Foundations of Programming Languages and Software Engineering

Universität Freiburg

June 19, 2012

- Abstract Data Types
Abstract Data Types

- We have learned about different datastructures, e.g. for dictionaries:
  - Search trees
  - Lists
  - Tables with hashing

- Implementations of these concepts may have different characteristics:
  - Memory usage
  - Efficiency

- Implementations should be exchangeable

- Abstract over the concepts, use ADTs!
  - Functional specification
  - Implementation independent
  - Different implementations of a single ADT are possible
ADTs are Special Signatures

Definition

Let $\Sigma$ be a signature.

- A $\Sigma$-identity is a pair $(s, t) \in T(\Sigma, X) \times T(\Sigma, X)$. We write a $\Sigma$-identity as $s \approx t$ for emphasis.
- An ADT is a pair $(\Sigma, \mathcal{E})$ where
  - $\Sigma$ is a signature,
  - $\mathcal{E} \subseteq T(\Sigma, X) \times T(\Sigma, X)$ is a set of $\Sigma$-identities.
An ADT for natural numbers

\[ \Sigma_{nat} = \{ \text{zero}^{(0)}, \text{succ}^{(1)} \} \]
\[ \mathcal{E}_{nat} = \emptyset \]
Examples

**An ADT for natural numbers**

\[ \Sigma_{nat} = \{ \text{zero}^{(0)}, \text{succ}^{(1)} \} \]
\[ \mathcal{E}_{nat} = \emptyset \]

**An ADT for integers**

\[ \Sigma_{int} = \{ \text{zero}^{(0)}, \text{pred}^{(1)}, \text{succ}^{(1)} \} \]
\[ \mathcal{E}_{int} = \{ \text{pred}(\text{succ}(x)) \approx x, \text{succ}(\text{pred}(x)) \approx x \} \]
Datatypes are $\Sigma$-Algebras

**Definition**

An identity $s \approx t$ is **valid** in a $\Sigma$-algebra $\mathcal{A} = (A, \mathcal{J})$ iff $\mathcal{J}_\alpha(s) = \mathcal{J}_\alpha(t)$ for all variable assignments $\alpha : X \rightarrow A$. 
Datatypes are $\Sigma$-Algebras

**Definition**

- An identity $s \approx t$ is valid in a $\Sigma$-algebra $A = (A, I)$ iff $I_\alpha(s) = I_\alpha(t)$ for all variable assignments $\alpha : X \to A$.

- A **datatype** is a $\Sigma$-algebra $\mathcal{D}$.

- A datatype $\mathcal{D}$ implements the ADT $(\Sigma, \mathcal{E})$ iff every identity $s \approx t \in \mathcal{E}$ is valid in $\mathcal{D}$.

*(Note: We shall refine this definition later.)*
Implementations of the ADT for Naturals

**Implementation 1**

\[ D\text{nat}' = (\mathbb{N}, J'), \quad J'(\text{zero}) = 0, \quad J'(\text{succ})(x) = x + 1 \]

(note that \( x \) is meta-notation!).

- No identities, so all are valid.
- The function \( J' \) is bijective.

**Implementation 2**

\[ D\text{nat}'' = (\{0, 1, 2, 3\}, J'') \]

\[ J''(\text{zero}) = 0, \quad J''(\text{succ})(x) = (x + 1) \mod 4. \]

No identities, so all are valid.

\( J''(\text{zero}) = 0 = J''(\text{succ}(\text{succ}(\text{succ}(\text{succ}(\text{zero}))))). \)
Implementations of the ADT for Naturals

Implementation 1

\[ \mathcal{D}_{\text{nat}'} = (\mathbb{N}, \mathcal{J}'), \mathcal{J}'(\text{zero}) = 0, \mathcal{J}'(\text{succ})(x) = x + 1 \]

(note that \( x \) is meta-notation!).

- No identities, so all are valid.
- The function \( \mathcal{J}' \) is bijective.

Implementation 2

\[ \mathcal{D}_{\text{nat}''} = (\{0, 1, 2, 3\}, \mathcal{J}''), \mathcal{J}''(\text{zero}) = 0, \]
\[ \mathcal{J}''(\text{succ})(x) = (x + 1) \mod 4. \]

- No identities, so all are valid.
- The function \( \mathcal{J}'' \) is not injective (but surjective).

\[ \mathcal{J}''(\text{zero}) = 0 = \mathcal{J}''(\text{succ}(\text{succ}(\text{succ}(\text{succ}(\text{zero}))))) \]
\( \mathcal{D}_{\mathbb{Z}} = (\mathbb{Z}, \mathcal{J}') \), \( \mathcal{J}'(\text{zero})(\cdot) = 0 \)
\( \mathcal{J}'(\text{succ})(x) = x + 1 \)
\( \mathcal{J}'(\text{pred})(x) = x - 1 \)

For arbitrary \( \alpha : \{x\} \to \mathbb{Z} \) we have
\[
\mathcal{J}'_{\alpha}(\text{pred}(\text{succ}(x))) = (\alpha(x) + 1) - 1 = \mathcal{J}'_{\alpha}(x)
\]
\[
\mathcal{J}'_{\alpha}(\text{succ}(\text{pred}(x))) = (\alpha(x) - 1) + 1 = \mathcal{J}'_{\alpha}(x)
\]

\( \mathcal{J}' \) is surjective but not injective. Consider
\( \mathcal{J}'(\text{zero}) = 0 = \mathcal{J}'(\text{succ}(\text{pred}(\text{zero}))). \)
Implementations of the ADT for Integers (2)

Implementation 2

\[ \mathcal{D}int'' = (\{0, 1, 2, 3\}, \mathcal{J}'') \]

\[ \mathcal{J}''(\text{zero})() = 0 \]
\[ \mathcal{J}''(\text{succ})(x) = x + 1 \mod 4 \]
\[ \mathcal{J}''(\text{pred})(x) = x - 1 \mod 4 \]

For arbitrary \( \alpha : \{x\} \rightarrow \mathbb{Z} \) we have

\[ \mathcal{J}_{\alpha}''(\text{pred}(\text{succ}(x))) = \mathcal{J}_{\alpha}''(x) \]
\[ \mathcal{J}_{\alpha}''(\text{succ}(\text{pred}(x))) = \mathcal{J}_{\alpha}''(x) \]

\[ \mathcal{J}'' \text{ is surjective but not injective.} \]
A Non-implementation

\[ Dint'' = (\mathbb{N}, J''') \]

\[ J'''(\text{zero})() = 0 \]

\[ J'''(\text{succ})(x) = x + 1 \]

\[ J'''(\text{pred})(x) = \begin{cases} x - 1 & x > 0 \\ 0 & x = 0 \end{cases} \]

Not an implementation:
For \( \alpha : X \rightarrow \mathbb{N} \) with \( \alpha(x) = 0 \) we have

\[ J_\alpha(\text{succ(\text{pred}(x)))) = 1 \neq 0 = J_\alpha(x) \]
Fixing the Problems

- Want to rule out implementations such as $\mathit{Dnat}'$ and $\mathit{Dint}'$.
- Definition of “implementation” is too weak.
- Needed: restriction on function $J$.
  - $J$ is not necessarily injective (see $\mathit{Dint}'$).
  - Idea: $J$ must be injective on the equivalence classes induced by the identities of an ADT.
  - In other words: $J$ should make those terms equal that are equal according to the identities of the ADT, but not more!
Equivalence Classes

Definition

Suppose $R$ is an equivalence relation on some set $M$.

- The set $[x]_R := \{y \in M \mid x R y\}$ is called the equivalence class of $x$.
- $y \in [x]_R$ is called a representative of $[x]_R$.
- The quotient of $M$ with respect to $R$ is the set of equivalence classes induced by $R$, written $M/_{R} := \{[x]_R \mid x \in M\}$.

Note: For equivalence classes $[x]_R$ and $[y]_R$ we have either $[x]_R = [y]_R$ or $[x]_R \cap [y]_R = \emptyset$. 

Foundations of Programming Languages and Software Engineering
Definition

Suppose $\Sigma$ is a signature and let $R$ be an equivalence relation on $T(\Sigma, X)$.

- $R$ is a congruence relation iff $R$ is closed under $\Sigma$-operations, i.e. $t_i R t'_i$ implies $f(t_1, \ldots, t_i, \ldots, t_n) R f(t_1, \ldots, t'_i, \ldots, t_n)$ for any $n \geq 0$, $f \in \Sigma^{(n)}$, and $t_1, \ldots, t_n, t'_i \in T(\Sigma, X)$. 

Congruence Relations
Syntactic Quotient Algebras

**Definition**

Let $\Sigma$ be a signature and $R$ be a congruence on $T(\Sigma, X)$. For all $n \geq 0$, $f \in \Sigma^{(n)}$, and $t_1, \ldots, t_n \in T(\Sigma, X)$, define $\mathcal{J}^R$ as follows:

$$\mathcal{J}^R(f)([t_1]_R, \ldots, [t_n]_R) = [f(t_1, \ldots, t_n)]_R$$

**Note**

- $(T(\Sigma, X)/_R, \mathcal{J}^R)$ is a $\Sigma$-algebra. Its carrier elements are sets of terms.
- The representatives are arbitrary: Let $n \geq 0$, $f \in \Sigma^{(n)}$, and $s_1, t_1, \ldots, s_n, t_n \in T(\Sigma, X)$. If $s_1 R t_1, \ldots, s_n R t_n$ then $f(s_1, \ldots, s_n) R f(t_1, \ldots, t_n)$. Hence, $[f(s_1, \ldots, s_n)]_R = [f(t_1, \ldots, t_n)]_R$, as $R$ is a congruence.
Let \((\Sigma, \mathcal{E})\) be an ADT. We define a relation \(\approx_{\mathcal{E}}\) on \(T(\Sigma, X)\) as the smallest relation such that

- \(\approx_{\mathcal{E}}\) is a congruence relation;
- \(\approx_{\mathcal{E}}\) contains \(\mathcal{E}\), i.e. \(s \approx t \in \mathcal{E}\) implies \(s \approx_{\mathcal{E}} t\);
- \(\approx_{\mathcal{E}}\) is closed under substitutions, i.e. \(s \approx_{\mathcal{E}} t\) implies \(\sigma(s) \approx_{\mathcal{E}} \sigma(t)\) for any substitution \(\sigma\) and all \(s, t \in T(\Sigma, X)\).
Example

Congruence classes of $\approx_{\text{int}}$

$$\Sigma_{\text{int}} = \{ \text{zero}^{(0)}, \text{pred}^{(1)}, \text{succ}^{(1)} \}$$

$$\mathcal{E}_{\text{int}} = \{ \text{pred}(\text{succ}(x)) \approx x, \quad \text{succ}(\text{pred}(x)) \approx x \}$$

$$[\text{zero}]_{\approx_{\text{int}}} = \{ \text{zero}, \quad \text{succ}(\text{pred}(\text{zero})),$$

$$\text{pred}(\text{succ}(\text{zero})), \quad \text{succ}(\text{succ}(\text{pred}(\text{pred}(\text{zero}))))), \ldots \}$$
Revised Definition for ADT Implementations

Definition

A datatype $\mathcal{D} = (M, \mathcal{J})$ implements ADT $(\Sigma, \mathcal{E})$ with constructors $\Gamma \subseteq \Sigma$ if

- $(M, \mathcal{J})$ is a $\Sigma$-algebra (as before)
- All identities from $\mathcal{E}$ are valid in $M$ (as before)
- For all $s, t \in T(\Gamma, \emptyset)$: $s \approx_\mathcal{E} t$ iff $\mathcal{J}(s) = \mathcal{J}(t)$ (new!)
Example

Nat as a constructor-based ADT (CADT)

CADT: $\Sigma = \{\text{zero, succ}\}, \mathcal{E} = \{\}, \Gamma = \Sigma$

Implementation: $(\mathbb{N}, \mathcal{J}_1)$ with $\mathcal{J}_1(\text{zero})(\cdot) = 0$ and $\mathcal{J}_1(\text{succ})(x) = x + 1$

- $(\mathbb{N}, \mathcal{J}_1)$ is $\Sigma$-algebra
- No identities to check
- Since $\mathcal{E} = \emptyset$, $\approx_{\mathcal{E}}$ is $=$. Suppose $s, t \in T(\Gamma, \emptyset)$.
  - If $s = t$ then $\mathcal{J}_1(s) = \mathcal{J}_1(t)$
  - Suppose $s \neq t$. Then $s = \text{succ}^n(t)$ with $n > 0$. Hence, $\mathcal{J}_1(s) = \mathcal{J}_1(t) + n \neq \mathcal{J}_1(t)$. 

Foundations of Programming Languages and Software Engineering
Dint′′ is not an implementation of the natural numbers

CADT: \( \Sigma = \{\text{zero}, \text{succ}\}, \mathcal{E} = \{\}, \Gamma = \Sigma \)
\(\{0, 1, 2, 3\}, \mathcal{J}''\) with \(\mathcal{J}''(\text{zero})() = 0, \mathcal{J}''(\text{succ})(x) = (x + 1) \mod 4\) is not an implementation.

- \(\{0, 1, 2, 3\}, \mathcal{J}''\) is \(\Sigma\)-algebra
- No identities to check
- Since \(\mathcal{E} = \emptyset\), \(\approx_{\mathcal{E}}\) is =.

We have \(\text{zero} \neq \text{succ}^4(\text{zero})\) but
\(\mathcal{J}''(\text{zero}) = 0 = \mathcal{J}''(\text{succ}^4(\text{zero}))\).
Example

Alternative implementation of the natural numbers

CADT: \( \Sigma = \{ \text{zero}, \text{succ} \}, \emptyset, \Gamma = \Sigma \)

Implementation: \( (\{a\}^*, J^{''''}) \) with \( J^{''''}(\text{zero})() = \epsilon, \)
\( J^{''''}(\text{succ})(w) = aw \)

- \( (\{a\}^*, J^{''''}) \) is \( \Sigma \)-algebra
- No identities to check
- Since \( \emptyset = \emptyset \), \( \approx_{\emptyset} \) is =.
  - If \( s = t \) then \( J^{''''}(s) = J^{''''}(t) \)
  - Suppose \( s \neq t \). Then \( s = \text{succ}^n(t) \) with \( n > 0 \). Hence,
    \( J^{''''}(s) = J^{''''}(t) \underbrace{a\ldots a}_{n} \neq J^{''''}(t) \).
Theorem

Let \((\Sigma, \mathcal{E})\) be an ADT with constructors \(\Gamma \subseteq \Sigma\). Then \(D = (T(\Sigma, \emptyset)/\approx_\varepsilon, J^{\approx_\varepsilon})\) is an implementation of \((\Sigma, \mathcal{E})\).

Proof. Omitted
Suppose $\Gamma = \Sigma = \{\text{zero, succ, pred}\}$,
$E = \{\text{succ(pred(x))} = x, \text{pred(succ(x))} = x\}$

**Question:** What is $T(\Sigma, \emptyset)/\approx_E$?

**Answer:** Give a representative for every equivalence class.

**Lemma**

For every term $t \in T(\Sigma, \emptyset)$, exactly one of the following propositions holds

A. There exists $n > 0$ such that $t \in [\text{succ}^n(\text{zero})]_{\approx_E}$.
B. $t \in [\text{zero}]_{\approx_E}$.
C. There exists $n > 0$ such that $t \in [\text{pred}^n(\text{zero})]_{\approx_E}$.
The proof is by term induction over $t$.

- Base case: $t = \text{zero}$. Then $B$ holds.
Proof (cont.)

- Induction Step for $t = \text{succ}(t')$. By the IH, one of the following holds for $t'$:

  A. $t' \approx_{\mathcal{E}} \text{succ}^{(n)}(\text{zero})$ for $n > 0$: then 
     $\text{succ}(t') \approx_{\mathcal{E}} \text{succ}(\text{succ}^{(n)}(\text{zero})) = \text{succ}^{(n+1)}(\text{zero})$. 
     Since $n + 1 > 0$ we have case A.

  B. $t' \approx_{\mathcal{E}} \text{zero}$: then $\text{succ}(t') \approx_{\mathcal{E}} \text{succ}(\text{zero})$. We have 
     case A with $n = 1$.

  C. $t' \approx_{\mathcal{E}} \text{pred}^{(n)}(\text{zero})$ for $n > 0$: Then 
     $\text{succ}(t') \approx_{\mathcal{E}} \text{succ}(\text{pred}^{(n)}(\text{zero}))$. 
     If $n = 1$ then $\text{succ}(\text{pred}(\text{zero})) \approx_{\mathcal{E}} \text{zero}$ so case B holds. 
     If $n > 1$ then 
     $\text{succ}(\text{pred}(\text{pred}^{(n-1)}(\text{zero}))) \approx_{\mathcal{E}} \text{pred}^{(n-1)}(\text{zero})$, so 
     case C holds.
Proof (cont.)

- Induction step for $\text{pred}(t)$ analogous.
Equivalence Classes for Terms Representing Integers

Lemma

Suppose \( n > 0, m > 0 \). Then we have

- \( \text{zero} \not\approxE \text{succ}^n(\text{zero}) \),
- \( \text{zero} \not\approxE \text{pred}^n(\text{zero}) \),
- \( \text{succ}^n(\text{zero}) \not\approxE \text{pred}^m(\text{zero}) \),
- \( \text{succ}^n(\text{zero}) \not\approxE \text{succ}^m(\text{zero}) \) provided \( n \neq m \), and
- \( \text{pred}^n(\text{zero}) \not\approxE \text{pred}^m(\text{zero}) \) provided \( n \neq m \).

It follows that

\[
\{\text{succ}^n(\text{zero})|n > 0\} \cup \{\text{zero}\} \cup \{\text{pred}^n(\text{zero})|n > 0\}
\]

is a set of representatives for \( T(\Sigma, \emptyset)/\approxE \).
Example

Integers as a CADT

CADT: $\Gamma = \Sigma = \{\text{zero}, \text{succ}, \text{pred}\}$,
$E = \{\text{succ}(\text{pred}(x)) = x, \text{pred}(\text{succ}(x)) = x\}$

Implementation: $(\mathbb{Z}, J)$ with
$J(z) = 0, J(\text{succ})(x) = x + 1, J(\text{pred})(x) = x - 1$

- $(\mathbb{Z}, J)$ is a $\Sigma$-algebra
- All identities are valid (as seen before)
- An easy term induction shows for all $t \in T(\Sigma, \emptyset)$ that
  - if $t \approx_{E} \text{zero}$ then $J(t) = 0$,
  - if $t \approx_{E} \text{succ}^{n}(\text{zero})$ then $J(t) = n$, and
  - if $t \approx_{E} \text{pred}^{n}(\text{zero})$ then $J(t) = -n$.

Hence, if $s \approx_{E} t$ then $J(s) = J(t)$.
Conversely, if $s \not\approx_{E} t$ then $J(s) \neq J(t)$ because $J$ maps different representatives to different integers.
Summary

Slogan

Calculating with ADT = applying term operations + determining set of representatives.
Richer ADTs

Definition (Linear Data Structure)
An ADT is called a linear data structure (LDS) iff there is an implementation with simple lists.

- LDS are aggregates, i.e. one element contains several elements of another sort.
- Types are now parameterized.
  - Examples: List(A), Array(A)
  - A is not a fixed type but rather a type parameter.
  - Compare with Java Generics: List&lt;A&gt;, A[]
Lists

**Definition (Signature for Lists)**

<table>
<thead>
<tr>
<th>data type</th>
<th>List(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>operations</td>
<td></td>
</tr>
<tr>
<td>empty</td>
<td>$\rightarrow$ List(A)</td>
</tr>
<tr>
<td>cons</td>
<td>$A \times$ List(A) $\rightarrow$ List(A)</td>
</tr>
<tr>
<td>head</td>
<td>List(A) $\rightarrow$ A</td>
</tr>
<tr>
<td>tail</td>
<td>List(A) $\rightarrow$ List(A)</td>
</tr>
<tr>
<td>empty?</td>
<td>List(A) $\rightarrow$ Boolean</td>
</tr>
<tr>
<td>app</td>
<td>List(A) $\times$ List(A) $\rightarrow$ List(A)</td>
</tr>
<tr>
<td>len</td>
<td>List(A) $\rightarrow$ Nat</td>
</tr>
</tbody>
</table>

- Definition parameterized over $A$
- Constructors: `empty`, `cons`
- Definition uses more than one type: More general notion of signature and arity needed
Lists (cont.)

**Definition (Identities for Lists)**

<table>
<thead>
<tr>
<th>identities</th>
<th>head(cons(a, l)) = a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tail(cons(a, l)) = l</td>
</tr>
<tr>
<td></td>
<td>empty?(empty) = true</td>
</tr>
<tr>
<td></td>
<td>empty?(cons(a, l)) = false</td>
</tr>
<tr>
<td></td>
<td>app(empty, v) = v</td>
</tr>
<tr>
<td></td>
<td>app(cons(a, l), v) = cons(a, app(l, v))</td>
</tr>
<tr>
<td></td>
<td>len(empty) = zero</td>
</tr>
<tr>
<td></td>
<td>len(cons(a, l)) = succ(len(l))</td>
</tr>
</tbody>
</table>
Heterogeneous Signatures

Definition

Let $S$ be a set of sorts. A heterogeneous signature $\Sigma$ is a set of function symbols where each $f \in \Sigma$ is associated with an arity $s \to s'$ where $s \in S^*$ and $s' \in S$.

Examples

- Arity of `empty`: $\epsilon \to \text{List}(A)$
- Arity of `cons`: $(A, \text{List}(A)) \to \text{List}(A)$

Previous definitions need to be generalized as well:

- An algebra has different carrier sets for every sort.
- Terms must respect the sorts associated with a function symbol to rule out illegal terms such as $\text{cons}(1, 1)$.
- Generalize congruence relation
Dealing with Partial Operations

- head and tail are partial operations:
  - head(empty) = ?
  - tail(empty) = ?

- One possible solution: Introduce a distinguished element $\bot_s$ for every sort $s$
  - head(empty) = $\bot_A$
  - tail(empty) = $\bot_{\text{List}(A)}$

- All operations are strict in $\bot_s$, i.e. if one argument is $\bot_s$ the result is $\bot_{s'}$
  - head($\bot_{\text{List}(A)}$) = $\bot_A$
  - tail($\bot_{\text{List}(A)}$) = $\bot_{\text{List}(A)}$
  - empty?($\bot_{\text{List}(A)}$) = $\bot_{\text{Boolean}}$
  - len($\bot_{\text{List}(A)}$) = $\bot_{\text{Nat}}$
  - ...

Proposition

Let \( t \) be a term of type \( \text{List}(A) \) without variables. Then one of the following holds:

- \( t \approx_e t' \) and \( t' \in T(\Gamma, \emptyset) \) where \( \Gamma = \{ \text{empty}, \text{cons} \} \).
- \( t \approx_e \bot_{\text{List}(A)} \)

**Proof.** The proof is by induction on \( t \).

1. **Case** \( t = \text{empty} \in T(\Gamma, \emptyset) \). Trivial.
2. **Case** \( t = \text{cons}(a, s) \) with \( s \approx_e s' \) and \( s' \in T(\Gamma, \emptyset) \). Hence, \( \text{cons}(a, s) \approx_e \text{cons}(a, s') \in T(\Gamma, \emptyset) \).
3. **Case** \( t = \text{head}(s) \) so \( t \) does not have type \( \text{List}(A) \).
4. **Case** \( t = \text{tail}(s) \) with \( s \approx_e s' \) and \( s' \in T(\Gamma, \emptyset) \). If \( s' = \text{empty} \) then \( t \approx_e \bot_{\text{List}(A)} \). If \( s' = \text{cons}(a, s'') \) then \( t \approx_e s'' \in T(\Gamma, \emptyset) \).
Proof (cont.)

- Case $t = \text{empty}(s)$ but then $t$ does not have type $\text{List}(A)$.

- Case $t = \text{app}(s_1, s_2)$ with $s_1 \approx_\varepsilon s_1'$ and $s_2 \approx_\varepsilon s_2'$ and $s_1', s_2' \in T(\Gamma, \emptyset)$.
  
  - If $s_1' = \text{empty}$ then $t \approx_\varepsilon \text{app}(\text{empty}, s_2) \approx_\varepsilon s_2 \approx_\varepsilon s_2'$.
  
  - If $s_1' = \text{cons}(a, s_1'')$ then
    
    \[
    t = \text{app}(s_1, s_2) \approx_\varepsilon \text{app}(\text{cons}(a, s_1''), s_2) \\
    \approx_\varepsilon \text{cons}(a, \text{app}(s_1'', s_2))
    \]
    
    By the IH, we have $\text{app}(s_1'', s_2) \approx_\varepsilon s'$ with $s' \in T(\Gamma, \emptyset)$.

- Case $t = \text{len}(s)$ but then $t$ does not have type $\text{List}(A)$. 
Another Example

Sequences
- Also known as Array or Vector
- Parameterized over type of elements
- Fixed number of elements
- Direct access to elements (constant time)
### Definition (Signature for Arrays)

<table>
<thead>
<tr>
<th>data type</th>
<th>Array($A$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>operations</td>
<td></td>
</tr>
<tr>
<td><strong>new</strong></td>
<td>$\text{Nat} \times A \rightarrow \text{Array}(A)$</td>
</tr>
<tr>
<td><strong>update</strong></td>
<td>$\text{Array}(A) \times \text{Nat} \times A \rightarrow \text{Array}(A)$</td>
</tr>
<tr>
<td><strong>get</strong></td>
<td>$\text{Array}(A) \times \text{Nat} \rightarrow A$</td>
</tr>
<tr>
<td><strong>len</strong></td>
<td>$\text{Array}(A) \rightarrow \text{Nat}$</td>
</tr>
</tbody>
</table>

- Functional arrays
- In imperative languages: the `update` operation changes the array
Definition (Identities for Arrays)

identities

(i.1) \( \text{get}(\text{new}(n, x), i) = x \) if \( i < n \)

(i.2) \( \text{get}(\text{update}(a, i, x), i) = x \) if \( i < \text{len}(a) \)

(i.3) \( \text{get}(\text{update}(a, j, x), i) = \text{get}(a, i) \) if \( i \neq j \)

(i.4) \( \text{update}(\text{update}(a, i, x), i, y) = \text{update}(a, i, y) \)

(i.5) \( \text{update}(\text{update}(a, j, x), i, y) = \text{update}(\text{update}(a, i, y), j, x) \) if \( i \neq j \)

(i.6) \( \text{update}(\text{new}(n, x), i, x) = \text{new}(n, x) \) if \( i < n \)

(i.7) \( \text{len}(\text{new}(n, x)) = n \)

(i.8) \( \text{len}(\text{update}(a, i, x)) = \text{len}(a) \)

New: Conditional identities
Calculating with representatives

Suppose the carrier set of $A$ is $\{a, b, c\}$.

1. $\text{get}(\text{new}(10, a), 5) \approx_{\mathcal{E}} a$
2. $\text{get}(\text{new}(10, a), 11) \approx_{\mathcal{E}} \bot_A$
3. $\text{get}(\text{update}(\text{update}(\text{new}(10, a), 5, b), 0, c), 5)$
   $\approx_{\mathcal{E}} \text{get}(\text{update}(\text{new}(10, a), 5, b), 5) \approx_{\mathcal{E}} b$
4. $\text{update}(\text{update}(\text{new}(10, a), 5, b), 5, a)$
   $\approx_{\mathcal{E}} \text{update}(\text{new}(10, a), 5, a) \approx_{\mathcal{E}} \text{new}(10, a)$
5. $\text{get}(\text{update}(\text{new}(0, a), 1, a), 1) \approx_{\mathcal{E}} \bot_A$

(i.2) is not applicable because $\text{len}(\text{update}(\text{new}(0, a), 1, a)) \approx_{\mathcal{E}} 0$. 

Foundations of Programming Languages and Software Engineering