Reasoning about Deltas —
Even Doing Nothing is Difficult

22nd annual HCSS Conference
18 May 22

Hira Syeda — Applied Scientist
Amazon Web Services

Takeaway

- Problem: reasoning about specifications deltas is difficult
- Need: automated proof engineering
- Solution: typeCart, an analysis tool

Takeaway

- Problem: reasoning about specifications deltas is difficult
- Need: automated proof engineering
- Solution: typeCart, an analysis tool
  - Compares two versions of specifications
  - Generates boilerplate code relating the two versions
  - Semi-automates reasoning about two versions of specifications

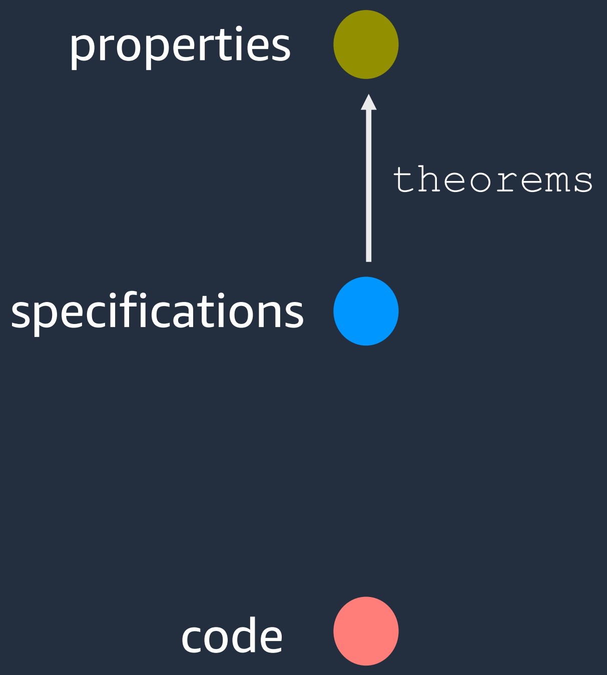
Takeaway

- Problem: reasoning about specifications deltas is difficult
- Need: automated proof engineering
- Solution: typeCart, an analysis tool
  - Compares two versions of specifications
  - Generates boilerplate code relating the two versions
  - Semi-automates reasoning about two versions of specifications
- Application: relating two specification versions of AWS authorization engine

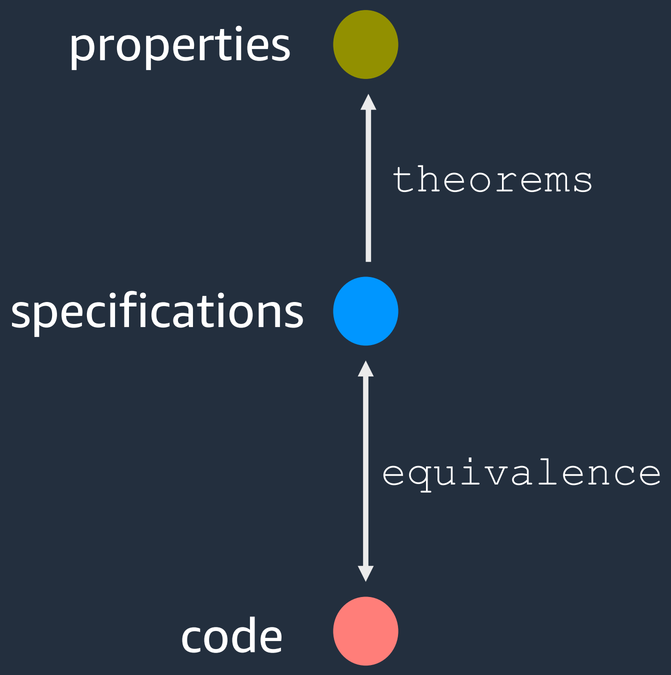
Evolving Specifications for Evolving Implementations

specifications
code

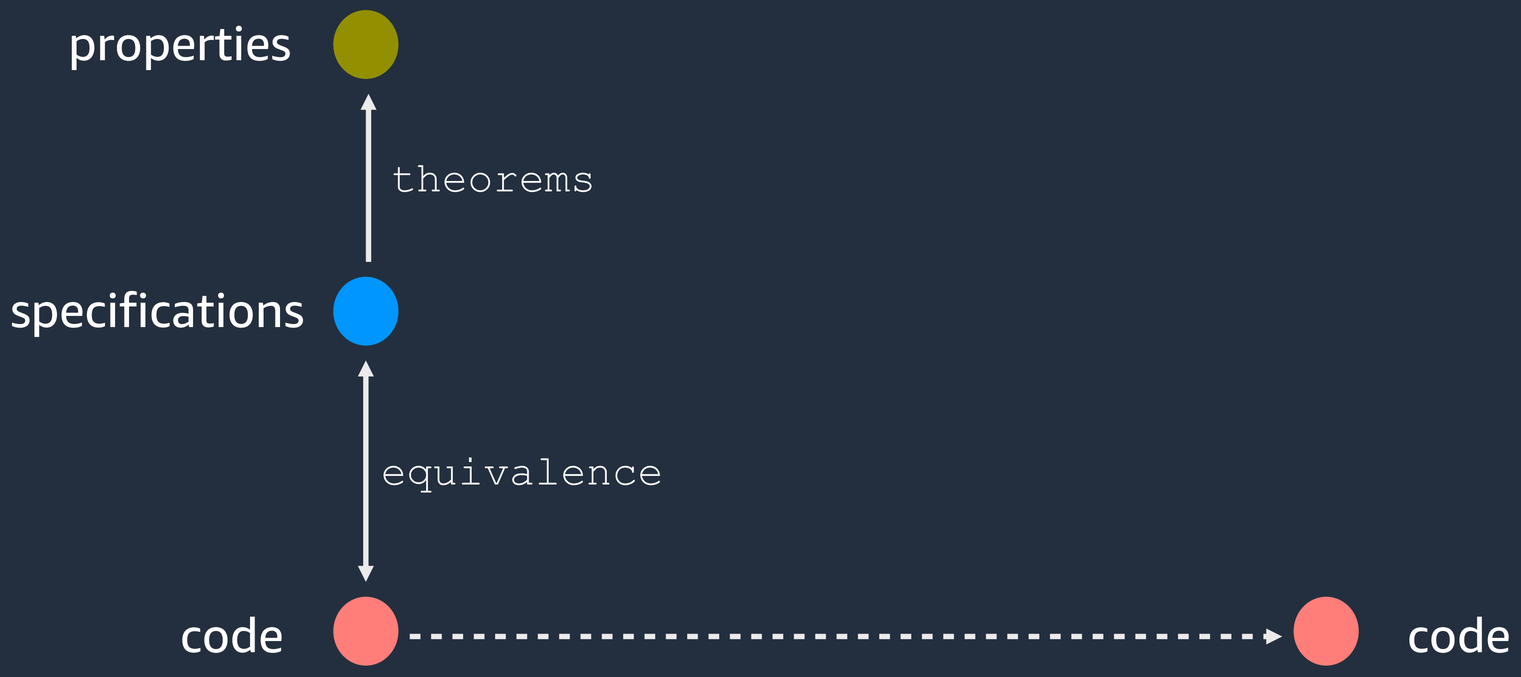
Evolving Specifications for Evolving Implementations



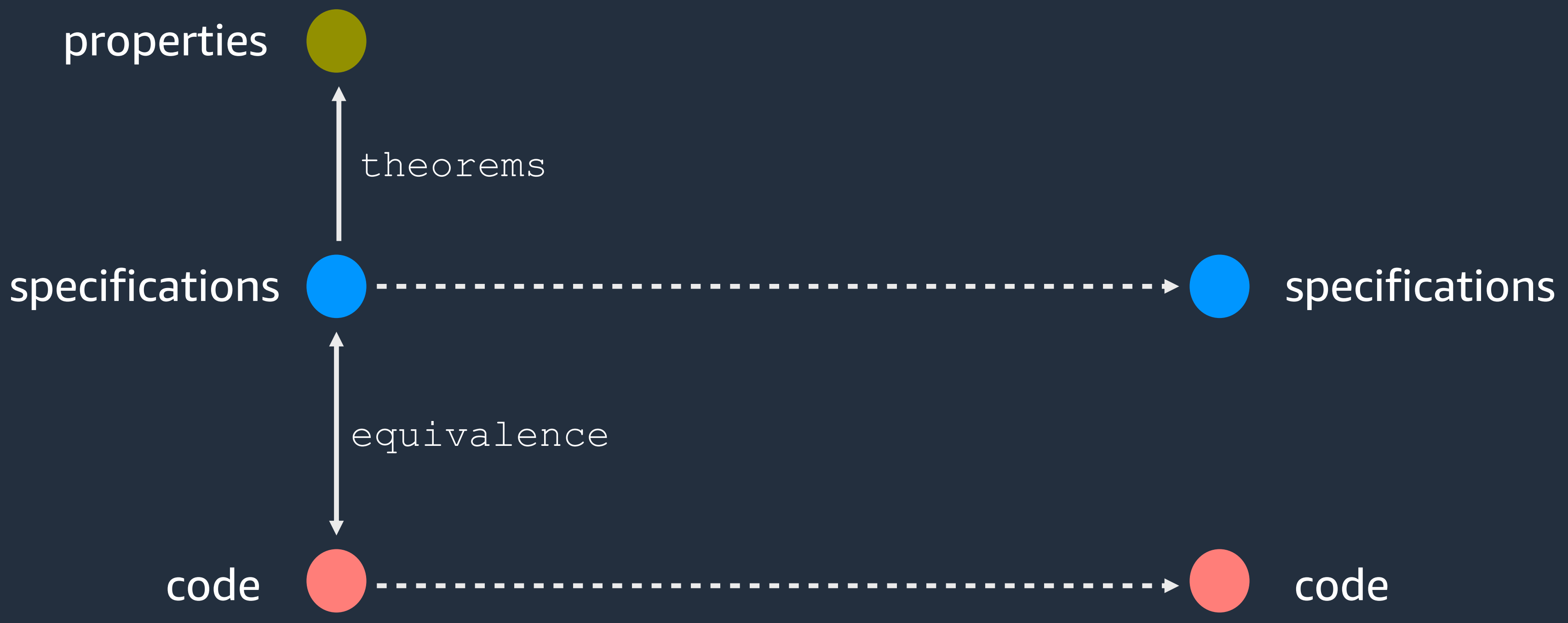
Evolving Specifications for Evolving Implementations



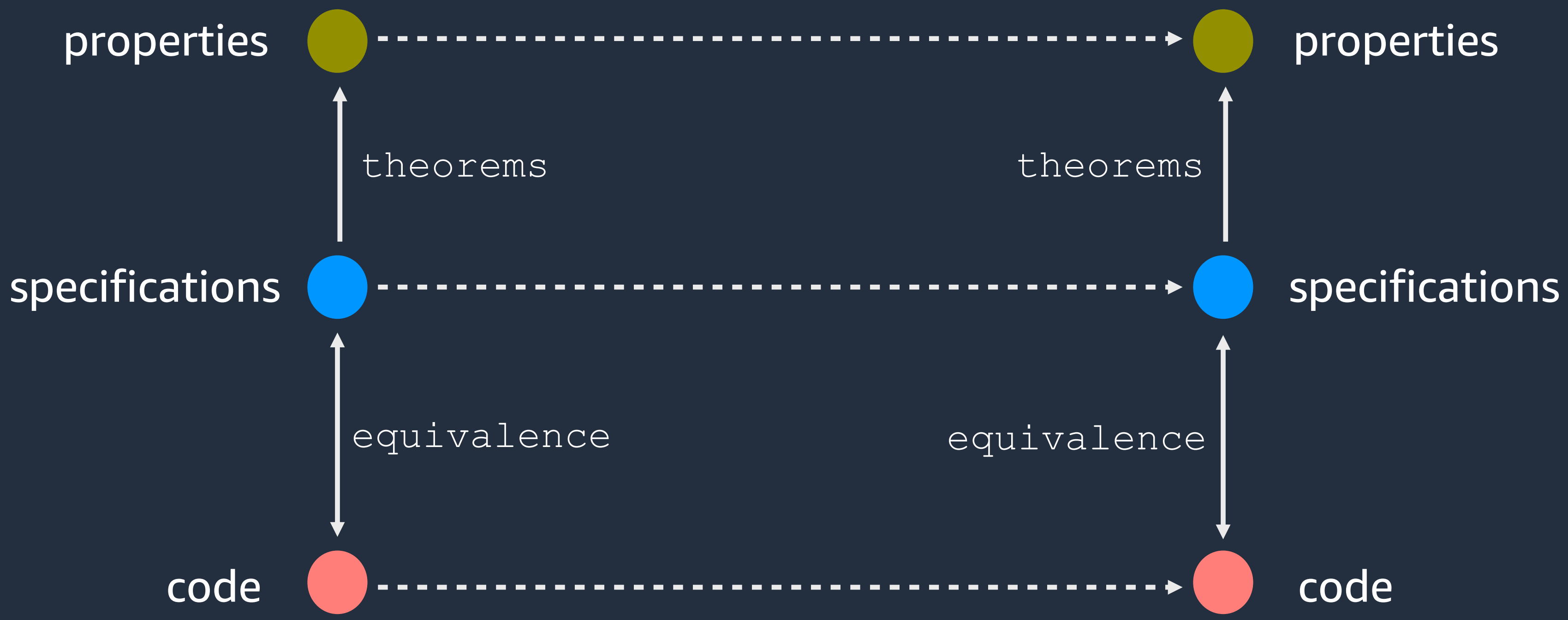
Evolving Specifications for Evolving Implementations



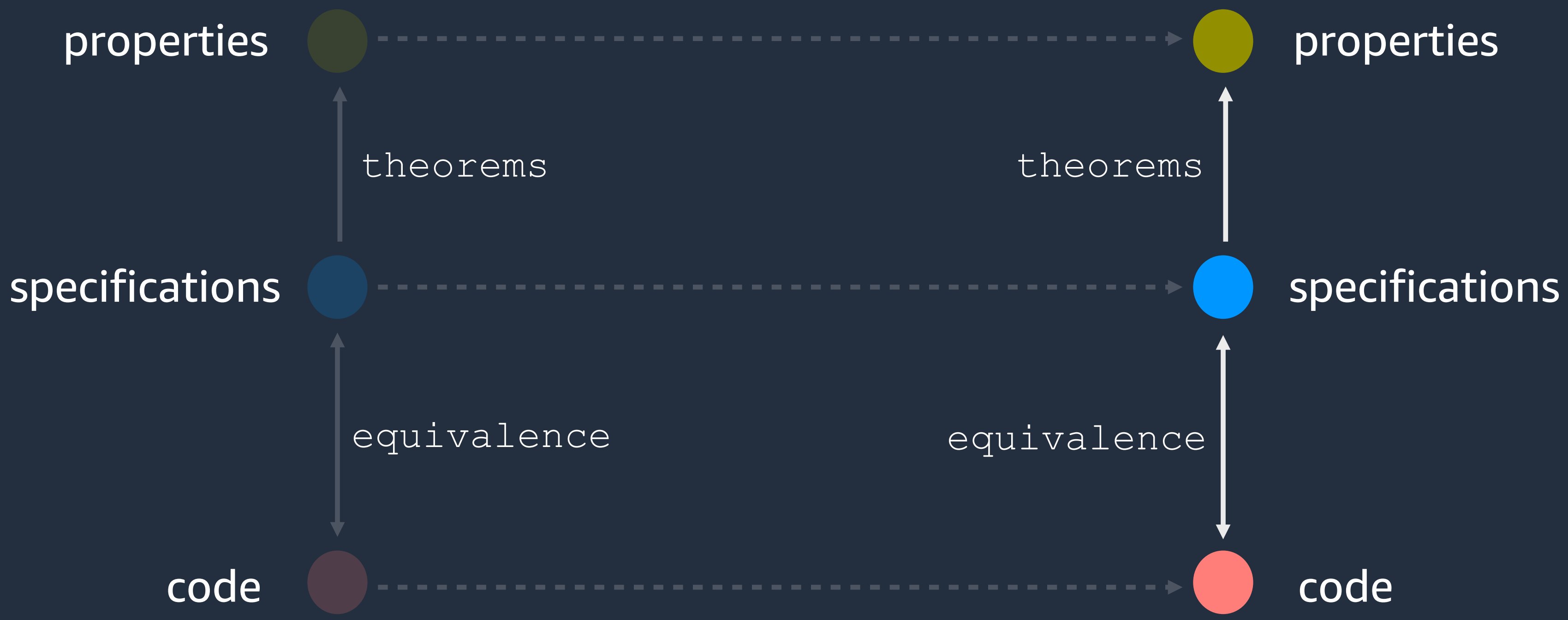
Evolving Specifications for Evolving Implementations



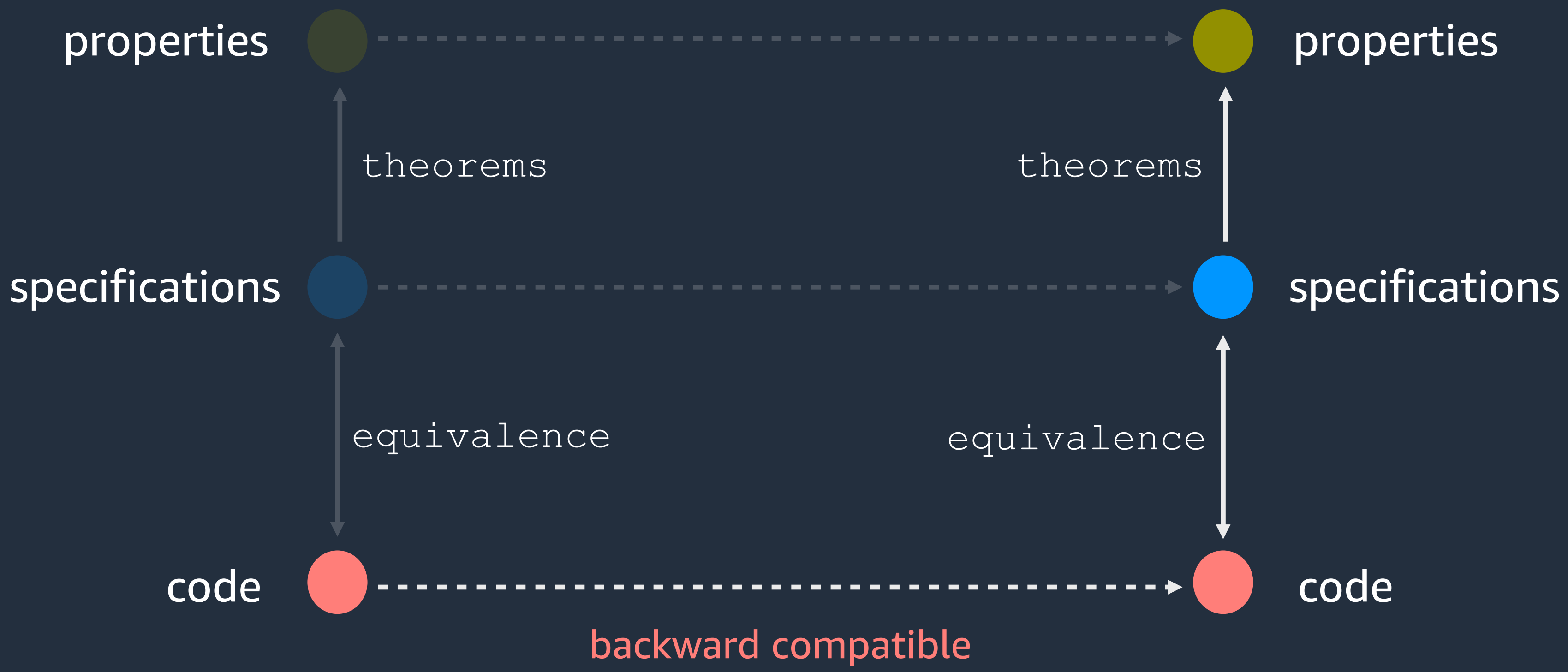
Evolving Specifications for Evolving Implementations



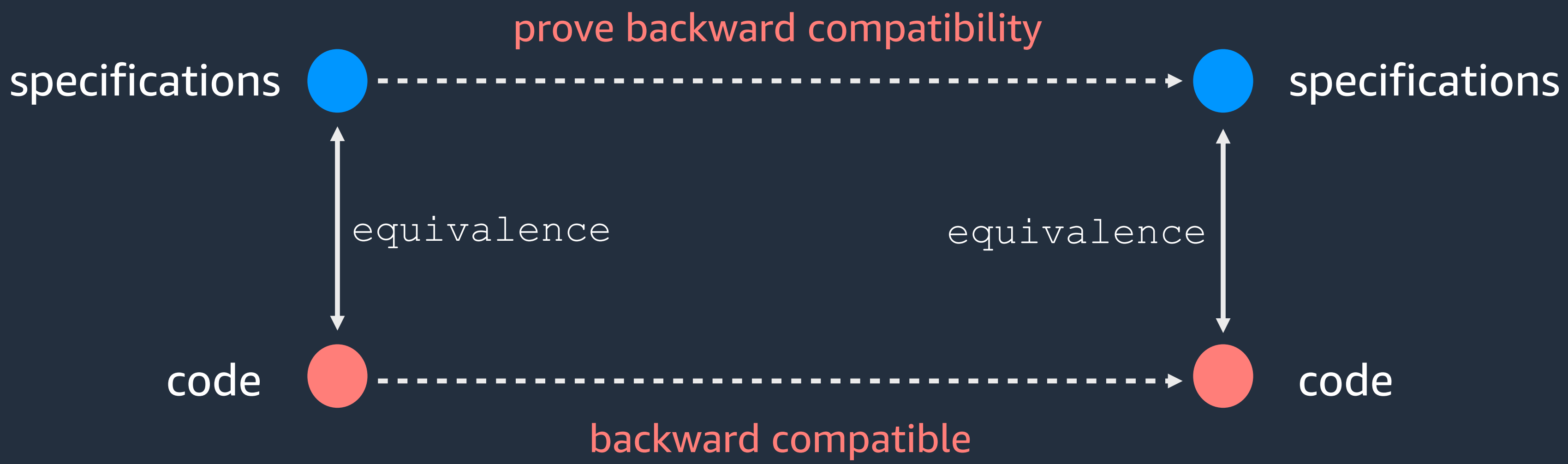
Evolving Specifications for Evolving Implementations



Evolving Specifications for Evolving Implementations

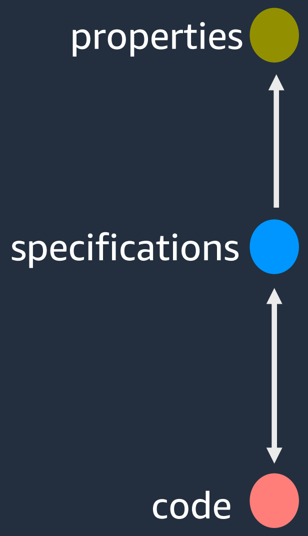


Evolving Specifications for Evolving Implementations



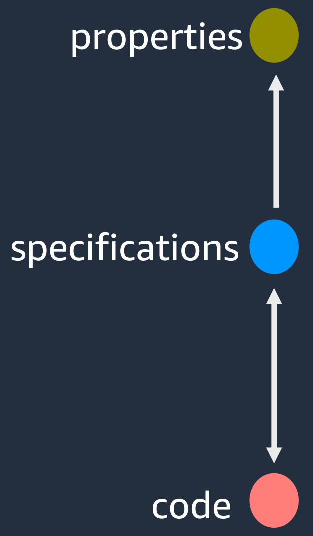
Verification is an Achievement

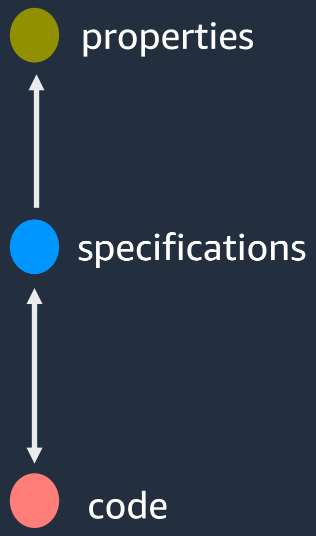
- Verifying a system once is already an achievement



Verification is an Achievement

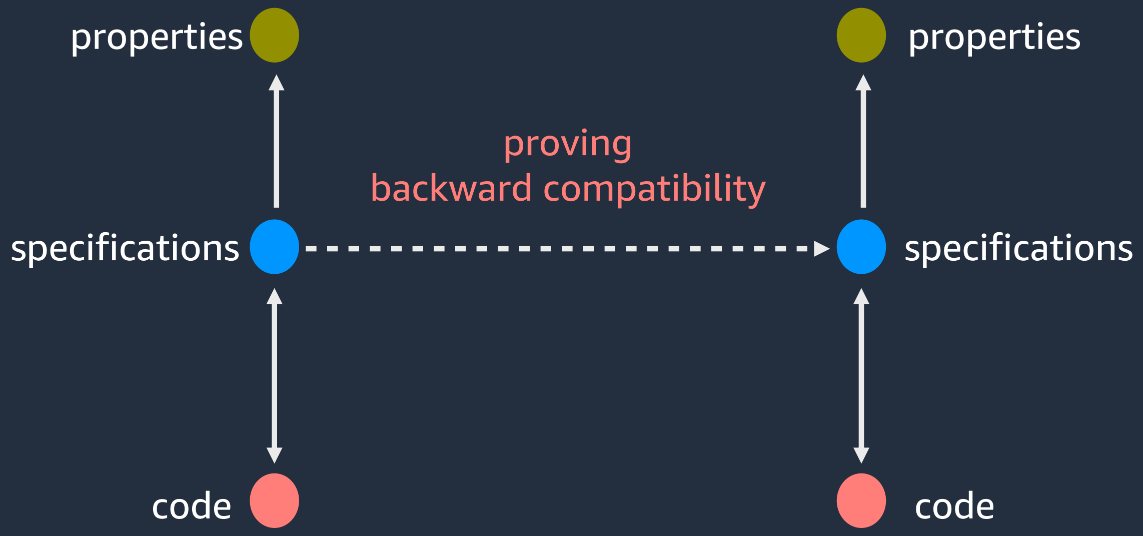
- Verifying a system once is already an achievement
- Maintaining verification artefacts is already a challenge





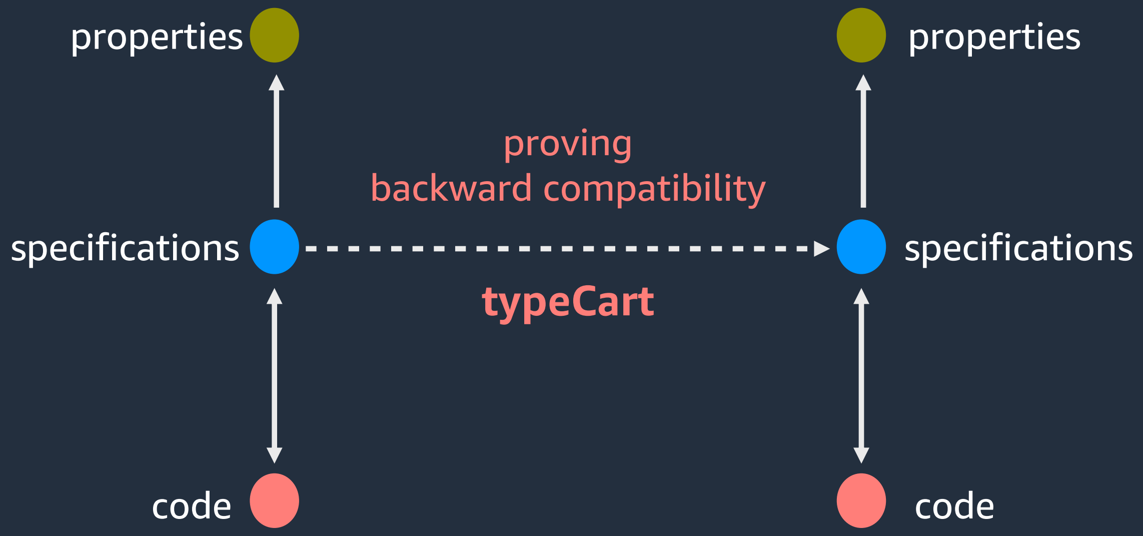
Verification is an Achievement

- Verifying a system once is already an achievement
- Maintaining verification artefacts is already a challenge
- Proving backward compatibility is an add-on



Verification is an Achievement

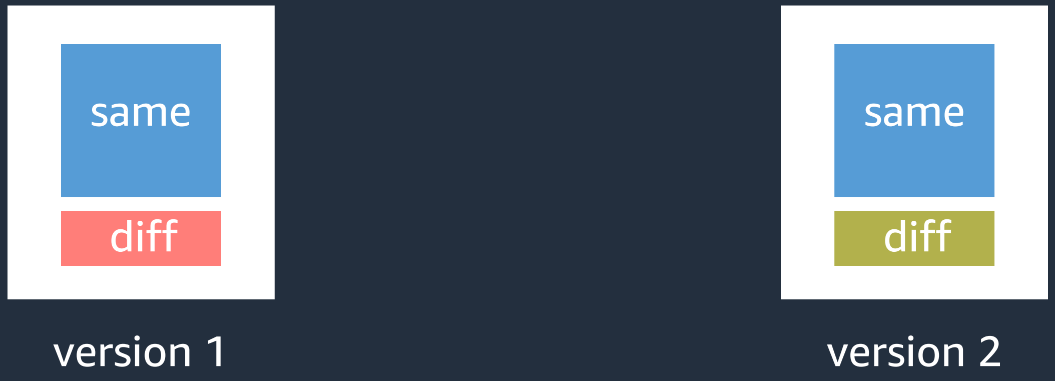
- Verifying a system once is already an achievement
- Maintaining verification artefacts is already a challenge
- Proving backward compatibility is an add-on
- typeCart: automates the process of relating specifications



Is Relating any Two Specifications Difficult?

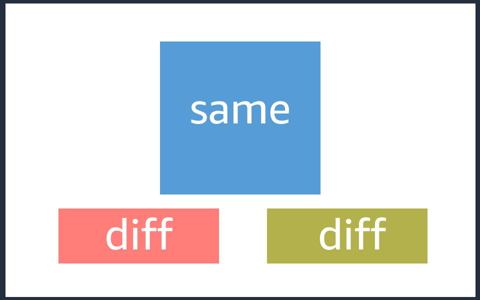
Is Relating any Two Specifications Difficult?

- Appears straightforward
  - The two versions represent the same implementation with some deltas
  - They model and specify relatable objects



Is Relating any Two Specifications Difficult?

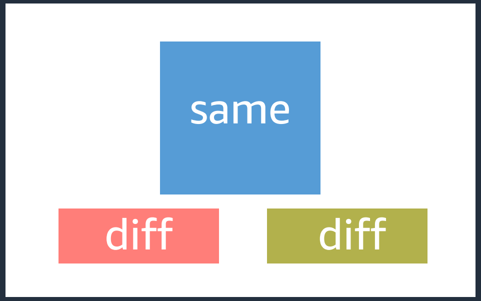
- Appears straightforward
  - The two versions represent the same implementation with some deltas
  - They model and specify relatable objects
- Relating equivalent objects between specifications should be trivial



version 1 and version 2

Is Relating any Two Specifications Difficult?

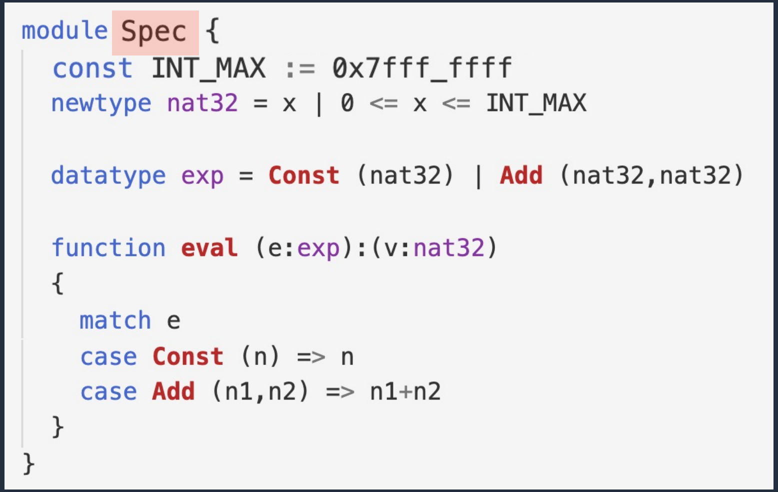
- Relating equivalent objects between specifications should be trivial
- Let's test our hypothesis!



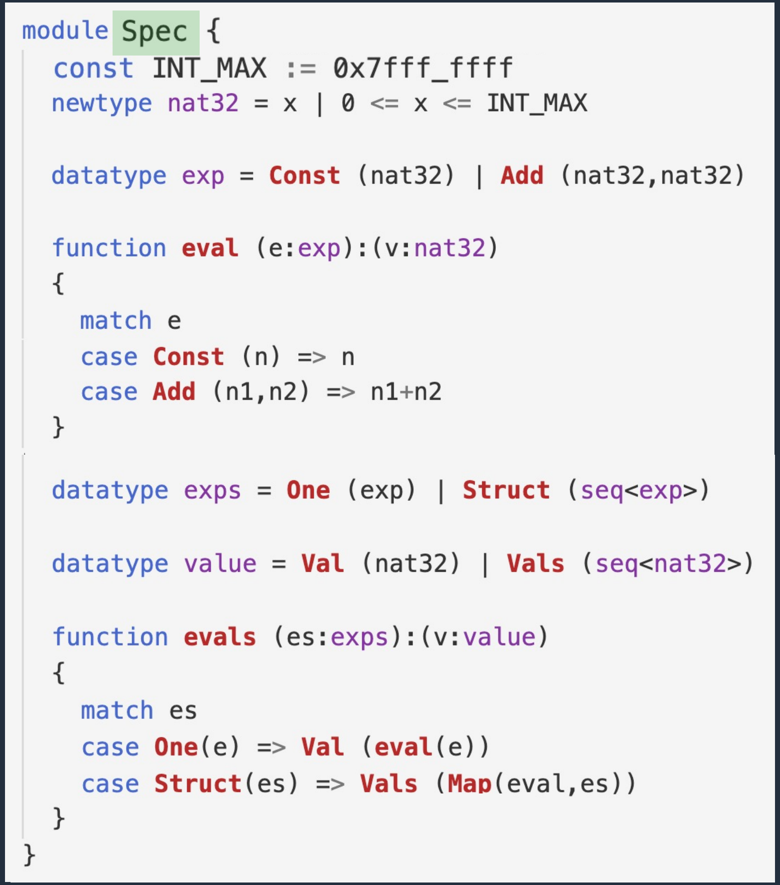
version 1 and version 2

Reasoning about Deltas

- Relating equivalent objects between specifications should be trivial



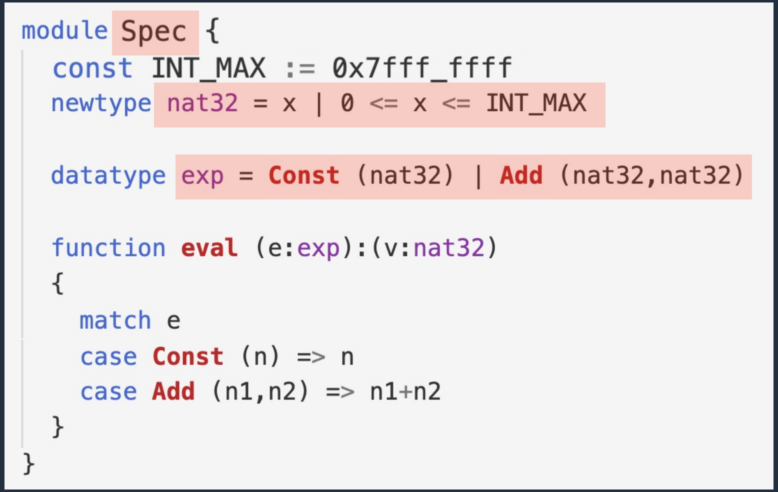
```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```



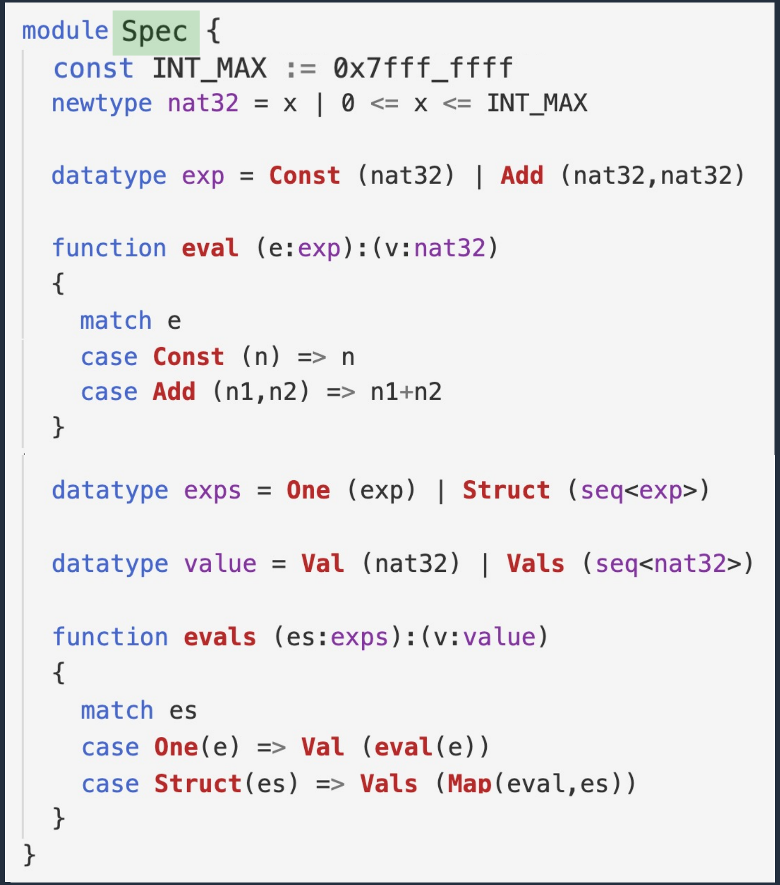
```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }

  datatype exps = One (exp) | Struct (seq<exp>)

  datatype value = Val (nat32) | Vals (seq<nat32>)

  function evals (es:exps):(v:value)
  {
    match es
    case One(e) => Val (eval(e))
    case Struct(es) => Vals (Map(eval,es))
  }
}
```

Reasoning about Deltas

- Relating equivalent objects between specifications should be trivial



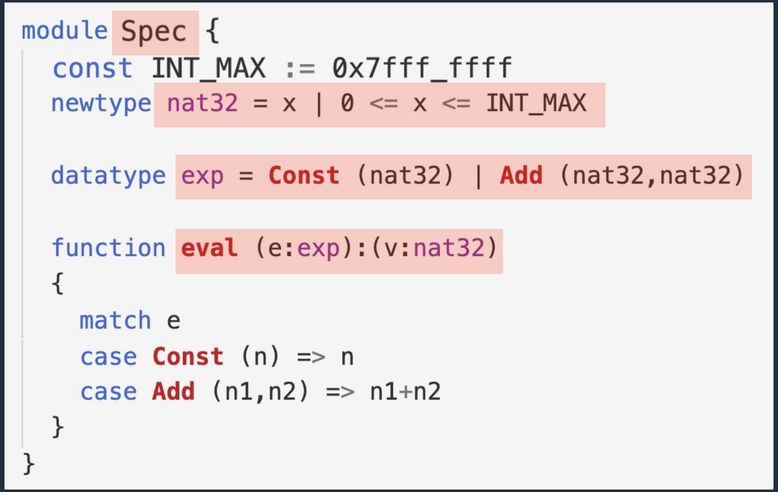
```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```



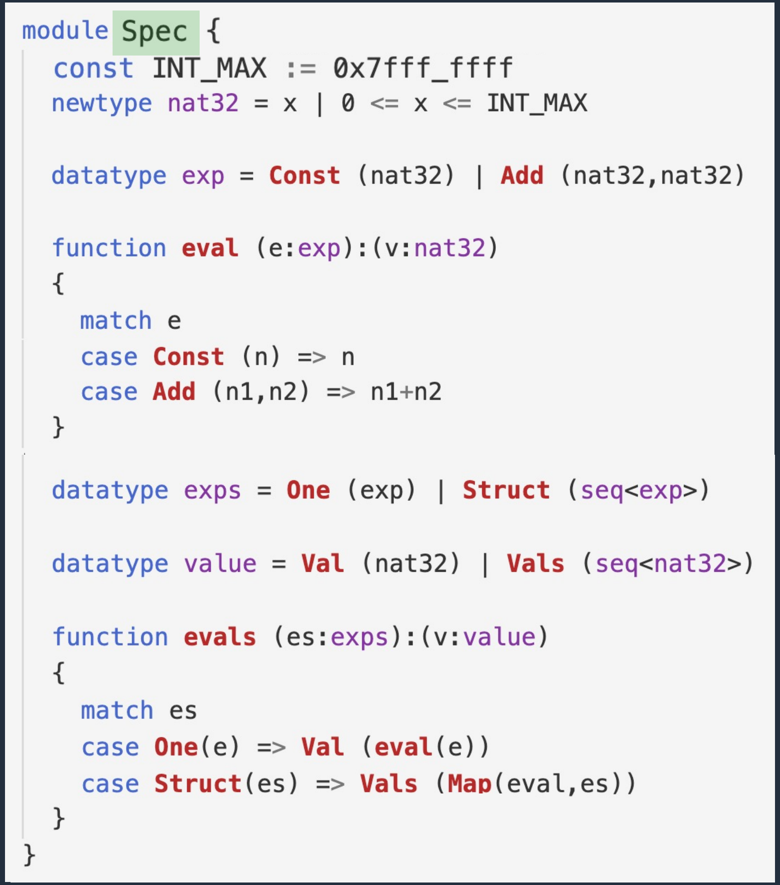
```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }

  datatype exps = One (exp) | Struct (seq<exp>)

  datatype value = Val (nat32) | Vals (seq<nat32>)

  function evals (es:exps):(v:value)
  {
    match es
    case One(e) => Val (eval(e))
    case Struct(es) => Vals (Map(eval,es))
  }
}
```

Reasoning about Deltas

- Relating equivalent objects between specifications should be trivial



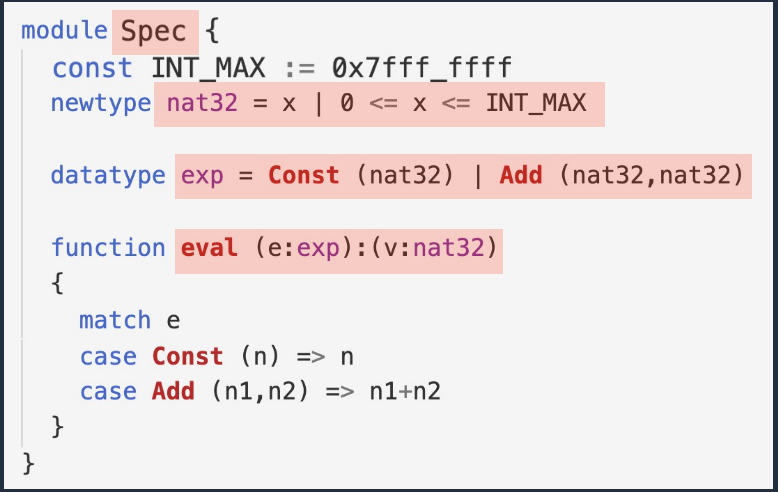
```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```



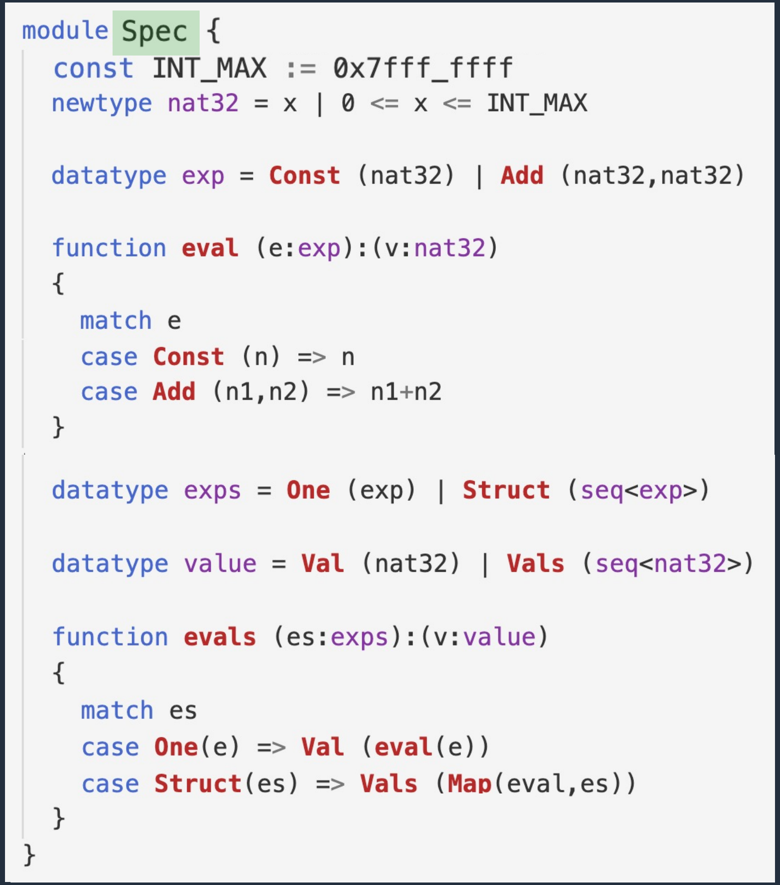
```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }

  datatype exps = One (exp) | Struct (seq<exp>)

  datatype value = Val (nat32) | Vals (seq<nat32>)

  function evals (es:exps):(v:value)
  {
    match es
    case One(e) => Val (eval(e))
    case Struct(es) => Vals (Map(eval,es))
  }
}
```

Reasoning about Deltas

- Relating equivalent objects between specifications should be trivial



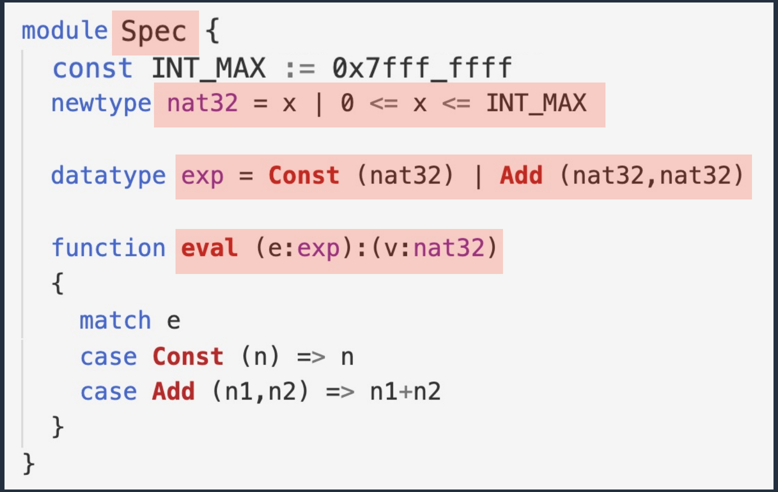
```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```



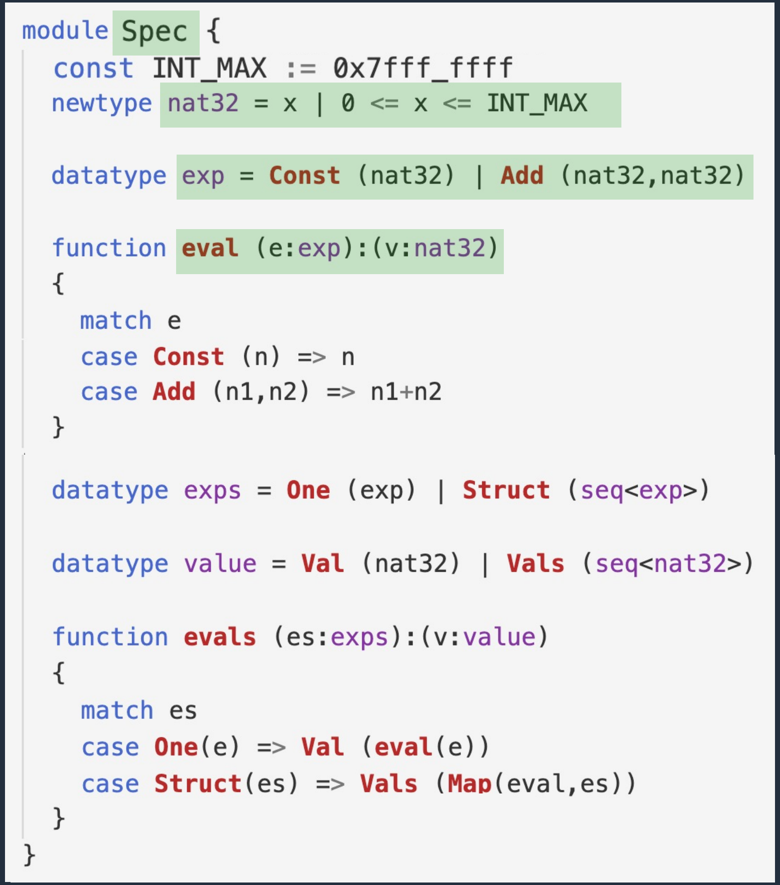
```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }

  datatype exps = One (exp) | Struct (seq<exp>)

  datatype value = Val (nat32) | Vals (seq<nat32>)

  function evals (es:exps):(v:value)
  {
    match es
    case One(e) => Val (eval(e))
    case Struct(es) => Vals (Map(eval,es))
  }
}
```

Reasoning about Deltas

- Relating equivalent objects between specifications should be trivial



```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```



```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }

  datatype exps = One (exp) | Struct (seq<exp>)

  datatype value = Val (nat32) | Vals (seq<nat32>)

  function evals (es:exps):(v:value)
  {
    match es
    case One(e) => Val (eval(e))
    case Struct(es) => Vals (Map(eval,es))
  }
}
```

Reasoning about Deltas

- Relating equivalent objects between specifications should be trivial

```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```

```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }

  datatype exps = One (exp) | Struct (seq<exp>)

  datatype value = Val (nat32) | Vals (seq<nat32>)

  function evals (es:exps):(v:value)
  {
    match es
    case One(e) => Val (eval(e))
    case Struct(es) => Vals (Map(eval,es))
  }
}
```

Reasoning about Deltas

- Relating **equivalent objects** between specifications should be trivial

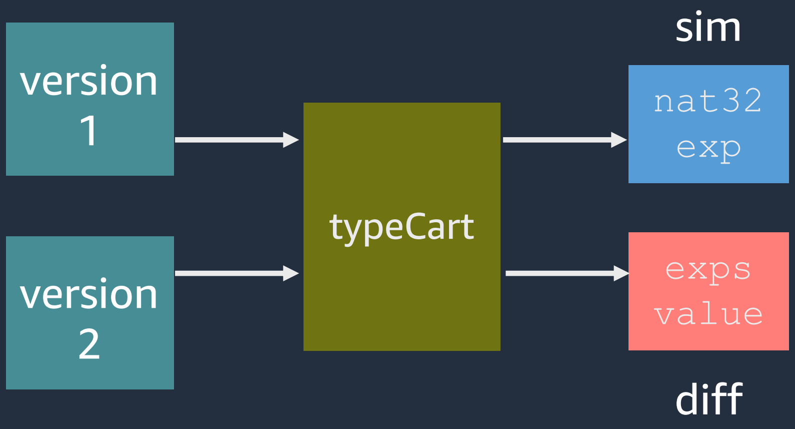
```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```

```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }

  datatype exps = One (exp) | Struct (seq<exp>)

  datatype value = Val (nat32) | Vals (seq<nat32>)

  function evals (es:exps):(v:value)
  {
    match es
    case One(e) => Val (eval(e))
    case Struct(es) => Vals (Map(eval,es))
  }
}
```

Reasoning about Deltas

- Relating **equivalent objects** between specifications should be trivial

```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```



```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }

  datatype exps = One (exp) | Struct (seq<exp>)

  datatype value = Val (nat32) | Vals (seq<nat32>)

  function evals (es:exps):(v:value)
  {
    match es
    case One(e) => Val (eval(e))
    case Struct(es) => Vals (Map(eval,es))
  }
}
```

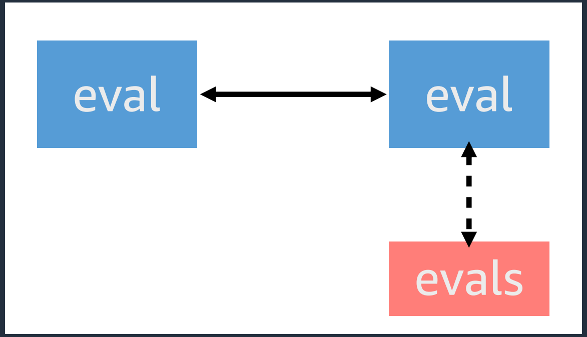
Reasoning about Deltas

- Relating equivalent objects between specifications should be trivial

```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```

```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }

  datatype exps = One (exp) | Struct (seq<exp>)

  datatype value = Val (nat32) | Vals (seq<nat32>)

  function evals (es:exps):(v:value)
  {
    match es
    case One(e) => Val (eval(e))
    case Struct(es) => Vals (Map(eval,es))
  }
}
```

sim



backward compatibility

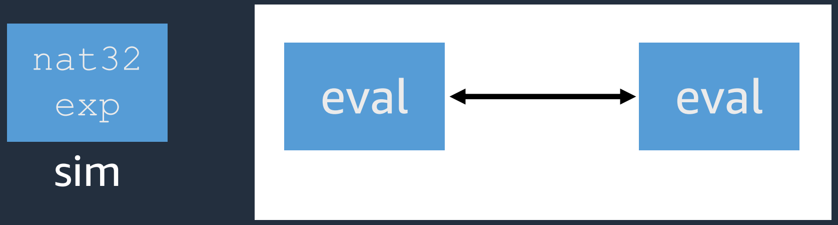
Reasoning about Deltas

- Relating equivalent objects between specifications should be trivial

```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```

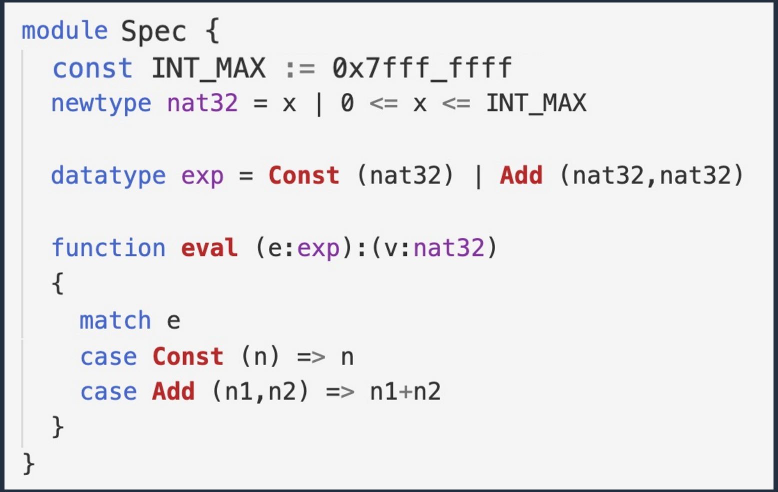
```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```

both `eval` functions evaluate to the same answer

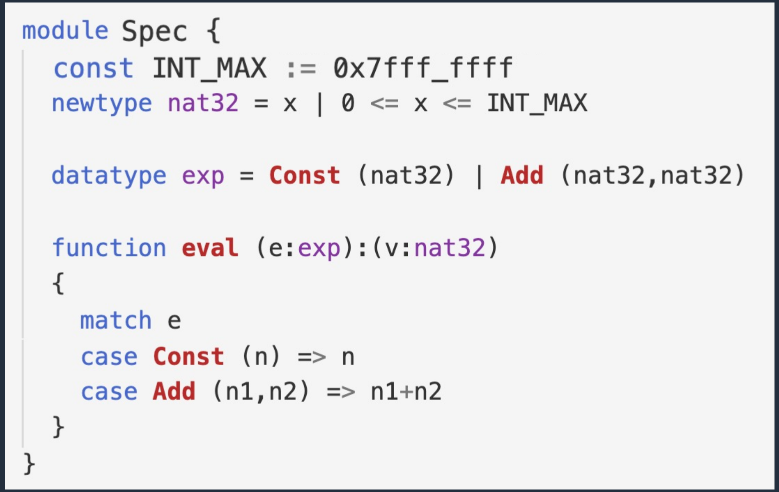


Reasoning about Deltas

- Relating equivalent objects between specifications should be trivial

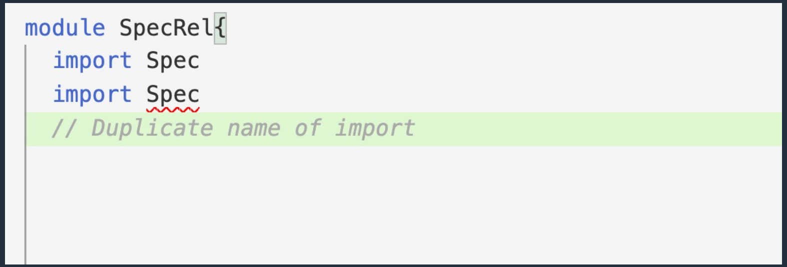


```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```



```
module Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```

both `eval` functions evaluate to the same answer



```
module SpecRel{
  import Spec
  import Spec
  // Duplicate name of import
```

Reasoning about Deltas

- Relating equivalent objects between specifications should be trivial

```
module Old.Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```

```
module New.Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```

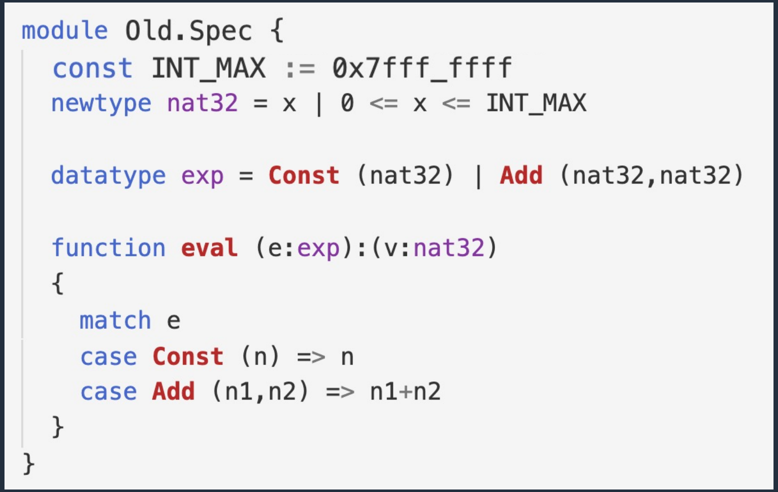
<u>both `eval` functions evaluate to the same answer</u>

namespaces for different versions

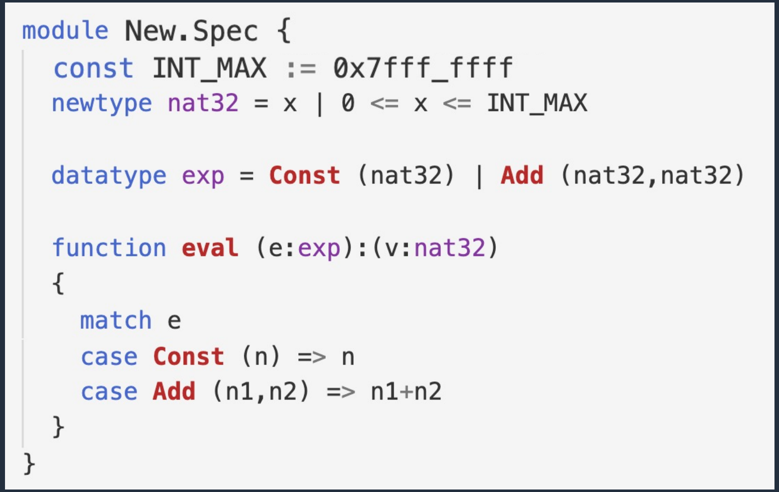
```
module SpecRel{
  import Old
  import New
```

Reasoning about Deltas

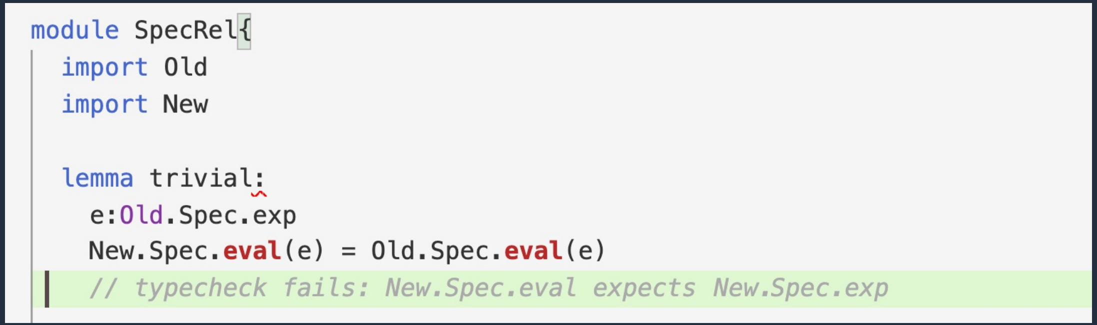
- Relating equivalent objects between specifications should be trivial



```
module Old.Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```



```
module New.Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```



```
module SpecRel{
  import Old
  import New

  lemma trivial:
    e:Old.Spec.exp
    New.Spec.eval(e) = Old.Spec.eval(e)
    // typecheck fails: New.Spec.eval expects New.Spec.exp
```

Reasoning about Deltas

- Relating equivalent objects between specifications should be trivial

```
module Old.Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```

```
module New.Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```

type casting
functions

```
module SpecRel{
  import Old
  import New

  lemma trivial:
   New.Spec.eval(expOldtoNew(e)) = Old.Spec.eval(e)
  // typecheck fails: New.Spec.nat32 and Old.Spec.nat32
```

Reasoning about Deltas

- Relating equivalent objects between specifications should be trivial

```
module Old.Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```

```
module New.Spec {
  const INT_MAX := 0x7fff_ffff
  newtype nat32 = x | 0 <= x <= INT_MAX

  datatype exp = Const (nat32) | Add (nat32,nat32)

  function eval (e:exp):(v:nat32)
  {
    match e
    case Const (n) => n
    case Add (n1,n2) => n1+n2
  }
}
```

type casting functions

```
module SpecRel{
  import Old
  import New

  lemma trivial:
    New.Spec.eval(expOldtoNew(e)) = nat32OldtoNew(Old.Spec.eval(e))
    // typecheck succeeds
```

Reasoning about Deltas

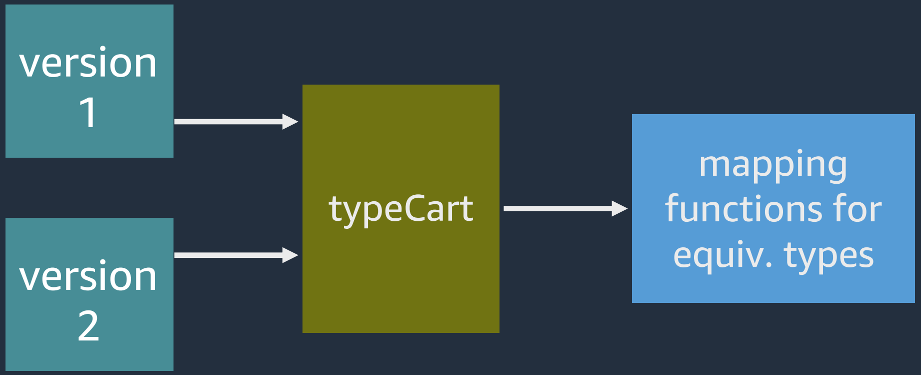
- Hypothesis: relating equivalent objects between specifications should be trivial
  - Yes, but only with type injections (mapping functions)

Reasoning about Deltas

- Hypothesis: relating equivalent objects between specifications should be trivial
  - Yes, but only with type injections (mapping functions)
- Different types in any verification effort, defined in multiple modules
  - Recursive, parametric, constrained, collection types

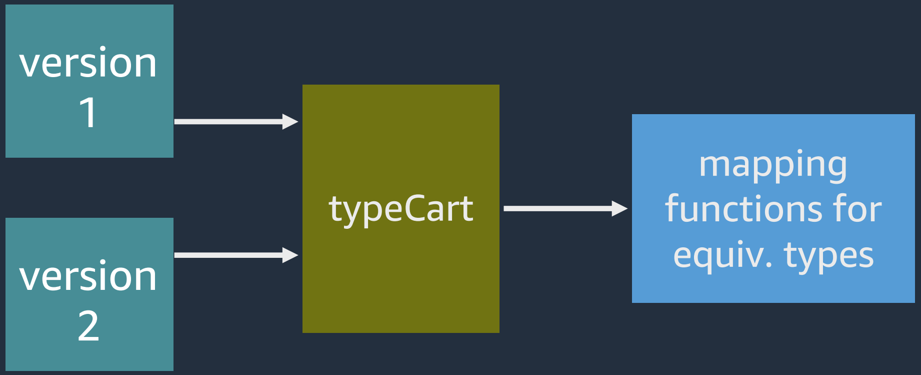
Reasoning about Deltas

- Hypothesis: relating equivalent objects between specifications should be trivial
  - Yes, but only with type injections (mapping functions)
- Different types in any verification effort, defined in multiple modules
  - Recursive, parametric, constrained, collection types
- typeCart
  - Identifies syntactically equivalent types between Dafny files
  - Generates mapping functions with verification conditions between equivalent types



Reasoning about Deltas

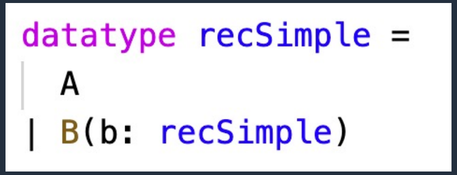
- Hypothesis: relating equivalent objects between specifications should be trivial
  - Yes, but only with type injections (mapping functions)
- Different types in any verification effort, defined in multiple modules
  - Recursive, parametric, constrained, collection types
- typeCart
  - Identifies syntactically equivalent types between Dafny files
  - Generates mapping functions with verification conditions between equivalent types



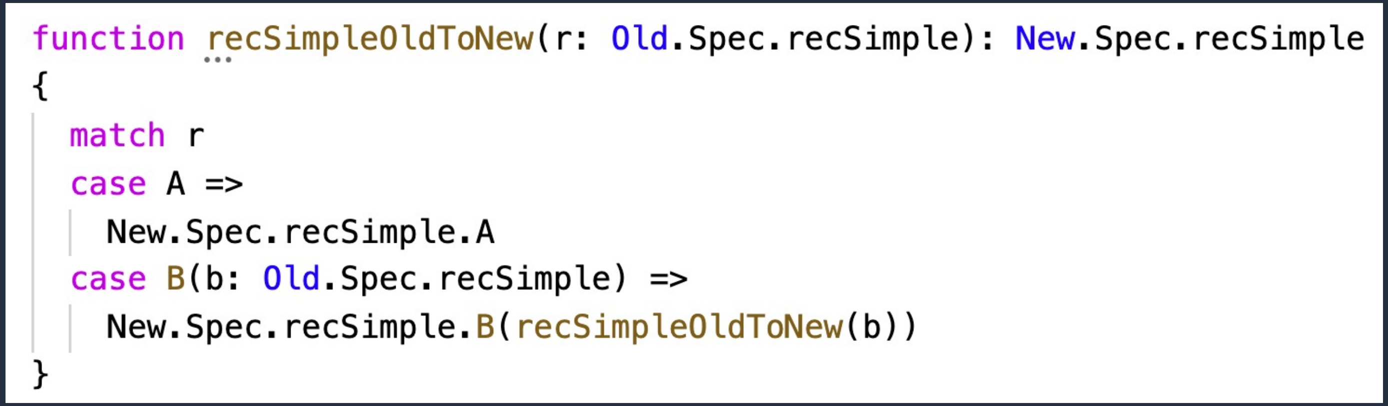
Fun Fact
typeCart: name-blend of **type** and **cartography** (practice of making maps)

typeCart — Examples

- Recursive datatype



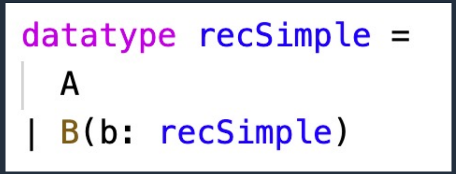
```
datatype recSimple =
    A
  | B(b: recSimple)
```



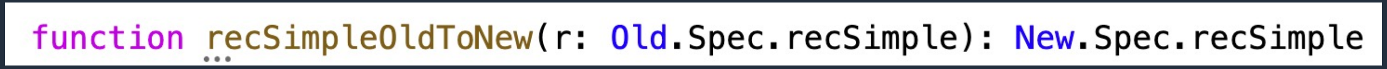
```
function recSimpleOldToNew(r: Old.Spec.recSimple): New.Spec.recSimple
{
  match r
  case A =>
    New.Spec.recSimple.A
  case B(b: Old.Spec.recSimple) =>
    New.Spec.recSimple.B(recSimpleOldToNew(b))
}
```

typeCart — Examples

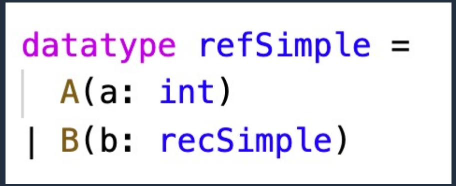
- Recursive datatype



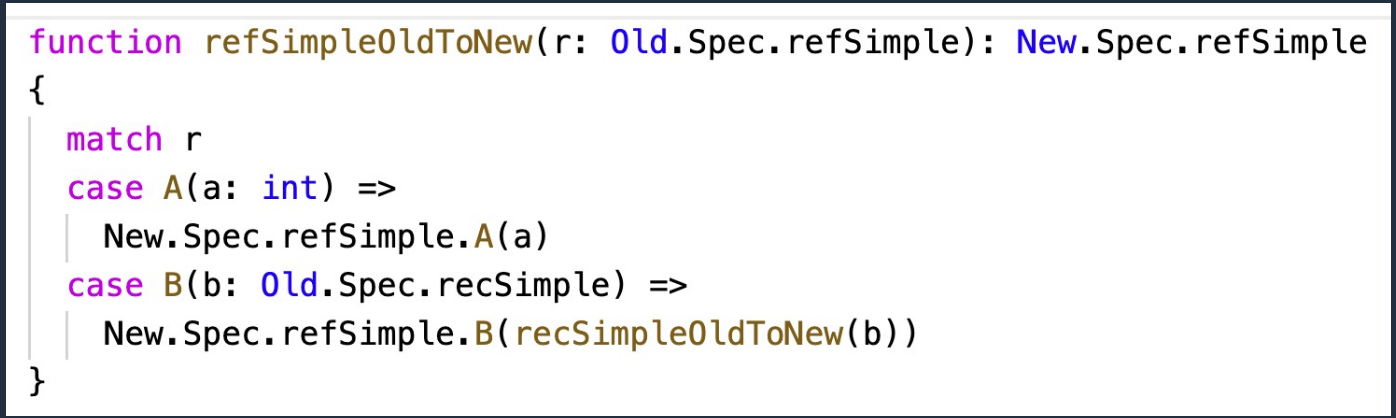
```
datatype recSimple =
  A
| B(b: recSimple)
```



```
function recSimpleOldToNew(r: Old.Spec.recSimple): New.Spec.recSimple
```



```
datatype refSimple =
  A(a: int)
| B(b: recSimple)
```



```
function refSimpleOldToNew(r: Old.Spec.refSimple): New.Spec.refSimple
{
  match r
  case A(a: int) =>
    New.Spec.refSimple.A(a)
  case B(b: Old.Spec.recSimple) =>
    New.Spec.refSimple.B(recSimpleOldToNew(b))
}
```

- Types that refer others

typeCart — Examples

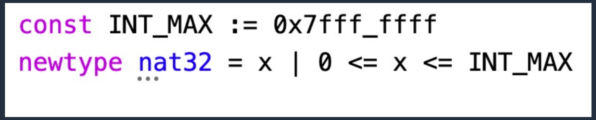
- Parametric types

```
datatype either<S, T> =
  Left(s: S)
| Right (t: T)
```

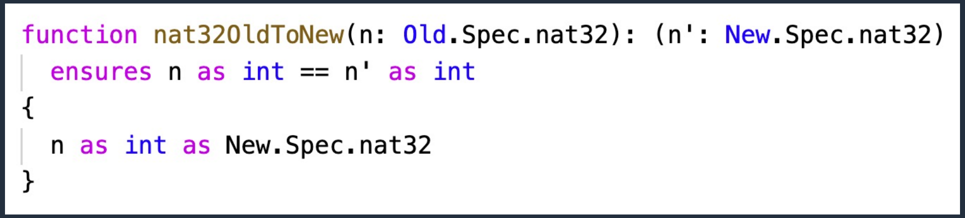
```
function eitherOldToNew<S, T, S', T'>
  (fS: S -> S', fT: T -> T', e: Old.Spec.either<S, T>):
  New.Spec.either<S', T'>
{
  match e
  case Left(s: S) =>
    New.Spec.either.Left(fS(s))
  case Right(t: T) =>
    New.Spec.either.Right(fT(t))
}
```

typeCart — Examples

- Bounded types



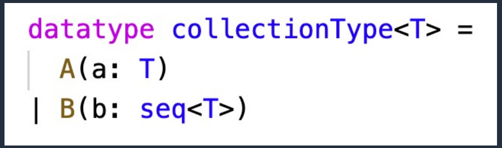
```
const INT_MAX := 0x7fff_ffff
newtype nat32 = x | 0 <= x <= INT_MAX
```



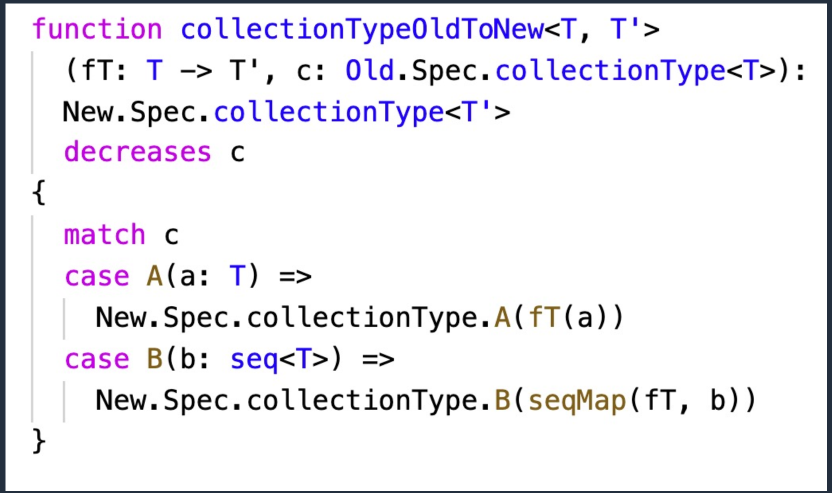
```
function nat32OldToNew(n: Old.Spec.nat32): (n': New.Spec.nat32)
  ensures n as int == n' as int
{
  n as int as New.Spec.nat32
}
```

typeCart — Examples

- Collection types

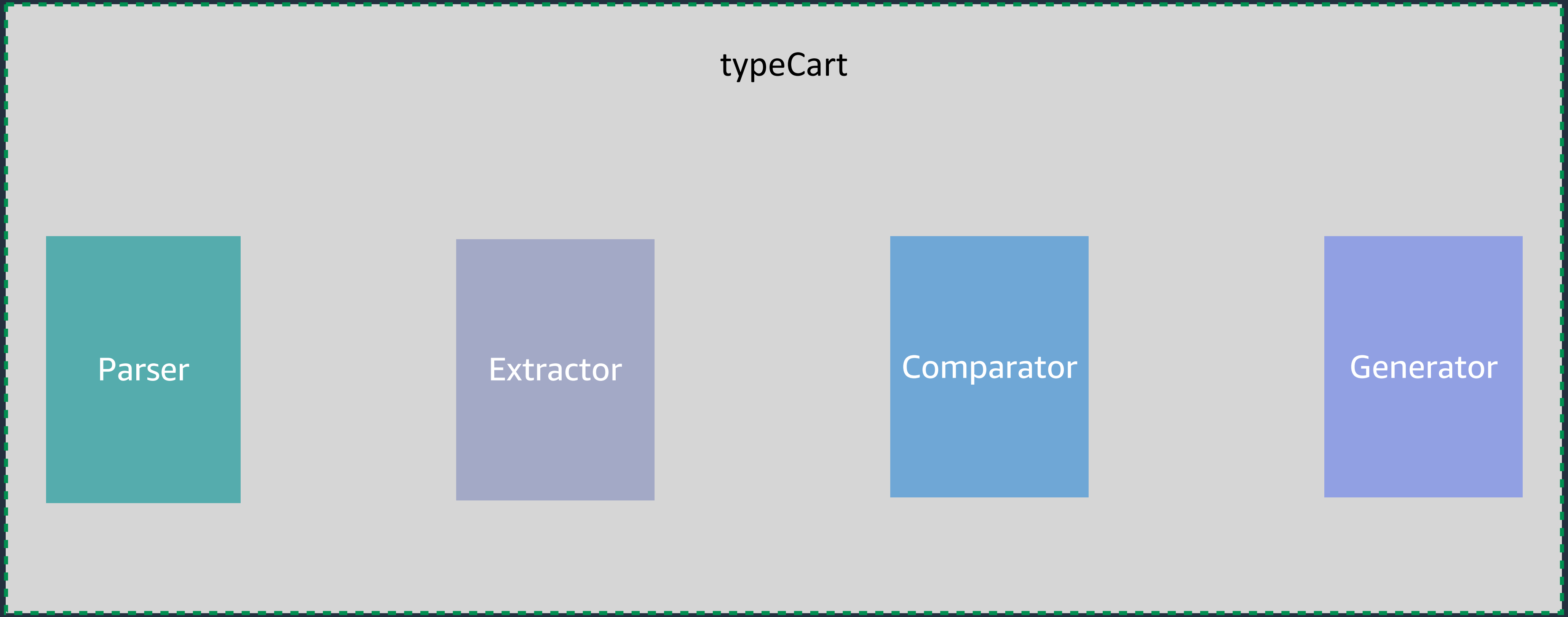


```
datatype collectionType<T> =
    A(a: T)
  | B(b: seq<T>)
```

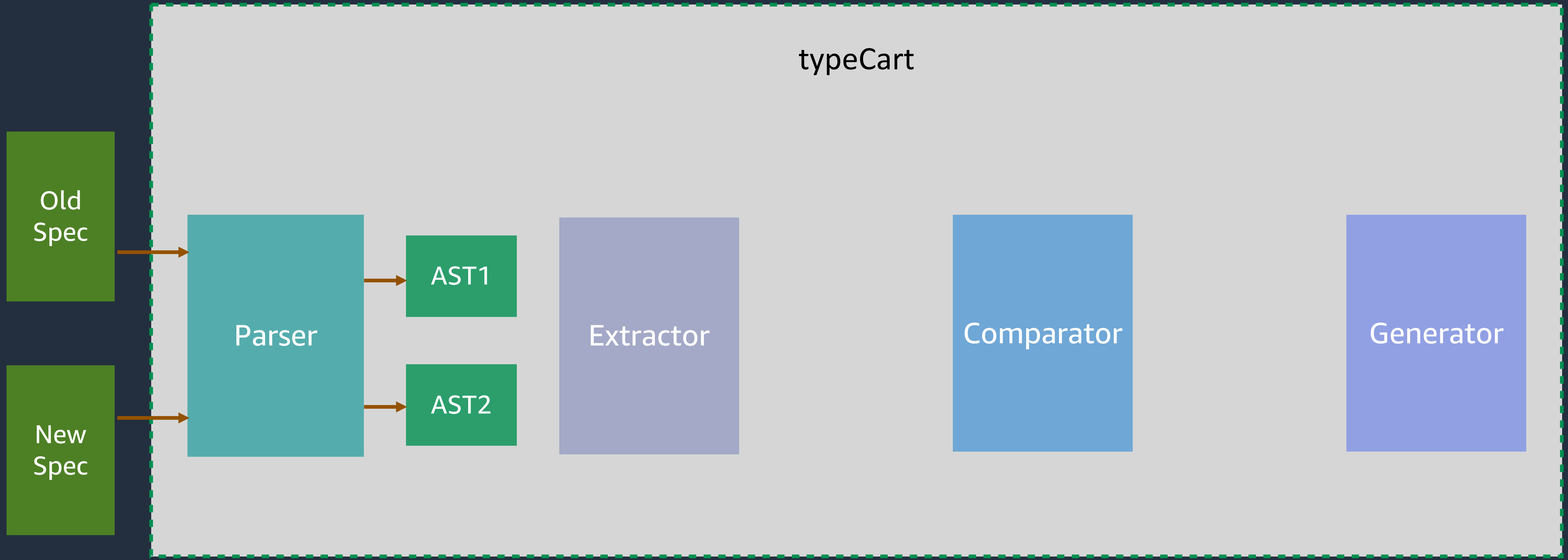


```
function collectionTypeOldToNew<T, T'>
  (fT: T -> T', c: Old.Spec.collectionType<T>):
  New.Spec.collectionType<T'>
  decreases c
{
  match c
  case A(a: T) =>
    New.Spec.collectionType.A(fT(a))
  case B(b: seq<T>) =>
    New.Spec.collectionType.B(seqMap(fT, b))
}
```

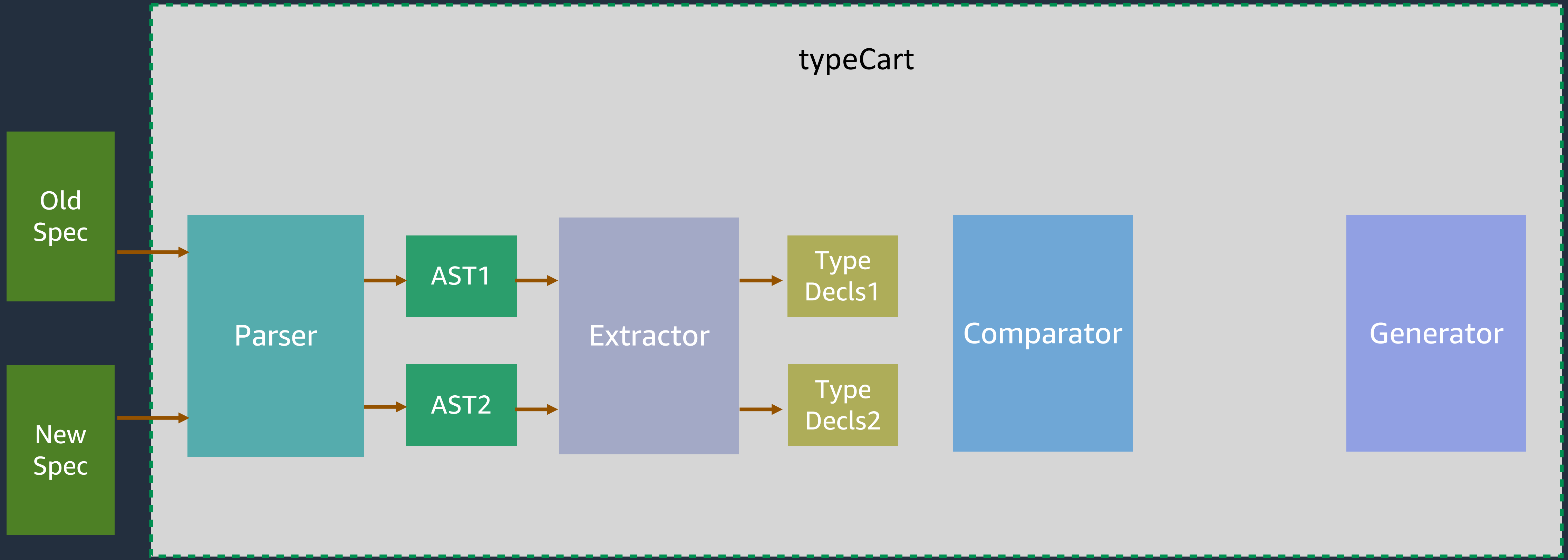
typeCart — Architecture



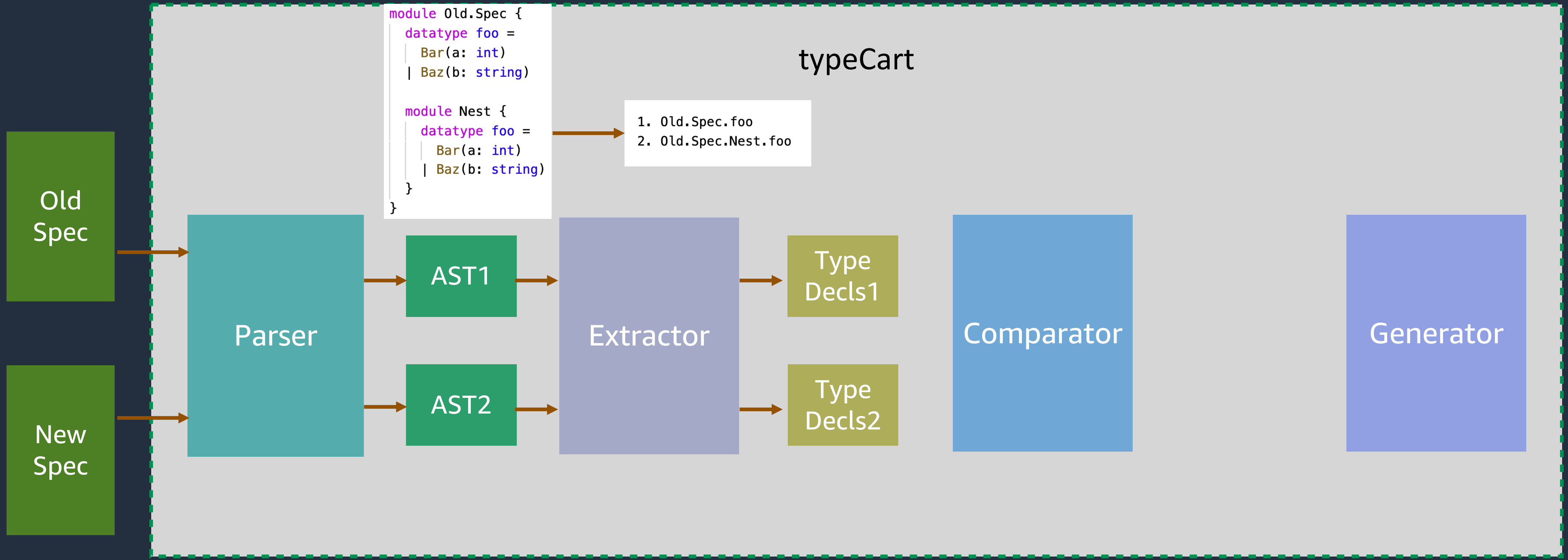
typeCart — Architecture



typeCart — Architecture



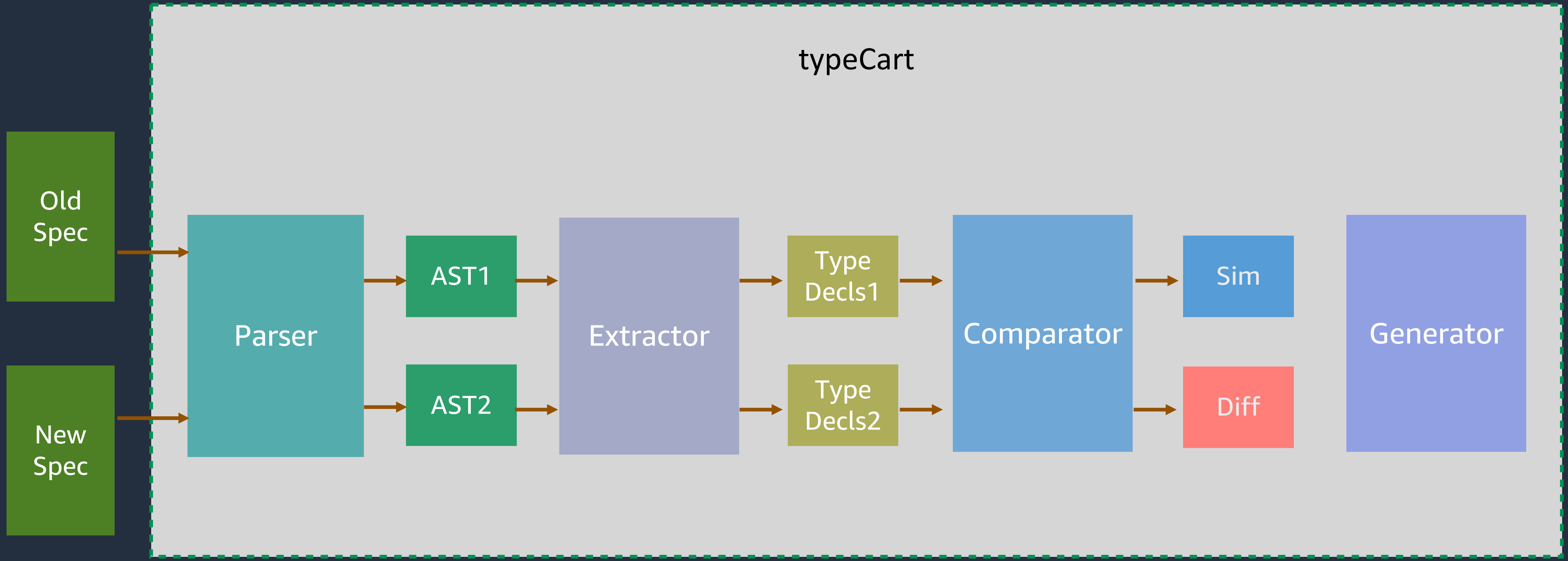
typeCart — Architecture



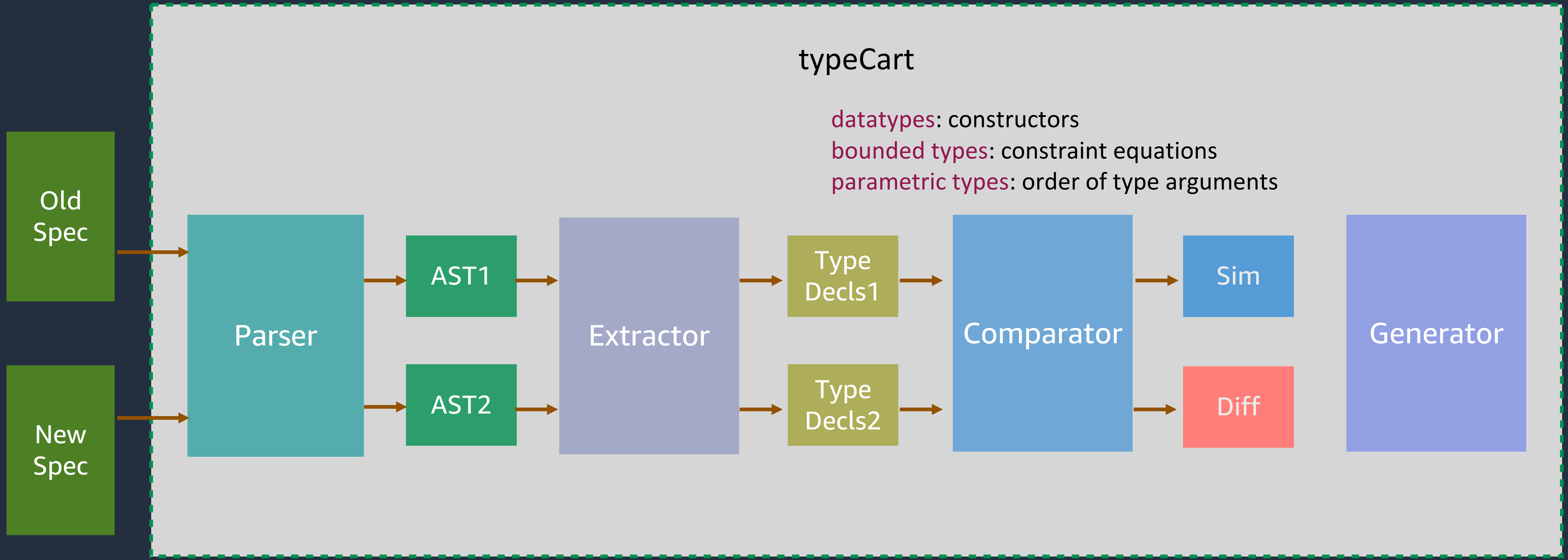
```
module Old.Spec {
  datatype foo =
    Bar(a: int)
  | Baz(b: string)

  module Nest {
    datatype foo =
      Bar(a: int)
    | Baz(b: string)
  }
}
```

```
1. Old.Spec.foo
2. Old.Spec.Nest.foo
```

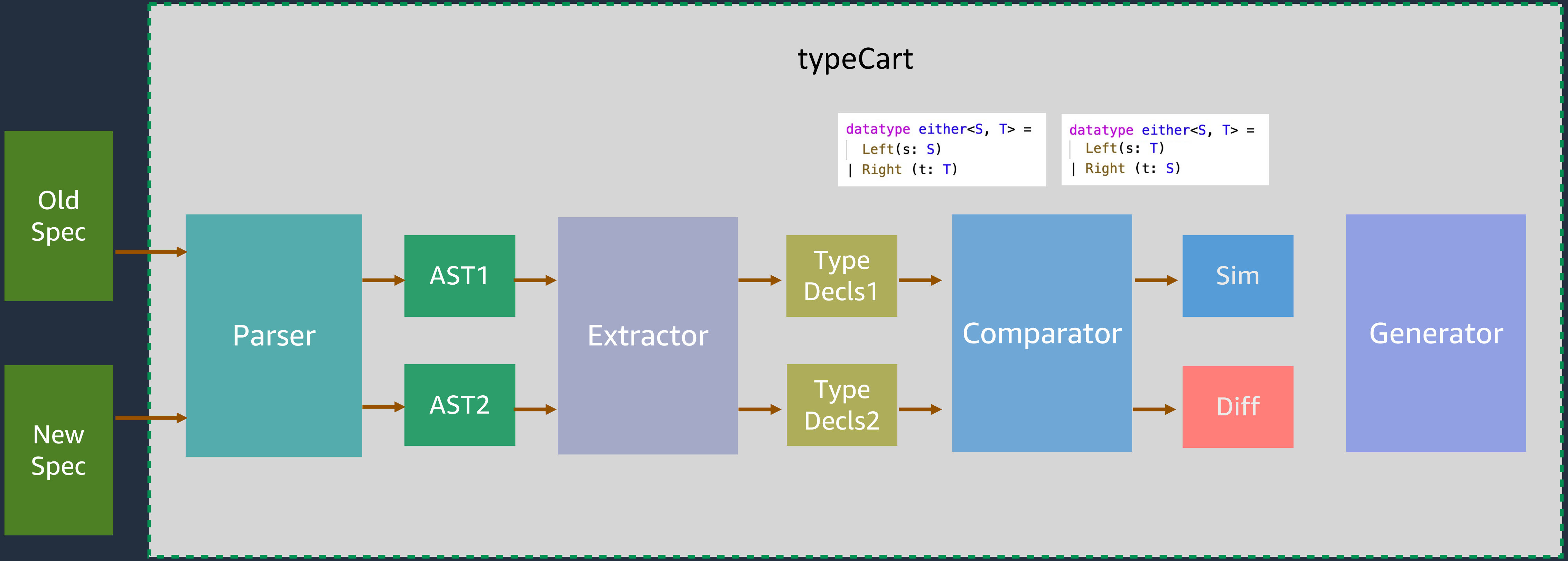
typeCart — Architecture



typeCart — Architecture



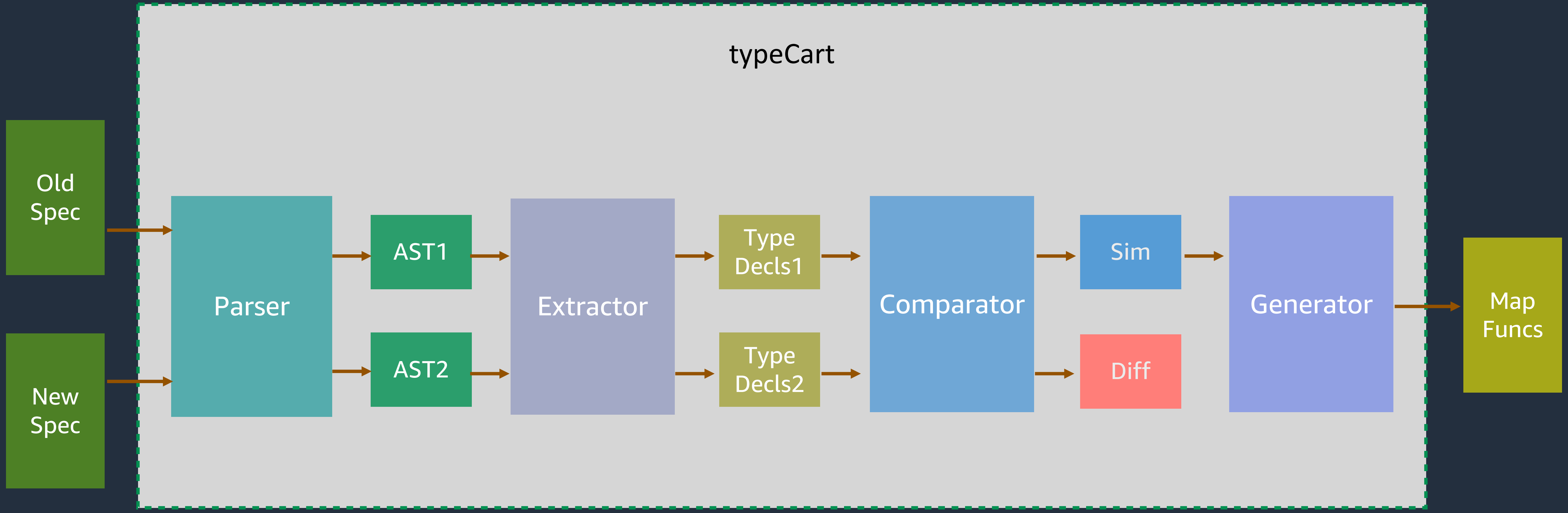
typeCart — Architecture



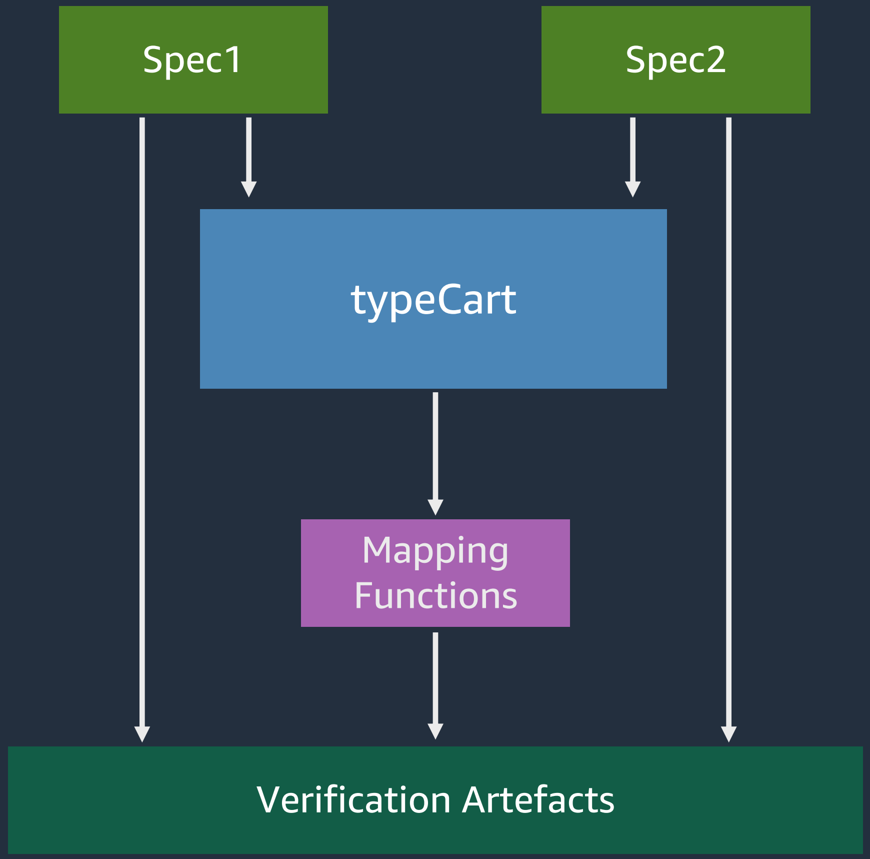
```
datatype either<S, T> =
  Left(s: S)
| Right (t: T)
```

```
datatype either<S, T> =
  Left(s: T)
| Right (t: S)
```

typeCart — Architecture



Integrating typeCart to Verification



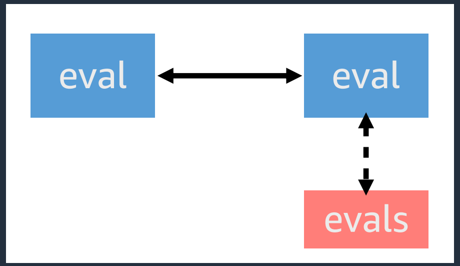
Recap

```
datatype exp = Const (nat32) | Add (nat32,nat32)

function eval (e:exp):(v:nat32)
```

version1

```
datatype exps = One (exp) | Struct (seq<exp>)

datatype value = Val (nat32) | Vals (seq<nat32>)

function evals (es:exps):(v:value)
```

version2



backward compatibility

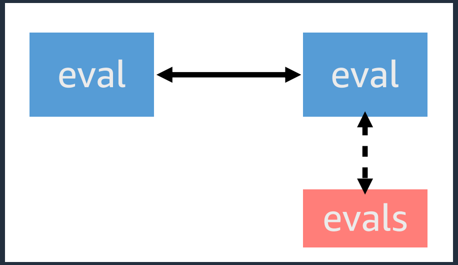
Recap

```
datatype exp = Const (nat32) | Add (nat32,nat32)

function eval (e:exp):(v:nat32)
```

version1

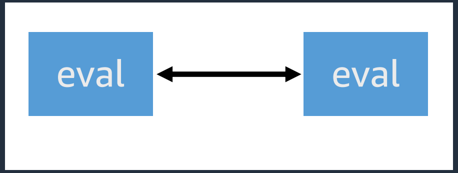
```
datatype exps = One (exp) | Struct (seq<exp>)

datatype value = Val (nat32) | Vals (seq<nat32>)

function evals (es:exps):(v:value)
```

version2



backward compatibility

- Doing nothing ~~is~~ was difficult: typeCart



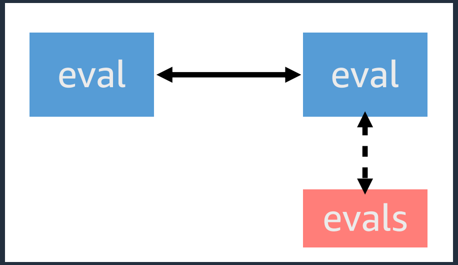
typeCart — Work in Progress

```
datatype exp = Const (nat32) | Add (nat32,nat32)

function eval (e:exp):(v:nat32)
```

version1

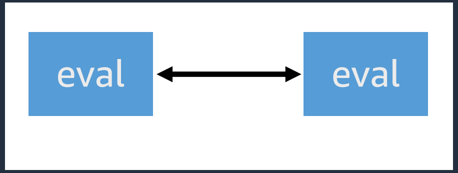
```
datatype exps = One (exp) | Struct (seq<exp>)

datatype value = Val (nat32) | Vals (seq<nat32>)

function evals (es:exps):(v:value)
```

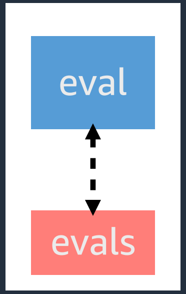
version2



backward compatibility

- Doing nothing ~~is~~ was difficult: typeCart
- Let's do something!
    - Reasoning about deltas



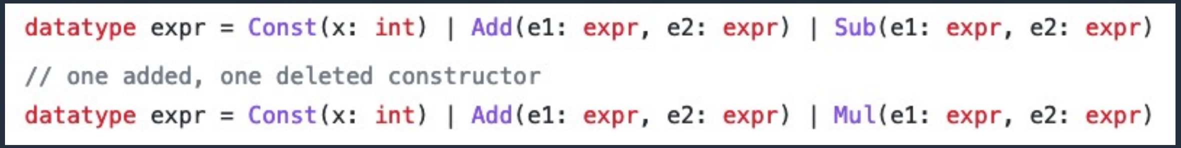


typeCart — Work in Progress

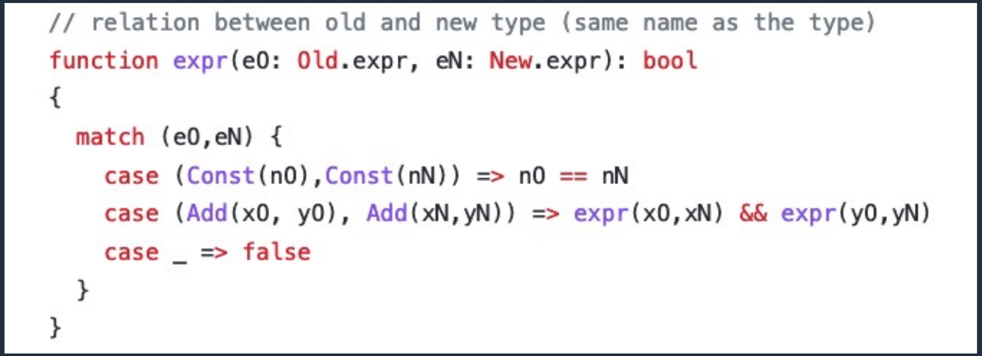
- Specifications deltas involve evolving types
- Evolving types require mapping relations instead of mapping functions

typeCart — Work in Progress

- Specifications deltas involve evolving types
- Evolving types require mapping relations instead of mapping functions



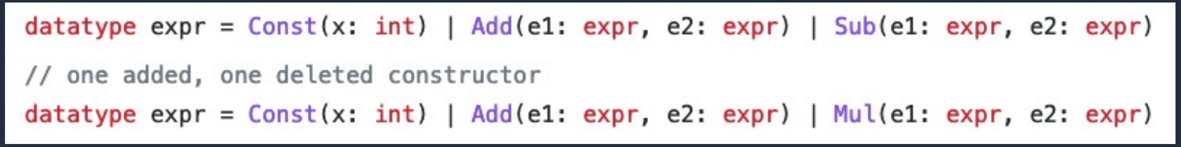
```
datatype expr = Const(x: int) | Add(e1: expr, e2: expr) | Sub(e1: expr, e2: expr)
// one added, one deleted constructor
datatype expr = Const(x: int) | Add(e1: expr, e2: expr) | Mul(e1: expr, e2: expr)
```



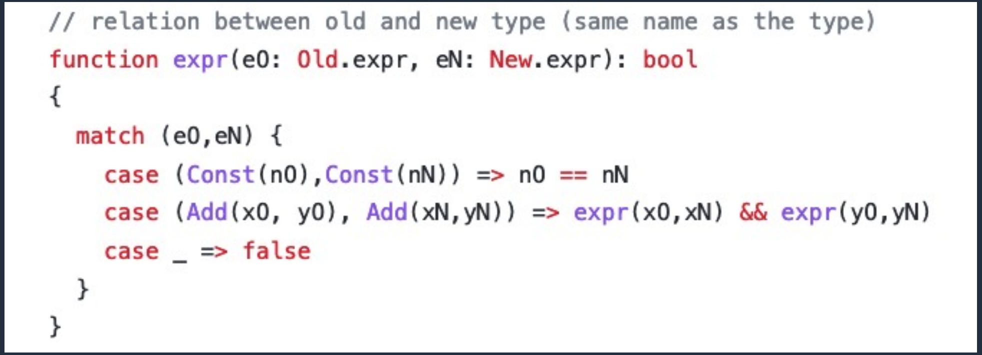
```
// relation between old and new type (same name as the type)
function expr(e0: Old.expr, eN: New.expr): bool
{
  match (e0,eN) {
    case (Const(n0),Const(nN)) => n0 == nN
    case (Add(x0, y0), Add(xN,yN)) => expr(x0,xN) && expr(y0,yN)
    case _ => false
  }
}
```

typeCart — Work in Progress

- Specifications deltas involve evolving types
- Evolving types require mapping relations instead of mapping functions



```
datatype expr = Const(x: int) | Add(e1: expr, e2: expr) | Sub(e1: expr, e2: expr)
// one added, one deleted constructor
datatype expr = Const(x: int) | Add(e1: expr, e2: expr) | Mul(e1: expr, e2: expr)
```



```
// relation between old and new type (same name as the type)
function expr(eO: Old.expr, eN: New.expr): bool
{
  match (eO,eN) {
    case (Const(nO),Const(nN)) => nO == nN
    case (Add(xO, yO), Add(xN,yN)) => expr(xO,xN) && expr(yO,yN)
    case _ => false
  }
}
```

upcoming typeCart V2:
- will identify evolving+equivalent types
- will generate mapping relations

typeCart — Work in Progress

- Type constructors with higher order arguments require nested translation

typeCart — Work in Progress

- Type constructors with higher order arguments require nested translation

```
datatype foo<A> = Foo (a:A)

datatype bar<A> = Bar (b:foo<foo<A>>)
```

typeCart — Work in Progress

- Type constructors with higher order arguments require nested translation

```
datatype foo<A> = Foo (a:A)

datatype bar<A> = Bar (b:foo<foo<A>>)
```

```
function barOldToNew<A, A'>(fA: A -> A', b: Old.bar<A>): New.bar<A'>
{
  match b
  case Bar(Foo (a)) =>
    New.bar.Bar(New.foo.Foo(fooOldToNew(fA,a)))
}
```

typeCart — Work in Progress

- Generic interface
  - TIL: typeCart Intermediate Language

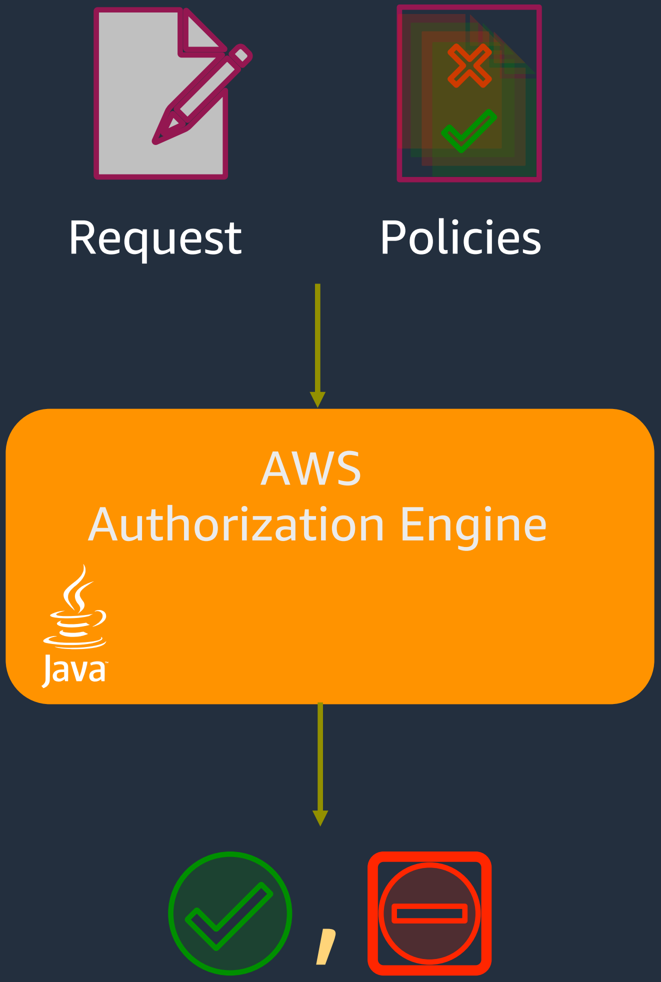
typeCart — Work in Progress

- Generic interface
  - TIL: typeCart Intermediate Language

- Revision control integration

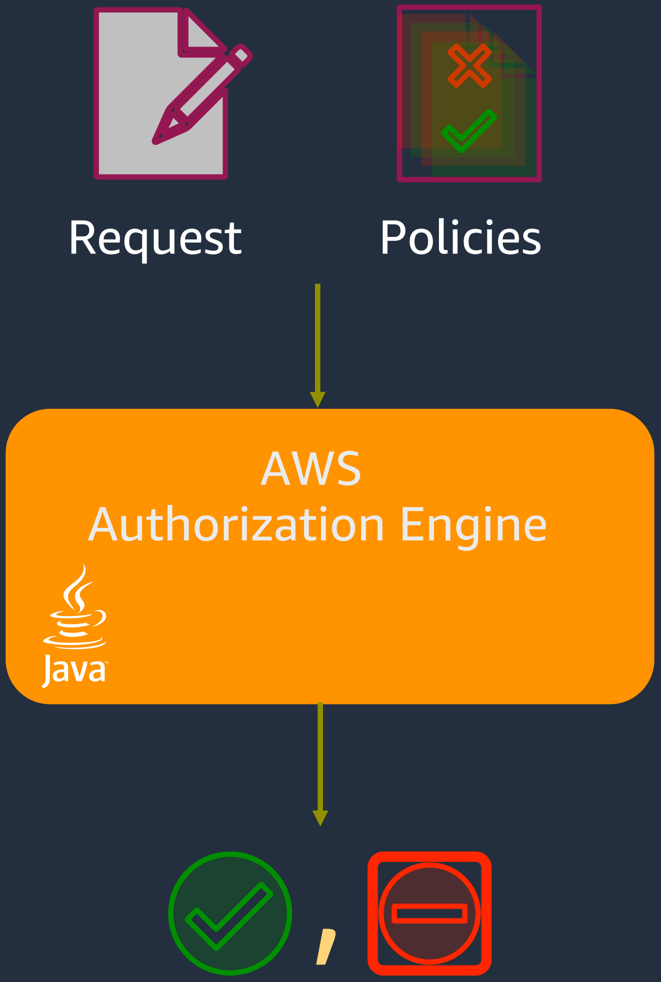
Backward Compatibility of AWS Authorization Engine

Backward Compatibility of AWS Authorization Engine



Is principal P allowed to perform action A with resource R under conditions C?

Backward Compatibility of AWS Authorization Engine



Is principal P allowed to perform action A with resource R under conditions C?

AWS authorizes **~34 trillion** API calls / day

Backward Compatibility of AWS Authorization Engine

- Changes to the Authorization Engine
  - Performance updates
  - Feature updates
  - Algorithmic updates

Backward Compatibility of AWS Authorization Engine

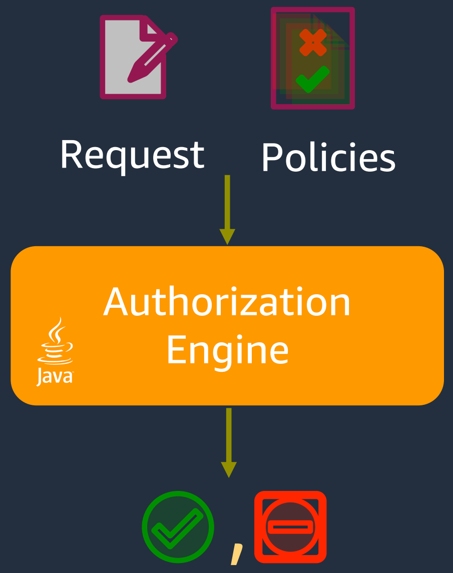
- Changes to the Authorization Engine
  - Performance updates
  - Feature updates
  - Algorithmic updates

Engineers seek answers to:

- How does this change affect the existing implementation?
- Is it backward compatible?
- Does this change affect the existing authorization strategy?

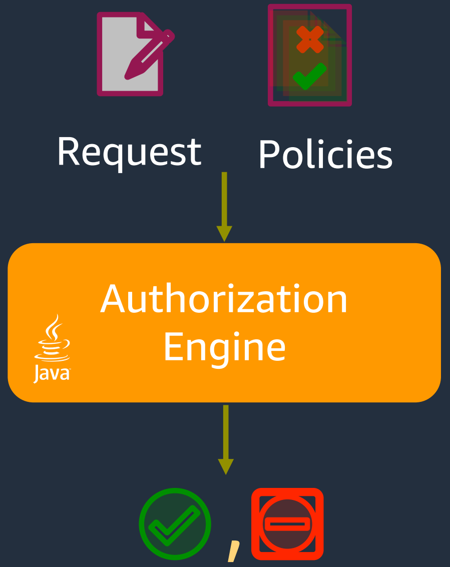
Yucca — Verified AWS Authorization Engine

- The Yucca project — started in the spring 2020
- Goals
  - Have a high level specifications of the Authorization Engine
  - Prove authorization properties
  - Prove backward compatibility going forward



Yucca — Verified AWS Authorization Engine

- The Yucca project — started in the spring 2020
- Goals
  - Have a high level specifications of the Authorization Engine
  - Prove authorization properties
  - Prove backward compatibility going forward
    - Apply typeCart to semi-automate the process



Can You use typeCart?

- Absolutely yes!

available on GitHub:
`awslabs/typecart`

- Want to analyze the effect of introducing changes to your Dafny development?
  - typeCart is here

Can You use typeCart?

- Absolutely yes!

available on GitHub:
`awslabs/typecart`

- Want to analyze the effect of introducing changes to your Dafny development?
  - typeCart is here

- Questions!