Using Proofs-from-Tests to Verify Higher-Order Programs

Suresh Jagannathan

Joint work with He Zhu

Introduction

Introduction

- How can we integrate specification inference and automated verification techniques within an optimizing compiler for Standard ML?
    - ★ Enrich the class of *provably* correct optimizations
    - ★ Facilitate better specialization and structure representation decisions

Introduction

- How can we integrate specification inference and automated verification techniques within an optimizing compiler for Standard ML?
  - Enrich the class of *provably* correct optimizations
  - Facilitate better specialization and structure representation decisions
- What do we need the system to be?
  - *Automated*
  - *Modular*
  - *Precise*
    - Incorporate notions of {path, context} - sensitivity
  - *Scalable and lightweight*
    - Use off-the-shelf verification tools
  - *Understandable*
    - Analysis over a high-level intermediate representation
    - Useful for error checking

Introduction

- How can we integrate specification inference and automated verification techniques within an optimizing compiler for Standard ML?
  - ★ Enrich the class of *provably* correct optimizations
  - ★ Facilitate better specialization and structure representation decisions
- What do we need the system to be?
  - ★ *Automated*
  - ★ *Modular*
  - ★ *Precise*
    - ✦ Incorporate notions of {path, context} - sensitivity
  - ★ *Scalable and lightweight*
    - ✦ Use off-the-shelf verification tools
  - ★ *Understandable*
    - ✦ Analysis over a high-level intermediate representation
    - ✦ Useful for error checking

Failure to infer a "rich" specification only implies a missed optimization opportunity, not a violation of compiler correctness

Challenges

Consider

```
fun arraymax a g =
  let fun am h j m =
       let val k = h j
           val a = assert (k>=0 /\ k<len a)
           val u = sub a k
           val p = max u m
       in assert (p >= m); p
       end
      fun am’ = am g
  in foldl (len a) 0 am’
  end
```

Challenges

Consider

```
fun arraymax a g =
  let fun am h j m =
       let val k = h j
           val a = assert (k>=0 /\ k<len a)
           val u = sub a k
           val p = max u m
       in assert (p >= m); p
       end
      fun am’ = am g
  in foldl (len a) 0 am’
  end
```

k within bounds of array a

p is a maximal element

Expressive assertion language

Challenges

Consider

```
fun arraymax a g =
 let fun am h j m =
    let val k = h j
      val a = assert (k>=0 /\ k<len a)
      val u = sub a k
      val p = max u m
    in assert (p >= m); p
    end
   fun am' = am g
 in foldl (len a) 0 am'
 end
```

k within bounds of array a

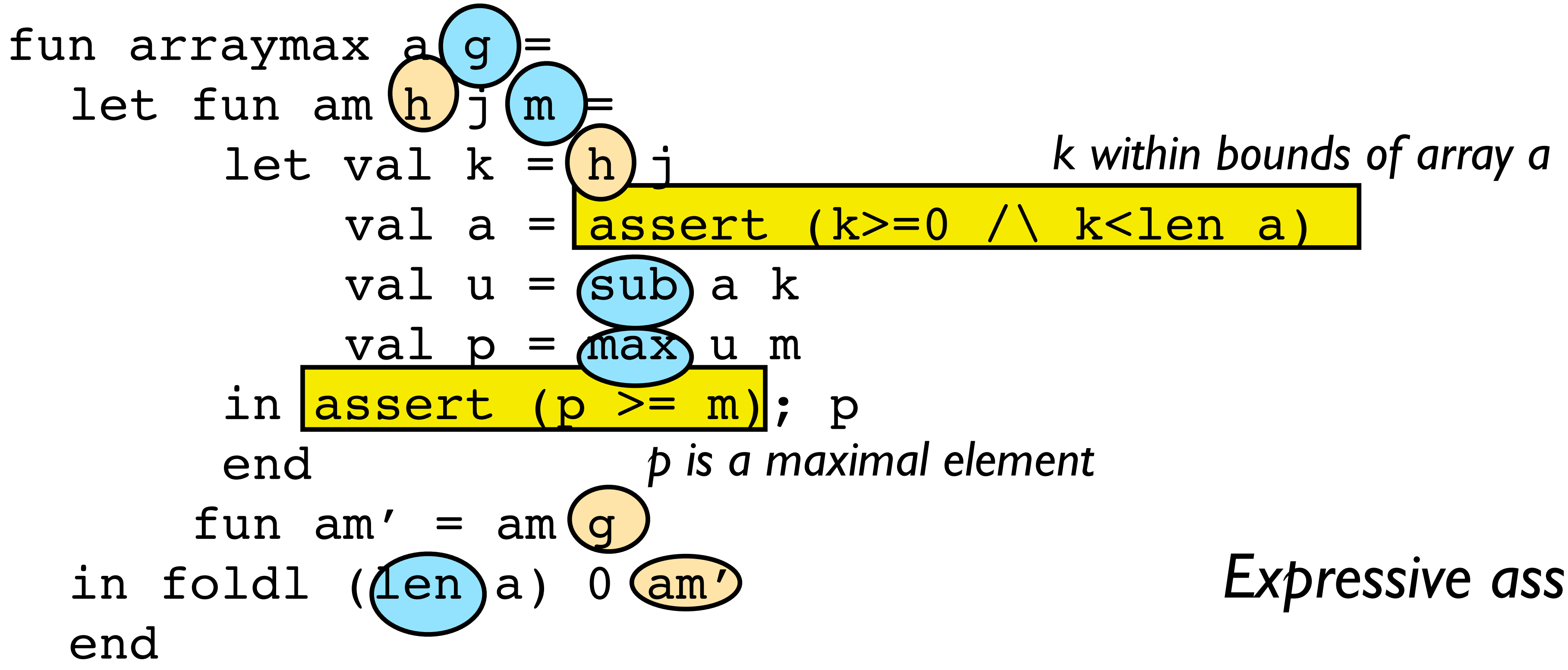
p is a maximal element

Expressive assertion language

Complex dataflow

Challenges

Consider



```
fun arraymax a g =
  let fun am h j m =
       let val k = h j
           val a = assert (k>=0 /\ k<len a)
           val u = sub a k
           val p = max u m
       in assert (p >= m); p
       end
      fun am' = am g
  in foldl (len a) 0 am'
  end
```

k within bounds of array a

p is a maximal element

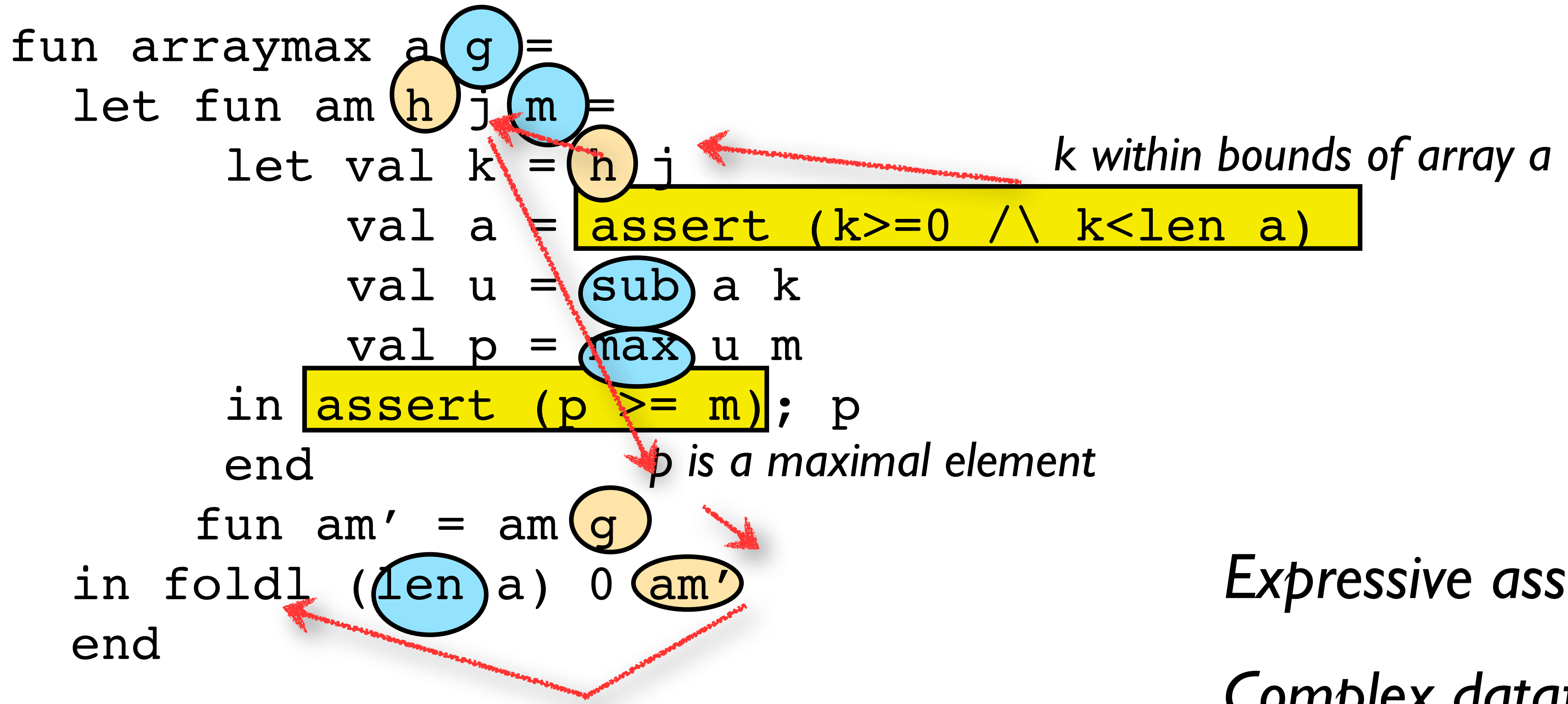
Expressive assertion language

Complex dataflow

Unknown procedures

Challenges

Consider



```
fun arraymax a g =
  let fun am h j m =
         let val k = h j
             val a = assert (k>=0 /\ k<len a)
             val u = sub a k
             val p = max u m
         in assert (p >= m); p
         end
      fun am' = am g
  in foldl (len a) 0 am'
  end
```

k within bounds of array a

p is a maximal element

Expressive assertion language

Complex dataflow

Unknown procedures

Specifications must propagate across procedure boundaries

Liquid Types

```
fun max x y  =
  if x > y
      then x
      else y



val r =
   max a b

val _ = assert (r >= a)
```

Liquid Types

```
fun max x y  = {x : {ν: int | true} → y : {ν : int | true} →{ν : int | ν >= x /\ ν >= y}}
  if x > y
     then x {ν : int | ν = x}
     else y {ν : int | ν = y}
  {ν : int | ν = x}

val r =  {ν >= a /\ ν >= b}
   max a b

val _ = assert (r >= a)
```

Extend standard types with refinement predicates that refer to program variables and primitive functions

Well-typed program implies correctness

Refinement Predicates

```
fun foldn n b f {n : true → b : true → f : {x1: {ν >= 0 /\ ν < n} → x2 : true → true} → true} =
   let fun loop i c {i : {(ν < n) ⇒ (ν >= 0)} → c : true → true} =
      if (i < n) then loop (i+1) (f i c) {true} else c {true} {true}
   in loop 0 b {true}
   end

fun g x {∀y. x : {ν >= 0 /\ ν < y} →{ν >= 0 /\ ν < y}} =
   x {∀y. ν = x}

fun arraymax a {a : {true} → true} =
   let fun am h j m
   {h : {x1 : {ν >= 0 /\ ν < len a} → {ν >= 0 /\ ν < len a}} →
    j : {ν >= 0 /\ ν < len a} →
    m : {true} → true} =
      let val k {ν >= 0 /\ ν < len a} = h j
           val _ {true} = assert (k>=0 /\ k < len a)
         u {true} = sub a k
         p {ν >= m} = max u m
      in assert (p >= m); p {ν = p} end
      fun am’ {x1: {ν >= 0 /\ ν < len a} → x2 : true → true} = am g
   in foldn (len a) 0 am’ {true} end
```

Set of logical qualifiers is potentially quite large

Would like to infer the potential set of qualifiers from context and refine them as appropriate

Refinement Predicates

```
fun foldn n b f {n : true → b : true → f : {x1: {v >= 0 /\ v < n} → x2 : true → true} → true} =
   let fun loop i c {i : {(v < n) ⇒ (v >= 0)} → c : true → true} =
      if (i < n) then loop (i+1) (f i c) {true} else c {true} {true}
   in loop 0 b {true}
   end

fun g x {∀y. x : {v >= 0 /\ v < y} →{v >= 0 /\ v < y}} =
   x {∀y. v = x}

fun arraymax a {a : {true} → true} =
   let fun am h j m
   {h : {x1 : {v >= 0 /\ v < len a} → {v >= 0 /\ v < len a}} →
    j : {v >= 0 /\ v < len a} →
    m : {true} → true} =
      let val k {v >= 0 /\ v < len a} = h j
           val _ {true} = assert (k>=0 /\ k < len a)
         u {true} = sub a k
         p {v >= m} = max u m
      in assert (p >= m); p {v = p} end
      fun am' {x1: {v >= 0 /\ v < len a} → x2 : true → true} = am g
   in foldn (len a) 0 am' {true} end
```

Set of logical qualifiers is potentially quite large

Would like to infer the potential set of qualifiers from context and refine them as appropriate

materialize quantifiers based on constraints introduced in non-lexical scope

Basic Idea

Basic Idea

- Analyze higher-order programs using first-order verification engine
  - ★ Use a modular (inter-procedural) analysis to abstract dataflow through higher-order procedures.
  - ★ First-order verification engine treats higher-order functions as abstract values.
  - ★ Use subtyping to propagate dependent type information across function boundaries

Basic Idea

- Analyze higher-order programs using first-order verification engine
  - ★ Use a modular (inter-procedural) analysis to abstract dataflow through higher-order procedures.
  - ★ First-order verification engine treats higher-order functions as abstract values.
  - ★ Use subtyping to propagate dependent type information across function boundaries
- Counterexample Guided Type Refinement
  - ★ Iteratively refine dependent types using information gleaned from counterexample program paths
  - ★ Use concrete tests to generate and mine dependent type predicates

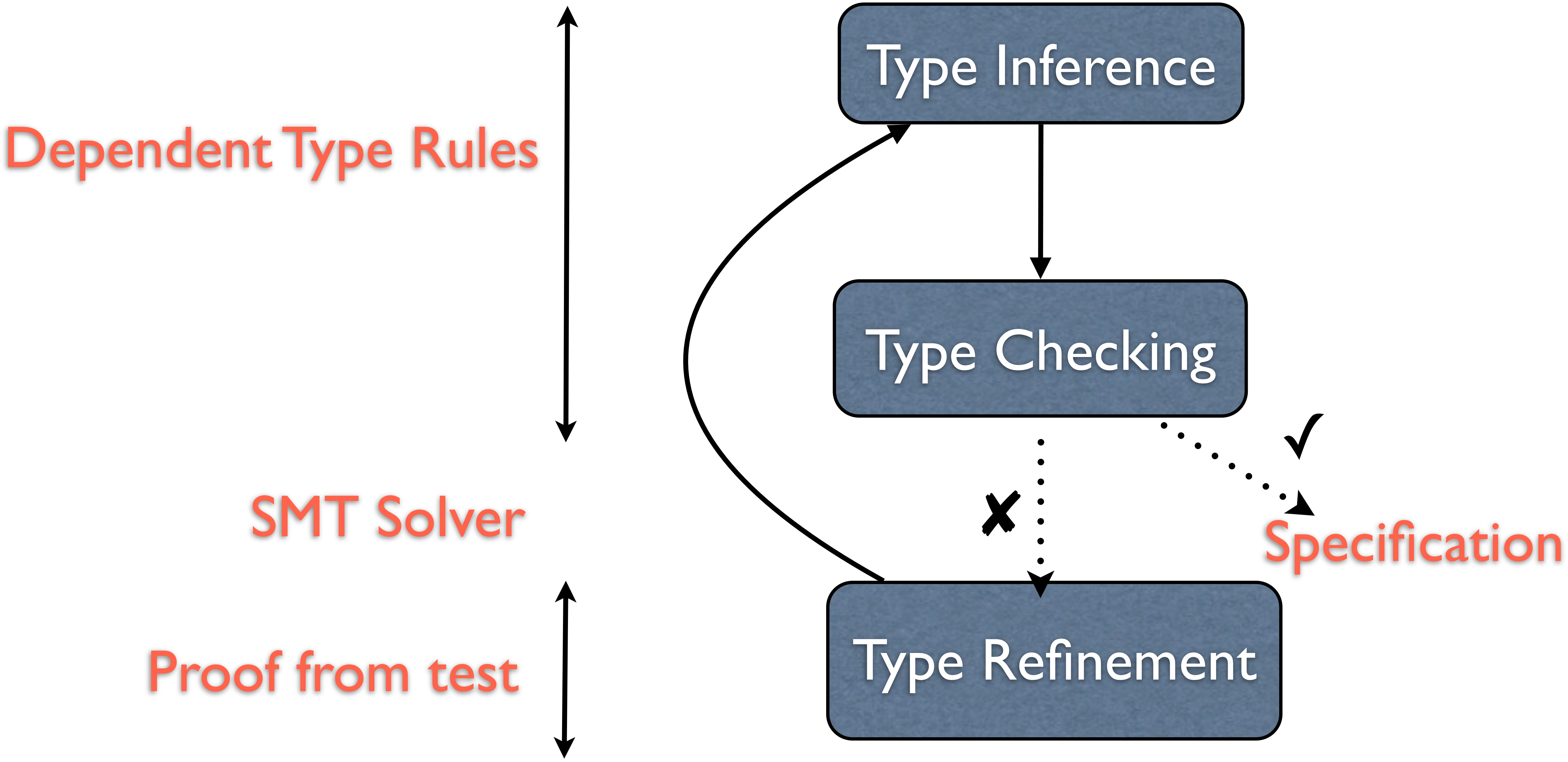
Basic Idea

- Analyze higher-order programs using first-order verification engine
  - ★ Use a modular (inter-procedural) analysis to abstract dataflow through higher-order procedures.
  - ★ First-order verification engine treats higher-order functions as abstract values.
  - ★ Use subtyping to propagate dependent type information across function boundaries
- Counterexample Guided Type Refinement
  - ★ Iteratively refine dependent types using information gleaned from counterexample program paths
  - ★ Use concrete tests to generate and mine dependent type predicates
- Fixpoint algorithm
  - ★ Iterative type refinement based on new verification facts
  - ★ Iterative type-checking based on new qualifier inferences

Basic Idea

- Analyze higher-order programs using first-order verification engine
  - ★ Use a modular (inter-procedural) analysis to abstract dataflow through higher-order procedures.
  - ★ First-order verification engine treats higher-order functions as abstract values.
  - ★ Use subtyping to propagate dependent type information across function boundaries
- Counterexample Guided Type Refinement
  - ★ Iteratively refine dependent types using information gleaned from counterexample program paths
  - ★ Use concrete tests to generate and mine dependent type predicates
- Fixpoint algorithm
  - ★Iterative type refinement based on new verification facts
  - ★Iterative type-checking based on new qualifier inferences
- Integrate these steps as a separate compiler phase

Framework



Dependent Type Rules

SMT Solver

Proof from test

Type Inference

Type Checking

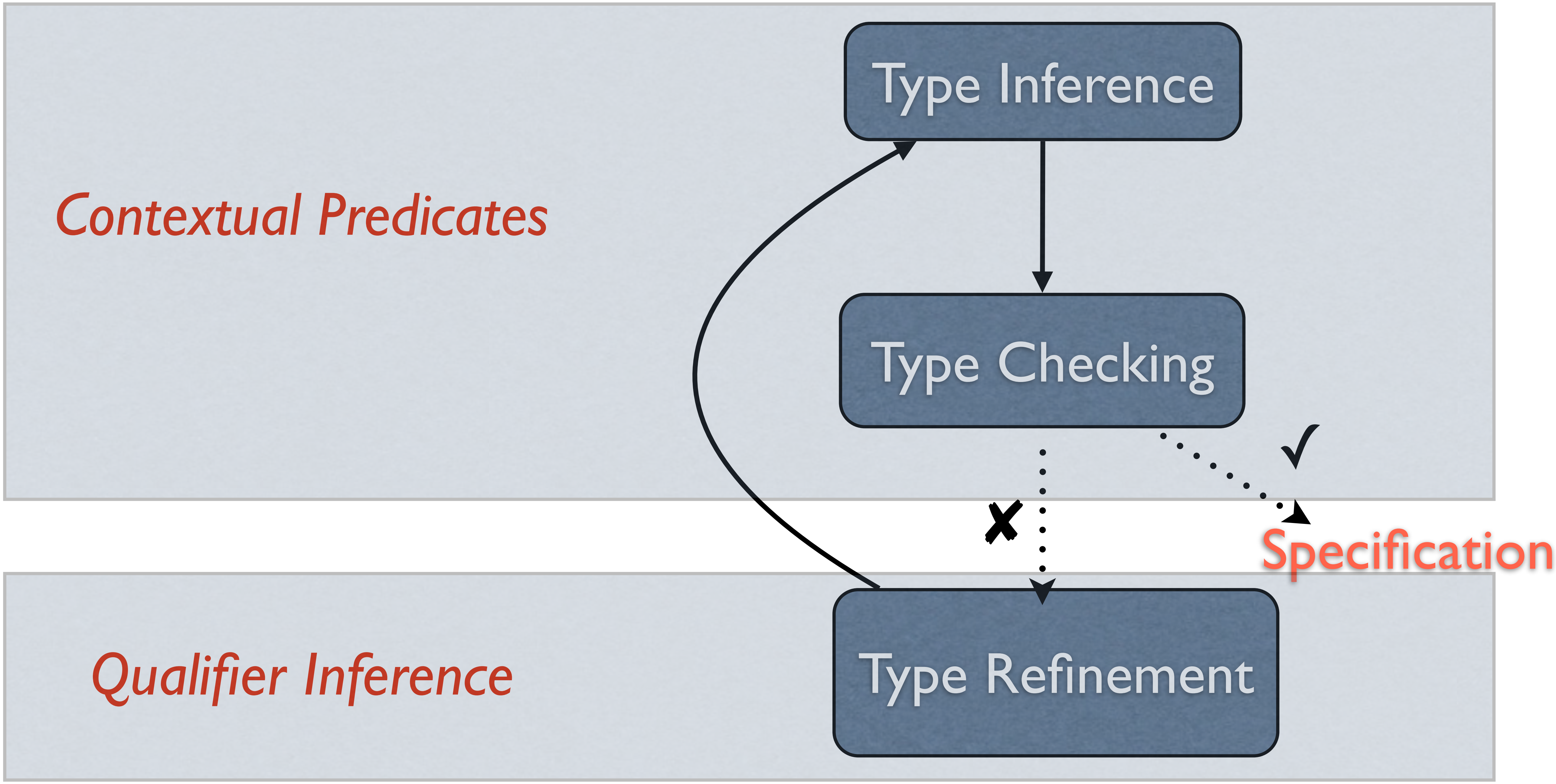
✗

✓

Specification

Type Refinement

Framework


Contextual Predicates
Type Inference
Type Checking
✓
✗
Specification
Qualifier Inference
Type Refinement

Type Inference

```
fun foldn n b f = ...
fun g x = x
fun arraymax a {a : {true} → true} =
   let fun am h j m  =
      let val k {ν >= 0 /\ ν < len a} = h j
           val u = (assert (k>=0 /\ k < len a); sub a k)
           val p = max u m
      in assert (p >= m); p
      end
      fun am’= am g
   in foldn (len a) 0 am’
   end
```

Want to infer type of h in this context

propagate dependent type constraints for
function signatures along subtyping chains

Type Inference

```
fun foldn n b f = ...
fun g x = x
fun arraymax a {a : {true} → true} =
  let fun am h j m  = {h : .... →j : .... → m : {true} → true}
    let val k {ν >= 0 /\ ν < len a} = h j
        val u = (assert (k>=0 /\ k < len a); sub a k)
        val p = max u m
    in assert (p >= m); p
    end
    fun am’= am g
  in foldn (len a) 0 am’
  end
```

Want to infer type of h in this context

propagate dependent type constraints for function signatures along subtyping chains

Type Inference

```
fun foldn n b f = ...
fun g x = x
fun arraymax a {a : {true} → true} =
   let fun am h j m  =
      let val k {v >= 0 /\ v < len a} = h j
           val u = (assert (k>=0 /\ k < len a); sub a k)
           val p = max u m
      in assert (p >= m); p
      end
      fun am’= am g
   in foldn (len a) 0 am’
   end
```

Want to infer type of h in this context

propagate dependent type constraints for
function signatures along subtyping chains

Type Inference

```
fun foldn n b f = ...
fun g x = x
fun arraymax a {a : {true} → true} =
  let fun am h j m  = {h : {... -> {ν >= 0 /\ ν < len a}} →j : .... → m : {true} → tru
    let val k {ν >= 0 /\ ν < len a} = h j
        val u = (assert (k>=0 /\ k < len a); sub a k)
        val p = max u m
    in assert (p >= m); p
    end
    fun am'= am g
  in foldn (len a) 0 am'
  end
```

Want to infer type of h in this context

propagate dependent type constraints for function signatures along subtyping chains

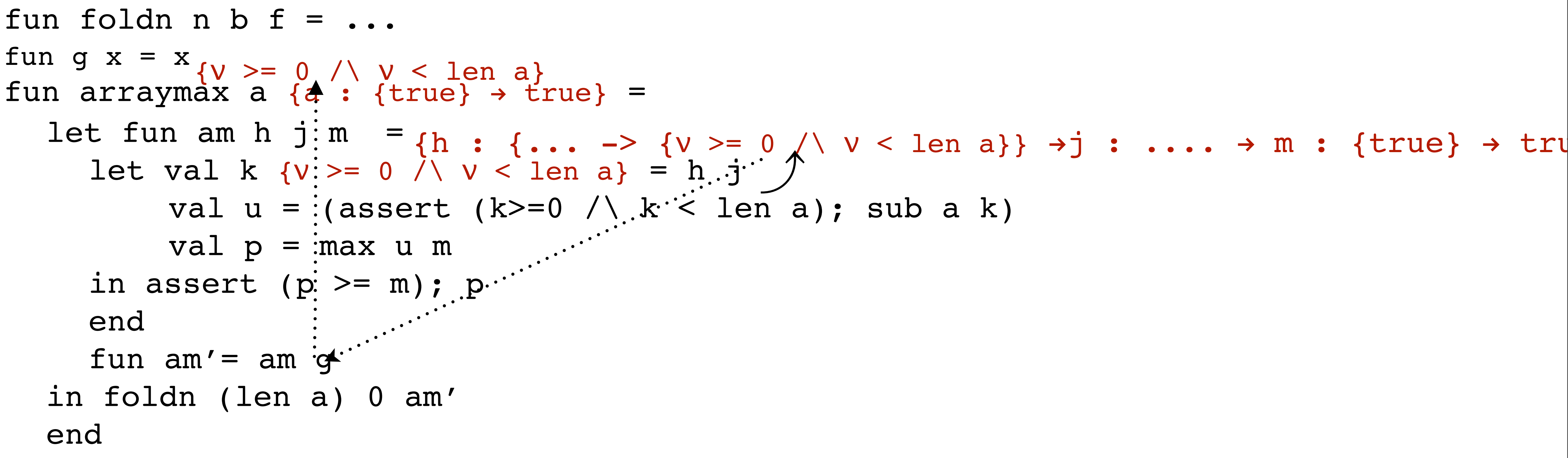
Type Inference

```
fun foldn n b f = ...
fun g x = x
fun arraymax a {a : {true} → true} =
   let fun am h j m  = {h : {... -> {v >= 0 /\ v < len a}} →j : .... → m : {true} → tru
      let val k {v >= 0 /\ v < len a} = h j
           val u = (assert (k>=0 /\ k < len a); sub a k)
           val p = max u m
      in assert (p >= m); p
      end
      fun am'= am g
   in foldn (len a) 0 am'
   end
```

Want to infer type of h in this context

propagate dependent type constraints for
function signatures along subtyping chains

Type Inference

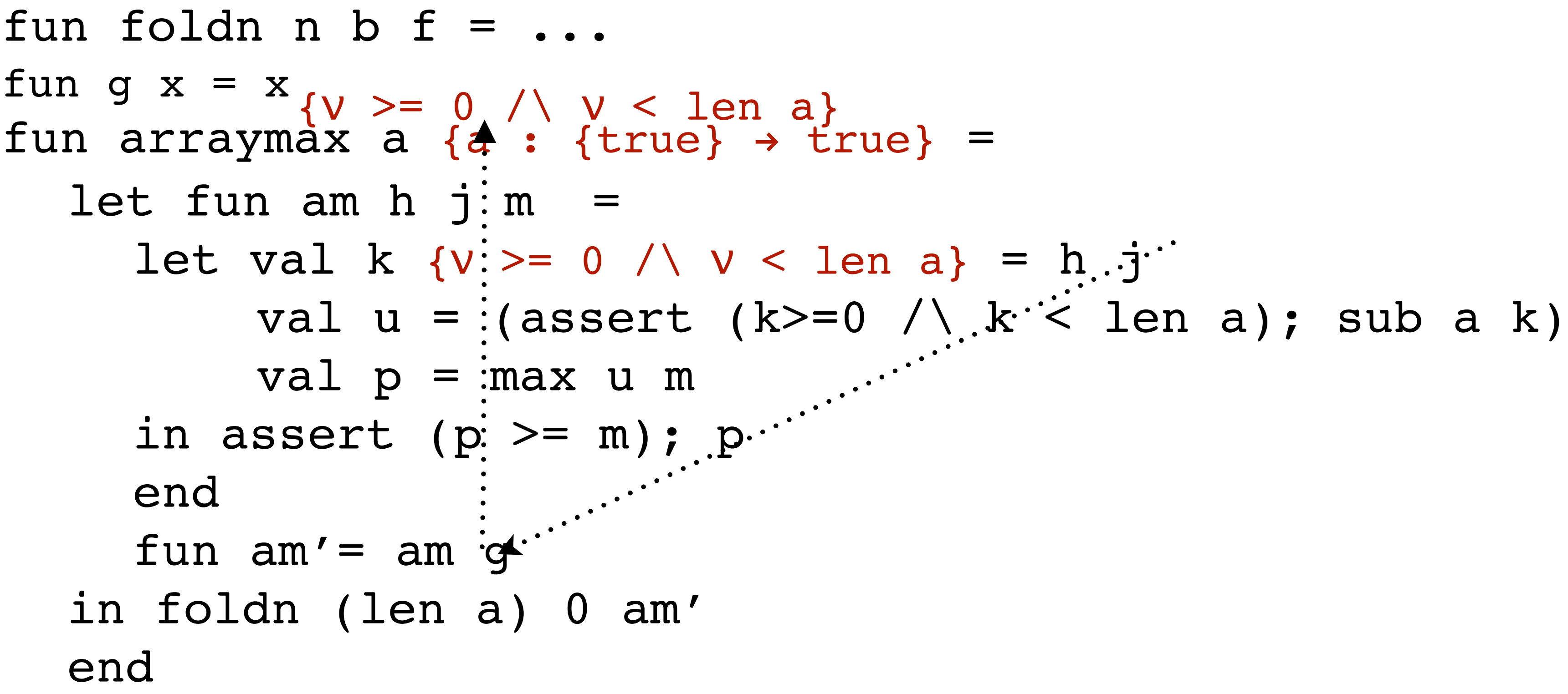


```
fun foldn n b f = ...
fun g x = x
                     {v >= 0 /\ v < len a}
fun arraymax a {a : {true} → true} =
   let fun am h j m  = {h : {... -> {v >= 0 /\ v < len a}} →j : .... → m : {true} → tru
      let val k {v >= 0 /\ v < len a} = h j
           val u = (assert (k>=0 /\ k < len a); sub a k)
           val p = max u m
      in assert (p >= m); p
      end
      fun am'= am g
   in foldn (len a) 0 am'
   end
```

Want to infer type of h in this context

propagate dependent type constraints for
function signatures along subtyping chains

Type Inference



```
fun foldn n b f = ...
fun g x = x
            {v >= 0 /\ v < len a}
fun arraymax a {a : {true} → true} =
   let fun am h j m  =
      let val k {v >= 0 /\ v < len a} = h j
           val u = (assert (k>=0 /\ k < len a); sub a k)
           val p = max u m
      in assert (p >= m); p
      end
      fun am'= am g
   in foldn (len a) 0 am'
   end
```

Want to infer type of h in this context

propagate dependent type constraints for
function signatures along subtyping chains

Type Inference

```
fun foldn n b f = ...
fun g x = x
          {v >= 0 /\ v < len a}
fun arraymax a {a : {true} → true} =
  let fun am h j m  =
    let val k {v >= 0 /\ v < len a} = h j
        val u = (assert (k>=0 /\ k < len a); sub a k)
        val p = max u m
    in assert (p >= m); p
    end
    fun am’= am g
  in foldn (len a) 0 am’
  end
```

Want to infer type of h in this context

propagate dependent type constraints for
function signatures along subtyping chains

Type Inference

```
fun foldn n b f = ...
fun g x = x {v >= 0 /\ v < len a}
fun arraymax a {a : {true} → true} =
  let fun am h j m  = {h : {x1: {v >= 0 /\ v < len a} -> {v >= 0 /\ v < len a}} →j : ...
    let val k {v >= 0 /\ v < len a} = h j
                                                            → m : {true} → true}
        val u = (assert (k>=0 /\ k < len a); sub a k)
        val p = max u m
    in assert (p >= m); p
    end
    fun am'= am g
  in foldn (len a) 0 am'
  end
```

Want to infer type of h in this context

propagate dependent type constraints for
function signatures along subtyping chains

Type Checking

- Construct a verification condition (VC) as a first-order formula from inferred dependent types
  - ★ Typing rules track path conditions that are encoded in the structure of the VC
- The condition to be verified by an SMT solver is the negation of the VC
  - ★ unsat => type checking successful
  - ★ sat => additional strengthening required to derive a consistent specification

Example

```
fun foldn n b f
  {n : true → b : true → f : {x1: {ν >= 0 /\ ν < n} → x2 : true → true} → true} =
  let fun loop i c
    {i : true → c : true → true} =
    if (i < n) then loop (i+1) (f i c) else c
  in loop 0 b end
```

Example

```
fun foldn n b f
  {n : true → b : true → f : {x1: {ν >= 0 /\ ν < n} → x2 : true → true} → true} =
  let fun loop i c
    {i : true → c : true → true} =
    if (i < n) then loop (i+1) (f i c) else c
  in loop 0 b end
```

Want to type-check the call

Example

```
fun foldn n b f
  {n : true → b : true → f : {x1: {ν >= 0 /\ ν < n} → x2 : true → true} → true} =
  let fun loop i c
    {i : true → c : true → true} =
    if (i < n) then loop (i+1) (f i c) else c
  in loop 0 b end
```

Want to type-check the call

Expected constraint on argument i is: `{ν >= 0 /\ ν < n}`

Example

```
fun foldn n b f
  {n : true → b : true → f : {x1: {ν >= 0 /\ ν < n} → x2 : true → true} → true} =
  let fun loop i c
    {i : true → c : true → true} =
    if (i < n) then loop (i+1) (f i c) else c
  in loop 0 b end
```

Want to type-check the call

Expected constraint on argument i is: `{ν >= 0 /\ ν < n}`

Need to solve: ¬`(i < n ) /\ (ν = i) => {ν >= 0 /\ ν < n}` to strengthen invariants

Example

```
fun foldn n b f
  {n : true → b : true → f : {x1: {ν >= 0 /\ ν < n} → x2 : true → true} → true} =
  let fun loop i c
    {i : true → c : true → true} =
    if (i < n) then loop (i+1) (f i c) else c
  in loop 0 b end
```

Want to type-check the call

Expected constraint on argument i is: `{ν >= 0 /\ ν < n}`

Need to solve: ¬`(i < n ) /\ (ν = i) => {ν >= 0 /\ ν < n}` to strengthen invariants

App

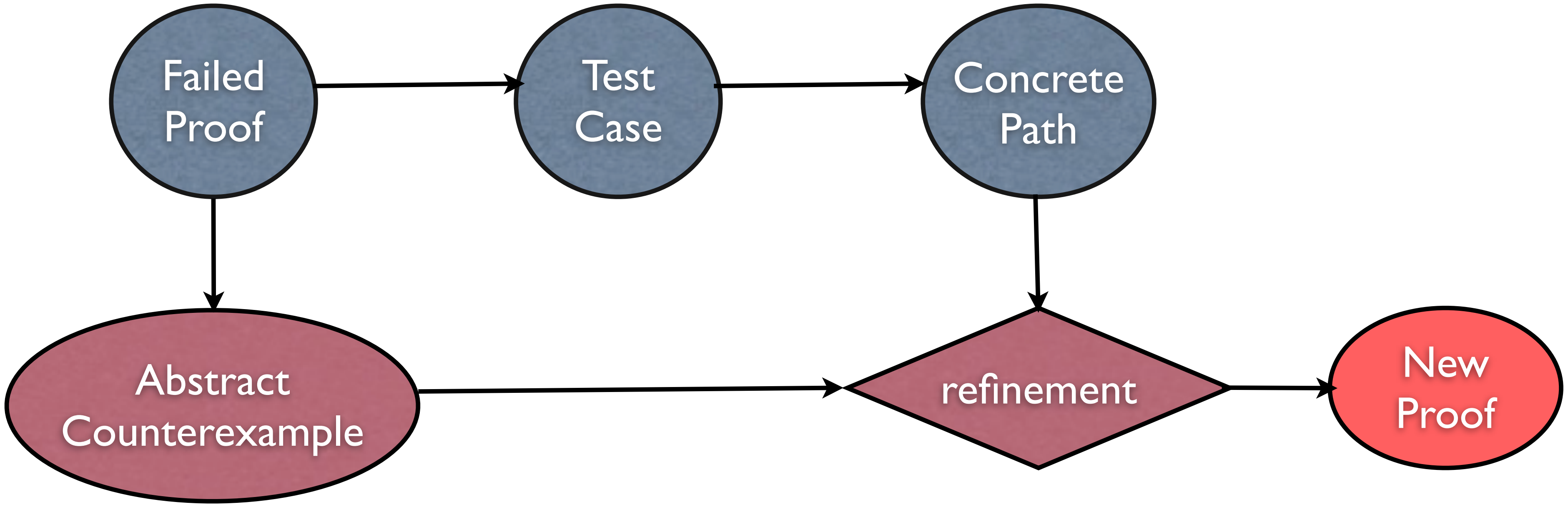
$$\frac{\Gamma \vdash v_2 : P_x \quad \Gamma \vdash e_f : (x : P_x \rightarrow P)}{\Gamma \vdash e_f(v_2) : [v_2/x]P}$$

IF

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash P \quad \Gamma; e_1 \vdash e_2 : P \quad \Gamma; \neg e_1 \vdash e_3 : P}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : P}$$

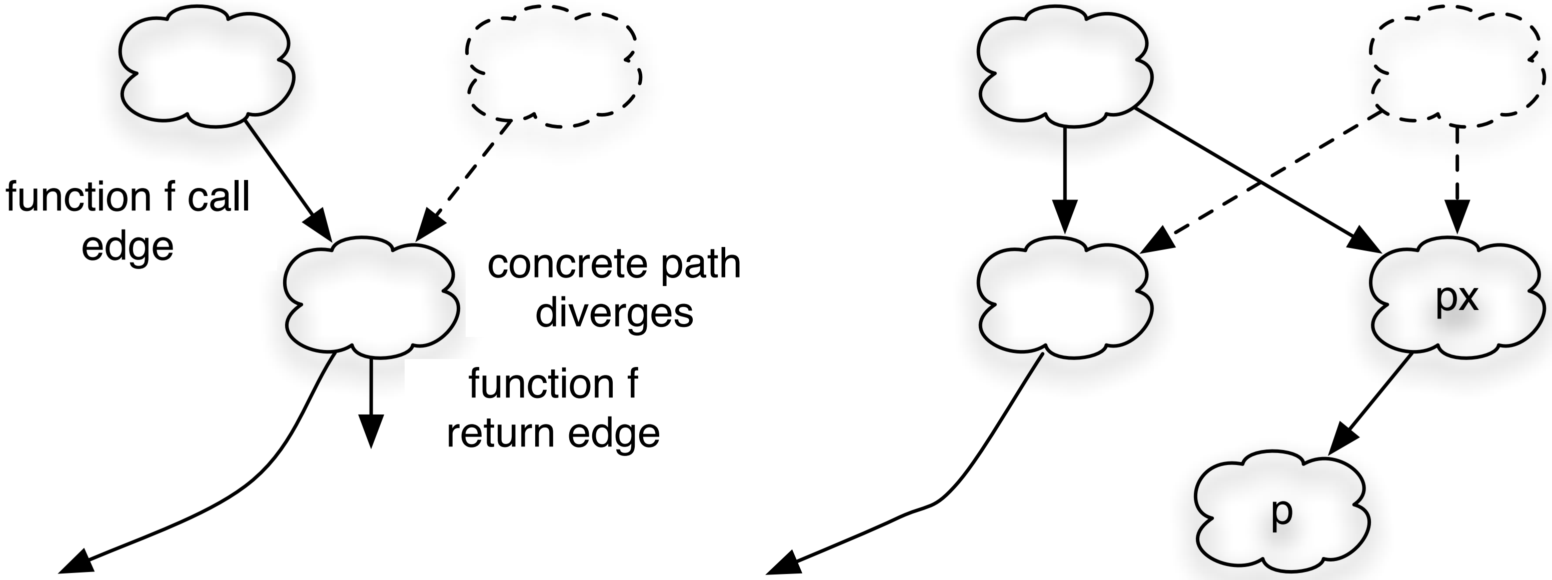
Type Refinement

- Type Refinement is used to augment the set of qualifiers
  - ★ We could re-analyze the body of called functions along a counterexample (inter-procedural) path from a call-site.
  - ★ But, within a compiler, can explore concrete paths fairly easily
    - ✦ Internally, build a compile-test-run loop over an optimized well-typed IR
- Use lightweight testing to determine where and how to refine the type system. (Proofs-from-tests aka Dash)



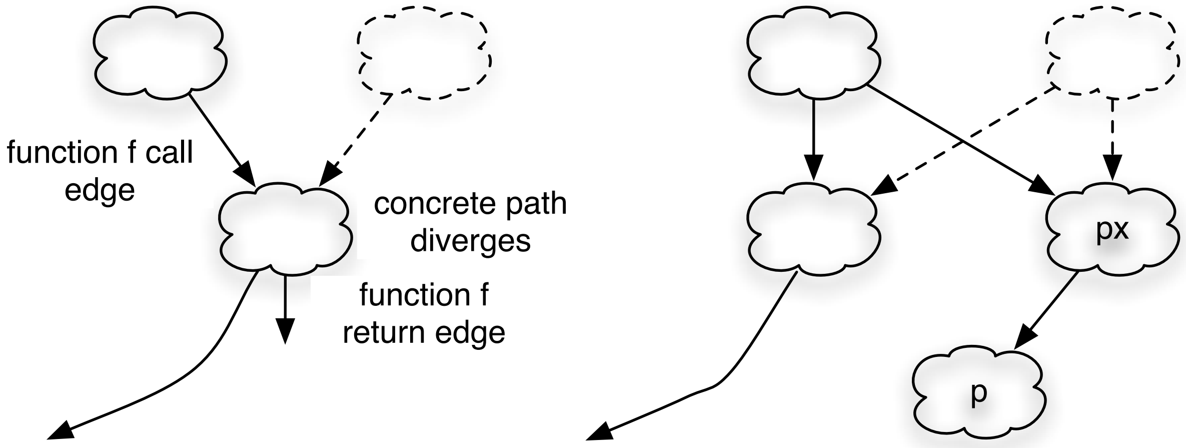
- Testing provides a concrete witness to guide how existing invariants can be strengthened.

Directed Testing



px is not implied by the environment

Directed Testing



px is not implied by the environment

- If environment at a call-site
  - ★ implies the called function’s pre-condition, then original counterexample will no longer be reported if we strengthen called function’s post-condition and compute weakest precondition.
  - ★ does not imply the called function’s precondition, then must strengthen function’s pre-condition to force divergence.
- Goal: either eliminate the counterexample by strengthening callee’s post-condition, or direct test case execution to converge to counterexample, strengthening caller’s pre-condition

Example

```
let fun g x = x
    fun f x = if x < 2
                 then let s = g x
                      in assert (s <= 0)
                      end
                 else ()
  ...
```

Example

```
let fun g x = x
    fun f x = if x < 2
                 then let s = g x
                      in assert (s <= 0)
                      end
                 else ()
  ...
```

Initially, solve:

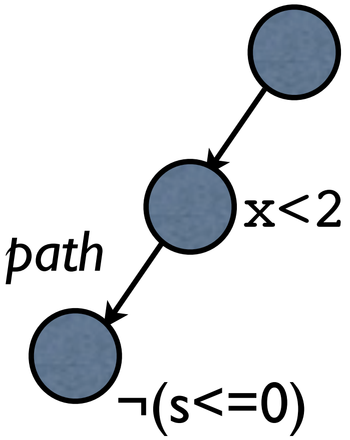
$$\neg VC : \neg(\mathrm{x} < 2 \Rightarrow s \leq 0)$$

Example

```
let fun g x = x
    fun f x = if x < 2
                 then let s = g x
                      in assert (s <= 0)
                      end
                 else ()
  ...
```

Initially, solve:

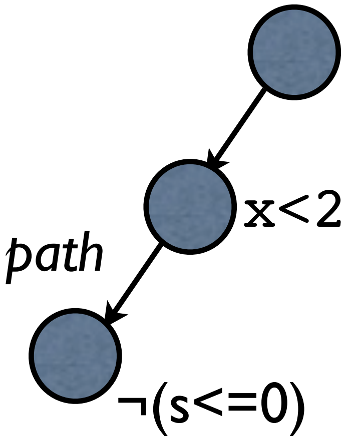
$$\neg VC : \neg(\mathrm{x} < 2 \Rightarrow s \leq 0)$$



counterexample path

Example

```
let fun g x = x
    fun f x = if x < 2
                 then let s = g x
                      in assert (s <= 0)
                      end
                 else ()
  ...
```



Initially, solve:

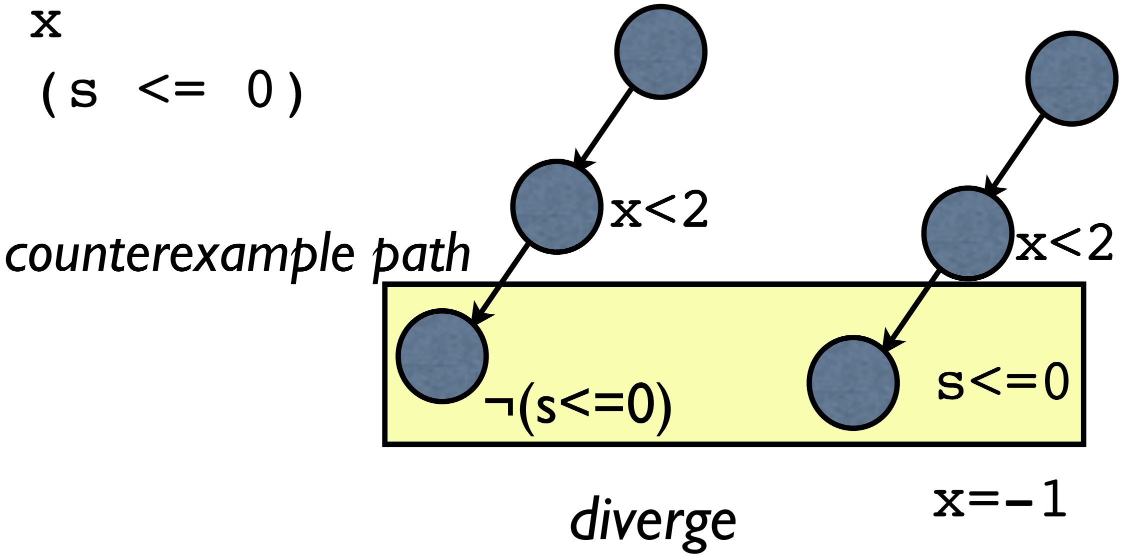
$$\neg VC : \neg(\mathrm{x} < 2 \Rightarrow s \leq 0)$$

Generate a concrete test supplying

$$\mathrm{x} = -1$$

Example

```
let fun g x = x
    fun f x = if x < 2
                then let s = g x
                     in assert (s <= 0)
                     end
                else ()
  ...
```



Initially, solve:

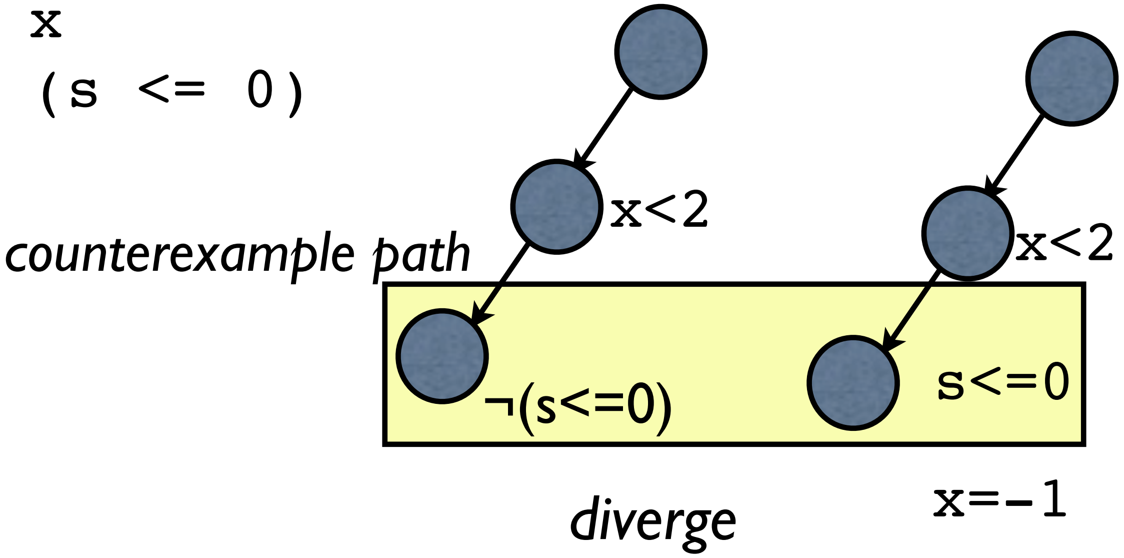
$$\neg VC : \neg(\mathrm{x} < 2 \Rightarrow s \leq 0)$$

Generate a concrete test supplying

$$\mathrm{x} = -1$$

Example

```
let fun g x = x
    fun f x = if x < 2
                 then let s = g x
                      in assert (s <= 0)
                      end
                 else ()
  . . .
```



Initially, solve:

$$\neg VC : \neg(\mathtt{x} < 2 \Rightarrow s \leq 0)$$

Generate a concrete test supplying

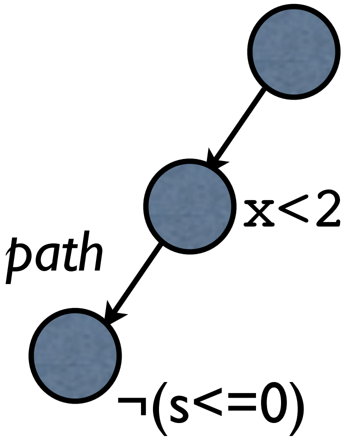
$$\mathtt{x} = -1$$

Strengthen g's type which is initially $\{\mathtt{true}\} \rightarrow \{\mathtt{true}\}$

$$\{\nu \leq 0\} \rightarrow \{\nu \leq 0\}$$

Example

```
let fun g x = x
    fun f x = if x < 2
                 then let s = g x
                      in assert (s <= 0)
                      end
                 else ()
  ...
```



counterexample path

Initially, solve:

$\neg VC : \neg(\mathrm{x} < 2 \Rightarrow s \leq 0)$

Generate a concrete test supplying

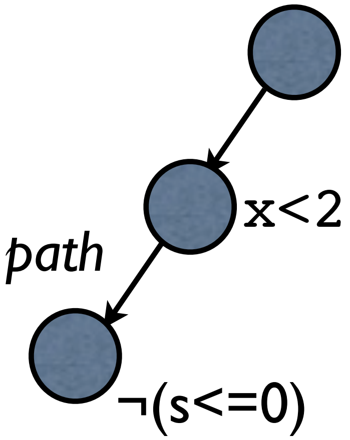
$\mathrm{x} = -1$

Strengthen g's type which is initially $\{\mathtt{true}\} \rightarrow \{\mathtt{true}\}$

$\{\nu \leq 0\} \rightarrow \{\nu \leq 0\}$

Example

```
let fun g x = x
    fun f x = if x < 2
                 then let s = g x
                      in assert (s <= 0)
                      end
                 else ()
  ...
```



Initially, solve:

$$\neg VC : \neg(\mathrm{x} < 2 \Rightarrow s \leq 0)$$

Generate a concrete test supplying

$$\mathrm{x} = -1$$

Strengthen g's type which is initially $\{\mathtt{true}\} \rightarrow \{\mathtt{true}\}$

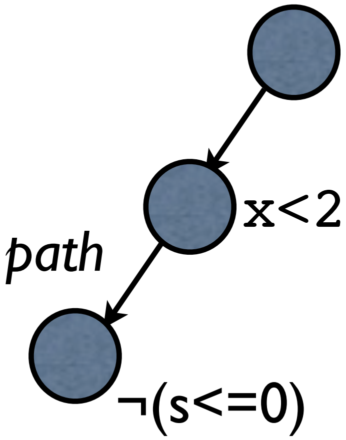
$$\{\nu \leq 0\} \rightarrow \{\nu \leq 0\}$$

Generate new verification condition based on g's refinement

$$\neg VC : \neg((\mathrm{x} < 2 \wedge (x \leq 0 \Rightarrow s \leq 0) \Rightarrow (s \leq 0))$$

Example

```
let fun g x = x
    fun f x = if x < 2
                 then let s = g x
                      in assert (s <= 0)
                      end
                 else ()
  ...
```



counterexample path

Initially, solve:

$$\neg VC : \neg(\mathrm{x} < 2 \Rightarrow s \leq 0)$$

Generate a concrete test supplying

$$\mathrm{x} = -1$$

Strengthen g's type which is initially $\{\texttt{true}\} \rightarrow \{\texttt{true}\}$

$$\{\nu \leq 0\} \rightarrow \{\nu \leq 0\}$$

Generate new verification condition based on g's refinement

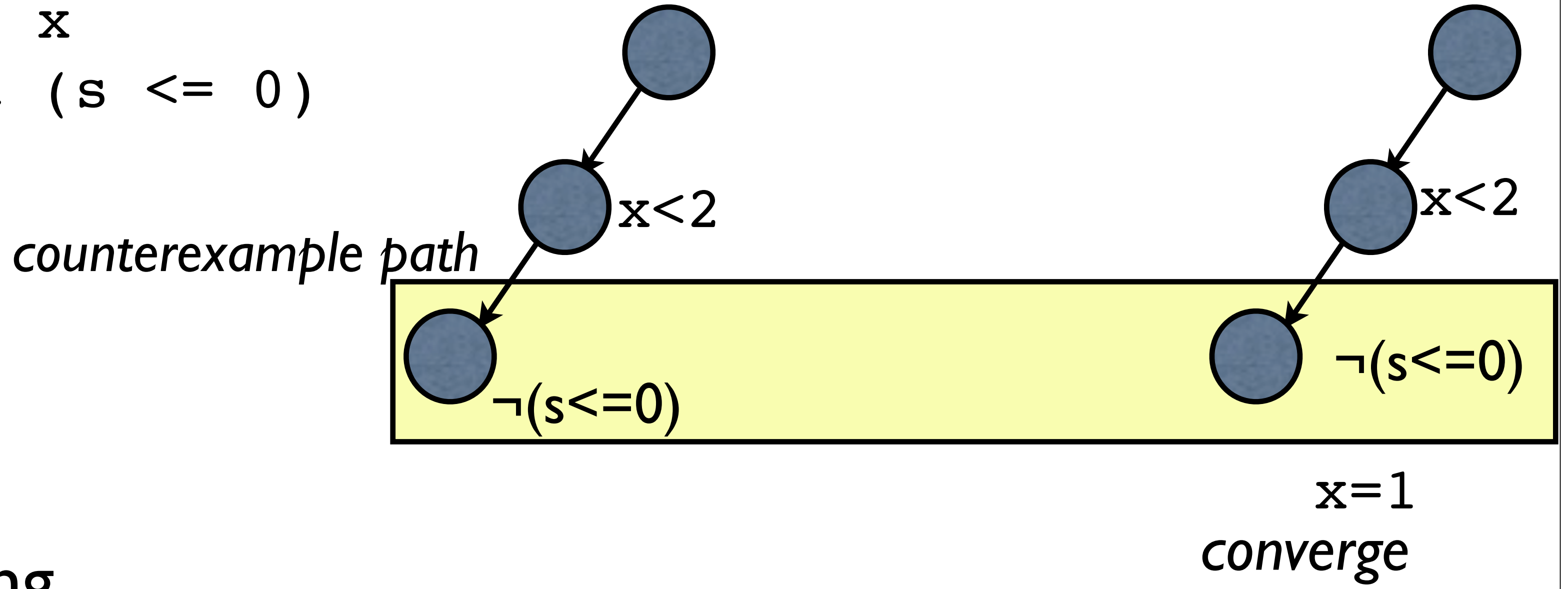
$$\neg VC : \neg((\mathrm{x} < 2 \wedge (x \leq 0 \Rightarrow s \leq 0) \Rightarrow (s \leq 0))$$

Generate new concrete test supplying

$$\mathrm{x} = 1$$

Example

```
let fun g x = x
    fun f x = if x < 2
                 then let s = g x
                      in assert (s <= 0)
                      end
                 else ()
  ...
```



Initially, solve:

$$\neg VC : \neg(\mathrm{x} < 2 \Rightarrow s \leq 0)$$

Generate a concrete test supplying

$$\mathrm{x} = -1$$

Strengthen g's type which is initially $\{\mathtt{true}\} \rightarrow \{\mathtt{true}\}$

$$\{\nu \leq 0\} \rightarrow \{\nu \leq 0\}$$

Generate new verification condition based on g's refinement

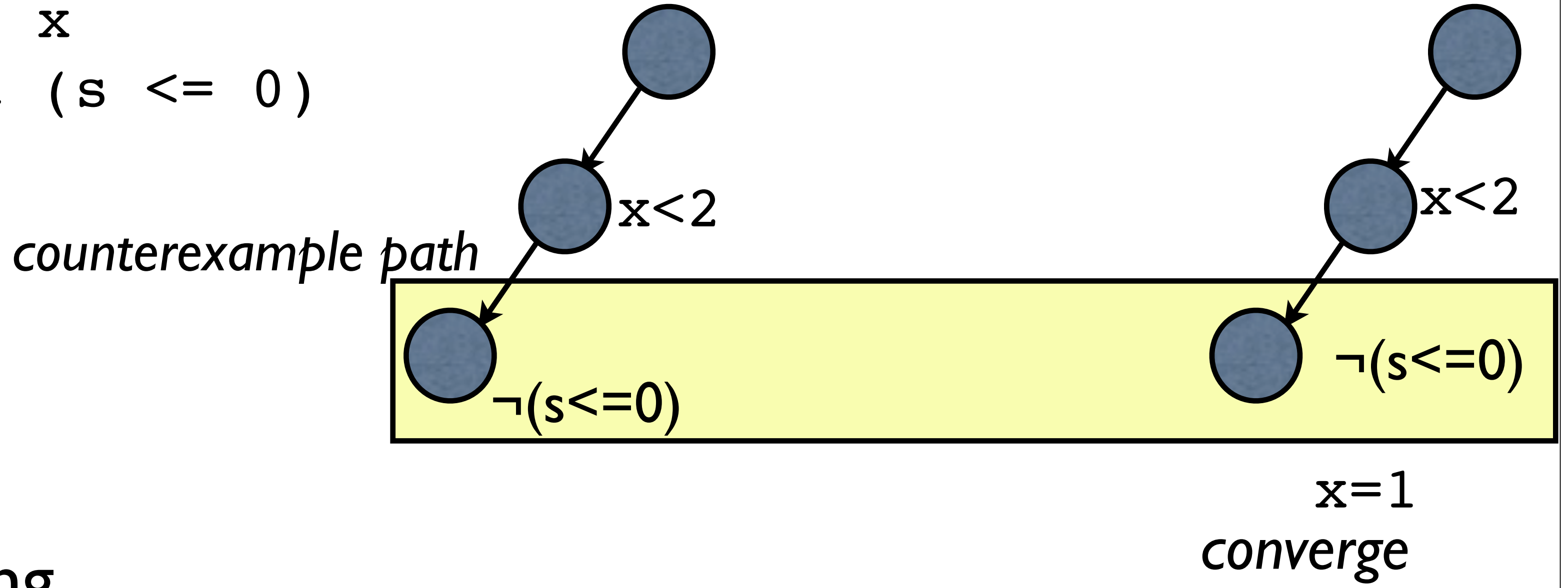
$$\neg VC : \neg((\mathrm{x} < 2 \wedge (x \leq 0 \Rightarrow s \leq 0) \Rightarrow (s \leq 0))$$

Generate new concrete test supplying

$$\mathrm{x} = 1$$

Example

```
let fun g x = x
    fun f x = if x < 2
                 then let s = g x
                      in assert (s <= 0)
                      end
                 else ()
  ...
```



Initially, solve:

$$\neg VC : \neg(\mathrm{x} < 2 \Rightarrow s \leq 0)$$

Generate a concrete test supplying

$$\mathrm{x} = -1$$

Strengthen g's type which is initially $\{\mathtt{true}\} \rightarrow \{\mathtt{true}\}$

$$\{\nu \leq 0\} \rightarrow \{\nu \leq 0\}$$

Generate new verification condition based on g's refinement

$$\neg VC : \neg((\mathrm{x} < 2 \wedge (x \leq 0 \Rightarrow s \leq 0) \Rightarrow (s \leq 0))$$

Generate new concrete test supplying

$$\mathrm{x} = 1$$

Strengthen pre-condition for f to $\{\mathrm{x} : \nu \leq 0\}$to eliminate the counter-example

Pre-Condition Refinement

```
fun foldn n b f
  {n : true → b : true → f : {x1: {v >= 0 /\ v < n} → x2 : true → true} → true} =
  let fun loop i c {i : true → c : true → true} =
    if (i < n) then loop (i+1) (f i c) else c
  in loop 0 b
  end
```

Pre-Condition Refinement

```
fun foldn n b f
  {n : true → b : true → f : {x1: {ν >= 0 /\ ν < n} → x2 : true → true} → true} =
  let fun loop i c {i : true → c : true → true} =
    if (i < n) then loop (i+1) (f i c) else c
  in loop 0 b
  end
```

Type-checking loop involves solving:

$$\neg VC : \neg\{(\mathtt{i} < \mathtt{n} \wedge (\nu = \mathtt{i}) \Rightarrow (\nu \geq 0 \wedge \nu < \mathtt{n})\}$$

Pre-Condition Refinement

```
fun foldn n b f
  {n : true → b : true → f : {x1: {ν >= 0 /\ ν < n} → x2 : true → true} → true} =
  let fun loop i c {i : true → c : true → true} =
    if (i < n) then loop (i+1) (f i c) else c
  in loop 0 b
  end
```

Type-checking loop involves solving:

$$\neg VC : \neg\{(\mathtt{i} < \mathtt{n} \wedge (\nu = \mathtt{i}) \Rightarrow (\nu \geq 0 \wedge \nu < \mathtt{n})\}$$

A concrete test case: { i = -1, n = 0 }

Test case run does not diverge from counterexample path

Pre-Condition Refinement

```
fun foldn n b f
  {n : true → b : true → f : {x1: {ν >= 0 /\ ν < n} → x2 : true → true} → true} =
  let fun loop i c {i : true → c : true → true} =
    if (i < n) then loop (i+1) (f i c) else c
  in loop 0 b
  end
```

Type-checking loop involves solving:

$$\neg VC : \neg\{(\mathtt{i} < \mathtt{n} \wedge (\nu = \mathtt{i}) \Rightarrow (\nu \geq 0 \wedge \nu < \mathtt{n})\}$$

A concrete test case: { i = -1, n = 0 }

Test case run does not diverge from counterexample path

Weakest pre-condition generation: $\mathtt{i} < \mathtt{n} \Rightarrow (\mathtt{i} \geq 0)$

Pre-Condition Refinement

```
fun foldn n b f
  {n : true → b : true → f : {x1: {ν >= 0 /\ ν < n} → x2 : true → true} → true} =
  let fun loop i c {i : true → c : true → true} =
    if (i < n) then loop (i+1) (f i c) else c
  in loop 0 b
  end
```

Type-checking loop involves solving:

$$\neg VC : \neg\{(\mathtt{i} < \mathtt{n} \wedge (\nu = \mathtt{i}) \Rightarrow (\nu \geq 0 \wedge \nu < \mathtt{n})\}$$

A concrete test case: { i = -1, n = 0 }

Test case run does not diverge from counterexample path

Weakest pre-condition generation: $\mathtt{i} < \mathtt{n} \Rightarrow (\mathtt{i} \geq 0)$

Strengthen dependent type for loop:

$$\{\mathtt{i} : \{(\nu < \mathtt{n}) \Rightarrow (\nu \geq 0)\} \rightarrow \mathtt{c} : \mathrm{true} \rightarrow \mathrm{true}$$

Post-Condition Refinement

```
fun arraymax a g =
  let fun am h j m
    {h : {x1 : {true}→{ν >= 0 /\ ν < len a}} →j : {true} → m : {true} → true} =
    let val k {ν >= 0 /\ ν < len a} = h j
        val u {true} = sub a k
        val p {true}     = max u m
    in assert (p >=m); p {ν=p} end {true}
    fun am’ = am g
  in fold (len a) 0 am’ end
```

Post-Condition Refinement

```
fun arraymax a g =
  let fun am h j m
    {h : {x1 : {true}→{ν >= 0 /\ ν < len a}} →j : {true} → m : {true} → true} =
    let val k {ν >= 0 /\ ν < len a} = h j
        val u {true} = sub a k
        val p {true}     = max u m
    in assert (p >=m); p {ν=p} end {true}
    fun am’ = am g
  in fold (len a) 0 am’ end
```

Type-checking function am involves solving:

$$\neg VC : \neg\{(\mathtt{k} \geq 0) \wedge (\mathtt{k} < \mathtt{len\ a}) \Rightarrow (\mathtt{p} \geq \mathtt{m})\}$$

Post-Condition Refinement

```
fun arraymax a g =
  let fun am h j m
    {h : {x1 : {true}→{ν >= 0 /\ ν < len a}} →j : {true} → m : {true} → true} =
    let val k {ν >= 0 /\ ν < len a} = h j
        val u {true} = sub a k
        val p {true}    = max u m
    in assert (p >=m); p {ν=p} end {true}
    fun am’ = am g
  in fold (len a) 0 am’ end
```

Type-checking function am involves solving:

$$\neg VC : \neg\{(\texttt{k} \geq 0) \wedge (\texttt{k} < \texttt{len a}) \Rightarrow (\texttt{p} \geq \texttt{m})\}$$

A possible test case generated by the solver:

$$\{\texttt{h} = \lambda x.0, \texttt{j} = 0, \texttt{m} = 0, \texttt{a} = \text{Array}[0]\}$$

Post-Condition Refinement

```
fun arraymax a g =
  let fun am h j m
    {h : {x1 : {true}→{ν >= 0 /\ ν < len a}} →j : {true} → m : {true} → true} =
    let val k {ν >= 0 /\ ν < len a} = h j
        val u {true} = sub a k
        val p            = max u m
    in assert (p >=m); p {ν=p} end {true}
    fun am’ = am g
  in fold (len a) 0 am’ end
```

Type-checking function am involves solving:

$$\neg VC : \neg\{(\mathtt{k} \geq 0) \wedge (\mathtt{k} < \mathtt{len\ a}) \Rightarrow (\mathtt{p} \geq \mathtt{m})\}$$

A possible test case generated by the solver:

$$\{\mathtt{h} = \lambda x.0, \mathtt{j} = 0, \mathtt{m} = 0, \mathtt{a} = \mathsf{Array}[0]\}$$

Post-Condition Refinement

```
fun arraymax a g =
  let fun am h j m
    {h : {x1 : {true}→{ν >= 0 /\ ν < len a}} →j : {true} → m : {true} → true} =
    let val k {ν >= 0 /\ ν < len a} = h j
        val u {true} = sub a k
        val p          = max u m
    in assert (p >=m); p {ν=p} end {true}
    fun am′ = am g
  in fold (len a) 0 am′ end
```

Type-checking function am involves solving:

$$\neg VC : \neg\{(\mathtt{k} \geq 0) \wedge (\mathtt{k} < \mathtt{len\ a}) \Rightarrow (\mathtt{p} \geq \mathtt{m})\}$$

A possible test case generated by the solver:

$$\{\mathtt{h} = \lambda x.0, \mathtt{j} = 0, \mathtt{m} = 0, \mathtt{a} = \text{Array}[0]\}$$

Concrete test diverges at call to max

Strengthen post-condition for max to $\{\nu \geq \mathtt{y}\}$ *where* $\mathtt{y}$ *is the second formal parameter to max*

Post-Condition Refinement

```
fun arraymax a g =
  let fun am h j m
    {h : {x1 : {true}→{ν >= 0 /\ ν < len a}} →j : {true} → m : {true} → true} =
    let val k {ν >= 0 /\ ν < len a} = h j
        val u {true} = sub a k
        val p { ν >= m }= max u m
    in assert (p >=m); p {ν=p} end {true}
    fun am’ = am g
  in fold (len a) 0 am’ end
```

Type-checking function am involves solving:

$$\neg VC : \neg\{(\mathtt{k} \geq 0) \wedge (\mathtt{k} < \mathtt{len\ a}) \Rightarrow (\mathtt{p} \geq \mathtt{m})\}$$

A possible test case generated by the solver:

$$\{\mathtt{h} = \lambda x.0, \mathtt{j} = 0, \mathtt{m} = 0, \mathtt{a} = \text{Array}[0]\}$$

Concrete test diverges at call to max

Strengthen post-condition for max to $\{\nu \geq \mathtt{y}\}$ *where* $\mathtt{y}$ *is the second formal parameter to max*

Algorithmic Type Inference

- Type inference
  - *Rule Generation*
    - Encode first-order formula from liquid type rules
  - *Constraint Propagation*
    - Subtyping chains
- Type checking
  - Verification condition generation and solve
- Type Refinement
  - Directed testing to generate functional precondition and postcondition constraints

Algorithmic Type Inference

- Type inference
  - *Rule Generation*
    - Encode first-order formula from liquid type rules
  - *Constraint Propagation*
    - Subtyping chains
- Type checking
  - Verification condition generation and solve
- Type Refinement
  - Directed testing to generate functional precondition and postcondition constraints

Language

Expressions

similar to Core ML intermediate representation in MLton

$$
\begin{aligned}
e' ::= & \\
& |\nu, v, x \in Var \\
& |c \\
& |\lambda x.e' \\
& |\text{if } e' \text{ then } e' \text{ else } e' \\
& |\text{let } x = e' \text{ in } e' \\
& |\text{assert } e'; e' \\
e_f ::= & \\
& |v \\
& |e_f \ v \\
e ::= & \\
& |[\Lambda\alpha]e' \\
& |[\gamma]e'
\end{aligned}
$$

Base Types

$$
\begin{aligned}
B ::= & \\
& |int \\
& |bool
\end{aligned}
$$

Dependent Types, Schemas

$$
\begin{aligned}
P ::= & \\
& |\{\nu : B|e\} \\
& |P \rightarrow P \\
& |\alpha \\
& |\forall\alpha.\forall x.P \\
& |P \wedge P
\end{aligned}
$$

Intersection types to distinguish predicates at different call-sites

Materialization to express non-lexical constraints

Derived from weakest precondition generation and propagated along subtyping chains

Dependent type encoding

- Liquid type rules provide well-formedness and subtyping constraints on dependent types
- Encode dependent types of local variables and terms using program terms and pre/post-conditions of functions
  - Local dependent types encode intra-procedural path information
    - Extract program error path from a (weak) dependent type system
- Abstract dependent types of functions
  - Initially, all argument and return of functions are abstracted to true
  - Strengthen pre- and post-conditions based on the program error path

Encoding Type Rules

- Encode type rules as constraints over first-order formula
- Encoding should preserve {path,context} sensitivity
- Facilitate test generation and predicate refinement

Example:

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash P \qquad \Gamma; e_1 \vdash e_2 : P \qquad \Gamma; \neg e_1 \vdash e_3 : P}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : P}$$

Generate constraint for P from this rule as:

$$P = \mathcal{C}(e_1) \Rightarrow \mathcal{C}(e_2) \wedge \neg\mathcal{C}(e_1) \Rightarrow \mathcal{C}(e_3)$$

Constraint relates value of guard to specific branches

$\mathcal{C}(e)$ defines the constraints generated for e based on the structure of the liquid type rules

Verification Condition Generation

Verification Condition Generation

Fun

$$\frac{\Gamma \vdash x : P_x \rightarrow P \quad \Gamma; x : P_x \vdash e : P}{\Gamma \vdash \lambda x.e : (x : P_x \rightarrow P)}$$

Verify postcondition P with precondition Px

App

$$\frac{\Gamma \vdash v_2 : P_x \quad \Gamma \vdash e_f : (x : P_x \rightarrow P)}{\Gamma \vdash e_f(v_2) : [v_2/x]P}$$

Postcondition P is assumed if and only if its precondition Px is asserted. Encoding?

Verification Condition Generation

Fun

$$\frac{\Gamma \vdash x : P_x \rightarrow P \quad \Gamma ; x : P_x \vdash e : P}{\Gamma \vdash \lambda x.e : (x : P_x \rightarrow P)}$$

Verify postcondition P with precondition Px

App

$$\frac{\Gamma \vdash v_2 : P_x \quad \Gamma \vdash e_f : (x : P_x \rightarrow P)}{\Gamma \vdash e_f(v_2) : [v_2/x]P}$$

Postcondition P is assumed if and only if its precondition Px is asserted. Encoding?

ASSERT

$$\frac{\Gamma \vdash e' : bool \quad \Gamma \vdash \{\_|true\} <: \{\_|e'\} \quad \Gamma \vdash e : P}{\Gamma \vdash \text{assert } e'; e : P}$$

Prove [Γ@e']⇒e' where [Γ@e'] represent the conjunction of dependent types of variables in e' from Γ

Encoding Application

App

$$\frac{\Gamma \vdash v_2 : P_x \quad \Gamma \vdash e_f : (x : P_x \to P)}{\Gamma \vdash e_f(v_2) : [v_2/x]P}$$

Two issues:

1. Want to preserve some measure of context-sensitivity

   keep pre- and post-condition constraints at different call sites distinct
2. Make "must-hold" properties defined by assertions within the function body explicit

Address these issues using intersection types

$$P_x \to P \bigwedge_i P_{x_i} \to P_i$$

The left conjunct captures constraints induced by assertions on function arguments that occur within the function body:

These constraints must always hold

Verification condition Γ |- v_2 : Px $\mathcal{C}(P) = [v_2/x]P$

The right conjunct discriminates over different call-sites

Constraints deduced for post-conditions at a call inform structure of pre-conditions

$\mathcal{C}(P) = \wedge_i([v2/\nu)P_{x_i} \Rightarrow [v_2/x]P_i$ if v_2 is of base type

$\mathcal{C}(P) = \wedge_i[v2/x]P_i$ otherwise because implication is made to hold by subtyping relation for functions

Type Checking/Refinement Algorithm

```
verify f λx.e WL
   let vcs = vcgen ∅ λx.e (abs_ty (λ x. e))
   in
      foreach vc in vcs
         let (result, vc_assignment, abstract_ce) = solve vc
         in
            if result then ()
            else
               let
                  t = genTestcase vc_assignment
                  concrete\_path = run (f, t)
                  if(abstract_ce = concrete_path) then
                     genPrecondition f concrete_path WL
                     verify f e
                  else
                     pred = divergePred f abstract_ce concrete_path
                     genPostcondtion f pred WL
   end
```

Type Checking/Refinement Algorithm

```
verify f λx.e WL
   let vcs = vcgen ∅ λx.e (abs_ty (λ x. e)) Generate verification conditions
   in
      foreach vc in vcs
         let (result, vc_assignment, abstract_ce) = solve vc
         in
            if result then ()
            else
               let
                  t = genTestcase vc_assignment
                  concrete\_path = run (f, t)
                  if(abstract_ce = concrete_path) then
                     genPrecondition f concrete_path WL
                     verify f e
                  else
                     pred = divergePred f abstract_ce concrete_path
                     genPostcondtion f pred WL
   end
```

Type Checking/Refinement Algorithm

```
verify f λx.e WL
   let vcs = vcgen ∅ λx.e (abs_ty (λ x. e)) Generate verification conditions
   in
      foreach vc in vcs  solve each condition using SMT
         let (result, vc_assignment, abstract_ce) = solve vc
         in
            if result then ()
            else
               let
                  t = genTestcase vc_assignment
                  concrete\_path = run (f, t)
                  if(abstract_ce = concrete_path) then
                     genPrecondition f concrete_path WL
                     verify f e
                  else
                     pred = divergePred f abstract_ce concrete_path
                     genPostcondtion f pred WL
   end
```

Type Checking/Refinement Algorithm

```
verify f λx.e WL
   let vcs = vcgen ∅ λx.e (abs_ty (λ x. e)) Generate verification conditions
   in
      foreach vc in vcs  solve each condition using SMT
         let (result, vc_assignment, abstract_ce) = solve vc
         in
            if result then ()
            else
               let    examine counterexample
                  t = genTestcase vc_assignment
                  concrete\_path = run (f, t)
                  if(abstract_ce = concrete_path) then
                     genPrecondition f concrete_path WL
                     verify f e
                  else
                     pred = divergePred f abstract_ce concrete_path
                     genPostcondtion f pred WL
   end
```

Type Checking/Refinement Algorithm

```
verify f λx.e WL
   let vcs = vcgen ∅ λx.e (abs_ty (λ x. e))  Generate verification conditions
   in
      foreach vc in vcs  solve each condition using SMT
         let (result, vc_assignment, abstract_ce) = solve vc
         in
            if result then ()
            else
               let    examine counterexample
                  t = genTestcase vc_assignment
                  concrete\_path = run (f, t)   extract and run test case from assignment
                  if(abstract_ce = concrete_path) then
                     genPrecondition f concrete_path WL
                     verify f e
                  else
                     pred = divergePred f abstract_ce concrete_path
                     genPostcondtion f pred WL
   end
```

Type Checking/Refinement Algorithm

```
verify f λx.e WL
   let vcs = vcgen ∅ λx.e (abs_ty (λ x. e)) Generate verification conditions
   in
      foreach vc in vcs  solve each condition using SMT
         let (result, vc_assignment, abstract_ce) = solve vc
         in
            if result then ()
            else
               let    examine counterexample
                  t = genTestcase vc_assignment
                  concrete\_path = run (f, t)  extract and run test case from assignment
                  if(abstract_ce = concrete_path) then
                     genPrecondition f concrete_path WL  weakest precondition generation when
                     verify f e                          concrete and abstract paths converge
                  else
                     pred = divergePred f abstract_ce concrete_path
                     genPostcondtion f pred WL
   end
```

Type Checking/Refinement Algorithm

```
verify f λx.e WL
   let vcs = vcgen ∅ λx.e (abs_ty (λ x. e))   Generate verification conditions
   in
      foreach vc in vcs   solve each condition using SMT
         let (result, vc_assignment, abstract_ce) = solve vc
         in
            if result then ()
            else
               let   examine counterexample
                  t = genTestcase vc_assignment
                  concrete\_path = run (f, t)   extract and run test case from assignment
                  if(abstract_ce = concrete_path) then
                     genPrecondition f concrete_path WL   weakest precondition generation when
                     verify f e                           concrete and abstract paths converge
                  else
                     pred = divergePred f abstract_ce concrete_path
                     genPostcondtion f pred WL   strengthen post-condition when they diverge
   end
```

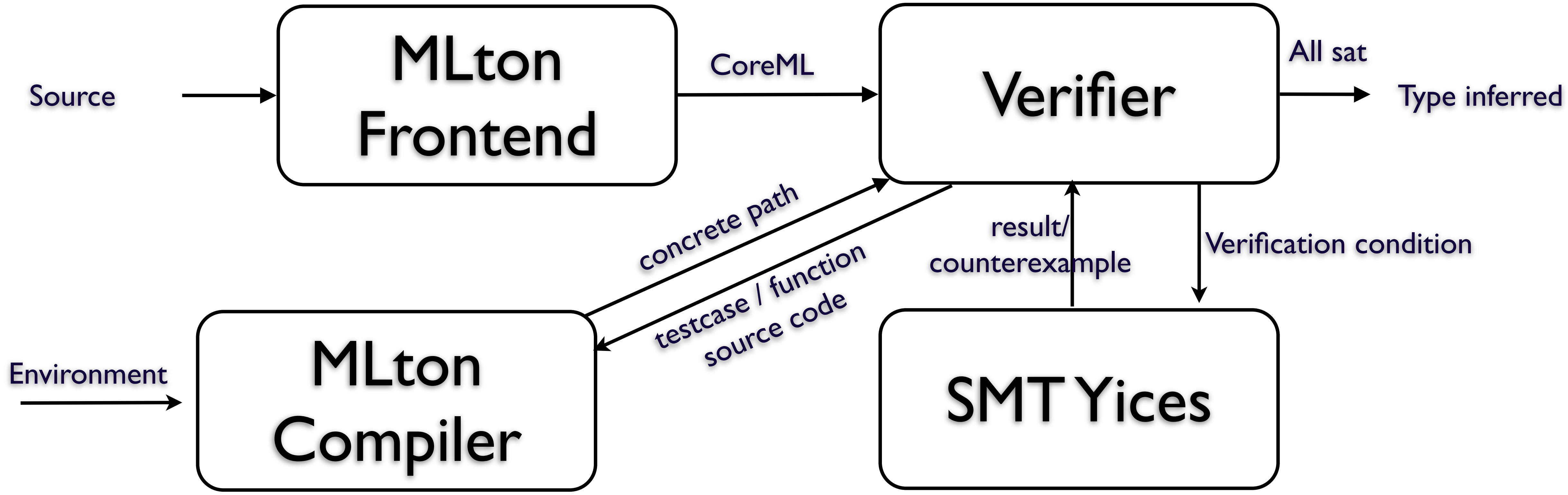
Type Checking/Refinement Algorithm

- Functions are put into worklist (WL) based on call relations
- Each iteration of the fixed-point picks a function from WL and verifies it using `verify`
- Generate pre/postcondition
  - ★ Can strengthen the dependent type of this function and callee function.
  - ★ Put all function whose dependent type has changed into WL (worklist)
- When precondition of a function is strengthened, invoke type constraint propagation algorithm

Implementation

- Implementation is based on MLton, an open-source optimizing Standard ML compiler
  - Incorporate verification mechanism as a front-end pass in Core ML
    - After defunctorization (no modules) and local simplification
      - Type information and original program variables still available
    - Before closure conversion, monomorphisation, and SSA translation
- Use theorem prover Yices as the verification engine
  - Use MLton's FFI interface to interact with a binary version of this prover
- A total of 7.5 KLOC

Implementation


Source

MLton Frontend

CoreML

Verifier

All sat

Type inferred

concrete path

testcase / function source code

result/ counterexample

Verification condition

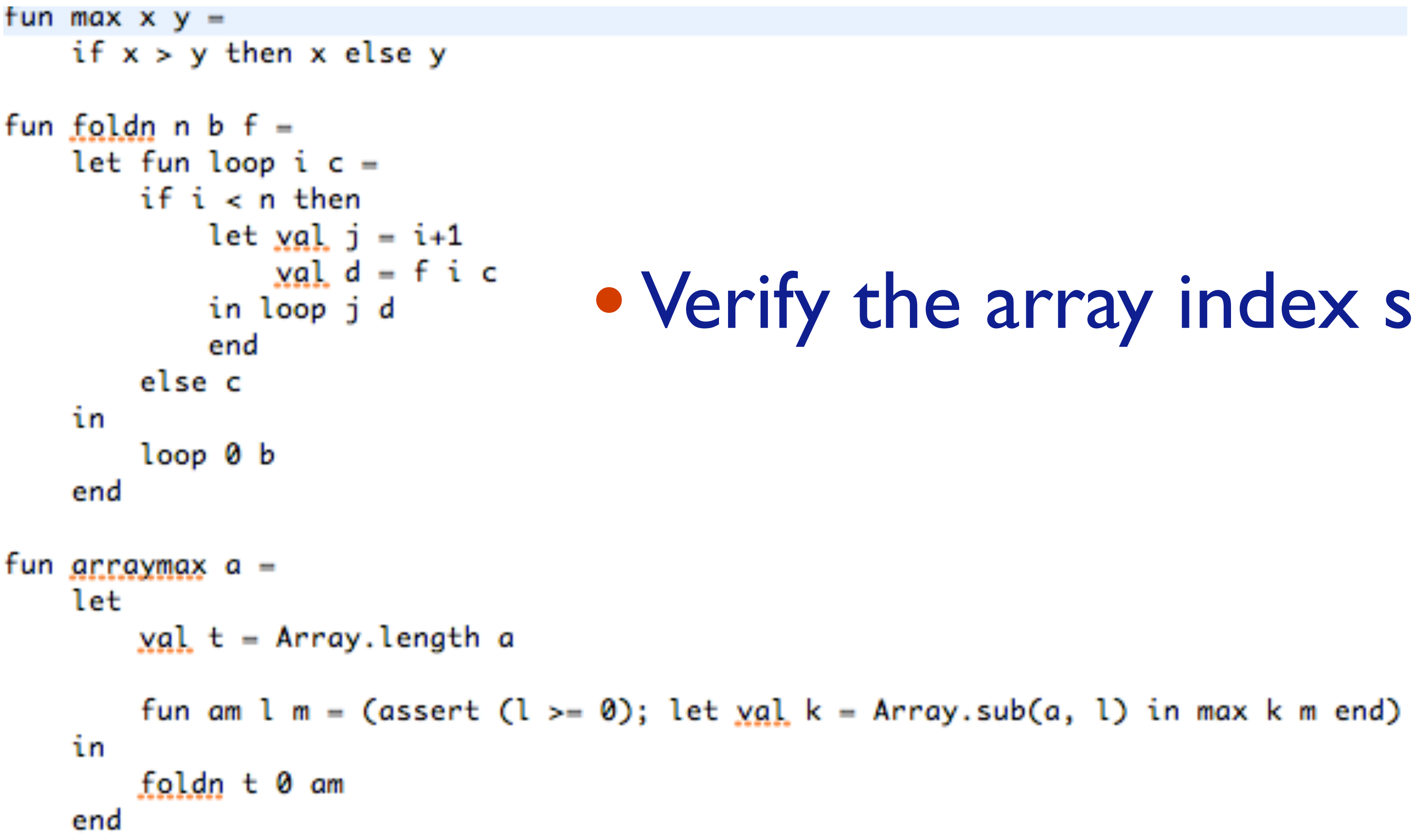
Environment

MLton Compiler

SMT Yices

- MLton frontend defunctorizes, type-checks, and does local simplification yielding CoreML, an intermediate representation which is polymorphic, higher-order and has nested patterns.
- Verifier generates verification conditions from typing rules for CoreML and feeds them to Yices.
- Obtain counterexample along with a testcase witness.
- Compile and run the function under verification with the testcase to obtain a concrete path
- Verifier uses the concrete path and counterexample to strengthen type system

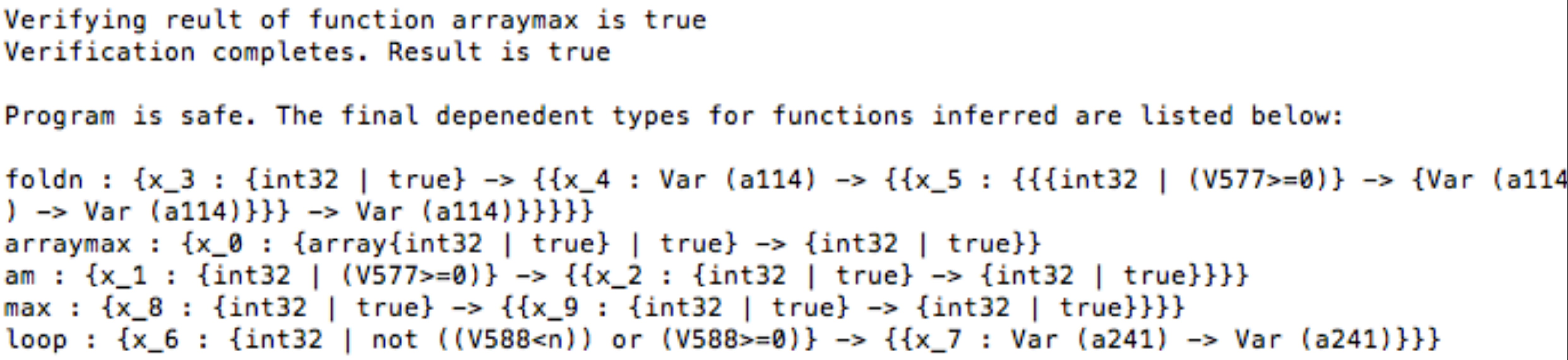
Preliminary Results



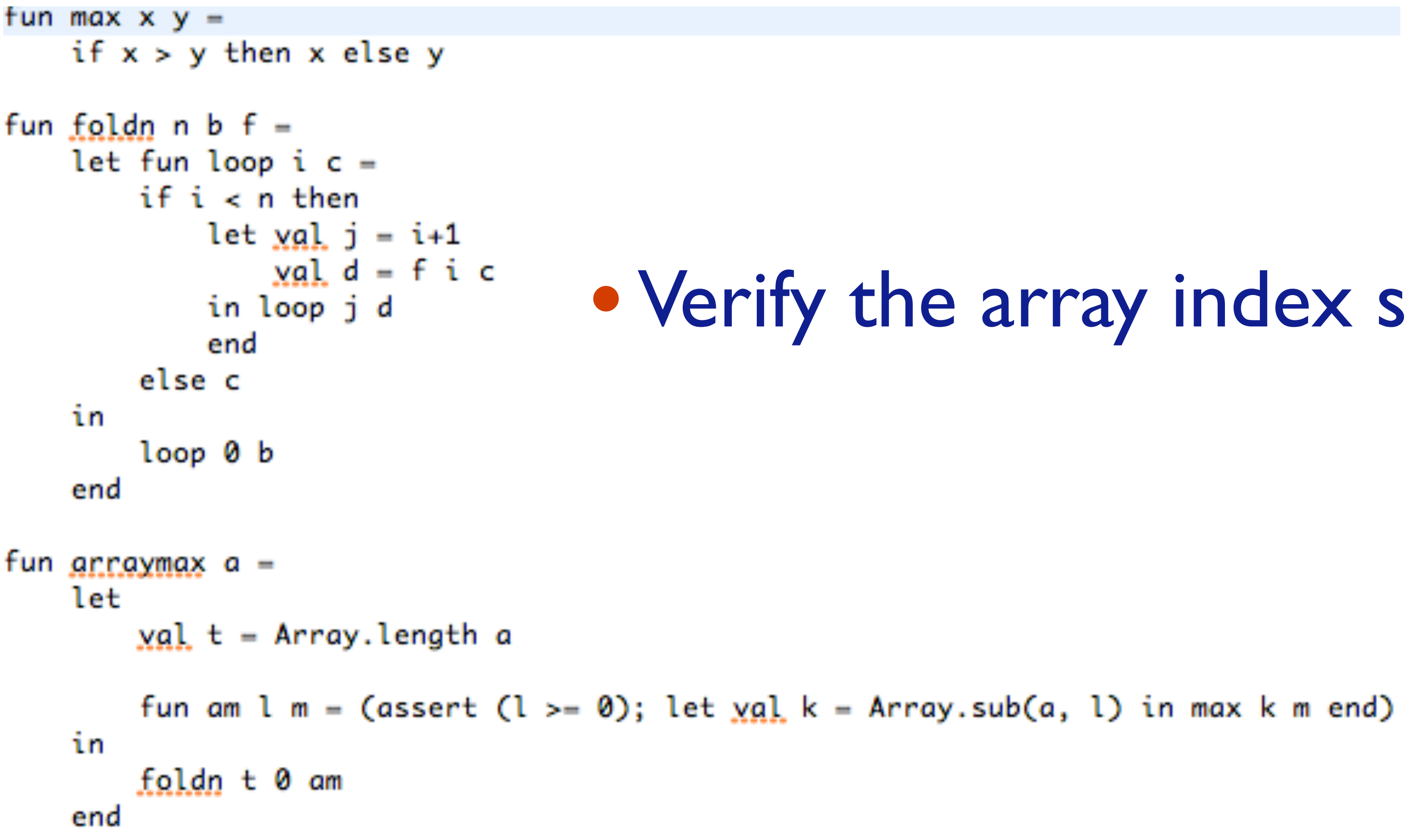
```
fun max x y =
    if x > y then x else y

fun foldn n b f =
    let fun loop i c =
        if i < n then
            let val j = i+1
                val d = f i c
            in loop j d
            end
        else c
    in
        loop 0 b
    end

fun arraymax a =
    let
        val t = Array.length a

        fun am l m = (assert (l >= 0); let val k = Array.sub(a, l) in max k m end)
    in
        foldn t 0 am
    end
```

- Verify the array index should always be no less than 0

Preliminary Results



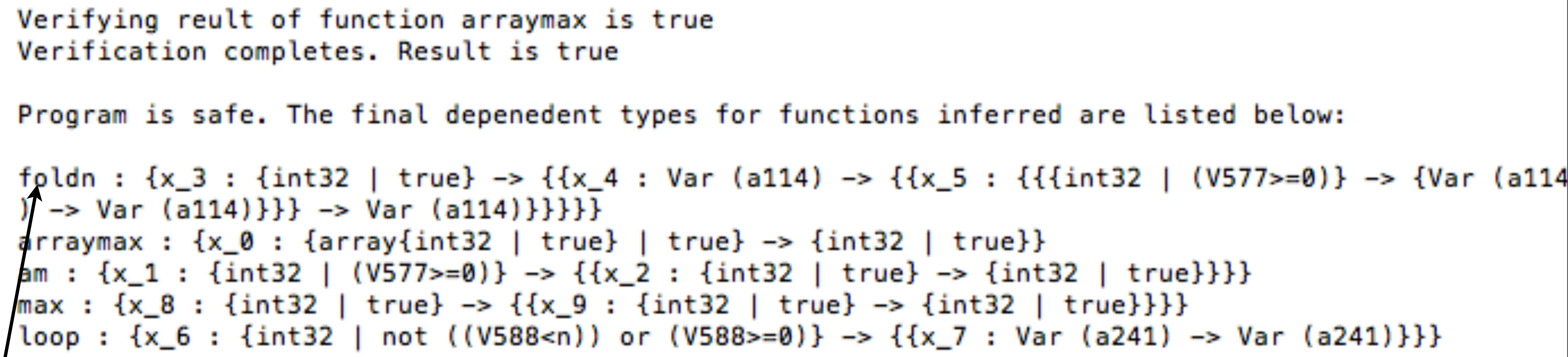
```
Verifying reult of function arraymax is true
Verification completes. Result is true

Program is safe. The final depenedent types for functions inferred are listed below:

foldn : {x_3 : {int32 | true} -> {{x_4 : Var (a114) -> {{x_5 : {{{int32 | (V577>=0)} -> {Var (a114
) -> Var (a114)}}} -> Var (a114)}}}}}
arraymax : {x_0 : {array{int32 | true} | true} -> {int32 | true}}
am : {x_1 : {int32 | (V577>=0)} -> {{x_2 : {int32 | true} -> {int32 | true}}}}
max : {x_8 : {int32 | true} -> {{x_9 : {int32 | true} -> {int32 | true}}}}
loop : {x_6 : {int32 | not ((V588<n)) or (V588>=0)} -> {{x_7 : Var (a241) -> Var (a241)}}}
```



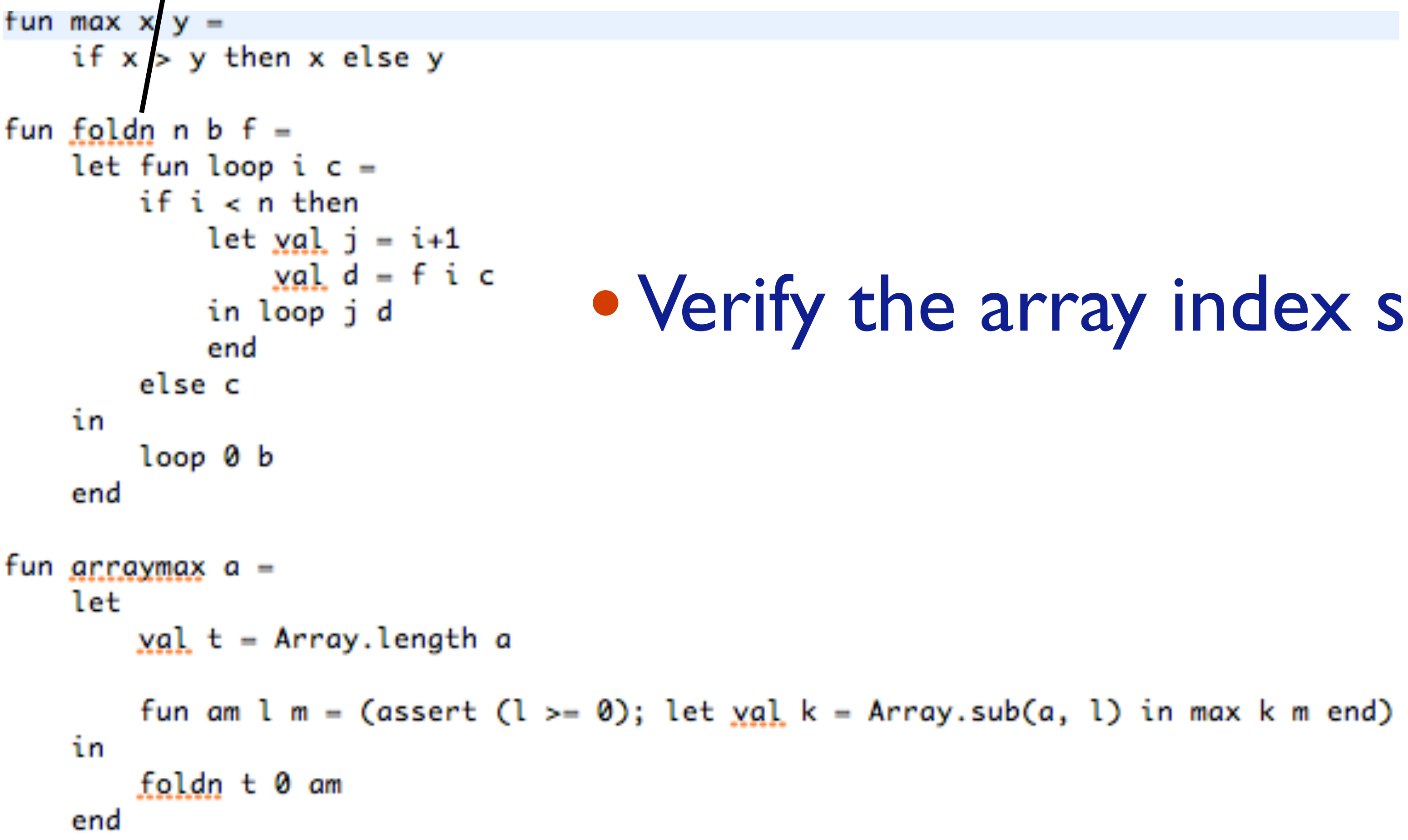
```
fun max x y =
    if x > y then x else y

fun foldn n b f =
    let fun loop i c =
        if i < n then
            let val j = i+1
                val d = f i c
            in loop j d
            end
        else c
    in
        loop 0 b
    end

fun arraymax a =
    let
        val t = Array.length a

        fun am l m = (assert (l >= 0); let val k = Array.sub(a, l) in max k m end)
    in
        foldn t 0 am
    end
```

- Verify the array index should always be no less than 0

Preliminary Results



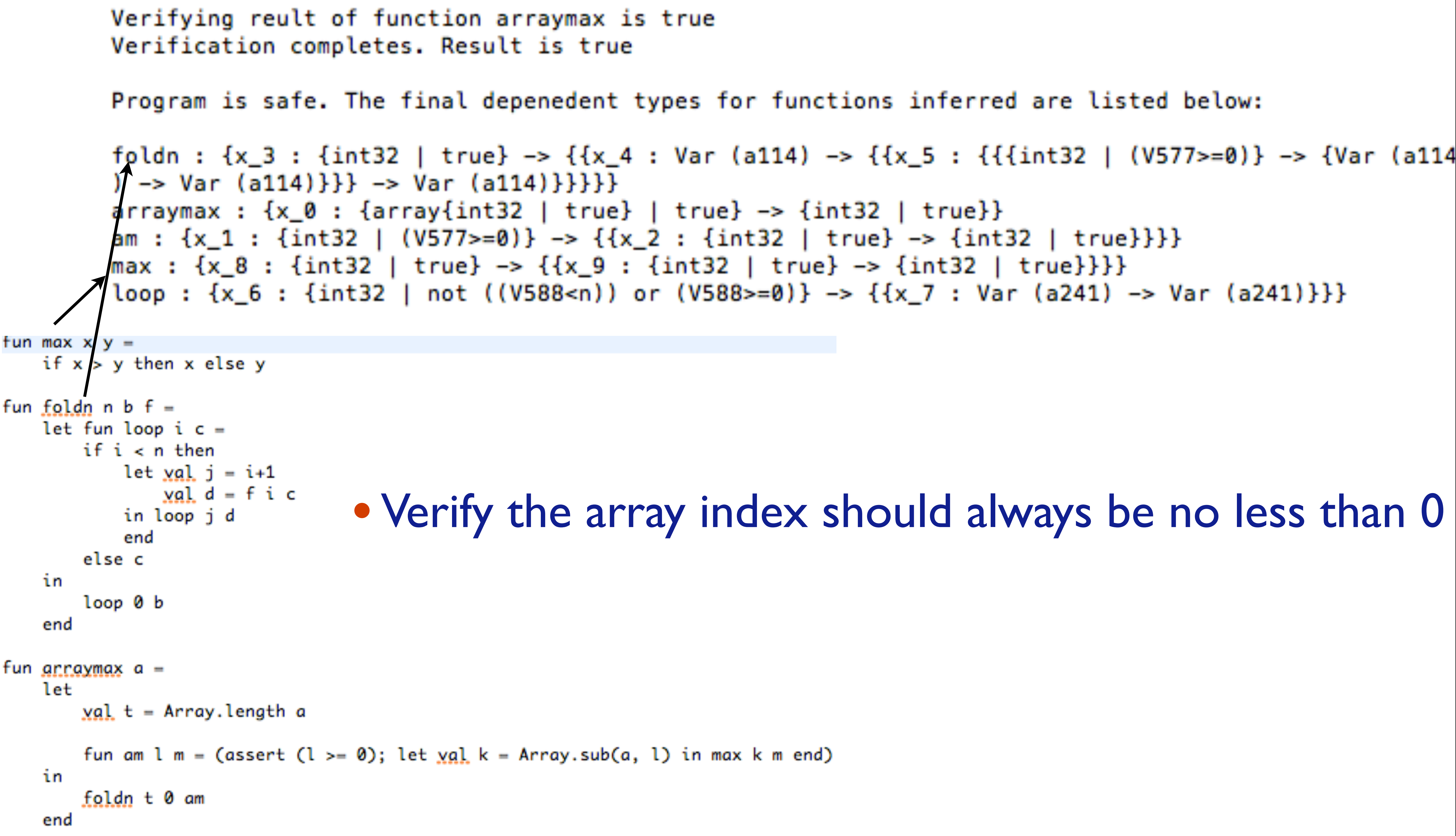
```
Verifying reult of function arraymax is true
Verification completes. Result is true

Program is safe. The final depenedent types for functions inferred are listed below:

foldn : {x_3 : {int32 | true} -> {{x_4 : Var (a114) -> {{x_5 : {{{int32 | (V577>=0)} -> {Var (a114
)} -> Var (a114)}}} -> Var (a114)}}}}}
arraymax : {x_0 : {array{int32 | true} | true} -> {int32 | true}}
am : {x_1 : {int32 | (V577>=0)} -> {{x_2 : {int32 | true} -> {int32 | true}}}}
max : {x_8 : {int32 | true} -> {{x_9 : {int32 | true} -> {int32 | true}}}}
loop : {x_6 : {int32 | not ((V588<n)) or (V588>=0)} -> {{x_7 : Var (a241) -> Var (a241)}}}
```



```
fun max x y =
    if x > y then x else y

fun foldn n b f =
    let fun loop i c =
        if i < n then
            let val j = i+1
                val d = f i c
            in loop j d
            end
        else c
    in
        loop 0 b
    end

fun arraymax a =
    let
        val t = Array.length a

        fun am l m = (assert (l >= 0); let val k = Array.sub(a, l) in max k m end)
    in
        foldn t 0 am
    end
```

- Verify the array index should always be no less than 0

Preliminary Results

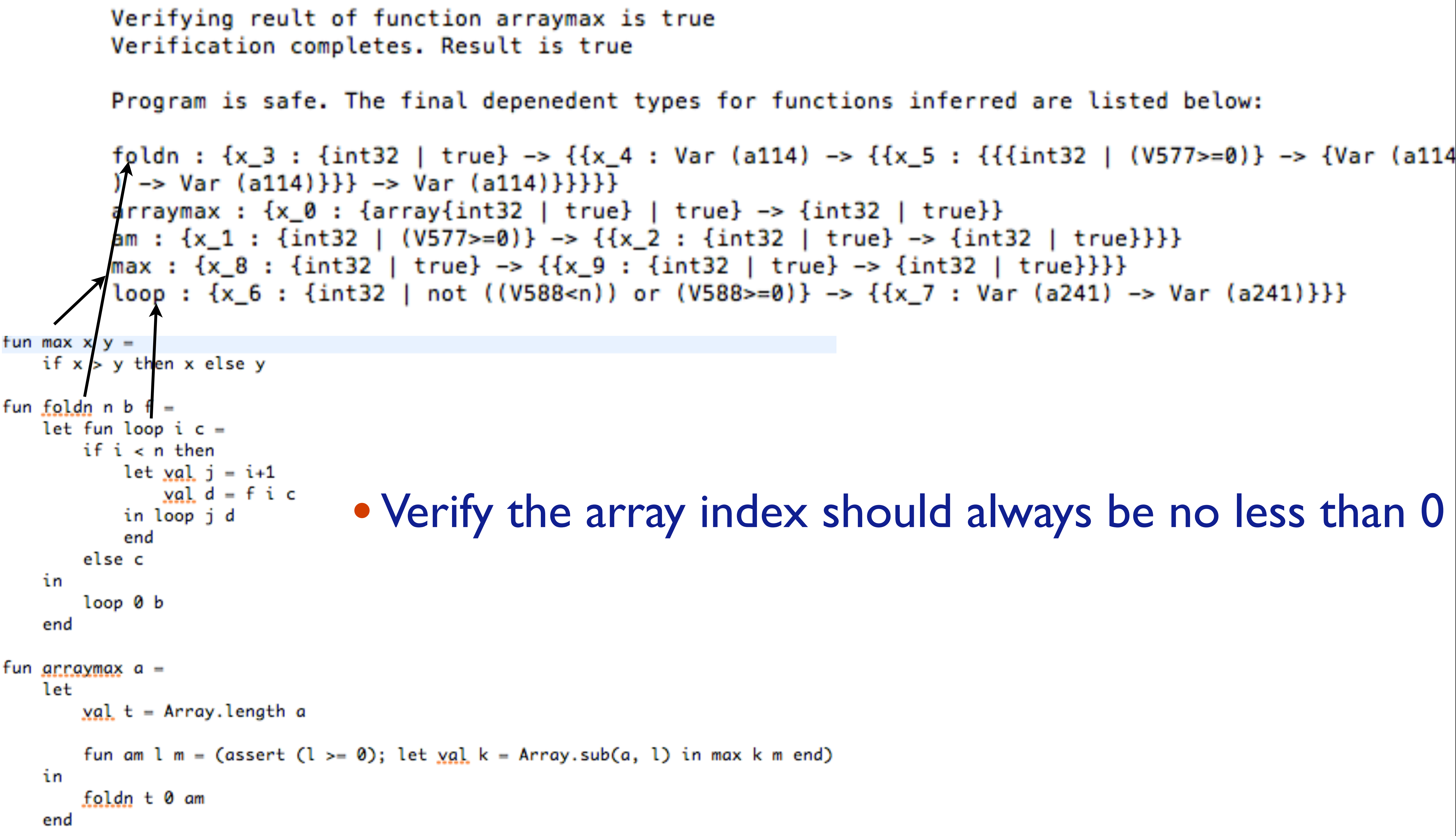


```
Verifying reult of function arraymax is true
Verification completes. Result is true

Program is safe. The final depenedent types for functions inferred are listed below:

foldn : {x_3 : {int32 | true} -> {{x_4 : Var (a114) -> {{x_5 : {{{int32 | (V577>=0)} -> {Var (a114
)} -> Var (a114)}}} -> Var (a114)}}}}}
arraymax : {x_0 : {array{int32 | true} | true} -> {int32 | true}}
am : {x_1 : {int32 | (V577>=0)} -> {{x_2 : {int32 | true} -> {int32 | true}}}}
max : {x_8 : {int32 | true} -> {{x_9 : {int32 | true} -> {int32 | true}}}}
loop : {x_6 : {int32 | not ((V588<n)) or (V588>=0)} -> {{x_7 : Var (a241) -> Var (a241)}}}
```

```
fun max x y =
    if x > y then x else y

fun foldn n b f =
    let fun loop i c =
        if i < n then
            let val j = i+1
                val d = f i c
            in loop j d
            end
        else c
    in
        loop 0 b
    end

fun arraymax a =
    let
        val t = Array.length a

        fun am l m = (assert (l >= 0); let val k = Array.sub(a, l) in max k m end)
    in
        foldn t 0 am
    end
```

- Verify the array index should always be no less than 0

Preliminary Results

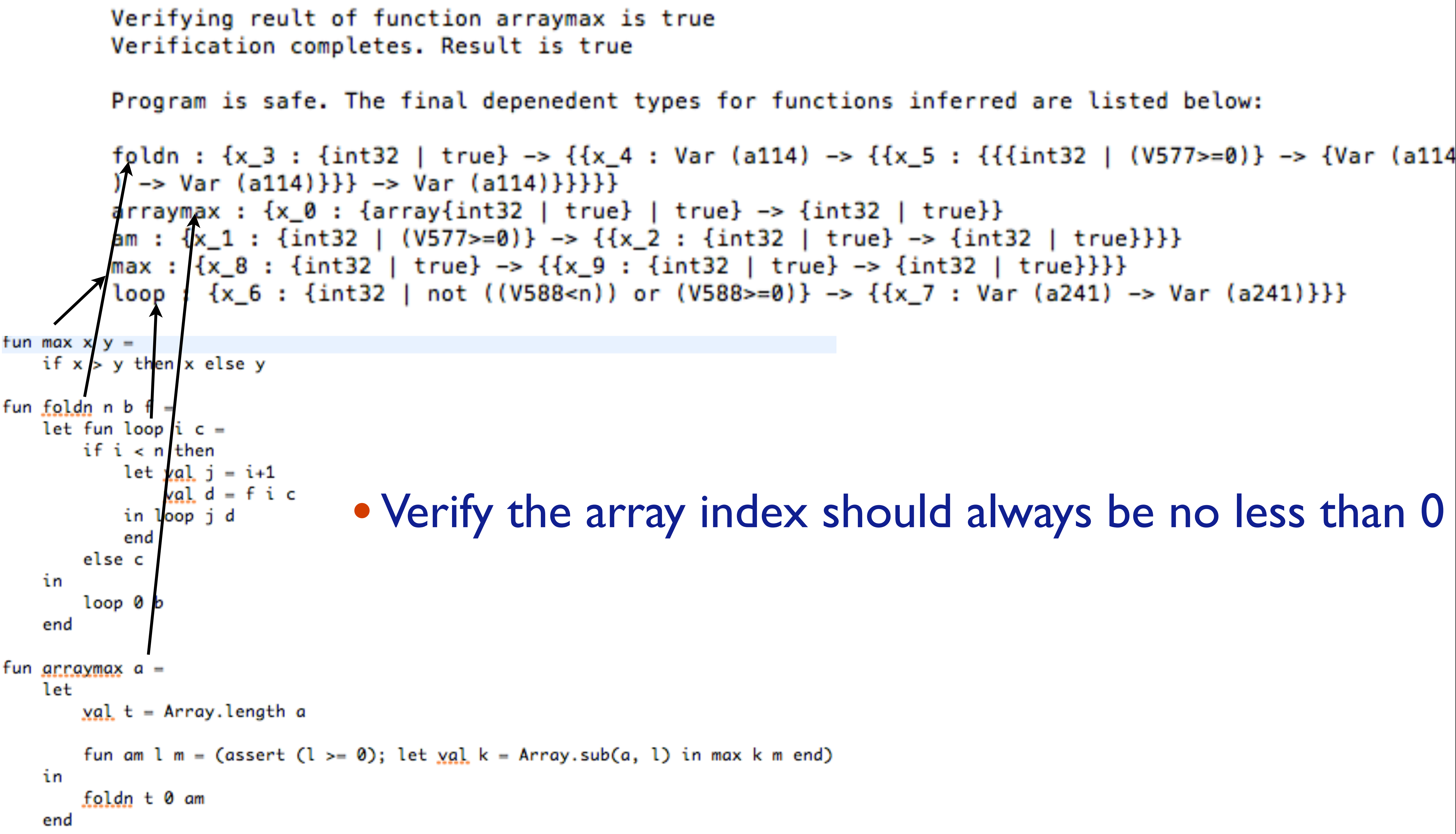


```
Verifying reult of function arraymax is true
Verification completes. Result is true

Program is safe. The final depenedent types for functions inferred are listed below:

foldn : {x_3 : {int32 | true} -> {{x_4 : Var (a114) -> {{x_5 : {{{int32 | (V577>=0)} -> {Var (a114
)} -> Var (a114)}}} -> Var (a114)}}}}}
arraymax : {x_0 : {array{int32 | true} | true} -> {int32 | true}}
am : {x_1 : {int32 | (V577>=0)} -> {{x_2 : {int32 | true} -> {int32 | true}}}}
max : {x_8 : {int32 | true} -> {{x_9 : {int32 | true} -> {int32 | true}}}}
loop : {x_6 : {int32 | not ((V588<n)) or (V588>=0)} -> {{x_7 : Var (a241) -> Var (a241)}}}
```

```
fun max x y =
    if x > y then x else y

fun foldn n b f =
    let fun loop i c =
        if i < n then
            let val j = i+1
                val d = f i c
            in loop j d
            end
        else c
    in
        loop 0 b
    end

fun arraymax a =
    let
        val t = Array.length a

        fun am l m = (assert (l >= 0); let val k = Array.sub(a, l) in max k m end)
    in
        foldn t 0 am
    end
```

- Verify the array index should always be no less than 0

Preliminary Results

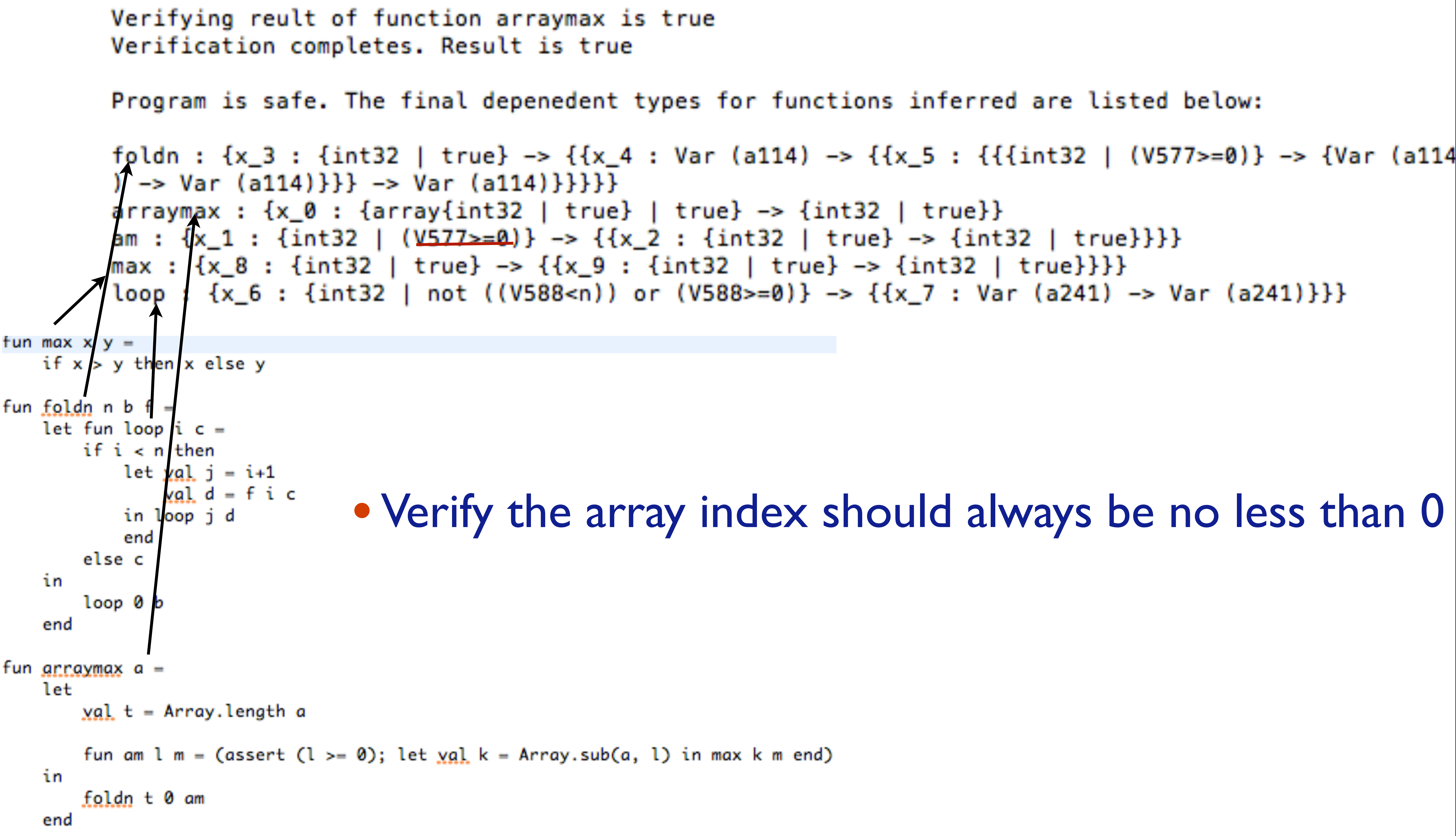


```
Verifying reult of function arraymax is true
Verification completes. Result is true

Program is safe. The final depenedent types for functions inferred are listed below:

foldn : {x_3 : {int32 | true} -> {{x_4 : Var (a114) -> {{x_5 : {{{int32 | (V577>=0)} -> {Var (a114
)} -> Var (a114)}}} -> Var (a114)}}}}}
arraymax : {x_0 : {array{int32 | true} | true} -> {int32 | true}}
am : {x_1 : {int32 | (V577>=0)} -> {{x_2 : {int32 | true} -> {int32 | true}}}}
max : {x_8 : {int32 | true} -> {{x_9 : {int32 | true} -> {int32 | true}}}}
loop : {x_6 : {int32 | not ((V588<n)) or (V588>=0)} -> {{x_7 : Var (a241) -> Var (a241)}}}
```

```
fun max x y =
    if x > y then x else y

fun foldn n b f =
    let fun loop i c =
        if i < n then
            let val j = i+1
                val d = f i c
            in loop j d
            end
        else c
    in
        loop 0 b
    end

fun arraymax a =
    let
        val t = Array.length a

        fun am l m = (assert (l >= 0); let val k = Array.sub(a, l) in max k m end)
    in
        foldn t 0 am
    end
```

- Verify the array index should always be no less than 0

Preliminary Results

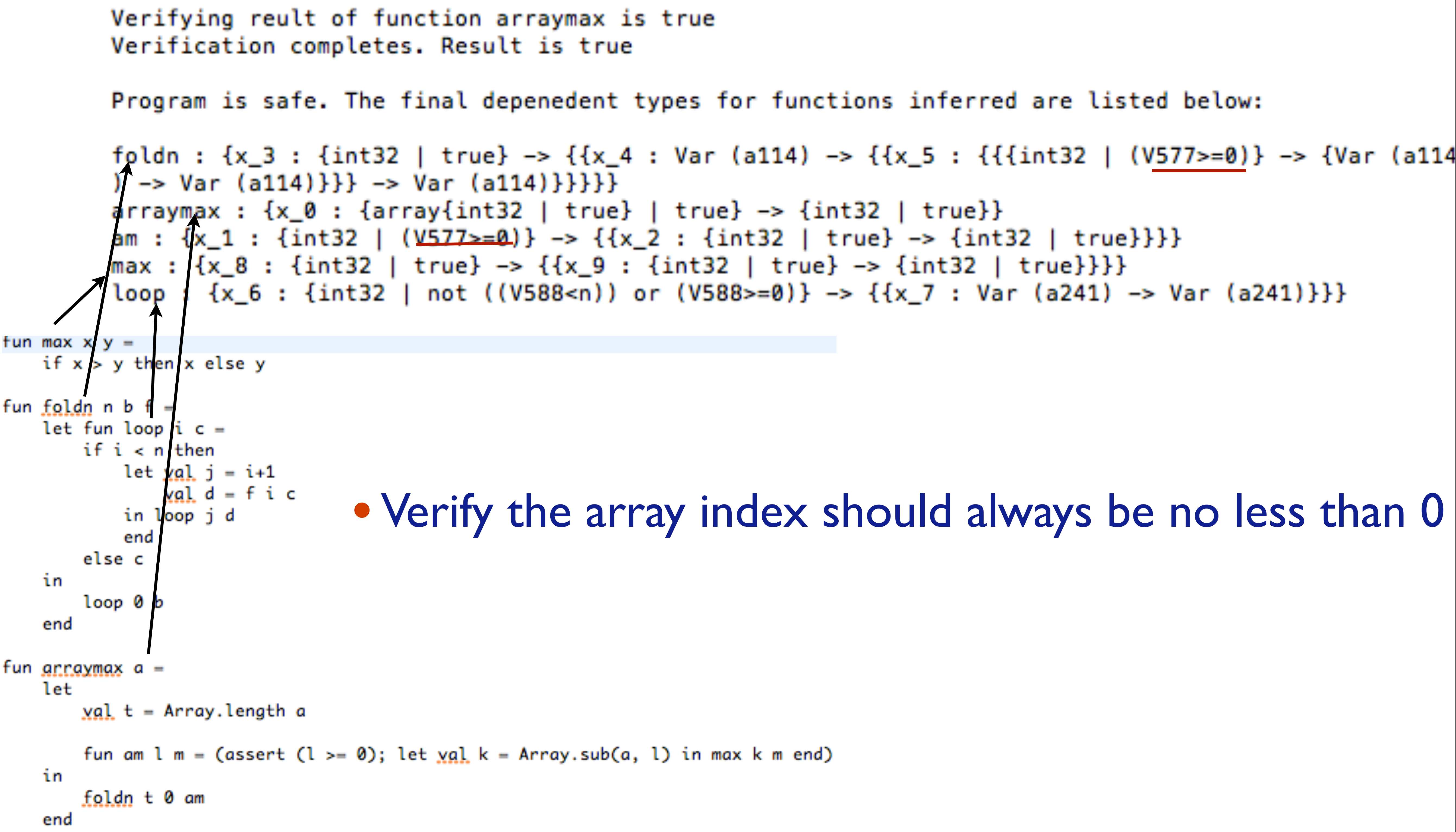


```
Verifying reult of function arraymax is true
Verification completes. Result is true

Program is safe. The final depenedent types for functions inferred are listed below:

foldn : {x_3 : {int32 | true} -> {{x_4 : Var (a114) -> {{x_5 : {{{int32 | (V577>=0)} -> {Var (a114
)} -> Var (a114)}}} -> Var (a114)}}}}}
arraymax : {x_0 : {array{int32 | true} | true} -> {int32 | true}}
am : {x_1 : {int32 | (V577>=0)} -> {{x_2 : {int32 | true} -> {int32 | true}}}}
max : {x_8 : {int32 | true} -> {{x_9 : {int32 | true} -> {int32 | true}}}}
loop : {x_6 : {int32 | not ((V588<n)) or (V588>=0)} -> {{x_7 : Var (a241) -> Var (a241)}}}
```

```
fun max x y =
    if x > y then x else y

fun foldn n b f =
    let fun loop i c =
        if i < n then
            let val j = i+1
                val d = f i c
            in loop j d
            end
        else c
    in
        loop 0 b
    end

fun arraymax a =
    let
        val t = Array.length a

        fun am l m = (assert (l >= 0); let val k = Array.sub(a, l) in max k m end)
    in
        foldn t 0 am
    end
```

- Verify the array index should always be no less than 0

Preliminary Results

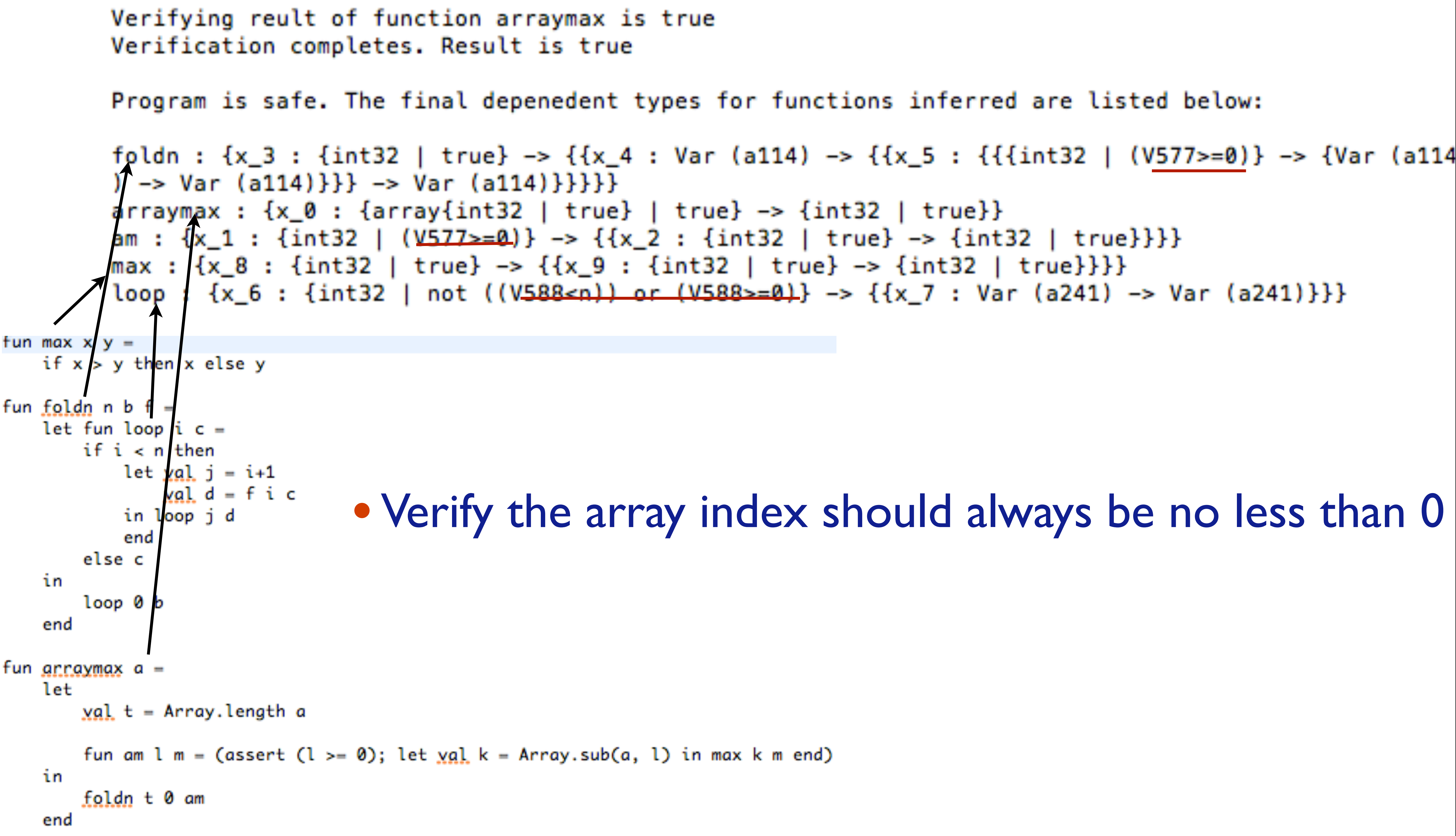


```
Verifying reult of function arraymax is true
Verification completes. Result is true

Program is safe. The final depenedent types for functions inferred are listed below:

foldn : {x_3 : {int32 | true} -> {{x_4 : Var (a114) -> {{x_5 : {{{int32 | (V577>=0)} -> {Var (a114
)} -> Var (a114)}}} -> Var (a114)}}}}}
arraymax : {x_0 : {array{int32 | true} | true} -> {int32 | true}}
am : {x_1 : {int32 | (V577>=0)} -> {{x_2 : {int32 | true} -> {int32 | true}}}}
max : {x_8 : {int32 | true} -> {{x_9 : {int32 | true} -> {int32 | true}}}}
loop : {x_6 : {int32 | not ((V588<n)) or (V588>=0)} -> {{x_7 : Var (a241) -> Var (a241)}}}
```

```
fun max x y =
    if x > y then x else y

fun foldn n b f =
    let fun loop i c =
        if i < n then
            let val j = i+1
                val d = f i c
            in loop j d
            end
        else c
    in
        loop 0 b
    end

fun arraymax a =
    let
        val t = Array.length a

        fun am l m = (assert (l >= 0); let val k = Array.sub(a, l) in max k m end)
    in
        foldn t 0 am
    end
```

- Verify the array index should always be no less than 0

Preliminary Results

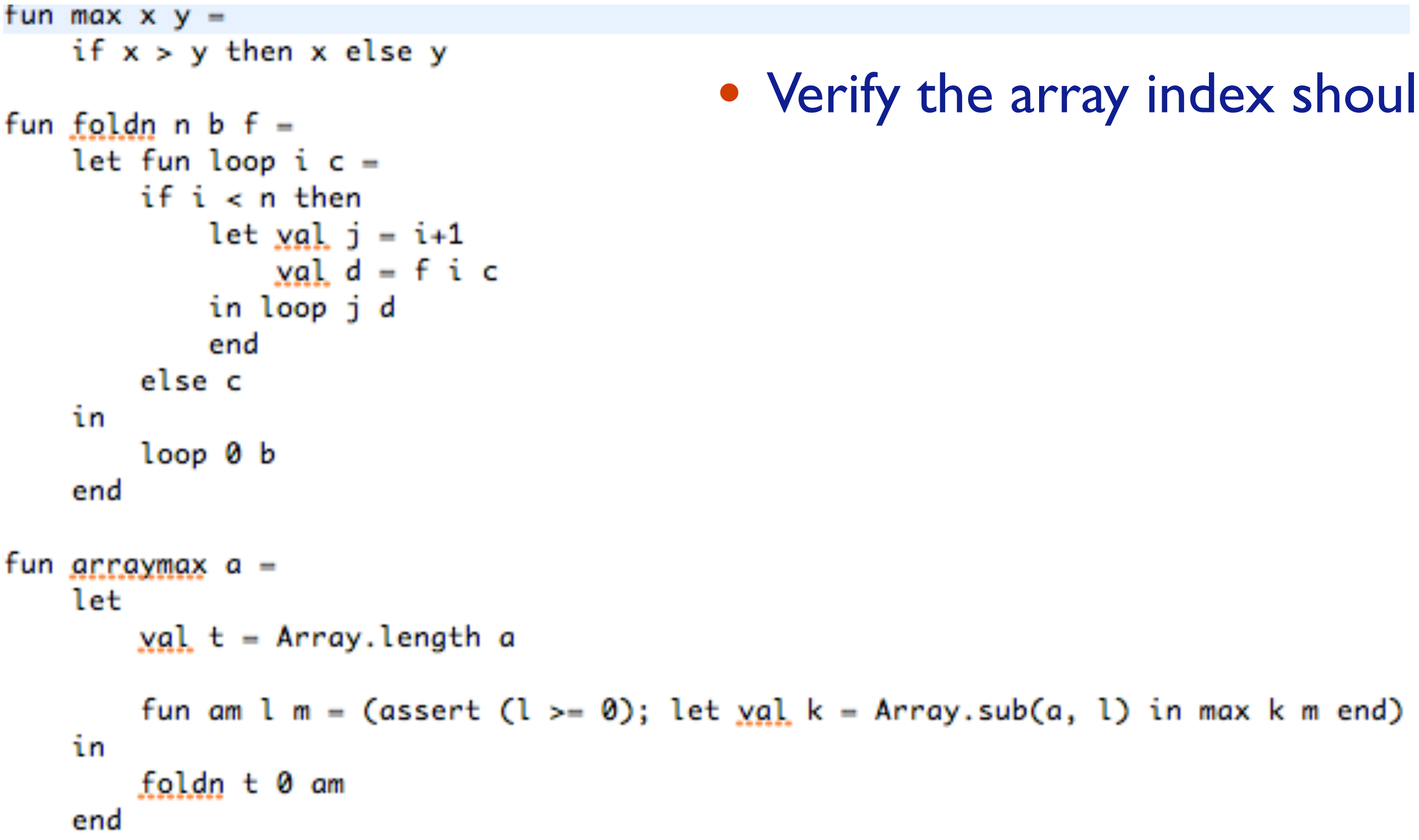


```
Verifying reult of function arraymax is true
Verification completes. Result is true

Program is safe. The final depenedent types for functions inferred are listed below:

foldn : {x_3 : {int32 | true} -> {{x_4 : Var (a114) -> {{x_5 : {{{int32 | (V577>=0)} -> {Var (a114
)} -> Var (a114)}}} -> Var (a114)}}}}}
arraymax : {x_0 : {array{int32 | true} | true} -> {int32 | true}}
am : {x_1 : {int32 | (V577>=0)} -> {{x_2 : {int32 | true} -> {int32 | true}}}}
max : {x_8 : {int32 | true} -> {{x_9 : {int32 | true} -> {int32 | true}}}}
loop : {x_6 : {int32 | not ((V588<n)) or (V588>=0)} -> {{x_7 : Var (a241) -> Var (a241)}}}
```

```
fun max x y =
    if x > y then x else y

fun foldn n b f =
    let fun loop i c =
        if i < n then
            let val j = i+1
                val d = f i c
            in loop j d
            end
        else c
    in
        loop 0 b
    end

fun arraymax a =
    let
        val t = Array.length a

        fun am l m = (assert (l >= 0); let val k = Array.sub(a, l) in max k m end)
    in
        foldn t 0 am
    end
```

- Verify the array index should always be no less than 0

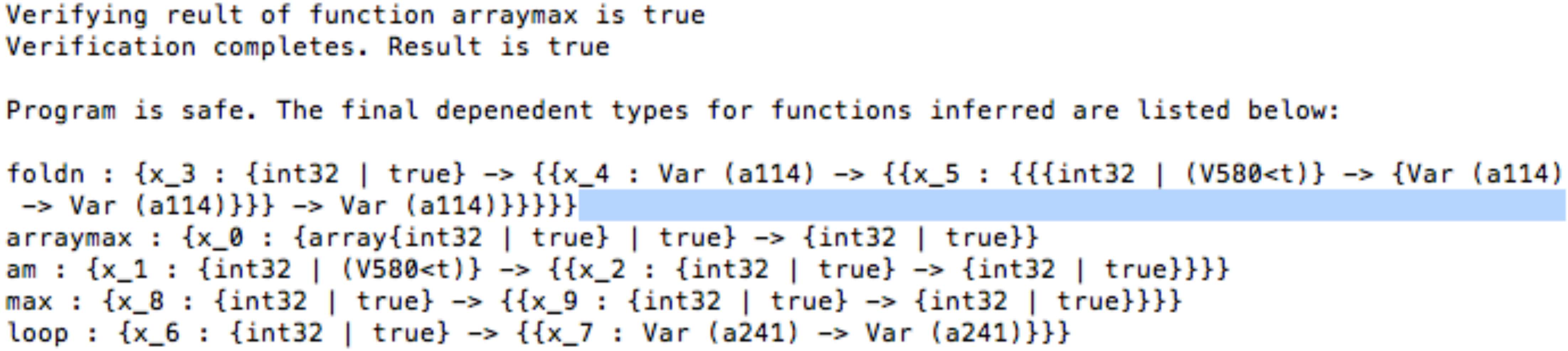
Preliminary Results



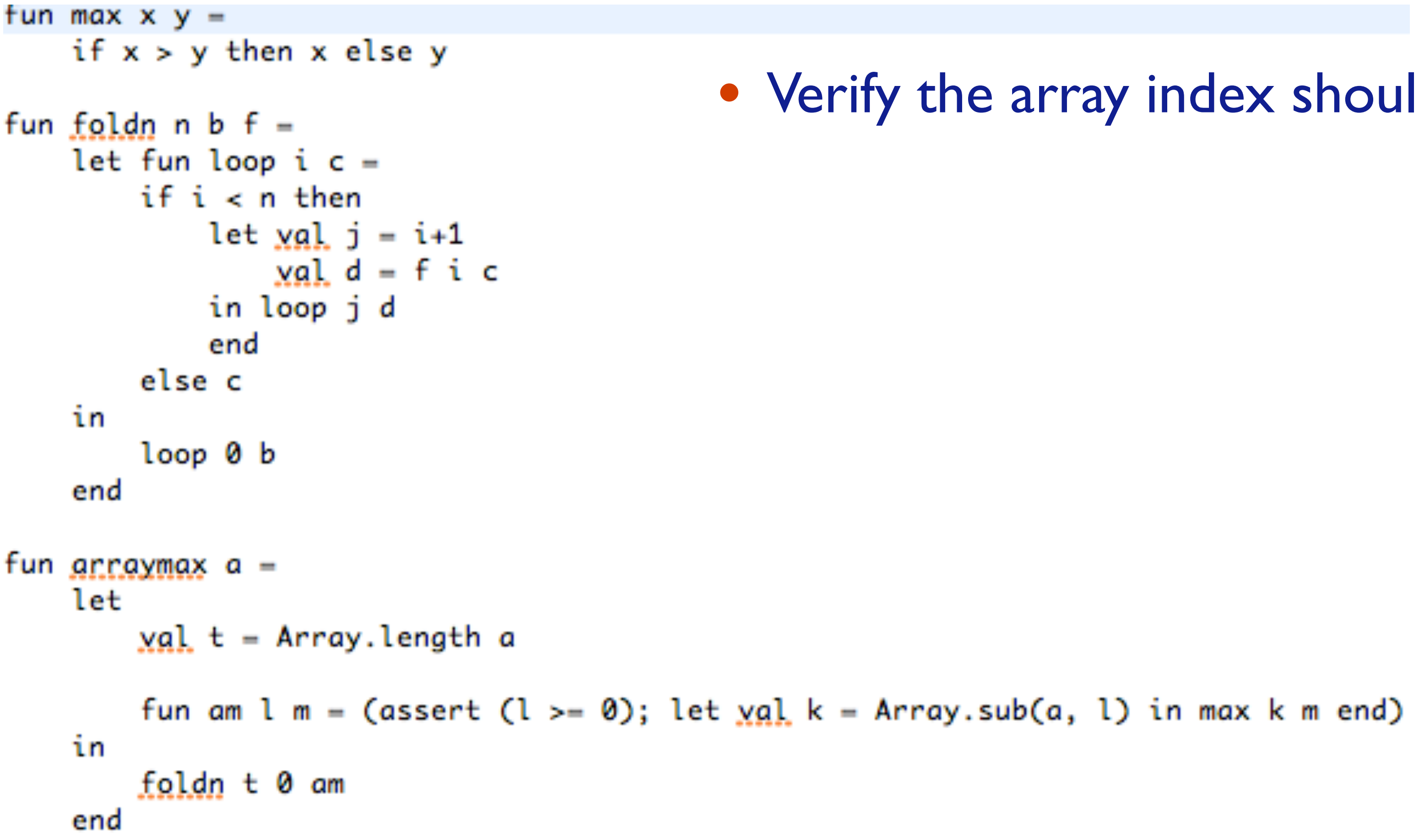
```
fun max x y =
    if x > y then x else y

fun foldn n b f =
    let fun loop i c =
        if i < n then
            let val j = i+1
                val d = f i c
            in loop j d
            end
        else c
    in
        loop 0 b
    end

fun arraymax a =
    let
        val t = Array.length a

        fun am l m = (assert (l >= 0); let val k = Array.sub(a, l) in max k m end)
    in
        foldn t 0 am
    end
```

- Verify the array index should always be less than the length of the array

Preliminary Results



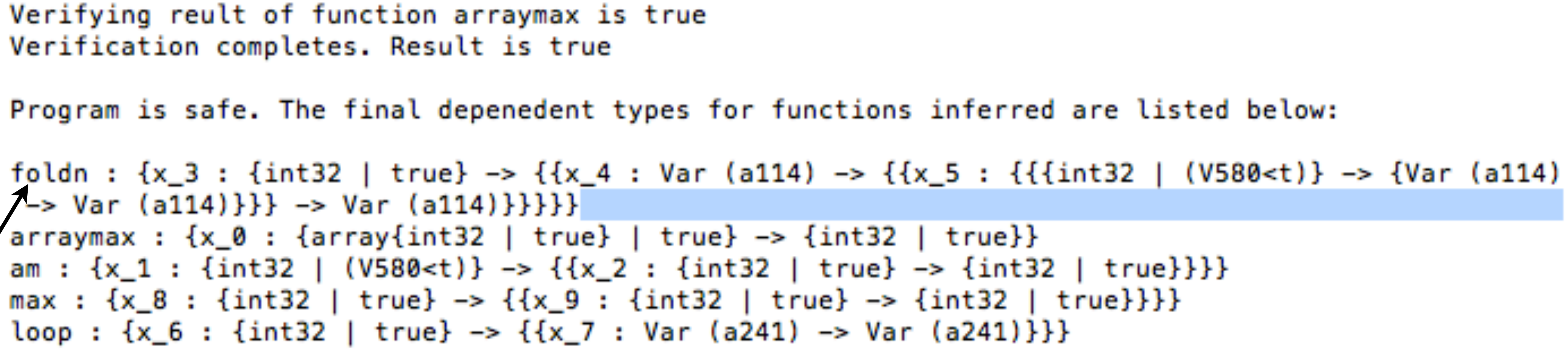
```
Verifying reult of function arraymax is true
Verification completes. Result is true

Program is safe. The final depenedent types for functions inferred are listed below:

foldn : {x_3 : {int32 | true} -> {{x_4 : Var (a114) -> {{x_5 : {{{int32 | (V580<t)} -> {Var (a114)
 -> Var (a114)}}} -> Var (a114)}}}}}
arraymax : {x_0 : {array{int32 | true} | true} -> {int32 | true}}
am : {x_1 : {int32 | (V580<t)} -> {{x_2 : {int32 | true} -> {int32 | true}}}}
max : {x_8 : {int32 | true} -> {{x_9 : {int32 | true} -> {int32 | true}}}}
loop : {x_6 : {int32 | true} -> {{x_7 : Var (a241) -> Var (a241)}}}
```



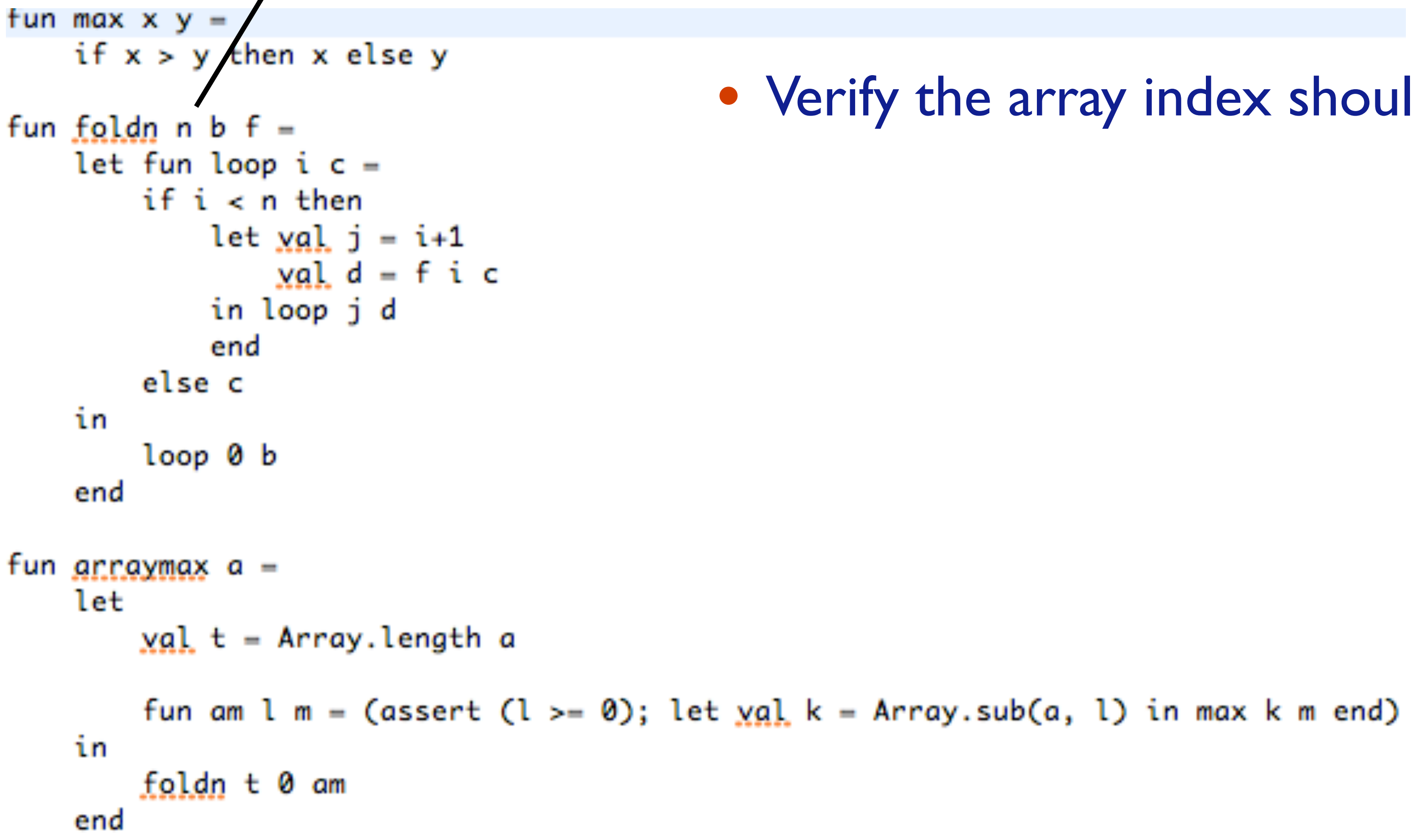
```
fun max x y =
    if x > y then x else y

fun foldn n b f =
    let fun loop i c =
        if i < n then
            let val j = i+1
                val d = f i c
            in loop j d
            end
        else c
    in
        loop 0 b
    end

fun arraymax a =
    let
        val t = Array.length a

        fun am l m = (assert (l >= 0); let val k = Array.sub(a, l) in max k m end)
    in
        foldn t 0 am
    end
```

- Verify the array index should always be less than the length of the array

Preliminary Results



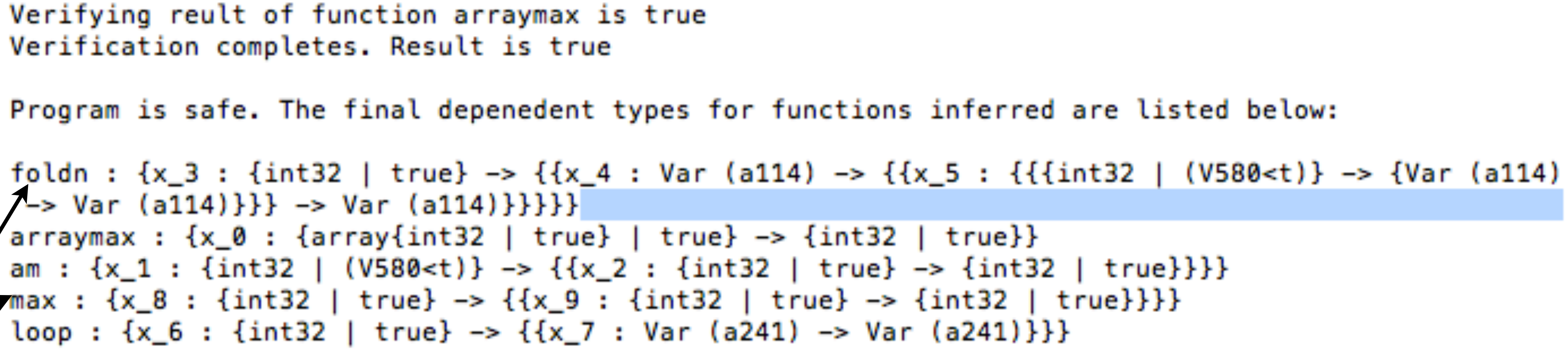
```
Verifying reult of function arraymax is true
Verification completes. Result is true

Program is safe. The final depenedent types for functions inferred are listed below:

foldn : {x_3 : {int32 | true} -> {{x_4 : Var (a114) -> {{x_5 : {{{int32 | (V580<t)} -> {Var (a114)
-> Var (a114)}}} -> Var (a114)}}}}}
arraymax : {x_0 : {array{int32 | true} | true} -> {int32 | true}}
am : {x_1 : {int32 | (V580<t)} -> {{x_2 : {int32 | true} -> {int32 | true}}}}
max : {x_8 : {int32 | true} -> {{x_9 : {int32 | true} -> {int32 | true}}}}
loop : {x_6 : {int32 | true} -> {{x_7 : Var (a241) -> Var (a241)}}}
```



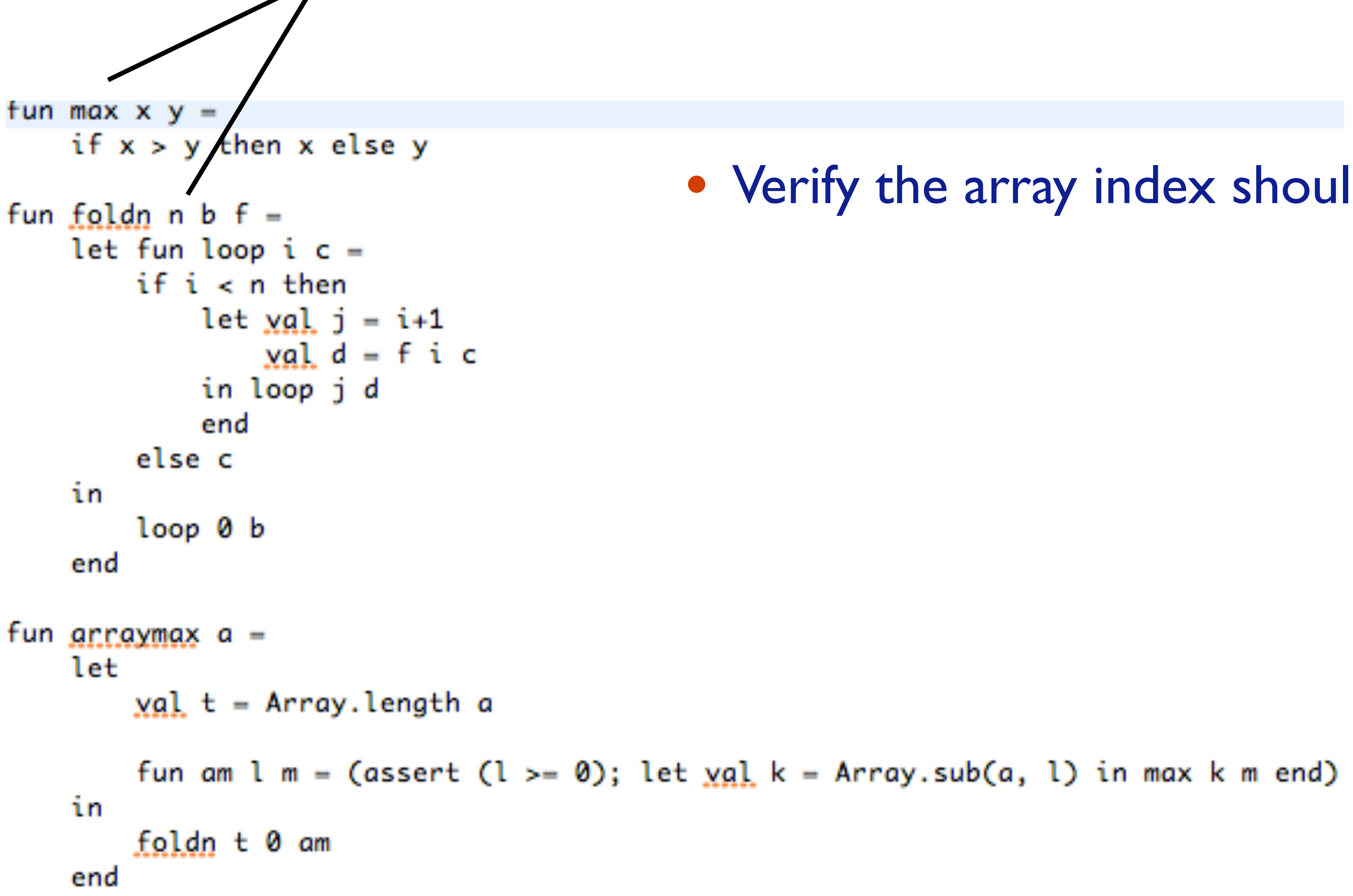
```
fun max x y =
    if x > y then x else y

fun foldn n b f =
    let fun loop i c =
        if i < n then
            let val j = i+1
                val d = f i c
            in loop j d
            end
        else c
    in
        loop 0 b
    end

fun arraymax a =
    let
        val t = Array.length a

        fun am l m = (assert (l >= 0); let val k = Array.sub(a, l) in max k m end)
    in
        foldn t 0 am
    end
```

- Verify the array index should always be less than the length of the array

Preliminary Results



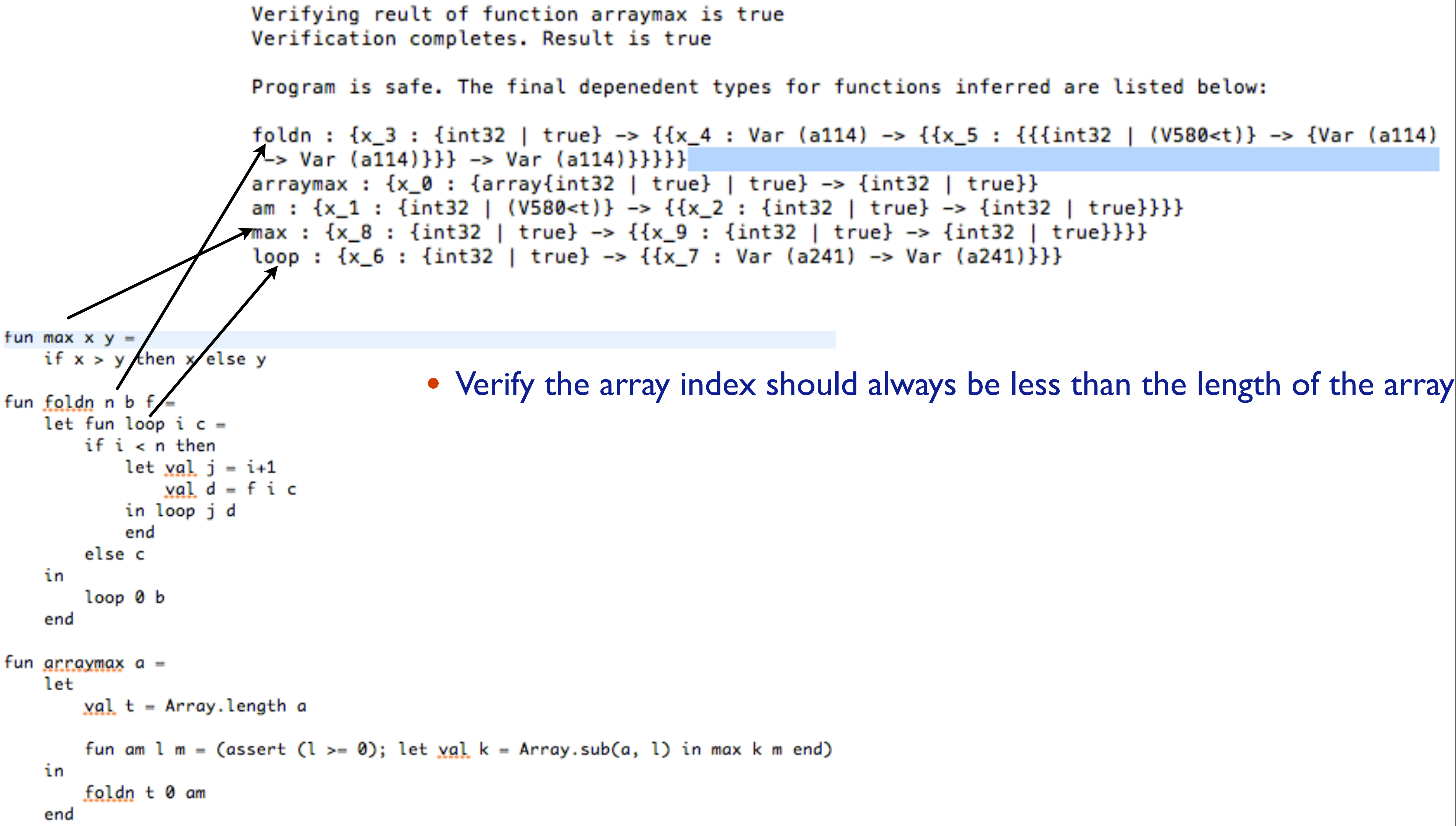
```
Verifying reult of function arraymax is true
Verification completes. Result is true

Program is safe. The final depenedent types for functions inferred are listed below:

foldn : {x_3 : {int32 | true} -> {{x_4 : Var (a114) -> {{x_5 : {{{int32 | (V580<t)} -> {Var (a114)
-> Var (a114)}}} -> Var (a114)}}}}}
arraymax : {x_0 : {array{int32 | true} | true} -> {int32 | true}}
am : {x_1 : {int32 | (V580<t)} -> {{x_2 : {int32 | true} -> {int32 | true}}}}
max : {x_8 : {int32 | true} -> {{x_9 : {int32 | true} -> {int32 | true}}}}
loop : {x_6 : {int32 | true} -> {{x_7 : Var (a241) -> Var (a241)}}}
```



```
fun max x y =
    if x > y then x else y

fun foldn n b f =
    let fun loop i c =
        if i < n then
            let val j = i+1
                val d = f i c
            in loop j d
            end
        else c
    in
        loop 0 b
    end

fun arraymax a =
    let
        val t = Array.length a

        fun am l m = (assert (l >= 0); let val k = Array.sub(a, l) in max k m end)
    in
        foldn t 0 am
    end
```

- Verify the array index should always be less than the length of the array

Preliminary Results

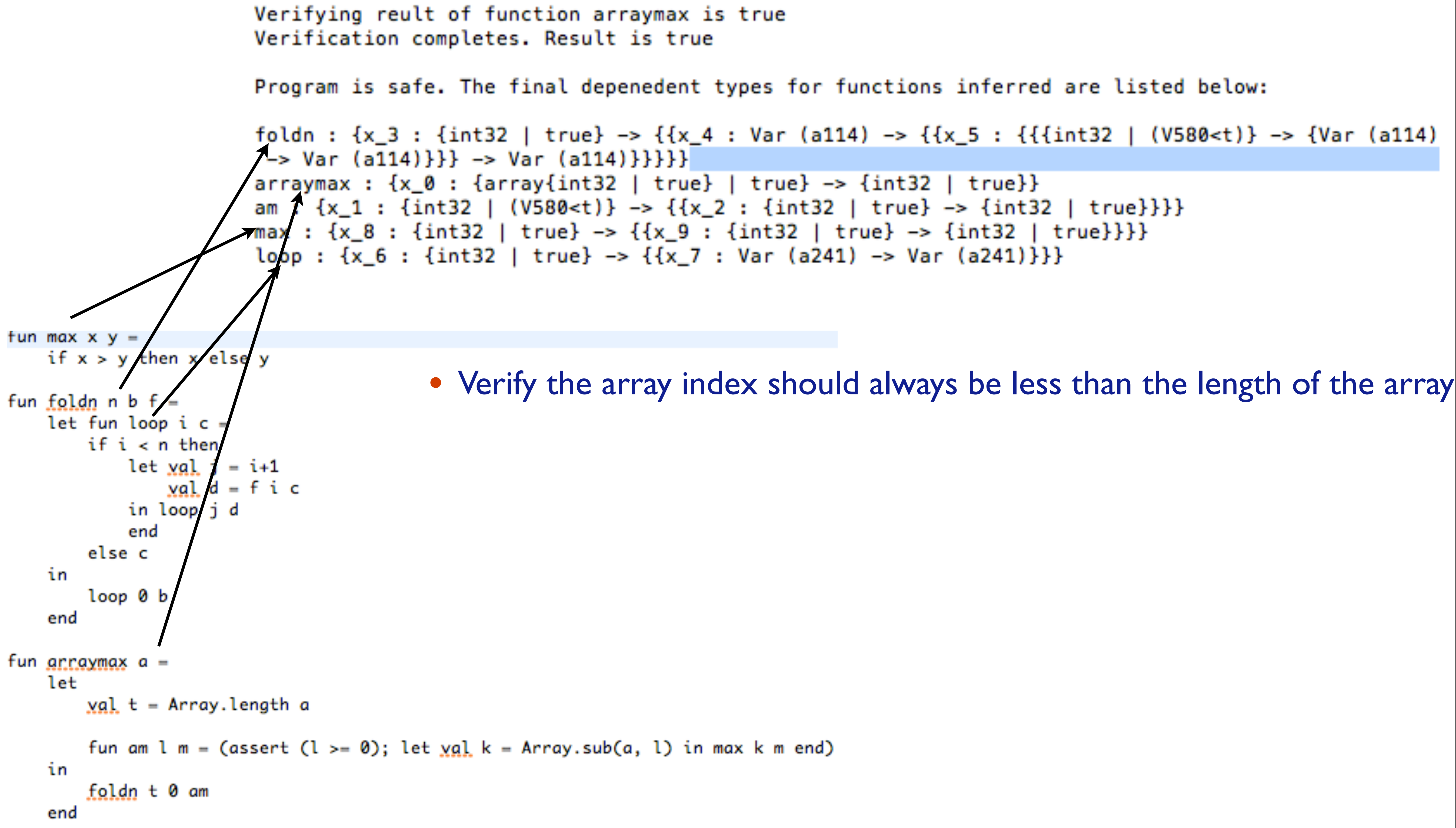


```
Verifying reult of function arraymax is true
Verification completes. Result is true

Program is safe. The final depenedent types for functions inferred are listed below:

foldn : {x_3 : {int32 | true} -> {{x_4 : Var (a114) -> {{x_5 : {{{int32 | (V580<t)} -> {Var (a114)
-> Var (a114)}}} -> Var (a114)}}}}}
arraymax : {x_0 : {array{int32 | true} | true} -> {int32 | true}}
am : {x_1 : {int32 | (V580<t)} -> {{x_2 : {int32 | true} -> {int32 | true}}}}
max : {x_8 : {int32 | true} -> {{x_9 : {int32 | true} -> {int32 | true}}}}
loop : {x_6 : {int32 | true} -> {{x_7 : Var (a241) -> Var (a241)}}}
```

```
fun max x y =
    if x > y then x else y

fun foldn n b f =
    let fun loop i c =
        if i < n then
            let val j = i+1
                val d = f i c
            in loop j d
            end
        else c
    in
        loop 0 b
    end

fun arraymax a =
    let
        val t = Array.length a

        fun am l m = (assert (l >= 0); let val k = Array.sub(a, l) in max k m end)
    in
        foldn t 0 am
    end
```

- Verify the array index should always be less than the length of the array

Preliminary Results

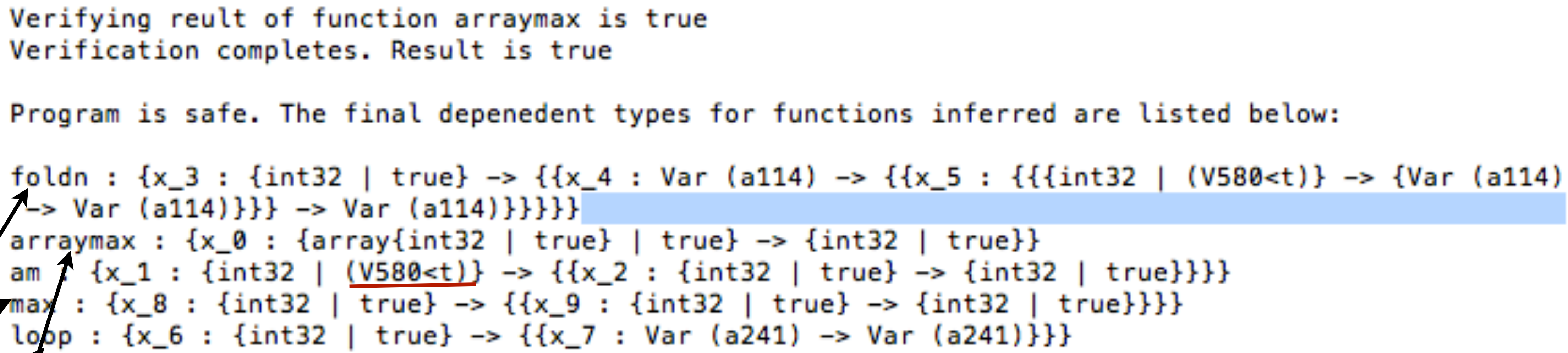


```
Verifying reult of function arraymax is true
Verification completes. Result is true

Program is safe. The final depenedent types for functions inferred are listed below:

foldn : {x_3 : {int32 | true} -> {{x_4 : Var (a114) -> {{x_5 : {{{int32 | (V580<t)}} -> {Var (a114)
-> Var (a114)}}} -> Var (a114)}}}}}
arraymax : {x_0 : {array{int32 | true} | true} -> {int32 | true}}
am : {x_1 : {int32 | (V580<t)} -> {{x_2 : {int32 | true} -> {int32 | true}}}}
max : {x_8 : {int32 | true} -> {{x_9 : {int32 | true} -> {int32 | true}}}}
loop : {x_6 : {int32 | true} -> {{x_7 : Var (a241) -> Var (a241)}}}
```

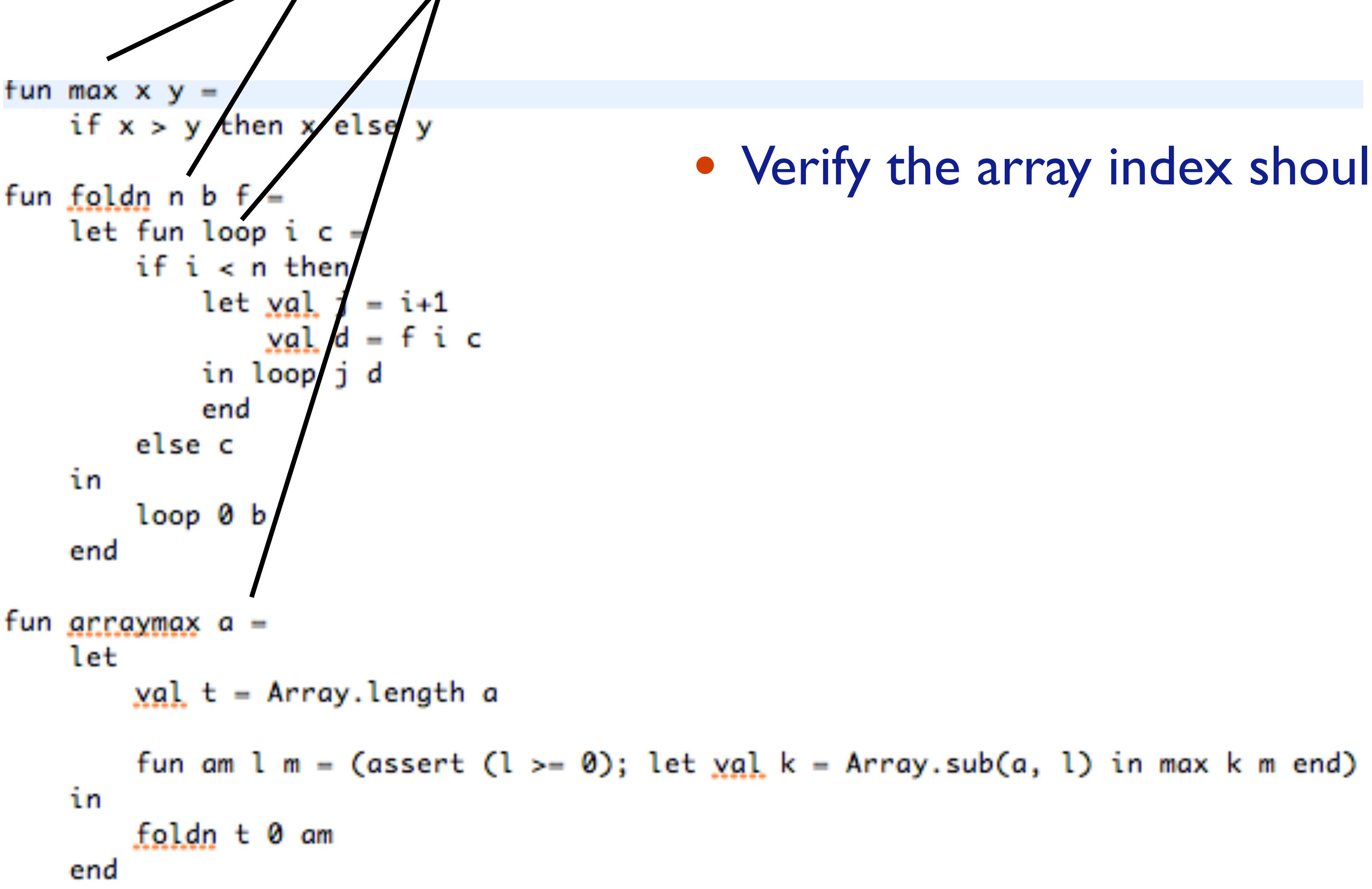
```
fun max x y =
    if x > y then x else y

fun foldn n b f =
    let fun loop i c =
        if i < n then
            let val j = i+1
                val d = f i c
            in loop j d
            end
        else c
    in
        loop 0 b
    end

fun arraymax a =
    let
        val t = Array.length a

        fun am l m = (assert (l >= 0); let val k = Array.sub(a, l) in max k m end)
    in
        foldn t 0 am
    end
```

- Verify the array index should always be less than the length of the array

Preliminary Results



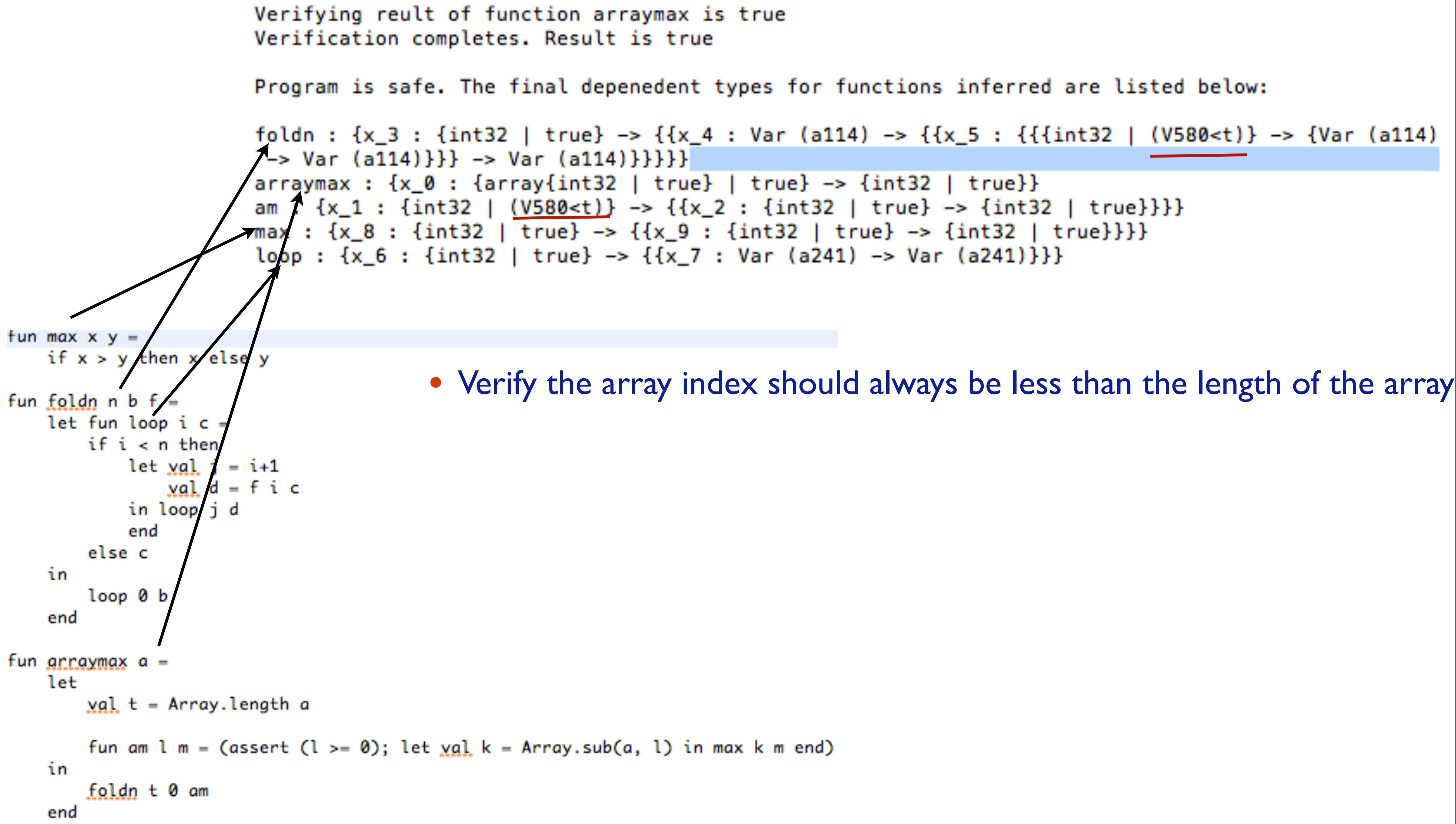
```
Verifying reult of function arraymax is true
Verification completes. Result is true

Program is safe. The final depenedent types for functions inferred are listed below:

foldn : {x_3 : {int32 | true} -> {{x_4 : Var (a114) -> {{x_5 : {{{int32 | (V580<t)} -> {Var (a114)
-> Var (a114)}}} -> Var (a114)}}}}}
arraymax : {x_0 : {array{int32 | true} | true} -> {int32 | true}}
am : {x_1 : {int32 | (V580<t)} -> {{x_2 : {int32 | true} -> {int32 | true}}}}
max : {x_8 : {int32 | true} -> {{x_9 : {int32 | true} -> {int32 | true}}}}
loop : {x_6 : {int32 | true} -> {{x_7 : Var (a241) -> Var (a241)}}}
```



```
fun max x y =
    if x > y then x else y

fun foldn n b f =
    let fun loop i c =
        if i < n then
            let val j = i+1
                val d = f i c
            in loop j d
            end
        else c
    in
        loop 0 b
    end

fun arraymax a =
    let
        val t = Array.length a

        fun am l m = (assert (l >= 0); let val k = Array.sub(a, l) in max k m end)
    in
        foldn t 0 am
    end
```

- Verify the array index should always be less than the length of the array

Preliminary Results



```
Verifying reult of function arraymax is true
Verification completes. Result is true

Program is safe. The final depenedent types for functions inferred are listed below:

foldn : {x_3 : {int32 | true} -> {{x_4 : Var (a114) -> {{x_5 : {{{int32 | (V580<t)} -> {Var (a114)
-> Var (a114)}}} -> Var (a114)}}}}}
arraymax : {x_0 : {array{int32 | true} | true} -> {int32 | true}}
am : {x_1 : {int32 | (V580<t)} -> {{x_2 : {int32 | true} -> {int32 | true}}}}
max : {x_8 : {int32 | true} -> {{x_9 : {int32 | true} -> {int32 | true}}}}
loop : {x_6 : {int32 | true} -> {{x_7 : Var (a241) -> Var (a241)}}}
```

```
fun max x y =
    if x > y then x else y

fun foldn n b f =
    let fun loop i c =
        if i < n then
            let val j = i+1
                val d = f i c
            in loop j d
            end
        else c
    in
        loop 0 b
    end

fun arraymax a =
    let
        val t = Array.length a

        fun am l m = (assert (l >= 0); let val k = Array.sub(a, l) in max k m end)
    in
        foldn t 0 am
    end
```

- Verify the array index should always be less than the length of the array

Related Work

- Liquid Types (Rondon et. al [PLDI'08], Kawaguchi et. al [PLDI'09])
  - ★ Qualifier discovery vs. selection
- Higher-Order Program Model Checking (Kobayashi et. al [PLDI'11]),
  - ★ First-order vs. higher-order verification engine
- Dependent Types from Counterexamples (Terauchi [POPL'10])
  - ★ Concrete tests vs. abstract counterexamples
- Verifying Functional Programs using Abstract Interpreters (Jhala et. al, [CAV'11])
  - ★ Program analysis vs. program transformation

Conclusions

Preliminary evidence that incorporating modular verification techniques into an optimizing compiler is feasible

- ★ A first step towards devising optimizations that leverage automatically derived "rich" specifications