Can we make use of ADTs in KeY?

Richard Bubel

June 28, 2005

Abstract Data Types (ADT)

```
\sorts {
  \object LString;
}

\functions {
  LString nil;
  // first is char modeled as int
  LString cat(int, LString);

  \nonRigid[location]
    LString content(java.lang.String);
  int length(LString);
  LString substring(int, int);
  // first is char modeled as int
  int indexOf(int, LString);
}
```

```
\rules {
 compute_length_1 {
   \find (length(cat(ch, lstr))) \replacewith (1+length(lstr))
 };

 compute_length_2 {
   \find (length(nil)) \replacewith (0)
 };

 LString_is_generated { // needs length definition
    \find (lstr)
    \varcond(\notFreeIn(chV, lstr), \notFreeIn(tailV, lstr))
    \add(\exists chV; \exists tailV;
       ((lstr=cat(chV, tailV) & length(lstr)=length(tailV)+1) |
         lstr=nil | lstr=null) ==>)
 };
```

Abstract Data Types (ADT)

```
\sorts {
  \object LString;
}

\functions {
  LString nil;
  // first is char modeled as int
  LString cat(int, LString);

  \nonRigid[location]
    LString content(java.lang.String);
  int length(LString);
  LString substring(int, int);
  // first is char modeled as int
  int indexOf(int, LString);
}
```

```
\rules {
 compute_length_1 {
   \find (length(cat(ch, lstr))) \replacewith (1+length(lstr))
 };

 compute_length_2 {
   \find (length(nil)) \replacewith (0)
 };

 LString_is_generated { // needs length definition
    \find (lstr)
    \varcond(\notFreeIn(chV, lstr), \notFreeIn(tailV, lstr))
    \add(\exists chV; \exists tailV;
       ((lstr=cat(chV, tailV) & length(lstr)=length(tailV)+1) |
         lstr=nil | lstr=null) ==>)
 };
```

- focus on functional specification
- well-founded theory
    - initiality → structural induction
- executable (if axioms allow definition of a term rewriting system)

Where may abstract data types help in KeY?

Structural induction

- make structural induction available in *JavaCardDL*
- generate correctness proof obligation

Where may abstract data types help in KeY?

Structural induction

- make structural induction available in *JavaCardDL*
- generate correctness proof obligation

Specification of concrete data types

- for general use in proofs, e.g. `java.lang.String`
- for intermediate usage: use to model partial aspects of a *Java* data type, e.g. inherent list structures

Where may abstract data types help in KeY?

Structural induction

- make structural induction available in *JavaCardDL*
- generate correctness proof obligation

Specification of concrete data types

- for general use in proofs, e.g. `java.lang.String`
- for intermediate usage: use to model partial aspects of a *Java* data type, e.g. inherent list structures

Therefore

- concrete data type has to be (partially) mapped to an ADT
- mapping has to be proven correct

Structural Induction - Preliminaries

Definition (Constructors $\mathcal{C}$)

Set of n-ary functions containing at least one nullary function (constants/base elements).
The nullary constants are usually described by a characterizing formula $\phi_{basis}(x)$.

For example: $\mathcal{C} = \{\texttt{null}, \texttt{next}\}$ or $\mathcal{C} = \{\texttt{null}, (\texttt{left}, \texttt{right})\}$

Structural Induction - Preliminaries

Definition (Constructors $\mathcal{C}$)

Set of n-ary functions containing at least one nullary function (constants/base elements).
The nullary constants are usually described by a characterizing formula $\phi_{basis}(x)$.

For example: $\mathcal{C} = \{\texttt{null}, \texttt{next}\}$ or $\mathcal{C} = \{\texttt{null}, (\texttt{left}, \texttt{right})\}$

Definition (Generated)

A data type T is generated by $\mathcal{C}$, if for all objects $o \in T$ there exists a ground term only made up of elements in C.

Structural Induction - Rule

Let $\Psi(x)$ denote the induction hypothesis over type $\mathbf{T}$

Base Case: $\texttt{==>}\ \backslash\texttt{forall}\ \mathbf{T}\ x; (\phi_{basis}(x)\ \texttt{->}\ \Psi(x))$

Step Case: $\texttt{==>}\ \bigwedge_{c \in \mathcal{C}, \alpha(c)=n} \backslash\texttt{forall}\ \mathbf{T}\ y, x_1, \ldots, x_n;$
$(\bigwedge_{i=1\ldots n} \Psi(x_i)\ \texttt{\&}\ y \doteq c(x_1 \ldots x_n)\texttt{->}\Psi(y))$

Use Case: $\backslash\texttt{forall}\ \mathbf{T}\ x; \Psi(x)\ \texttt{==>}$

Structural Induction - Rule

Let $\Psi(x)$ denote the induction hypothesis over type $\mathbf{T}$

Base Case: $\texttt{==>} \setminus\texttt{forall}\ \mathbf{T}\ x; (\phi_{basis}(x)\ \texttt{->}\ \Psi(x))$

Step Case: $\texttt{==>} \bigwedge_{c\in\mathcal{C},\alpha(c)=n} \setminus\texttt{forall}\ \mathbf{T}\ y, x_1, \ldots, x_n;$
$(\bigwedge_{i=1\ldots n}\Psi(x_i)\ \texttt{\&}\ y \doteq c(x_1 \ldots x_n)\texttt{->}\Psi(y))$

Use Case: $\setminus\texttt{forall}\ \mathbf{T}\ x; \Psi(x)\ \texttt{==>}$

Example (Single Linked List)

$\mathbf{T} = \texttt{List},\ \Phi_{basis}(x) :\Leftrightarrow x \doteq \texttt{null},\ \mathcal{C} := \{\texttt{next}\}$

Base Case: $\texttt{==>} \setminus\texttt{forall List}\ x; (x = \texttt{null->}\ \Psi(x))$

Step Case: $\texttt{==>} \setminus\texttt{forall List}\ y, x_1; (\Psi(x_1)\ \texttt{\&}\ y.\texttt{next} \doteq x_1\texttt{->}\Psi(y))$

Use Case: $\setminus\texttt{forall List}\ x; \Psi(x)\ \texttt{==>}$

Induction Rule - Soundness

Soundness Proofobligation:

$$\forall y : \mathbf{T}.\mathit{generated}(y)$$

where

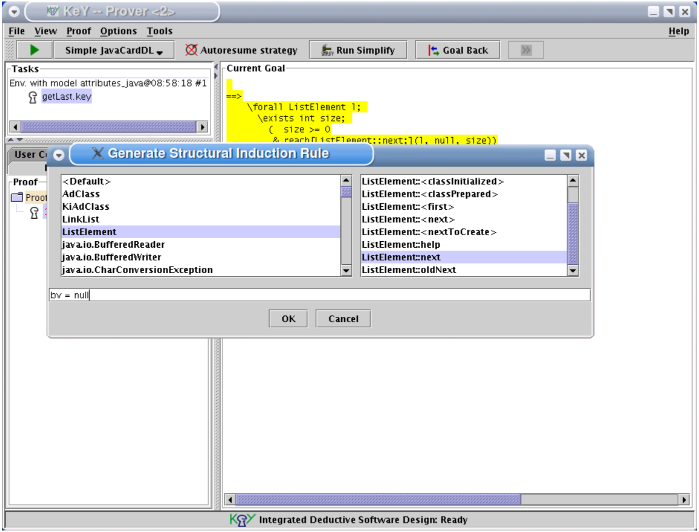
$$\mathit{generated}(y) :\Leftrightarrow \; \exists d : \mathtt{int}.(d >= 0 \;\&\; \mathit{generated}(y, d)) :\Leftrightarrow$$

$$\bigvee_{c \in \mathcal{C}, \alpha(c)=n} \exists x_1 \ldots x_n : \mathbf{T}.\exists d_1 \ldots d_n : \mathtt{int}.$$

$$(d_1 >= 0 \;\& \ldots \&\; d_n >= 0 \;\&$$

$$y = c(x_1 \ldots x_n) \;\&\; d = \mathit{max}\{d_1 \ldots d_n\} + 1 \;\&\; \bigwedge_{i=1 \ldots n} \mathit{generated}(x_i, d_i))$$

Structural Induction - In KeY



```
==>
  \forall ListElement l;
    \exists int size;
      (  size >= 0
       & reach[ListElement::next;](l, null, size))
```

bv = null

Specification of concrete data types

Claim: In some cases an ADT specification offers an easier treatment of data types

Example (`String` support in KeY)

- `Strings` as an array of characters clutters proof
- typical interested in the content of a `String`

Introduce a string ADT `LString` modeling string literals
Provide operations like `substring` or `indexOf`
Link to `java.lang.String` via `content:String->LString` function

Specification of concrete data types

Claim: In some cases an ADT specification offers an easier treatment of data types

Example (`String` support in KeY)

- `String`s as an array of characters clutters proof
- typical interested in the content of a `String`

Introduce a string ADT `LString` modeling string literals
Provide operations like `substring` or `indexOf`
Link to `java.lang.String` via `content:String->LString` function

```
\<{ s = "ab"; }\>s.content = cat('a',cat('b',nil))
```

Specification of concrete data types

Claim: In some cases an ADT specification offers an easier treatment of data types

Example (`String` support in KeY)

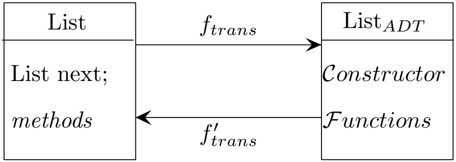
- `Strings` as an array of characters clutters proof
- typical interested in the content of a `String`

Introduce a string ADT `LString` modeling string literals
Provide operations like `substring` or `indexOf`
Link to `java.lang.String` via `content:String->LString` function

```
\<{ s = "ab"; }\>s.content = cat('a',cat('b',nil))
Apply: assign_string_lit
{s:=c_new, c_new.content:=cat('a',cat('b',nil))}
     \<{ }\>s.content = cat('a',cat('b',nil))
```

Mapping from Java to ADT

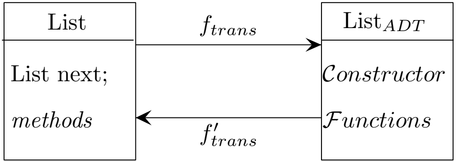
Mapping



Rules Symbolic Execution of Java works on the ADT

Mapping from Java to ADT

Mapping



List

List next;

methods

f_{trans}

f'_{trans}

List_{ADT}

$\mathcal{C}onstructor$

$\mathcal{F}unctions$

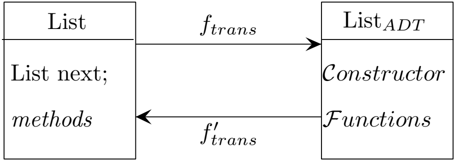
Rules Symbolic Execution of Java works on the ADT

```
rw_eqn {
  \find(ll1.#next = ll2 ==>)
  \replacewith(ll1=cons(head(ll1), ll2)==>) };

assign_abstract {
  \find (\<{.. #o.#next = #se; ...}\> post)
  \replacewith((!(#o=null)->{#o:=cons(head(#o), #se)}
                 \<{.. ...}\>post)) };
```

Mapping from Java to ADT

Mapping

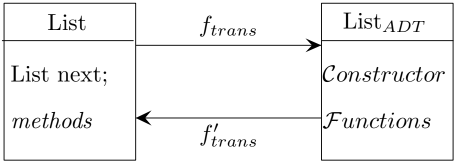


Rules Symbolic Execution of Java works on the ADT

```
list_induction {
  \varcond(\notFreeIn(ve,ind))
  "Base Case": \add(==> {\subst iv; null} ind);
  "Step Case": \add(==> \forall iv;(ind ->
       \forall ve;{\subst iv; cons(ve, iv)}ind));
  "Use Case":  \add(\forall iv;ind==>)
};
```

Mapping from Java to ADT

Mapping



Rules Symbolic Execution of Java works on the ADT

We want

$$\mathcal{D} \models f_{trans}(\phi) \Rightarrow \mathcal{D} \models f'_{trans}(f_{trans}(\phi)) \rightarrow \phi$$

Which properties of the mapping guarantee sound rules?

Future Work

- Functional verification of several *Java Collection Framework* classes (e.g. `LinkedList`, `ArrayList`, `TreeSet`)
- Optimising proofs of generateness and well-founded properties
- Reuse of known structures and proven properties in classes (signature homorphisms)