Anti-XML

Second Time's the Charm!

https://github.com/djspiewak/anti-xml

Outline

- Basic design principles
- Implementation notes
- TODO

Design Principles

- Functional (in the Rúnarian sense)
- Type safe
- Consistent
- Convenient

https://github.com/djspiewak/anti-xml

Functional

- `Group` is a specialization of `Vector`

Functional

- `Group` is a specialization of `Vector`
- Selection doesn't modify the old `Group`

Functional

- `Group` is a specialization of `Vector`
- Selection doesn’t modify the old `Group`
- `Group` creation is checked, not wrapped!

```scala
val nodes = Array(<a/>, <b/>, <c/>)
val ns = NodeSeq fromSeq nodes
nodes(0) = <a2/>
```

https://github.com/djspiewak/anti-xml

Type Safe

- `Group` is parameterized on element type
  - e.g. `Group[Elem] <: Group[Node]`

https://github.com/djspiewak/anti-xml

Type Safe

- `Group` is parameterized on element type
  - e.g. `Group[Elem] <: Group[Node]`
- Selectors know their selected element type

https://github.com/djspiewak/anti-xml

Type Safe

- `Group` is parameterized on element type
  - e.g. `Group[Elem] <: Group[Node]`
- Selectors know their selected element type
- Explicit converters retain specific type

```
val e = <foo/>
val e2 = e.anti      // e2: Elem

val n = e: xml.Node
val n2 = n.anti      // n2: Node

val ns = e: NodeSeq
val ns2 = ns.anti    // ns2: Group[Node]
```

https://github.com/djspiewak/anti-xml

Consistent

- XPath *cannot* be encoded as combinators
  - Selection behavior inconsistent w.r.t. level

Consistent

- XPath *cannot* be encoded as combinators
  - Selection behavior inconsistent w.r.t. level
- Selection lacks "magic" strings like `"_"`

https://github.com/djspiewak/anti-xml

Consistent

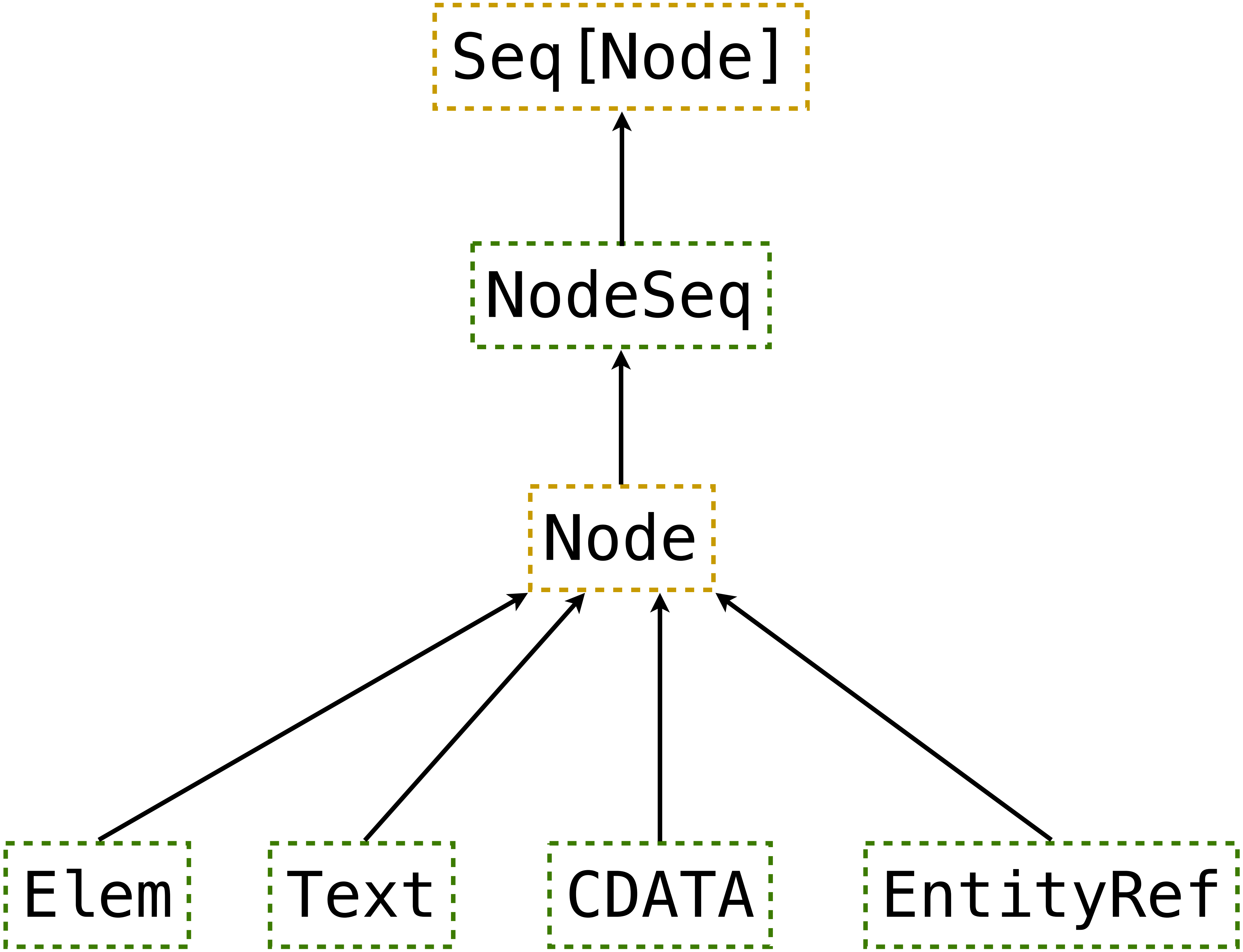
- XPath *cannot* be encoded as combinators
  - Selection behavior inconsistent w.r.t. level
- Selection lacks "magic" strings like `"_"`
- *Very* carefully designed implicit conversions!

https://github.com/djspiewak/anti-xml

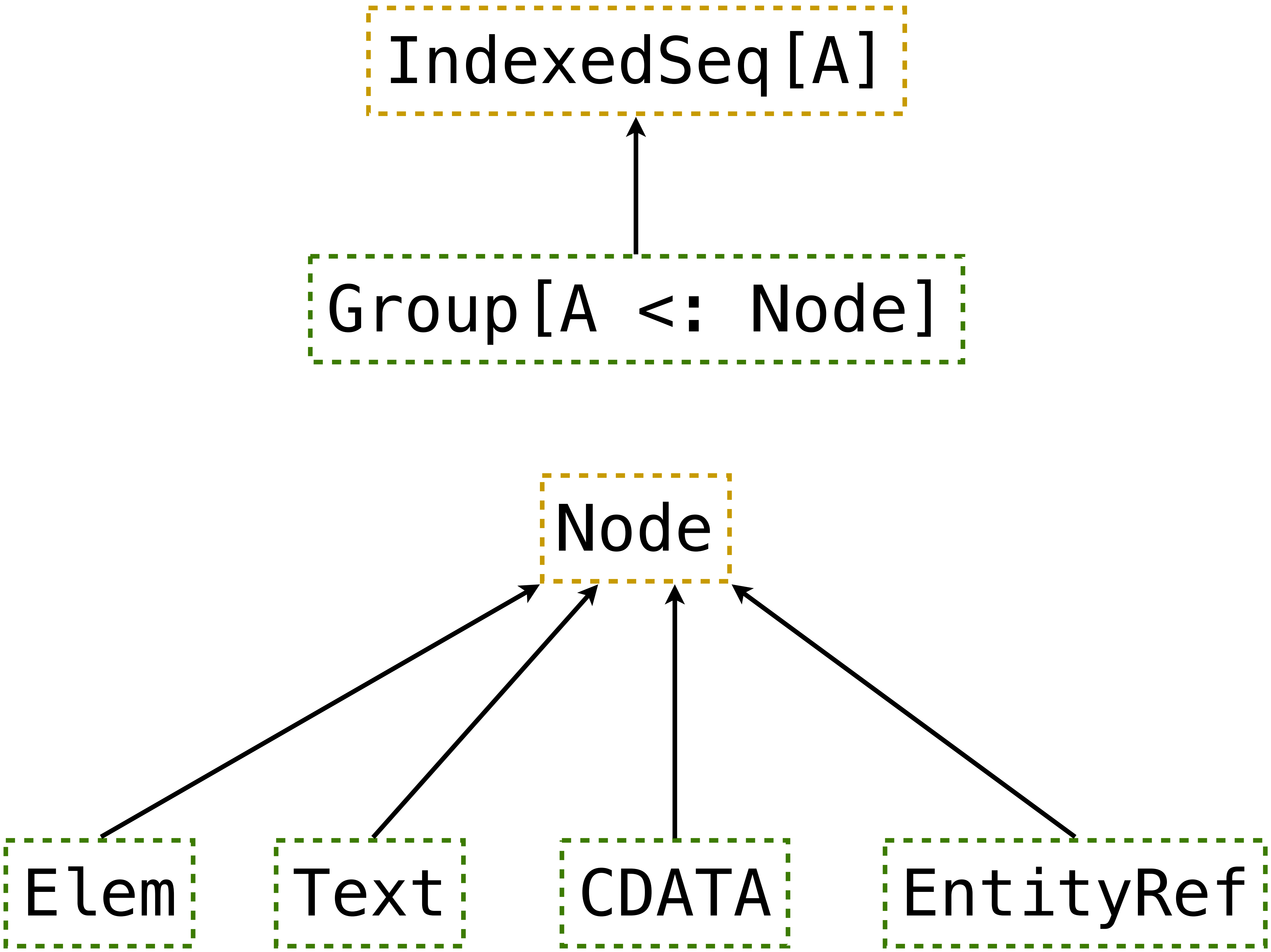
Consistent

- XPath *cannot* be encoded as combinators
  - Selection behavior inconsistent w.r.t. level
- Selection lacks "magic" strings like `"_"`
- *Very* carefully designed implicit conversions!
- Hierarchy avoids issues with subtyping

`https://github.com/djspiewak/anti-xml`



https://github.com/djspiewak/anti-xml



https://github.com/djspiewak/anti-xml

Convenient

- QNames and attributes are easy to use

Convenient

- QNames and attributes are easy to use
- Vector base allows random-access in $O(1)$

https://github.com/djspiewak/anti-xml

Convenient

- QNames and attributes are easy to use
- Vector base allows random-access in $O(1)$
- Explicit converters allow use of literals

Convenient

- QNames and attributes are easy to use
- Vector base allows random-access in $O(1)$
- Explicit converters allow use of literals
- N-Hole zipper for deep updates

https://github.com/djspiewak/anti-xml

```xml
<bookstore>
    <book>
        <title>For Whom the Bell Tolls</title>
        <author>Hemmingway</author>
    </book>
    <book>
        <title>I, Robot</title>
        <author>Isaac Asimov</author>
    </book>
    <book>
        <title>Programming Scala</title>
        <author>Dean Wampler</author>
        <author>Alex Payne</author>
    </book>
</bookstore>
```

https://github.com/djspiewak/anti-xml

```scala
val titledBooks = for {
  book <- bookstore \ "book"
  title <- book \ "title" \ text

  val filtered = book.children filter {
    case Elem(None, "title", _, _, _) => false
    case _ => true
  }
} yield {
  book.copy(
    attrs = book.attrs + ("title" -> title),
    children = filtered)
}

val bookstore2 = titledBooks.unselect
```

https://github.com/djspiewak/anti-xml

```
val titledBooks = for {
  book <- bookstore \ "book"
  title <- book \ "title" \ text

  val filtered = book.children filter {
    case Elem(None, "title", _, _, _) => false
    case _ => true
  }
} yield {
  book.copy(
    attrs = book.attrs + ("title" -> title),
    children = filtered)
}

val bookstore2 = titledBooks.unselect
```

https://github.com/djspiewak/anti-xml

Implementation

- `Group` is an `IndexedSeq`
  - Data managed by `VectorCase`

Implementation

- `Group` is an `IndexedSeq`
  - Data managed by `VectorCase`
- `Zipper` extends `Group`

Implementation

- `Group` is an `IndexedSeq`
  - Data managed by `VectorCase`
- `Zipper` extends `Group`
- Selection is handled by `Selectable`

Implementation

- `Group` is an `IndexedSeq`
  - Data managed by `VectorCase`
- `Zipper` extends `Group`
- Selection is handled by `Selectable`
- Parser is based on StAX (`javax.xml`)

`https://github.com/djspiewak/anti-xml`

Selection Impl

- Two forms of selection
  - Shallow (\) – move down and filter
  - Deep (\\) – in-order enum and filter
- `CanBuildFromWithZipper`
- `Selector`(s) are `PartialFunction`(s)!
- Bloom filters for element name selection

https://github.com/djspiewak/anti-xml

Zipper Impl

```haskell
data Ctx a = Gap (Forest a) | Hole Int

type Level a = ([Ctx a], [a])

type Zipper a = ([Level a], Forest a)
```

https://github.com/djspiewak/anti-xml

Converters Impl

- `Converter` is the pimp
- `XMLConvertable[A, B]` is the typeclass
  - Doesn't extend `Function1`!
- Specific type precedence
  - Enforced by inheritance in companion
  - Singleton self-types for the win!

`ScalaCompat`

- Dirty hack to handle `GenTraversable`
- Auto-generated by SBT at build-time
- Defined in *Project.scala*

TODO: Easy

- Documentation (scaladoc and website)
- Small, self-contained examples
- Update build to SBT 0.10
- More benchmarks
- MOAR TESTS!!

https://github.com/djspiewak/anti-xml

TODO: Moderate

- Level-selection (filter without descent)
- Context-preserving `Zipper` util methods
  - `drop`, `take`, etc
- Optimizations (especially memory)
- Converters *to* `scala.xml`

TODO: Hard

- `Zipper` for deep-selection
    - Patch from Daniel Beskin under review
- Compiler plugin for XML literals
- Lazy `Group`
    - Prototype done by David LaPalomento
- Type-safe(r) `Zipper#unselect` results

Have fun!

http://anti-xml.org

https://github.com/djspiewak/anti-xml