Model Checking Abstract Syntax Trees

Pascal Hof

Chair for Software-Technology - TU Dortmund

20.4.2012

Two harmless files in a repository

Html template page

```
<html>
 <head>
  <title>#title#</title>
 </head>
</html>
```

Function using the html template

```
replace :: (String,String) → String → String

buildHome :: IO ()
buildHome = do
  c ← readFile "template.html"
  let c’ = replace ("#titel#","Home") c
  writeFile "home.html" c’
```

The strings `"title"` and `"titel"` have to be equal in order to ensure correct behavior.

Yet again: two harmless files in a repository

Java class offering support for different GUI-languages

```
class I18N{
  public static final String OK;
  public static final String CANCEL;
  public static final String REFRESH;
  public static final String TITLE;

  static{
    // initialize attributes
    // attribute name and left side in properties file have to match
  }
}
```

Property file

```
#I18N_en.properties
OK=Ok
CANCEL=Cancel
REFRESH=Refresh
TITLE=Title
```

The attribute names and left sides have to match in order to ensure correct behavior.

Motivation

Definition

We call such relationships between two files a dependency.

Goal of this talk

Tool support for defining and checking dependencies.

IDEs

IDEs already check dependencies, but an IDE plugin for defining your own dependencies would be nice to have. It can check dependencies whenever you save a file.

Observation

We need to take abstract syntax trees (ast) into account, since a textual search can not distinguish keywords from identifiers.

How to define dependenies?

Two different approaches:

1. Identify a “interesting” node by its position in the ast.
2. Define a logical formula, which holds at “interesting” nodes.

We use the logical formula approach, because:

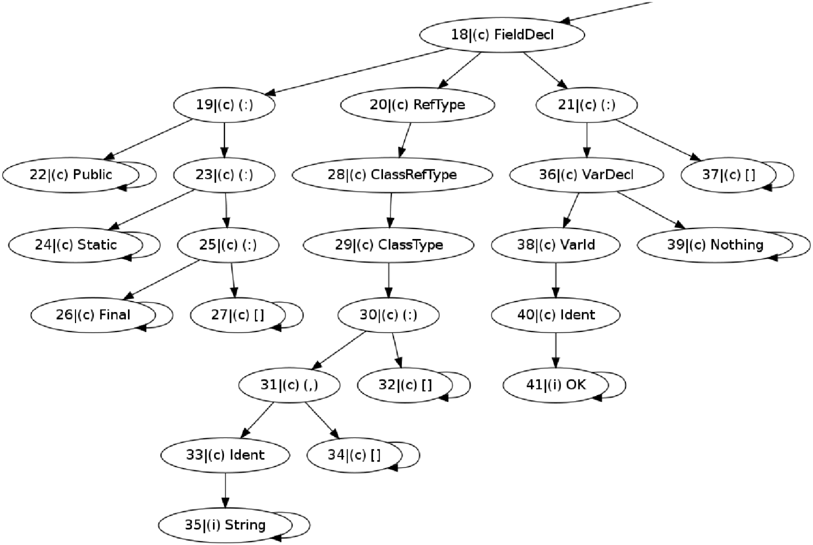
- after modification of a file the position has to be adjusted.
- using the first approach every single dependency has to be defined by hand. The logical formula has to be defined only once.

Agenda

- How to represent an ast?
- How to define dependencies?
- How to check dependencies?

Representing a abstract syntax tree (1/2)

```
data Label = Constr String | Ident String
```



Representing a abstract syntax tree (2/2)

Definition: Kripke structure

For a set of atomic propositions AP a Kripke structure $K = (S, R, I, lab)$ consists of the following components:

- a set of states S
- a transition relation $R \subseteq S \times S$
- a set of initial states $I \subseteq S$
- a state labeling function $lab : S \rightarrow AP$

```
type KripkeState = Int -- unique identifier

class Kripke (k :: ∗ → ∗) where
  states :: k l → [KripkeState]
  initStates :: k l → [KripkeState]
  rel :: KripkeState → KripkeState → k l → Bool
  label :: KripkeState → k l → l

  termToKripke :: Data t ⇒ t → k Label
```

How to define dependencies?

A type class for logics

```
class Logic logic where
  eval :: (Eq label,Kripke kripke)
       ⇒ kripke label → logic label → [KripkeState]
```

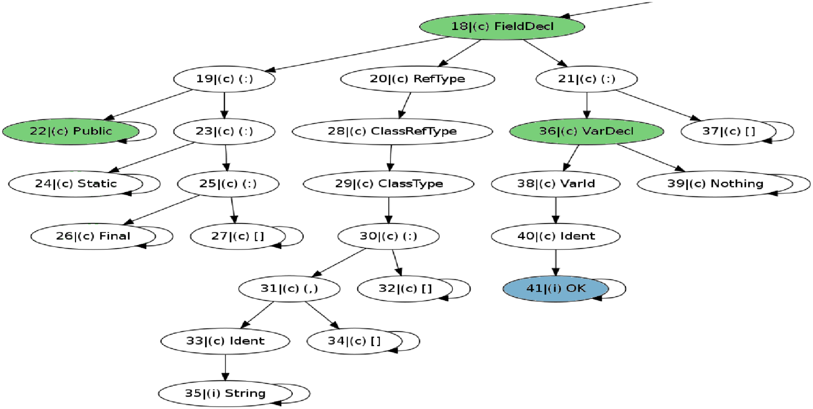
Basic temporal logic

```
data TL a
  = TT | FF | Ap a | Neg (TL a)
  | Disj (TL a) (TL a)
  | Conj (TL a) (TL a)
  | EX (TL a) -- exists next
  | EF (TL a) -- exists future
  | EY (TL a) -- exists yesterday (dual to EX)
  | EP (TL a) -- exists past (dual to EF)

instance Logic TL where -- ...
```

Defining a temporal formula

We want to define a formula which holds at every state representing a name of a public attribute.



```
(EP $ AP $ Constr "VarDecl")
‘Conj‘
(EP $ Conj (AP $ Constr "FieldDecl") (EF $ AP $ Constr "Public") )
```

Dependencies

A datatype for contexts

```
data Context t l where
  Context :: (Logic l,Data t)
   ⇒ FilePath → (String → Maybe t) → l Label → Context t l

evalC :: (Logic l,Data t) ⇒ Context t l → IO (Set Label)
```

A datatype for dependencies

```
data Dependency a b l where
  Dependency :: (Data a,Data b,Logic l)
    ⇒ (Context a l) → (Context b l) → Rel → Dependency a b l

data Rel = Subset | Equal

evalDep :: (Logic l,Data a,Data b) ⇒ Dependency a b l → IO Bool
```

Example: dependencies at work

Two harmless files. . .

```
class I18N{
  public static final String OK;
  public static final String CANCEL;
  public static final String REFRESH;
  public static final String TITLE;
}
```

```
#I18N_en.properties
OK=Ok
CANCEL=Cancel
REFRESH=Refresh
TITL=Title
```

Resulting output

```
No match for string "TITLE" found
The following candidate strings where found:
 "TILE"
 "OK"
 "CANCEL"
 "REFRESH"
```

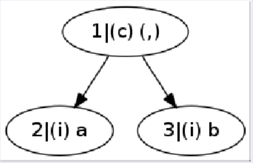
Problems

Definition of logical formulae

One needs a semantic knowlegde about the structure of the data type in order to define the logical formulae.
$\Rightarrow$ GUI for definition of formulae needed.

Separation of `Strings`

No temporal logic formula can seperate `"a"` and `"b"` in the following Kripke structure:



How to handle data types whose terms may contain more than one `String` as a successor of a constructor?

Separation of Strings

Extending temporal logic

As shown in the example one needs to state about specific successors of a constructor:

```
data TLex a
  = TL (TL a)
   | EX Int (TLex a) -- exists next at i-th successor
   | EF Int (TLex a) -- exists future in i-th subtree
```

No need to extend the yesterday and past operator (for our purposes) since every node has a unique predecessor in a tree.

Kripke structures

Kripke structures have a successor relation, but unfortunately the successors of a state are not ordered.

Separation of Strings

Solution

- extend successor relation of kripke structures, such that the successors of a state are ordered:

```
class Kripke k ⇒ KripkeOrd k where
  sucAtIdx :: Int → KripkeState → k l → Maybe KripkeState
```

- defining the evaluation algorithm for our new logic TLex based on a KripkeOrd is quite simple.
- KripkeOrd is a type class to represent rooted, labeled n-ary trees - ignoring the fact that Kripke and thus KripkeOrd allow multiple initial states, i.e. roots in a tree.

Idea

Why not evaluate logical formluae directly on abstract syntax trees?

Local model checking on Haskell terms

Local model checking (snipplet)

Let us start with local model checking:

```
-- holds :: TLex Label → Zipper → Bool
holds (AP (Constr a))   z = a ≡ (constructorname z)

holds (EX phi)          z = any (holds phi) (children z)

holds (EY phi) z = case up z of
  Nothing → False
  Just z' → holds phi z'

-- returns zippers to direct subtrees
-- children :: Zipper → [Zipper]
```

This works pretty nice!

Global model checking on Haskell terms

Global model checking

```
-- eval :: Data t ⇒ TLex Label → t → [String]
```

Naive algorithm: Traverse the ast and use the local model checking algorithm at every leaf representing an identifier.

Can we get better? Don't know yet...

Conclusion

What have we learned?

- dependencies between files matter.
- dependencies can be expressed using temporal logic.
- basic temporal logic is not expressive enough to define dependencies.
- one can apply local model checking directly to Haskell terms.

Further work

- Semiautomatic correction of violated dependencies
- How to define logical formulae? → GUI
- How to evaluate logical formulae on Haskell terms?

Sources can be found at `github.com/pascalh`

Questions?