Introduce yourself, your project and why you picked it (math proofs)

I will start with some history of the application:

Written in Standard ML and Scala

Developed at \_ and \_

Released in 1986 but updated and improved by many institutes/individuals

The version I will focus on in Isabelle/HOL, released in late 90s

This version is most widely used, the HOL is higher order logic

Fun fact: named after \_’s daughter by UK guy

So, what exactly is Isabelle:

Genaric proof assistant

Allows for math formulas to be represented in a formal language and

Provides tools to prove these in a logical calculus

This means the formalization of math proofs and

Formal verification – soft/hardware verif, prove props of comp langs/protocols

Isar – actual proof language cool bc it is super human readable (and machine)

Isabelle has many tools that are used to help proofs

Classical reasoner performs long chains of reasoning steps to prove exprs

Simplifier can reason with and about equations (think equality)

Linear arithmetic and various algebraic decision procedures are provided

External *first-order provers* (for first order logic)

jEdit UI for Isabelle IDE, download on linux mac and windows

Where has Isabell been used?

seL4 chip verif by NICTA – National Info and communications tech of Australia

the main theorem stated that the C code correctly implements the formal specification of the kernel, used on all parts (design, code, implementation)

bugs in early version and over 150 issues in each of design and specification

Isabelle AFP – lots of proofs, can use to prove other things

Formalize thms such as those on the slides

Demo:

Run through file syntax – think header, imports, begin, then func dec (relations to functional programming and use of recursion) – then simple lemma syntax

Example code:

theory Demo

imports Main

begin

(\*add function\*)

fun add :: "nat ⇒ nat ⇒ nat" where

"add 0 n = n" |

"add (Suc m) n = Suc (add m n)"

(\*Proof for add is communitive (x + y = y + x) \*)

(\*2 come to this for base case\*)

lemma add\_zero [simp]: "y = add y 0"

apply(induction y)

apply(auto)

done

(\*3 come to this for inductive\*)

lemma add\_suc [simp]: "Suc (add y x) = add y (Suc x)"

apply(induction y)

apply(auto)

done

(\*1 start with this\*)

lemma add\_comm [simp]: "add x y = add y x"

apply(induction x)

apply(auto)

done

(\*Define binary tree\*)

datatype 'a tree = Leaf | Node " 'a tree" 'a " 'a tree"

(\*Define mirror function\*)

fun mirror :: "'a tree ⇒ 'a tree" where

"mirror Leaf = Leaf" |

"mirror (Node l a r) = Node (mirror r) a (mirror l)"

(\*Simple proof for double mirror reverts back\*)

lemma mir2\_bt [simp]: "mirror(mirror t) = t"

apply(induction t)

apply(auto)

done

(\*Define sum nodes of nat bt function\*)

fun sum :: "nat tree ⇒ nat" where

"sum Leaf = 0"|

"sum (Node l x r) = x + sum l + sum r"

(\*Simple proof for sum always positive\*)

lemma "0 ≤ sum t"

apply(induction t)

apply(auto)

done

end