Towards Practical and Rigorous Automated Grading in Functional Programming Courses

Dragana Milovančević
EPFL, Switzerland
dragana.milovancevic@epfl.ch

July 31, 2023

EPFL

Women in EuroProofNet 2023

Warm-Up Exercise #1

Warm-Up Exercise #1

Write a function that deletes the duplicate elements from the stated list.

```
def uniq(lst: List[Int]): List[Int] = ???
```

Warm-Up Exercise #1

Write a function that deletes the duplicate elements from the stated list.

```
def uniq(lst: List[Int]): List[Int] =
  distinct(List(), lst)

def distinct(a: List[Int], b: List[Int]): List[Int] =
  b match
    case Nil() => a
    case Cons(x, xs) =>
      if isin(x, a) then distinct(a, xs)
      else distinct(a ++ List(x), xs)

def isin(n: Int, lst: List[Int]): Boolean =
  lst.foldRight(false){ (e, acc) =>
    (e == n || acc)
  }
```

Is this solution correct?

Warm-Up Exercise #1

Write a function that deletes the duplicate elements from the stated list.

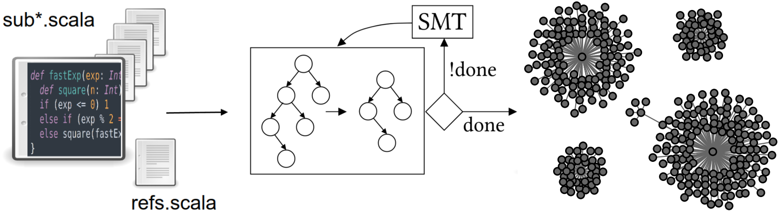
```
def uniq(lst: List[Int]): List[Int] =
  distinct(List(), lst)

def distinct(a: List[Int], b: List[Int]): List[Int] =
  b match
    case Nil() => a
    case Cons(x, xs) =>
      if isin(x, a) then distinct(a, xs)
      else distinct(a ++ List(x), xs)

def isin(n: Int, lst: List[Int]): Boolean =
  lst.foldRight(false){ (e, acc) =>
    (e == n || acc)
  }
```

Is this solution correct? ✓

How Does This Scale?

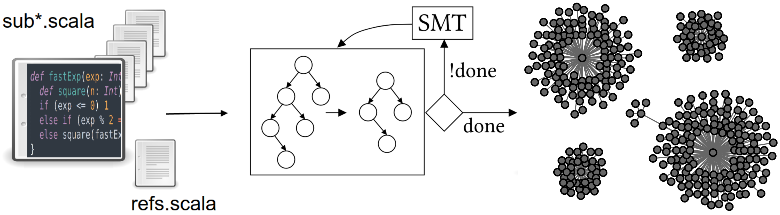
How Does This Scale?

Proving and Disproving Equivalence of Functional Programming Assignments[1]



[1]Dragana Milovancevic and Viktor Kunčak. 2023. Proving and Disproving Equivalence of Functional Programming Assignments. PLDI'23.

Proving and Disproving Equivalence of Functional Programming Assignments[1]



- An interesting combination of program verification and clustering
- Program decomposition and function call matching (modular programs)
- A large variety of recursive problems, challenging for equivalence proofs

[1] Dragana Milovancevic and Viktor Kunčak. 2023. Proving and Disproving Equivalence of Functional Programming Assignments. PLDI'23.

Why would students in an introductory course care about proofs?

*Automated techniques make it easy to perform **flawed** assessment at scale*[1]

- Because testing is not enough!
- Formal techniques as a guarantee that programs are never wrongly classified as correct, or incorrect

[1] John Wrenn, Shriram Krishnamurthi, and Kathi Fisler. 2018. Who Tests the Testers?. ICER'18.

Why equivalence checking?

- Because formal verification is hard!
- Case studies show ratios such as 9 lines of specifications per executable line[1]
- Impractical for automated grading

[1] Mario Bucev and Viktor Kunčak. Formally verified quite ok image format. In 2022 Formal Methods in Computer-Aided Design (FMCAD'22).

Overview of Our Approach

3. Clustering algorithm that finds the subset of correct solutions

2. Function call matching based on type- and test-directed search

1. Pairwise equivalence checking based on functional induction

Pairwise Equivalence Checking

Pairwise Equivalence Checking

A candidate program F is equivalent to a reference program M if:

- M and F have the same signature
- M and F terminate
- M and F return the same output for all inputs

Pairwise Equivalence Checking: Recursion and Functional Induction

```
def isinM(lst:List[Int], n:Int): Boolean =
  if lst.isEmpty then false
  else if lst.head == n then true
  else isinM(lst.tail, n)
```

```
def isin(lst:List[Int], n:Int): Boolean =
  lst.foldRight(false){ (e, acc) =>
    (e == n || acc)
  }
```

Pairwise Equivalence Checking: Recursion and Functional Induction

```
def isinM(lst:List[Int], n:Int): Boolean = {
  if lst.isEmpty then false
  else if lst.head == n then true
  else isinM(lst.tail, n)








} ensuring(result => result == isin(lst, n))
```

Pairwise Equivalence Checking: Recursion and Functional Induction

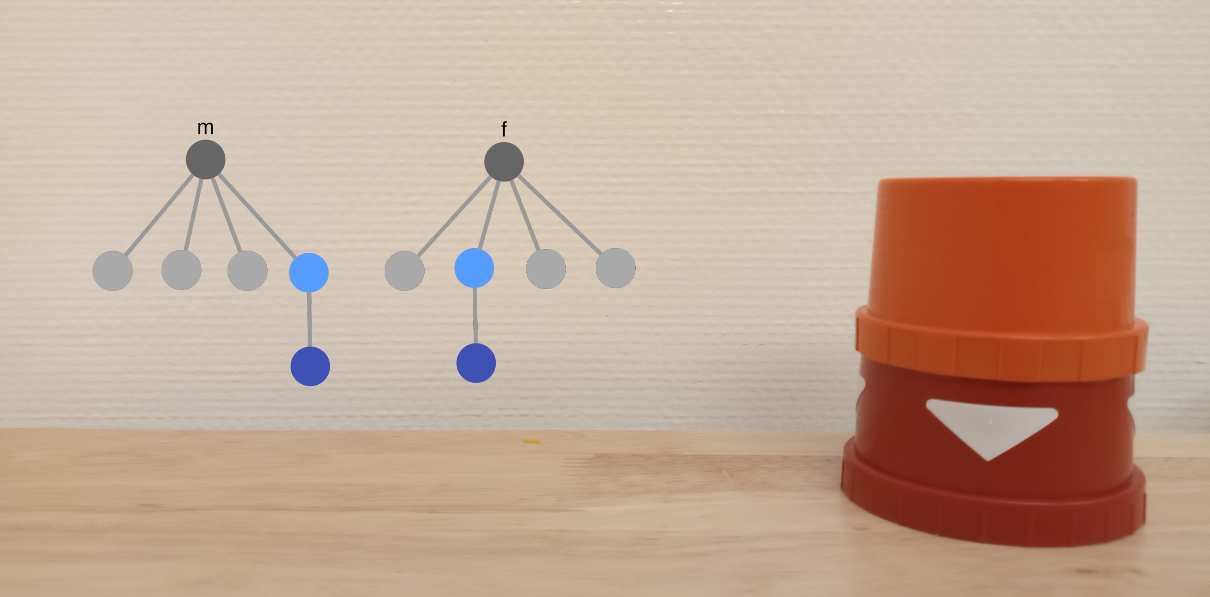
```
def isinM(lst:List[Int], n:Int): Boolean = {
  if lst.isEmpty then false
  else if lst.head == n then true
  else
    val tail = lst.tail
    val res =
      if tail.isEmpty then false
      else if tail.head == n then true
      else isinM(tail.tail, n)
    assume(res == isin(tail, n))
    res
} ensuring(result => result == isin(lst, n))
```

Pairwise Equivalence Checking: Recursion and Functional Induction

```
def isinM(lst:List[Int], n:Int): Boolean = {
  if lst.isEmpty then false
  else if lst.head == n then true
  else isinM(lst.tail, n)
    val tail = lst.tail
    val res =
      if tail.isEmpty then false
      else if tail.head == n then true
      else isinM(tail.tail, n)
    assume(res == isin(tail, n))
    res
} ensuring(result => result == isin(lst, n))
```

Function Call Matching

Function Call Matching



Function Call Matching: An Example

```scala
def uniqM(lst: List[Int]): List[Int] =
  distinctM(lst, Nil())

def distinctM(l: List[Int], r: List[Int]): List[Int] =
  l match
    case Nil() => r
    case Cons(x, xs) =>
      if !isinM(r, x) then distinctM(xs, r ++ List(x))
      else distinctM(xs, r)

def isinM(lst: List[Int], n: Int): Boolean =
  if lst.isEmpty then false
  else if lst.head == n then true
  else isinM(lst.tail, n)
```

Function Call Matching: Type- and Test-Directed Search

```
def uniqM(lst: List[Int]): List[Int]
def distinctM(l: List[Int], r: List[Int]): List[Int]
def isinM(lst: List[Int], n: Int): Boolean

def uniq(lst: List[Int]): List[Int]
def distinct(a: List[Int], b: List[Int]): List[Int]
def isin(n: Int, lst: List[Int]): Boolean
```

Function Call Matching: Type- and Test-Directed Search

```
def uniqM(lst: List[Int]): List[Int]
def distinctM(l: List[Int], r: List[Int]): List[Int]
def isinM(lst: List[Int], n: Int): Boolean

✗ def uniq(lst: List[Int]): List[Int]
✗ def distinct(a: List[Int], b: List[Int]): List[Int]
✓ def isin(n: Int, lst: List[Int]): Boolean
```

- Type-directed search
- Test-directed search

Function Call Matching: Type- and Test-Directed Search

```
def uniqM(lst: List[Int]): List[Int]
def distinctM(l: List[Int], r: List[Int]): List[Int]
def isinM( lst: List[Int] , n: Int ): Boolean

def uniq(lst: List[Int]): List[Int]
def distinct(a: List[Int], b: List[Int]): List[Int]
def isin( n: Int , lst: List[Int] ): Boolean
```

- Type-directed search

  Test-directed search

Function Call Matching:
Type- and Test-Directed Search

```
def uniqM(lst: List[Int]): List[Int]
def distinctM( l: List[Int] , r: List[Int] ): List[Int]
def isinM(lst: List[Int], n: Int): Boolean

def uniq(lst: List[Int]): List[Int]
def distinct( a: List[Int] , b: List[Int] ): List[Int]
def isin(n: Int, lst: List[Int]): Boolean
```

Type-directed search

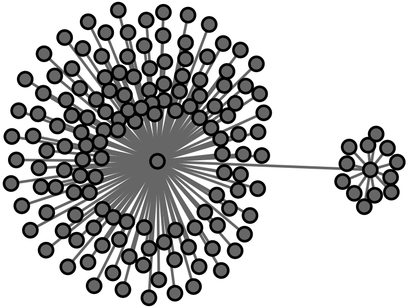
- Test-directed search

Clustering Algorithm

Clustering Algorithm

- Classifies candidate programs using pairwise equivalence as a subroutine
- Prioritization of reference programs, including candidate programs proven correct
- Reuse of generated counterexamples

Clustering Algorithm: Discovery of Intermediate Reference Solutions



- The single reference solution is in the center of the larger cluster
- Our system automatically finds the intermediate reference solution, from the set of student submissions
- Prioritization of reference programs to avoid the quadratic behavior

Evaluation

1. Results on equivalence checking examples
2. Results on functional programming assignments

Results on Equivalence Checking Examples

	ackermann[1]	mccarthy91[1]	limit1[1]	limit2[1]	limit3[1]	add-horn[1]	triangular[1]	inlining[1]	sum[2]	fibonacci-f[2]	pascal[2]	fibonacci-m[2]	fibonacci-t[2]	fibonacci-h[2]	fact4[3]	fact13[3]	fact14[3]
REVE[1]	✓	✓	✗	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
RVT[23]	✓	✓	✓	✗	✗	✓	✗	✗	✓	✓	✗	✗	✗	✓	✓	✗	✗
Our System	✗	✗	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

[1] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating Regression Verification. ASE'14.

[2] Chaked R. J. Sayedoff and Ofer Strichman. 2022. Regression verification of unbalanced recursive functions with multiple calls (long version).

[3] Ofer Strichman and Maor Veitsman. 2016. Regression Verification for Unbalanced Recursive Functions. FM'16

Results on FP Assignments

Results on FP Assignments

No	Name	#S	#P	#T	#F	LoC	Found/Term.
1	filter	1	210	210	1.1	9	99%
2	max	1	216	216	1.5	15	92%
3	mirror	1	97	96	1.1	19	100%
4	mem	1	136	136	1.1	18	100%
5	sigma	3	736	734	1.1	11	97%
6	natadd	4	447	381	1.4	13	93%
7	natmul	7	447	381	2.9	27	62%
8	change	1	9	8	2.6	26	100%
9	heap	1	20	11	6.4	39	100%
10	uniq	4	157	147	2.8	23	81%
11	iter	1	28	27	1.3	11	0%
12	crazy2add	1	240	-	2.0	52	NA
13	formula	1	708	680	2.1	52	97%
14	lambda	4	802	781	2.6	34	71%
15	diff	1	409	-	2.1	60	NA

- Scala translations of public benchmarks from an OCaml FP course
- Thousands of programs
- Typical FP problems, include user-defined types and higher-order functions

Example 1/3: Max

```scala
def maxM(lst: List[Int]): Int = lst match
  case Nil() => Int.MinValue
  case Cons(h, Nil()) => h
  case Cons(h, t) => if h > maxM(t) then h else maxM(t)

def maxT(lst: List[Int]): Int = lst match
  case Nil() => Int.MinValue
  case Cons(a, Nil()) => a
  case Cons(a, Cons(b, t)) =>
    if a > b then maxT(a :: t) else maxT(b :: t)

def maxF(lst: List[Int]): Int = lst match
  case Nil() => Int.MinValue
  case Cons(h, t) =>
    t.foldLeft(h)((a, b) => if a >= b then a else b)
```

✓

Example 2/3: Formula

```
def eval(p: Formula): Boolean =
  p match
    case True => true
    case False => false
    case Not(a) => !eval(a)
    case AndAlso(a, b) => eval(a) && eval(b)
    case OrElse(a, b) => eval(a) || eval(b)
    case Imply(a, b) => eval(Not(a)) || eval(b)
    case Equal(a, b) => if my_exp(a) == my_exp(b) then true else false

def my_exp(p: Exp): Int =
  p match
    case Num(b) => b
    case Plus(a, b) => my_exp(a) + my_exp(b)
    case Minus(a, b) => my_exp(a) - my_exp(b)
```

✓

Example 3/3 Change

```
def change(coins: List[BigInt], amount: BigInt): BigInt = {
  require(amount >= 0)
  def coin_reduce(lst: List[BigInt], v: BigInt): BigInt = {
    require(v >= 0)
    lst match
      case Nil() => 0
      case Cons(hd, tl) =>
        if hd <= 0 then 999
        else if v - hd == 0 then 1
        else if v - hd < 0 then 0
        else coin_reduce(tl, v) + coin_reduce(lst, v-hd)
  }
  if amount == 0 then if coins == Nil() then 0 else 1
  else if amount < 0 then 0
  else coin_reduce(coins, amount)
}
```

✗ coins: List(2, 0, 1), amount: 1

Results on FP Assignments

No	Name	#S	#P	#T	#F	LoC	Found/Term.
1	filter	1	210	210	1.1	9	99%
2	max	1	216	216	1.5	15	92%
3	mirror	1	97	96	1.1	19	100%
4	mem	1	136	136	1.1	18	100%
5	sigma	3	736	734	1.1	11	97%
6	natadd	4	447	381	1.4	13	93%
7	natmul	7	447	381	2.9	27	62%
8	change	1	9	8	2.6	26	100%
9	heap	1	20	11	6.4	39	100%
10	uniq	4	157	147	2.8	23	81%
11	iter	1	28	27	1.3	11	0%
12	crazy2add	1	240	-	2.0	52	NA
13	formula	1	708	680	2.1	52	97%
14	lambda	4	802	781	2.6	34	71%
15	diff	1	409	-	2.1	60	NA

Results on FP Assignments

No	Name	#S	#P	#T	#F	LoC	Found/Term.
1	filter	1	210	210	1.1	9	99%
2	max	1	216	216	1.5	15	92%
3	mirror	1	97	96	1.1	19	100%
4	mem	1	136	136	1.1	18	100%
5	sigma	3	736	734	1.1	11	97%
6	natadd	4	447	381	1.4	13	93%
7	natmul	7	447	381	2.9	27	62%
8	change	1	9	8	2.6	26	100%
9	heap	1	20	11	6.4	39	100%
10	uniq	4	157	147	2.8	23	81%
11	iter	1	28	27	1.3	11	0%
12	crazy2add	1	240	-	2.0	52	NA
13	formula	1	708	680	2.1	52	97%
14	lambda	4	802	781	2.6	34	71%
15	diff	1	409	-	2.1	60	NA

- Overall 86% success rate (Found/Term.)
- Overall 414/420 of counterexamples discovered (99%)

Results on FP Assignments

No	Name	#S	#P	#T	#F	LoC	Found/Term.
1	filter	1	210	210	1.1	9	99%
2	max	1	216	216	1.5	15	92%
3	mirror	1	97	96	1.1	19	100%
4	mem	1	136	136	1.1	18	100%
5	sigma	3	736	734	1.1	11	97%
6	natadd	4	447	381	1.4	13	93%
7	natmul	7	447	381	2.9	27	62%
8	change	1	9	8	2.6	26	100%
9	heap	1	20	11	6.4	39	100%
10	uniq	4	157	147	2.8	23	81%
11	iter	1	28	27	1.3	11	0%
12	crazy2add	1	240	-	2.0	52	NA
13	formula	1	708	680	2.1	52	97%
14	lambda	4	802	781	2.6	34	71%
15	diff	1	409	-	2.1	60	NA

- Large numbers of submissions (over 300 per benchmark on avg.)
- Variety of recursive problems (15 in total, with 28 LoC on avg.)
- Mostly correct submissions (∼90%)

Results on FP Assignments

No	Name	#S	#P	#T	#F	LoC	Found/Term.
1	filter	1	210	210	1.1	9	99%
2	max	1	216	216	1.5	15	92%
3	mirror	1	97	96	1.1	19	100%
4	mem	1	136	136	1.1	18	100%
5	sigma	3	736	734	1.1	11	97%
6	natadd	4	447	381	1.4	13	93%
7	natmul	7	447	381	2.9	27	62%
8	change	1	9	8	2.6	26	100%
9	heap	1	20	11	6.4	39	100%
10	uniq	4	157	147	2.8	23	81%
11	iter	1	28	27	1.3	11	0%
12	crazy2add	1	240	-	2.0	52	NA
13	formula	1	708	680	2.1	52	97%
14	lambda	4	802	781	2.6	34	71%
15	diff	1	409	-	2.1	60	NA

- Overall 96% success rate on single-function benchmarks
- One reference solution is typically sufficient

Results on FP Assignments

No	Name	#S	#P	#T	#F	LoC	Found/Term.
1	filter	1	210	210	1.1	9	99%
2	max	1	216	216	1.5	15	92%
3	mirror	1	97	96	1.1	19	100%
4	mem	1	136	136	1.1	18	100%
5	sigma	3	736	734	1.1	11	97%
6	natadd	4	447	381	1.4	13	93%
7	natmul	7	447	381	2.9	27	62%
8	change	1	9	8	2.6	26	100%
9	heap	1	20	11	6.4	39	100%
10	uniq	4	157	147	2.8	23	81%
11	iter	1	28	27	1.3	11	0%
12	crazy2add	1	240	-	2.0	52	NA
13	formula	1	708	680	2.1	52	97%
14	lambda	4	802	781	2.6	34	71%
15	diff	1	409	-	2.1	60	NA

Resources: Implementation, Benchmarks

- Implementation on top of the Stainless verifier for Scala
- Open source: github.com/epfl-lara/stainless
- Accompanying artifact with all our benchmarks: zenodo.org/record/7810840
- Additional examples: stainless/frontends/benchmarks/equivalence

Example Run

```
milovanc@thinkpadx1e:~/equivalence/benchmarks/fastexp$ ls
A14.scala  A2.scala  B11.scala  B16.scala  C18.scala  C6.scala
A1.scala   A5.scala  B14.scala  C15.scala  C1.scala   refs.scala
milovanc@thinkpadx1e:~/equivalence/benchmarks/fastexp$ cd ../../stainless
milovanc@thinkpadx1e:~/equivalence/stainless$ ./stainless ../benchmarks/fastexp/*.scala --batched=t
rue --comparefuns=fastExp --models=fastExpM --timeout=0.5
```

```
[  Info  ] Printing equivalence checking results:
[  Info  ] List of functions that are equivalent to model Model.fastExpM: B14.fastExp, A14.fastExp, C15.fastExp, A1.fastE
[  Info  ] List of erroneous functions: C1.fastExp
[  Info  ] List of timed-out functions: A5.fastExp
[  Info  ] List of wrong functions:
[  Info  ] Printing the final state:
[  Info  ] Path for the function B16.fastExp: Model.fastExpM
[  Info  ] Path for the function B11.fastExp: Model.fastExpM
[  Info  ] Path for the function C13.fastExp: Model.fastExpM
[  Info  ] Path for the function C6.fastExp: Model.fastExpM
[  Info  ] Path for the function A5.fastExp:
[  Info  ] Path for the function A2.fastExp: Model.fastExpM
[  Info  ] Path for the function A1.fastExp: Model.fastExpM
[  Info  ] Path for the function C1.fastExp: Model.fastExpM
[  Info  ] Path for the function C15.fastExp: Model.fastExpM
[  Info  ] Path for the function A14.fastExp: Model.fastExpM
[  Info  ] Path for the function B14.fastExp: Model.fastExpM
[  Info  ] Counterexample for the function C1.fastExp: Map(base -> BigInt("0"), exp -> BigInt("1"))
```

Resources: Related Work

- Automated grading
  - Clustering: OverCode, Clara, CoderAssist, LEGenT, ZEUS...
  - LearnML: counterexample + feedback generation (FixML, TestML, CAFE)
- Equivalence checking
  - Regression verification: REVE, RVT
  - Translation validation
- Automated proofs by induction
  - Recursion induction in Isabelle, functional induction in Coq
  - Functional induction is the default induction heuristic in ACL2

Warm-Up Exercise #2

Warm-Up Exercise #2

Given the following lemmas:

(MapNil)	`Nil.map(f) === Nil`
(MapCons)	`(x :: xs).map(f) === f(x) :: xs.map(f)`
(MapTrNil)	`Nil.mapTr(f, ys) === ys`
(MapTrCons)	`(x :: xs).mapTr(f, ys) === xs.mapTr(f, ys ++ (f(x) :: Nil))`
(NilAppend)	`Nil ++ xs === xs`
(ConsAppend)	`(x :: xs) ++ ys === x :: (xs ++ ys)`

Let us prove the following lemma for the base case: l is Nil.

- `Nil.mapTr(f, y::ys) === y::Nil.mapTr(f, ys)`

Warm-Up Exercise #2

Given the following lemmas:

(MapNil) `Nil.map(f) === Nil`

(MapCons) `(x :: xs).map(f) === f(x) :: xs.map(f)`

(MapTrNil) `Nil.mapTr(f, ys) === ys`

(MapTrCons) `(x :: xs).mapTr(f, ys) === xs.mapTr(f, ys ++ (f(x) :: Nil))`

(NilAppend) `Nil ++ xs === xs`

(ConsAppend) `(x :: xs) ++ ys === x :: (xs ++ ys)`

Let us prove the following lemma for the base case: l is Nil.

- `Nil.mapTr(f, y::ys) === y::Nil.mapTr(f, ys)`

```
Nil.mapTr(f, y::ys)
=== y::ys                (by MapTrNil(f, y::ys))
=== y::Nil.mapTr(f, ys)  (by MapTrNil(f, ys))
```

Is this solution correct?

Warm-Up Exercise #2

Given the following lemmas:

(MapNil) `Nil.map(f) === Nil`

(MapCons) `(x :: xs).map(f) === f(x) :: xs.map(f)`

(MapTrNil) `Nil.mapTr(f, ys) === ys`

(MapTrCons) `(x :: xs).mapTr(f, ys) === xs.mapTr(f, ys ++ (f(x) :: Nil))`

(NilAppend) `Nil ++ xs === xs`

(ConsAppend) `(x :: xs) ++ ys === x :: (xs ++ ys)`

Let us prove the following lemma for the base case: l is Nil.

- `Nil.mapTr(f, y::ys) === y::Nil.mapTr(f, ys)`

```
Nil.mapTr(f, y::ys)
=== y::ys                (by MapTrNil(f, y::ys))
=== y::Nil.mapTr(f, ys)  (by MapTrNil(f, ys))
```

Is this solution correct? ✓

How Does This Scale?

How Does This Scale?

- ☐ MapCons, IH, NilAppend, MapTrCons, IH
- ☐ MapCons, IH, IH, NilAppend, MapTrCons
- ☐ MapCons, AccOut, IH, NilAppend, MapTrCons
- ☐ MapTrCons, AccOut, NilAppend, IH, MapCons
- ☐ MapCons, IH, NilAppend, MapTrCons, AccOut
- ☐ MapCons, NilAppend, AccOut, MapTrCons, AccO
- ☐ MapCons, NilAppend, AccOut, MapTrCons, IH
- ☐ MapTrCons, IH, NilAppend, AccOut, MapCons
- ☐ MapCons, IH, AccOut, NilAppend, MapTrCons
- ☐ MapCons, NilAppend, IH, AccOut, MapTrCons
- ☐ MapCons, NilAppend, AccOut, AccOut, MapTrCo

How Does This Scale?

- [ ] ~~MapCons, IH, NilAppend, MapTrCons, IH~~
- [ ] ~~MapCons, IH, IH, NilAppend, MapTrCons~~
- [ ] ~~MapCons, AccOut, IH, NilAppend, MapTrCons~~
- [ ] ~~MapTrCons, AccOut, NilAppend, IH, MapCons~~
- [ ] ~~MapCons, IH, NilAppend, MapTrCons, AccOut~~
- [ ] ~~MapCons, NilAppend, AccOut, MapTrCons, AccOu~~
- [ ] ~~MapCons, NilAppend, AccOut, MapTrCons, IH~~
- [ ] ~~MapTrCons, IH, NilAppend, AccOut, MapCons~~
- [ ] ~~MapCons, IH, AccOut, NilAppend, MapTrCons~~
- [ ] ~~MapCons, NilAppend, IH, AccOut, MapTrCons~~
- [ ] ~~MapCons, NilAppend, AccOut, AccOut, MapTrC~~

Proving Correctness of Proof Assignments – Future Directions

```
val AccOutNil = Theorem( Nil.mapTr(f, (x :: xs)) === (x :: Nil.mapTr(f, xs)) ) {
  have     ( Nil.mapTr(f, (x :: xs)) === (x :: xs) )
                                by Apply(mapTr.NilCase of (acc → (x :: xs)))
  thenHave( Nil.mapTr(f, (x :: xs)) === (x :: Nil.mapTr(f, xs)) )
                                by Apply(mapTr.NilCase of (acc → xs)) }
```

[1] Simon Guilloud, Sankalp Gambhir, and Viktor Kuncak. 2023. LISA – A Modern Proof System. ITP'23.

Proving Correctness of Proof Assignments – Future Directions

```
val AccOutNil = Theorem( Nil.mapTr(f, (x :: xs)) === (x :: Nil.mapTr(f, xs)) ) {
  have     ( Nil.mapTr(f, (x :: xs)) === (x :: xs) )
                                    by Apply(mapTr.NilCase of (acc → (x :: xs)))
  thenHave( Nil.mapTr(f, (x :: xs)) === (x :: Nil.mapTr(f, xs)) )
                                    by Apply(mapTr.NilCase of (acc → xs)) }
```

- Our candidate: LISA[1], a new proof assistant based on set theory
- Why LISA?
  - Does not require deep knowledge of proof assistants
  - Provides an intuitive and programmer-friendly environment
  - Key features: DSL + high-level interface

[1]Simon Guilloud, Sankalp Gambhir, and Viktor Kuncak. 2023. LISA – A Modern Proof System. ITP'23.

Conclusion

- Practical and rigorous autograding of FP assignments
- Combination of simple techniques
- Fully automated
  - The only inputs are student submissions and reference solution
- Rigorous
  - Relies on program verifiers (or proof assistants)
- Effective in practice
  - Outperforms equivalence checking tools on their own benchmarks
  - 86% success rate on thousands of submissions from an actual FP course