Static Program Reduction via Specification Slicing

Martin Kellogg
New Jersey Institute of Technology

Motivation: analysis debugging

Motivation: analysis debugging

I'm getting many those errors while the project was already compilable with 3.35.

For example, I had to do this to stop this error being reported: apache/cassandra@`7b3c4ce`#diff-234d3942bee540163239ada08e27e2aee864d00a3a5f356f570e18684d1bae03R167

It is weird because neither `Iterator` nor `PaxosKeyState` has any `@MustCall` obligations. I get that in many more places, yet I don't see any pattern.

Another example (which you can reproduce because I haven't fixed that yet: `ant cf-only -Dcf.check.only=org/apache/cassandra/index/sasi/conf/IndexMode.java`):

```
[javac] /home/jlewandowski/dev/cassandra/c18239-static-analysis/src/java/org/apache/ n
[javac]             String literalOption = indexOptions.get(INDEX_IS_LITERAL_OPTION);
[javac]                          ^
[javac]   The type of object is: java.lang.String.
[javac]   Reason for going out of scope: regular method exit
```

where `indexOptions` is a method parameter of type `Map<String, String>`.

Motivation: analysis debugging

I'm getting many those errors while the project was already compilable with 3.35.

For example, I had to do this to stop this error being reported:

kelloggm commented on Jul 5, 2023 Member

These are caused by the soundness fix in #5912, which closed a (frankly pretty serious) bug related to how type variables are handled. We expected it to introduce a few false positives, but from your description it sounds like the impact is more serious than we'd anticipated. I'm sorry about that - I made a judgment call that the number of false positives would probably be worth fixing the soundness problem, but it sounds from your description that, at least in your case, that's not how it seemed.

I'll look into the specific examples that you cited (in `IndexMode.java` and the one you fixed in `PaxosUncommittedTracker.java`) and see if I can build small versions that I can use as test cases. If so, I might be able to improve these to avoid issuing too many of these errors.

Motivation: analysis debugging

I'm getting many those errors while the project was already compilable with 3.35.

For example, I had to do this to stop this error being reported:

kelloggm commented on Jul 5, 2023 Member

These are caused by the soundness fix in #5912, which closed a (frankly pretty serious) bug related to how type variables are handled. We expected it to introduce a few false positives, but from your description it sounds like the impact is more serious than we'd anticipated. I'm sorry about that - I made a judgment call that the number of false positives would probably be worth fixing the soundness problem, but it sounds from your description that, at least in your case, that's not how it seemed.

I'll look into the specific examples that you cited (in `IndexMode.java` and the one you fixed in `PaxosUncommittedTracker.java`) and see if I can build small versions that I can use as test cases. If so, I might be able to improve these to avoid issuing too many of these errors.

Motivation: analysis debugging

- Typical scenario:
  - you are the **developer** of some useful static analysis

Motivation: analysis debugging

- Typical scenario:
  - you are the **developer** of some useful static analysis
  - a **user** encounters a problem with your analysis (e.g., a crash, wrong output, etc.)

Motivation: analysis debugging

- Typical scenario:
  - you are the **developer** of some useful static analysis
  - a **user** encounters a problem with your analysis (e.g., a crash, wrong output, etc.)
  - the user helpfully **reports the bug**
    - but usually without a **small, reproducible test case**

Traditional solution: program reduction

- Underlying algorithm: **delta debugging**
  - uses **divide-and-conquer** to find a minimal “interesting” subset of a given target set
  - relies on the presence of a **runnable oracle** for “interesting”

Traditional solution: program reduction

- Underlying algorithm: **delta debugging**
  - uses **divide-and-conquer** to find a minimal "interesting" subset of a given target set
  - relies on the presence of a **runnable oracle** for "interesting"
- Program reduction today:
  - "interesting" = "does running the analysis on the program cause the bug we're interested in"
  - C-Reduce, Perses (ICSE 2018) work this way

Traditional solution: program reduction

- Delta-debugging based program reduction is a **dynamic analysis**
  - that is, it requires us to **run the analysis** that we are debugging repeatedly

Traditional solution: program reduction

- Delta-debugging based program reduction is a **dynamic analysis**
    - that is, it requires us to **run the analysis** that we are debugging repeatedly
        - works fine for ***fast, easy-to-deploy*** analyses (e.g., a C compiler)

Traditional solution: program reduction

- Delta-debugging based program reduction is a **dynamic analysis**
  - that is, it requires us to **run the analysis** that we are debugging repeatedly
    - works fine for ***fast, easy-to-deploy*** analyses (e.g., a C compiler)
  - what about **heavier-weight analyses** (e.g., program verifiers)?
    - e.g., the analysis in the example at the beginning is a "fast" resource leak verifier, but it still runs in tens of minutes on realistically-sized Java programs

Traditional solution: program reduction

- Delta-debugging based program reduction is a **dynamic analysis**
  - that is, it requires us to **run the analysis** that we are debugging repeatedly
    - works fine for ***fast, easy*** … compiler)
  - what about **heavier-weigh**…
    - e.g., the analysis in the example at the beginning is a "fast" resource leak verifier, but it still runs in tens of minutes on realistically-sized Java programs

Result: in practice, analysis developers don't use program reduction (it's too slow)

Static program reduction

- For any given ***dynamic*** analysis, there is usually a ***static*** analysis that could achieve the same goal (and vice-versa)
  - but usually with different tradeoffs

Static program reduction

- For any given ***dynamic*** analysis, there is usually a ***static*** analysis that could achieve the same goal (and vice-versa)
    - but usually with different tradeoffs
        - a static program reduction technique wouldn't have to scale with the cost of running the underlying analysis

Static program reduction

- For any given ***dynamic*** analysis, there is usually a ***static*** analysis that could achieve the same goal (and vice-versa)
  - but usually with different tradeoffs
    - a static program reduction technique wouldn't have to scale with the cost of running the underlying analysis
- What are the barriers to **static program reduction**?
  - if we can't run the analysis whose output we're trying to preserve, how do what know what parts of the program can be removed?

Key insight: modularity

Key insight: modularity

- most analyses we're interested in are **modular**
  - that is, for performance reason they don't perform arbitrary interprocedural analysis

Key insight: modularity

- most analyses we're interested in are **modular**
  - that is, for performance reason they don't perform arbitrary interprocedural analysis
- if we can **formalize modularity**, we can use that definition to build a static program reducer:
  - intuitively, modularity tells us what other program elements can be considered when analyzing a target program element

Specification slicing

Key theorem: a specification slicer preserves the ***compile-time behavior*** of a target program *P* (with respect to a modular program analysis *V*) at some target program location *L*

Specification slicing

Key theorem: a specification slicer preserves the ***compile-time behavior*** of a target program *P* (with respect to a modular program analysis *V*) at some target program location *L*

cf. "traditional" slicing: a "traditional" slicer preserves the ***run-time behavior*** of a target program *P* (with respect to concrete execution) at some target program location *L*

Specification slicing

Key theorem: a specification slicer preserves the ***compile-time behavior*** of a target program *P* (with respect to a modular program analysis *V*) at some target program location *L*

cf. "traditional" slicing: a "traditional" slicer preserves the ***run-time behavior*** of a target program *P* (with respect to concrete execution) at some target program location *L*

You can think of this as "abstract" slicing, in the sense of "abstract interpretation"

Specification slicing: algorithm

Specification slicing: algorithm

Input: program *P*, location *L* in *P*, and **definition of modularity** *M*

Specification slicing: algorithm

Input: program *P*, location *L* in *P*, and **definition of modularity** *M*

- *M* is a **syntax-directed map** from kinds of program elements to related program elements that the analysis of interest considers

Specification slicing: algorithm

Input: program *P*, location *L* in *P*, and **definition of modularity** *M*

- *M* is a **syntax-directed map** from kinds of program elements to related program elements that the analysis of interest considers
  - e.g., *M*("field read expression") might be "field's declaration, type of field, receiver expression"

Specification slicing: algorithm

Input: program *P*, location *L* in *P*, and **definition of modularity** *M*

- *M* is a **syntax-directed map** from kinds of program elements to related program elements that the analysis of interest considers
  - e.g., *M*("field read expression") might be "field's declaration, type of field, receiver expression"
  - *M* can be reused between similar analyses
    - e.g., javac, Checker Framework, OpenJML all have approximately the same *M* for Java

Specification slicing: algorithm

Input: program *P*, location *L* in *P*, and definition of modularity *M*

Algorithm:

Specification slicing: algorithm

Input: program *P*, location *L* in *P*, and definition of modularity *M*

Algorithm:

```
worklist = all elements e in L's scope
slice = ∅
while (worklist is not empty):
    toPreserve = worklist.pop()
    if (!slice.contains(toPreserve)):
        slice.add(toPreserve)
        worklist.add(M(toPreserve))
return slice
```

Specification slicing: examples

- remove bodies of used methods
- primitive field reads
- a more complex example with unsolved symbols

Specification slicing: unsolvable symbols

- So far, we have assumed that all symbols are **solvable**
  - that is, we're assuming that we have the whole program

Specification slicing: unsolvable symbols

- So far, we have assumed that all symbols are **solvable**
  - that is, we're assuming that we have the whole program
- In practice, we often **can't easily access** the whole program
  - e.g., it would require human effort to go collect the classpath of the target program in the example
    - need to run the build tool, etc.

Specification slicing: unsolvable symbols

- So far, we have assumed that all symbols are **solvable**
  - that is, we're assuming that we have the whole program
- In practice, we often **can't easily access** the whole program
  - e.g., it would require human effort to go collect the classpath of the target program in the example
    - need to run the build tool, etc.
  - this is an impediment to using a specification slicer in a fully-automated system

Specification slicing: unsolvable symbols

- An advantage of static program reduction is that we can minimize **incomplete programs**

Specification slicing: unsolvable symbols

- An advantage of static program reduction is that we can minimize **incomplete programs**
  - however, minimizing an incomplete program may introduce **ambiguity** into the modularity model

Specification slicing: unsolvable symbols

- An advantage of static program reduction is that we can minimize **incomplete programs**
  - however, minimizing an incomplete program may introduce **ambiguity** into the modularity model
  - for example, suppose the program reads a field that's not defined. More than one option for where to put that field:
    - the immediate superclass
    - the superclass' superclass
    - etc.

Specification slicing: exact vs approximate

- Our practical specification slicer has two modes:

Specification slicing: exact vs approximate

- Our practical specification slicer has two modes:
  - ***exact mode***: access to the whole program is assumed, and the slicer relies on that to find the definitions of program elements
    - this mode works just like the algorithm a few slides ago

Specification slicing: exact vs approximate

- Our practical specification slicer has two modes:
  - ***exact mode***: access to the whole program is assumed, and the slicer relies on that to find the definitions of program elements
    - this mode works just like the algorithm a few slides ago
  - ***approximate mode***: when the slicer finds a symbol it can't solve, it **generates** a program element that makes sense in context
    - uses heuristics to deal with ambiguity
    - but not guaranteed to preserve analysis behavior (even if modularity model is correct) if there is ambiguity

Specification slicing: project status

- We have a **prototype** for modular analyses of Java:
  - https://github.com/kelloggm/specimin
- Currently dealing with:

Specification slicing: project status

- We have a **prototype** for modular analyses of Java:
  - https://github.com/kelloggm/specimin
- Currently dealing with:
  - a long tail of engineering effort to get the **approximate mode heuristics** right

Specification slicing: project status

- We have a **prototype** for modular analyses of Java:
  - https://github.com/kelloggm/specimin
- Currently dealing with:
  - a long tail of engineering effort to get the **approximate mode heuristics** right
  - getting the **modularity model formalization** exactly right
- But we are getting closer to both

Static program reduction: usefulness

- Zooming out, why study program reduction?

Static program reduction: usefulness

- Zooming out, why study program reduction?
  - it's a useful **debugging tool** for analyses on its own, but…

Static program reduction: usefulness

- Zooming out, why study program reduction?
  - it's a useful **debugging tool** for analyses on its own, but…
- A fast program reduction technique that is guaranteed to preserve analysis behavior **unlocks interesting use cases**!

Static program reduction: usefulness

- Zooming out, why study program reduction?
  - it's a useful **debugging tool** for analyses on its own, but…
- A fast program reduction technique that is guaranteed to preserve analysis behavior **unlocks interesting use cases**!
  - staying **under the token limit** when combining an LLM + a modular analysis
  - analyzing **only a changeset** instead of the whole program
  - running an analysis in a **tight loop**

Static program reduction: usefulness

- Zooming out, why study program reduction?
  - it’s a useful **debugging tool** for analyses on its own, but…
- A fast program reduction technique that is guaranteed to preserve analysis behavior **unlocks interesting use cases**!
  - staying **under the token limit** when combining an LLM + a r

I’ll talk about these if you ask

  - analyzing **only a changeset** instead of the whole program
  - running an analysis in a **tight loop**

Static program reduction: usefulness

- Zooming out, why study program reduction?
  - it's a useful **debugging tool** for analyses on its own, but…
- A fast program reduction technique that is guaranteed to prese

definitely going to talk about this

use cases!
  - staying **under the token limit** when combining an LLM + a modular analysis
  - analyzing **only a changeset** instead of the whole program
  - running an analysis in a **tight loop**

LLMs + program analysis

- There's a lot of excitement around the idea of **combining LLMs + sound program analyzers** (CEGAR-style)

LLMs + program analysis

- There's a lot of excitement around the idea of **combining LLMs + sound program analyzers** (CEGAR-style)
- One hurdle so far is the **token limit** of LLMs
  - realistically-sized programs don't fit!

LLMs + program analysis

- There's a lot of excitement around the idea of **combining LLMs + sound program analyzers** (CEGAR-style)
- One hurdle so far is the **token limit** of LLMs
  - realistically-sized programs don't fit!
- **Program reduction** is an obvious solution to this problem

LLMs + program analysis

- There's a lot of excitement around the idea of **combining LLMs + sound program analyzers** (CEGAR-style)
- One hurdle so far is the **token limit** of LLMs
  - realistically-sized programs don't fit!
- **Program reduction** is an obvious solution to this problem
  - but traditional program reduction techniques are slow

LLMs + program analysis

- There's a lot of excitement around the idea of **combining LLMs + sound program analyzers** (CEGAR-style)
- One hurdle so far is the **token limit** of LLMs
  - realistically-sized programs don't fit!
- **Program reduction** is an obvious solution to this problem
  - but traditional program reduction techniques are slow
  - for **fully-automated** systems, static program reduction:
    - allows the verifier to interact **honestly** with the LLM
    - is **faster** than dynamic techniques and can operate on incomplete programs

Example fully-automated system

- Consider a system that:
  - chooses a method in an open-source project

Example fully-automated system

- Consider a system that:
  - chooses a method in an open-source project

Note that an approximate slicer means we **don’t even need to be able to build** the target project - it can be anything!

Example fully-automated system

- Consider a system that:
  - chooses a method in an open-source project
  - uses **static program reduction** to shrink that method

Example fully-automated system

- Consider a system that:
  - chooses a method in an open-source project
  - uses **static program reduction** to shrink that method
  - runs a verification tool on the method, making **pessimistic assumptions** about the input

Example fully-automated system

- Consider a system that:
  - chooses a method in an open-source project
  - uses **static program reduction** to shrink that method
  - runs a verification tool on the method, making **pessimistic assumptions** about the input
  - if there is a warning, **calls an LLM** and asks it to write a patch

Example fully-automated system

- Consider a system that:
  - chooses a method in an open-source project
  - uses **static program reduction** to shrink that method
  - runs a verification tool on the method, making **pessimistic assumptions** about the input
  - if there is a warning, **calls an LLM** and asks it to write a patch
  - **re-runs the verifier** on the patch

Example fully-automated system

- Consider a system that:
  - chooses a method in an open-source project
  - uses **static program reduction** to shrink that method
  - runs a verification tool on the method, making **pessimistic assumptions** about the input
  - if there is a warning, **calls an LLM** and asks it to write a patch
  - **re-runs the verifier** on the patch
    - if it passes, we have **fully-automatically** produced a bug-fixing pull request

Example fully-automated system

- Consider this bug in Netflix's spinnaker project: https://github.com/spinnaker/spinnaker/issues/6856
- Static program reduction would produce:

Exa

- C

  h
- S

```
import java.util.concurrent.*;
import java.util.concurrent.locks.Lock;
import org.checkerframework.framework.qual.DefaultQualifier; // for CF Nullness
import org.checkerframework.framework.qual.TypeUseLocation;
import org.checkerframework.checker.nullness.qual.Nullable;

@DefaultQualifier(value = Nullable.class, locations = TypeUseLocation.PARAMETER)
class CallableCache<Key, Result> {
  private final ConcurrentHashMap<Key, Future<Result>> cache = null;
  private final Lock lock = null;
  private static final org.slf4j.Logger log
     = org.slf4j.LoggerFactory.getLogger(CallableCache.class);

  void clear(Key key) {
    try {
      lock.lock();
      var future = cache.get(key);
      if (future != null && (future.isDone() || future.isCancelled())) {
        cache.remove(key);
        log.debug("Removing element from cache identified by key: " + key);
      }
    } finally {
      lock.unlock();
    }
  }
}
```

Example fully-automated system

- Consider this bug in Netflix's spinnaker project: https://github.com/spinnaker/spinnaker/issues/6856
- Static program reduction would produce…
- Running a nullability analysis produces:

```
CallableCache.java:16: error: [argument] incompatible argument for parameter
key of ConcurrentHashMap.get.
      var future = cache.get(key);
                             ^
  found   : Key extends @Initialized @Nullable Object
  required: @Initialized @NonNull Object
```

Example fully-automated system

- Consider this bug in Netflix's spinnaker project: https://github.com/spinnaker/spinnaker/issues/6856
- Static program reduction would produce…
- Running a nullability analysis produces…
- Prompting an LLM (GPT 3.5) with the code and the warning I've just shown you produces a fix that passes the typechecker:

Example fully-automated system

- Consider this bug in Netflix's spinnaker project:
  http…innaker/spinnaker/issues/6856
- S…
- F…
- F… e
  j…

LLM's fix

```java
void clear(Key key) {
  try {
    lock.lock();
    Key nonNullKey = key;
    if (nonNullKey != null) {
      var future = cache.get(nonNullKey);
      if (future != null && (future.isDone() || future.isCancelled())) {
        cache.remove(nonNullKey);
        log.debug("Removing element from cache identified by key: " + nonNullKey);
      }
    }
  } finally {
    lock.unlock();
  }
}
```

Example fully-automated system

- Consider this bug in Netflix's spinnaker project: https://github.com/spinnaker/spinnaker/issues/6856
- Static program reduction would produce…
- Running a nullability analysis produces…
- Prompting an LLM (GPT 3.5) with the code and the warning I've just shown you produces a fix that passes the typechecker…
    - that is **very similar** to the real human fix that was merged

Example fully-automated system

- Consider this bug in Netflix's spinnaker project:
  https://github.com/spinnaker/spinnak

human fix

```
void clear(Key key) {
  if (key == null) {
    return;
  }
  try {
    lock.lock();
    var future = cache.get(key);
    if (future != null && (future.isDone() || future.isCancelled())) {
      cache.remove(key);
      log.debug("Removing element from cache identified by key: " + key);
    }
  } finally {
    lock.unlock();
  }
}
```

Example fully-automated system

- Consider this bug in Netflix's spinnaker project: https://github.com/spinnaker/spinnaker/issues/6856
- Static program reduction would produce…
- Running a nullability analysis produces…
- Prompting an LLM (GPT 3.5) with the code and the warning I've just shown you produces a fix that passes the typechecker…
  - that is **very similar** to the real human fix that was merged
    - LLM's fix locks and then unlocks the lock unnecessarily :(
    - but could have fixed the bug **fully automatically**, without any human intervention (except to review the PR)

Summary

- **static program reduction** could be a **useful debugging aid** for modular program analyses

Summary

- **static program reduction** could be a **useful debugging aid** for modular program analyses
    - **specification slicing** exploits **analysis modularity** to accomplish static program reduction

Summary

- **static program reduction** could be a **useful debugging aid** for modular program analyses
  - **specification slicing** exploits **analysis modularity** to accomplish static program reduction
- **fast, sound** static program reduction **unlocks new use cases**:
  - LLMs + analysis for fully-automated code improvement
  - code-review-time verification, verifier-guided refactoring, etc.

Summary

- **static program reduction** could be a **useful debugging aid** for modular program analyses
  - **specification slicing** exploits **analysis modularity** to accomplish static program reduction
- **fast, sound** static program reduction **unlocks new use cases**:
  - LLMs + analysis for fully-automated code improvement
  - code-review-time verification, verifier-guided refactoring, etc.
- **prototype** specification slicer for Java:
  - https://github.com/kelloggm/specimin

Summary

Thanks to all of my collaborators who have made this work possible: Loi Nguyen, Tahiatul Islam, Jonathan Phillips, Oscar Chaparro, Michael Ernst, et al.

- **static program reduction** could be a **useful debugging aid** for modular program analyses
  - **specification slicing** exploits **analysis modularity** to accomplish static program reduction
- **fast, sound** static program reduction **unlocks new use cases**:
  - LLMs + analysis for fully-automated code improvement
  - code-review-time verification, verifier-guided refactoring, etc.
- **prototype** specification slicer for Java:
  - https://github.com/kelloggm/specimin

Backup slides

Backup slides

- Code review time verification
- VGR
- More involved Specimin example in slideware (not finished)

Code-review-time verification

Code-review-time verification

- Modular analyses typically require users to **write specifications**
    - e.g., type annotations for a pluggable typechecker

Code-review-time verification

- Modular analyses typically require users to **write specifications**
    - e.g., type annotations for a pluggable typechecker
- This is an **obstacle to adoption** of such analyses

Code-review-time verification

- Modular analyses typically require users to **write specifications**
  - e.g., type annotations for a pluggable typechecker
- This is an **obstacle to adoption** of such analyses
  - doesn't match how developers work with legacy code, which happens at **changeset granularity** (i.e., code review)

Code-review-time verification

- Modular analyses typically require users to **write specifications**
  - e.g., type annotations for a pluggable typechecker
- This is an **obstacle to adoption** of such analyses
  - doesn't match how developers work with legacy code, which happens at **changeset granularity** (i.e., code review)
- **Idea**: what if we could analyze just a changeset in isolation?
  - and ask developers to write specs just for what has changed

Code-review-time verification

- Modular analyses typically require users to **write specifications**
  - e.g., type annotations for a pluggable typechecker
- This is an **obstacle to adoption** of such analyses
  - doesn't match how developers work with legacy code, which happens at **changeset granularity** (i.e., code review)
- **Idea**: what if we could **analyze just a changeset** in isolation?
  - and ask developers to write specs just for what has changed
  - we can **use our static program reducer** to do this

Code-review-time verification

- Modular analyses typically require users to **write specifications**
  - e.g., type annotations for a pluggable typechecker
- This is an **obstacle to adoption** of such analyses
  - doesn't match how developers work with legacy code, which happens at **changeset granularity** (i.e., code review)
- **Idea**: what if we could analyze just a changeset in isolation?
  - and ask developers to write specs just for what has changed
  - we can use our static program reducer to do this
  - **key scientific question**: will specs written this way for many changesets **contradict** each other or tend to **converge**?

Verifier-guided refactoring

- Motivation: sound program analyses always have **false positives**

Verifier-guided refactoring

- Motivation: sound program analyses always have **false positives**
- Many **semantics-preserving** program transformations exist

Verifier-guided refactoring

- Motivation: sound program analyses always have **false positives**
- Many **semantics-preserving** program transformations exist
- **Idea**: if an analysis is sound and can **verify any variant** of a piece of code created by applying only semantics-preserving transformations, then the code is **definitely safe**

Verifier-guided refactoring

- Motivation: sound program analyses always have **false positives**
- Many **semantics-preserving** program transformations exist
- **Idea**: if an analysis is sound and can **verify any variant** of a piece of code created by applying only semantics-preserving transformations, then the code is **definitely safe**
    - requires us to run the verifier in a **tight loop** on many variants

Verifier-guided refactoring

- Motivation: sound program analyses always have **false positives**
- Many **semantics-preserving** program transformations exist
- **Idea**: if an analysis is sound and can **verify any variant** of a piece of code created by applying only semantics-preserving transformations, then the code is **definitely safe**
  - requires us to run the verifier in a **tight loop** on many variants
    - **too expensive** if we're considering the whole program

Verifier-guided refactoring

- Motivation: sound program analyses always have **false positives**
- Many **semantics-preserving** program transformations exist
- **Idea**: if an analysis is sound and can **verify any variant** of a piece of code created by applying only semantics-preserving transformations, then the code is **definitely safe**
  - requires us to run the verifier in a **tight loop** on many variants
    - **too expensive** if we're considering the whole program
    - our fast, sound static program reducer allows the analysis to run much faster in such a loop -> **makes this practical**?

Specification slicing: example

```
boolean isLiteral = false;
try {
  String literalOption = indexOptions.get(INDEX_IS_LITERAL_OPTION);
  AbstractType<?> validator = column.cellValueType();
  isLiteral = literalOption == null ?
   (validator instanceof UTF8Type || validator instanceof AsciiType)
     : Boolean.parseBoolean(literalOption);
} catch (Exception e){
  logger.error("failed to parse {} option, defaulting to 'false'.",
     INDEX_IS_LITERAL_OPTION);
}
```

Specification slicing: examp

This is the code from the example at the beginning of the talk

```
boolean isLiteral = false;
try {
  String literalOption = indexOptions.get(INDEX_IS_LITERAL_OPTION);
  AbstractType<?> validator = column.cellValueType();
  isLiteral = literalOption == null ?
   (validator instanceof UTF8Type || validator instanceof AsciiType)
     : Boolean.parseBoolean(literalOption);
} catch (Exception e){
  logger.error("failed to parse {} option, defaulting to 'false'.",
     INDEX_IS_LITERAL_OPTION);
}
```

Specification slicing: example

preserved (+ its type...)

```
boolean isLiteral = false;
try {
  String literalOption = indexOptions.get(INDEX_IS_LITERAL_OPTION);
  AbstractType<?> validator = column.cellValueType();
  isLiteral = literalOption == null ?
   (validator instanceof UTF8Type || validator instanceof AsciiType)
     : Boolean.parseBoolean(literalOption);
} catch (Exception e){
  logger.error("failed to parse {} option, defaulting to 'false'.",
     INDEX_IS_LITERAL_OPTION);
}
```

Specification slicing: example

```
boolean isLiteral = false;
try {
  String literalOption = indexOptions.get(INDEX_IS_LITERAL_OPTION);
  AbstractType<?> validator = column.cellValueType();
  isLiteral = literalOption == null ?
   (validator instanceof UTF8Type || validator instanceof AsciiType)
     : Boolean.parseBoolean(literalOption);
} catch (Exception e){
  logger.error("failed to parse {} option, defaulting to 'false'.",
     INDEX_IS_LITERAL_OPTION);
}
```

preserved (+ wherever it is defined...)

Specification slicing: example

preserved (+ its supertypes...)

```
boolean isLiteral = false;
try {
  String literalOption = indexOptions.get(INDEX_IS_LITERAL_OPTION);
  AbstractType<?> validator = column.cellValueType();
  isLiteral = literalOption == null ?
    (validator instanceof UTF8Type || validator instanceof AsciiType)
      : Boolean.parseBoolean(literalOption);
} catch (Exception e){
  logger.error("failed to parse {} option, defaulting to 'false'.",
      INDEX_IS_LITERAL_OPTION);
}
```

Specification slicing: example

```
boolean isLiteral = false;
try {
  String literalOptio ... get(INDEX_IS_LITERAL_OPTION);
  AbstractType<?> vali ... ellValueType();
  isLiteral = literalOp ...
   (validator instanceof UTF8Type || validator instanceof AsciiType)
     : Boolean.parseBoolean(literalOption);
} catch (Exception e){
  logger.error("failed to parse {} option, defaulting to 'false'.",
     INDEX_IS_LITERAL_OPTION);
}
```

preserved (+ its supertypes...)

And so on...