➜ **Russell** (1901): Paradox $R \equiv \{X | X \notin X\}$
- ➜ **Whitehead & Russell** (Principia Mathematica, 1910-1913): Fix the problem
- ➜ **Church** (1930): λ calculus as logic, `true`, `false`, $\wedge$, ... as λ terms

Problem:

Nice, but ...

As a mathematical foundation, λ does not work. **It resulted in an inconsistent logic.**

- ➜ **Frege** (Predicate Logic, $\sim$ 1879): allows arbitrary quantification over predicates
- ➜ **Russell** (1901): Paradox $R \equiv \{X | X \notin X\}$
- ➜ **Whitehead & Russell** (Principia Mathematica, 1910-1913): Fix the problem
- ➜ **Church** (1930): λ calculus as logic, `true`, `false`, $\wedge$, ... as λ terms

Problem:

with $\{x |\ P\ x\} \equiv \lambda x.\ P\ x \qquad x \in M \equiv M\ x$

Nice, but ...

As a mathematical foundation, λ does not work. **It resulted in an inconsistent logic.**

- ➜ **Frege** (Predicate Logic, $\sim$ 1879): allows arbitrary quantification over predicates
- ➜ **Russell** (1901): Paradox $R \equiv \{X | X \notin X\}$
- ➜ **Whitehead & Russell** (Principia Mathematica, 1910-1913): Fix the problem
- ➜ **Church** (1930): λ calculus as logic, `true`, `false`, $\wedge$, ... as λ terms

Problem:

with	$\{x \mid P\ x\} \equiv \lambda x.\ P\ x$	$x \in M \equiv M\ x$
you can write	$R \equiv \lambda x.\ \texttt{not}\ (x\ x)$	

Nice, but ...

As a mathematical foundation, λ does not work. **It resulted in an inconsistent logic.**

- ➜ **Frege** (Predicate Logic, $\sim$ 1879): allows arbitrary quantification over predicates
- ➜ **Russell** (1901): Paradox $R \equiv \{X | X \notin X\}$
- ➜ **Whitehead & Russell** (Principia Mathematica, 1910-1913): Fix the problem
- ➜ **Church** (1930): λ calculus as logic, `true`, `false`, $\wedge$, ... as λ terms

Problem:

with	$\{x \mid P\ x\} \equiv \lambda x.\ P\ x$	$x \in M \equiv M\ x$
you can write	$R \equiv \lambda x.\ \texttt{not}\ (x\ x)$	
and get	$(R\ R) =_\beta \texttt{not}\ (R\ R)$	

Nice, but ...

As a mathematical foundation, λ does not work. **It resulted in an inconsistent logic.**

- ➜ **Frege** (Predicate Logic, $\sim$ 1879): allows arbitrary quantification over predicates
- ➜ **Russell** (1901): Paradox $R \equiv \{X | X \notin X\}$
- ➜ **Whitehead & Russell** (Principia Mathematica, 1910-1913): Fix the problem
- ➜ **Church** (1930): λ calculus as logic, `true`, `false`, $\wedge$, ... as λ terms

Problem:

with	$\{x \mid P\ x\} \equiv \lambda x.\ P\ x$	$x \in M \equiv M\ x$
you can write	$R \equiv \lambda x.\ \texttt{not}\ (x\ x)$	
and get	$(R\ R) =_\beta \texttt{not}\ (R\ R)$	
because	$(R\ R) = (\lambda x.\ \texttt{not}\ (x\ x))\ R \longrightarrow_\beta \texttt{not}\ (R\ R)$	

We have learned so far...

- ➜ λ calculus syntax
- ➜ free variables, substitution
- ➜ β reduction
- ➜ α and η conversion
- ➜ β reduction is confluent
- ➜ λ calculus is very expressive (turing complete)
- ➜ λ calculus results in an inconsistent logic

