Obfuscating Set Representations

Stephen Drape
Oxford University Computing Laboratory

with thanks to Jeff Sanders
What is obfuscation?

Obfuscation is a program transformation:

- Used to make a program "harder to understand"
- Try to make reverse engineering harder
- Must preserve functionality
- Concerns about efficiency
Obfuscation is usually applied to object-oriented languages such as Java and C#.

When compiling these languages, an intermediate representation is produced.

It is possible to recover the original code from this representation – obfuscation can make this process harder.
Fresh Approach

Instead of obfuscating an imperative program, we consider obfuscating operations of a data-type – we can then exploit properties of that data-type (see later!).

We model the data-types and the operations in Haskell.
Deriving and Proving

We want to obfuscate some set operations.

Using functional programs, we can:
- **Derive** obfuscations
- Easily establish **proofs of correctness**

Both of these are difficult to do in imperative languages.
List splitting

Adapt "array splitting" – consider a particular example "alternating split"

Write \( xs \sim \langle l, r \rangle_a \) to denote \( xs \) is split into two lists \( l \) and \( r \) – \( xs \) is data refined by \( \langle l, r \rangle_a \)

\[
[5, 7, 5, 4, 3, 1, 1] \sim \langle [5, 5, 3, 1], [7, 4, 1] \rangle_a
\]

Invariant: \(|r| \leq |l| \leq |r|+1\)
Splitting Function

\[
\begin{align*}
\text{split}([ ]) &= \langle [ ], [ ] \rangle_a \\
\text{split}([p]) &= \langle [p], [ ] \rangle_a \\
\text{split}(p:q:xs) &= \langle p:l, q:r \rangle_a \\
\text{where } \langle l, r \rangle_a &= \text{split}(xs)
\end{align*}
\]

\[
\begin{align*}
\text{unsplit } \langle [ ], [ ] \rangle_a &= [ ] \\
\text{unsplit } \langle [p], [ ] \rangle_a &= [p] \\
\text{unsplit } \langle p:l, q:r \rangle_a &= p:q: \text{unsplit}(\langle l, r \rangle_a)
\end{align*}
\]
Derivations

\[ \text{split} \cdot \text{op} = \text{sp\_op} \cdot \text{split} \]

\[ \text{sp\_op} = \text{split} \cdot \text{op} \cdot \text{unsplit} \]

\[ \text{op} = \text{unsplit} \cdot \text{sp\_op} \cdot \text{split} \]
List operations

\[ p : \langle l, r \rangle_a = \langle p : r, l \rangle_a \]

\[ \langle l_0, r_0 \rangle_a \mathbin{++} \langle l_1, r_1 \rangle_a \]

| \text{|} |l_0| = |r_0| | = \langle l_0 \mathbin{++} l_1, r_0 \mathbin{++} r_1 \rangle_a |

| \text{|otherwise} | = \langle l_0 \mathbin{++} r_1, r_0 \mathbin{++} l_1 \rangle_a |

: \text{and } ++ \text{ distribute over } \text{split}
Unordered Lists without Duplicates

\[
\text{member } p \, xs = \text{or}(\text{map } (==p) \, xs)
\]

\[
\text{insert } p \, xs = \text{if member } p \, xs \text{ then } xs \\
\text{else } p:xs
\]

\[
\text{delete } p \, xs = ys ++ (\text{if null } zs \\
\text{then } zs \\
\text{else } \text{tail } zs)
\]

\[
\text{where } (ys,zs) = \text{span } (/=p) \, xs
\]
Deriving the delete operation for split lists

\[ \text{delete}_a p = \text{split.delete } p \cdot \text{unsplit} \]

Let 
\[ l = [l_0, l_1, \ldots, l_j, l_{j+1}, \ldots, l_n] \quad r = [r_0, r_1, \ldots, r_{n'}] \]

and \( xs \sim \langle l, r \rangle_a \)

We have three cases

(a) \( p \in l \)
(b) \( p \in r \)
(c) \( p \notin \langle l, r \rangle_a \)
Case (a) \hspace{1cm} (p \in l)

Suppose that \( l_j = p \)

Let \((l_y, l_z) = \text{span}(\neq p)\) \( l \)

\[
= ([l_0, l_1, \ldots, l_{j-1}], [l_j, l_{j+1}, \ldots, l_n])
\]

\((r_y, r_z) = \text{splitAt} \mid l_y \mid r\)

\[
= ([r_0, r_1, \ldots, r_{j-1}], [r_j, \ldots, r_{n'}])
\]
Case (a) \( (p \in l) \)

\[
delete_a p \langle l, r \rangle_a
\]

\( = \{\text{derivation equation}\} \)

\[
\text{split (delete } p \text{ (unsplit } \langle l, r \rangle_a \text{))}
\]

\( = \{\text{definition of unsplit}\} \)

\[
\text{split (delete } p \text{ [l}_0, r_0, l_1,\ldots\text{])}
\]

\( = \{\text{definition of delete, } l_j = p\} \)

\[
\text{split ([l}_0, r_0,\ldots, r_{j-1}] + + [r_j,\ldots])
\]
Case (a) \quad (p \in l)

\text{split}([l_0, r_0, \ldots, r_{j-1}] ++ [r_j, \ldots])

= \{\text{split distributes over ++}\}

\text{split}([l_0, r_0, \ldots]) ++ \text{split}([r_j, \ldots])

= \{\text{definition of split}\}

\langle[l_0, \ldots], [r_0, \ldots]\rangle_a ++ \langle[r_j, \ldots, r_n], [l_{j+1}, \ldots, l_n]\rangle_a

= \{\text{earlier definitions}\}

\langle ly, ry\rangle_a ++ \langle rz, \text{tail } lz\rangle_a
Case (a) \((p \in l)\)

\[
\langle ly, ry \rangle_a ++ \langle rz, \text{tail } lz \rangle_a
\]

\[
= \{ \text{definition of } ++, \ | ly | = | ry | \}\]

\[
\langle ly ++ rz, ry ++ \text{tail } lz \rangle_a
\]

We cannot simplify this further, but as lists are unordered:

\[
\langle ly, ry \rangle_a \text{ is equivalent to } \langle ry, ly \rangle_a
\]
Case (a) \hspace{1cm} (p \in l)

\begin{align*}
\langle ry, ly \rangle_a &+\!+ \langle rz, \text{tail } l_z \rangle_a \\
=&\{\text{definition of } +\!+\}
\langle ry +\!+ rz, ly +\!+ \text{tail } l_z \rangle_a \\
=&\{\text{definitions}\}
\langle r, \text{delete } p \ l \rangle_a
\end{align*}
Case (b) \hspace{1cm} \left( p \in r \right)

\[
\text{delete}_a \ p \langle l, r \rangle_a \\
= \left\{ l = (\text{head} \ l) : (\text{tail} \ l), \ l \neq [] \right\} \\
\text{delete}_a \ p \langle (\text{head} \ l) : (\text{tail} \ l), r \rangle_a \\
= \{ \text{definition of : } \} \\
\text{delete}_a \ p \langle (\text{head} \ l) : \langle r, \text{tail} \ l \rangle_a \rangle \\
= \{ \text{head} \ l \neq p \} \\
(\text{head} \ l) : (\text{delete}_a \ p \langle r, \text{tail} \ l \rangle_a)
\]
Case (b) \ (p \in \ r) \\

\[(\text{head} \ l):(\text{delete}_a \ p \ (r, \ \text{tail} \ l)_a)\]
\[= \{\text{previous definition of delete}_a \}\]
\[(\text{head} \ l):(\langle \text{tail} \ l, \ \text{delete} \ p \ r \rangle_a)\]
\[= \{\text{definition of} \ : \}\]
\[\langle (\text{head} \ l):(\text{delete} \ p \ r), \ \text{tail} \ l \rangle_a \]
Finally

\[(c) \quad p \notin l \text{ and } p \notin r \]

\[
delete_a p \langle l, r \rangle_a = \langle l, r \rangle_a
\]

Final definition

\[
delete_a p \langle l, r \rangle_a
\]

\[
| \quad \text{member } p l = \langle r, delete p l \rangle_a
\]

\[
| \quad \text{member } p r =
\]

\[
\langle (\text{head } l) : (\text{delete } p r), \text{tail } l \rangle_a
\]

| otherwise = \langle l, r \rangle_a
Insert operation

\[ \text{insert}_a p \langle l, r \rangle_a = \]
\[ \text{if member}_a p \langle l, r \rangle_a \]
\[ \text{then } \langle l, r \rangle_a \]
\[ \text{else } \langle p:r, l \rangle_a \]

We will now prove that

\[ \text{insert } p = \text{unsplit. (insert}_a p). \text{split} \]
Proof

Case for $p \in xs$ is trivial.

Otherwise, suppose that:

$$xs \sim \langle l, r \rangle_a \text{ and } p \notin xs$$

$$\text{unsplit}(\text{insert}_a \ p \ \text{split}(xs))$$

$$= \{xs \sim \langle l, r \rangle_a \}$$

$$\text{unsplit}(\text{insert}_a \ p \ (\langle l, r \rangle_a))$$
Proof

\[
\text{unsplit}(\text{insert}_a \ p \ (\langle l, r \rangle_a))
\]
\[
=\{\text{definition of } \text{insert}_a\}
\]
\[
\text{unsplit}(\langle p : r, l \rangle_a)
\]
\[
=\{\text{definition of :}\}
\]
\[
\text{unsplit}(p : \langle l, r \rangle_a)
\]
\[
=\{\text{property of :}\}
\]
\[
p: \text{unsplit}(\langle l, r \rangle_a)
\]
Proof

\[ p : \text{uns\text{u}\text{m}split} (\langle l, r \rangle_a) \]
\[ = \{ xs \sim \langle l, r \rangle_a \} \]

\[ p : xs \]
\[ = \{ \text{definition of } \text{insert} \} \]

**insert** \( p \) \( xs \)  \( \Box \)
Complexity

```
delete \ p \ xs = ys ++ (if \ null \ zs
then \ zs
else \ tail \ zs)
where (ys, zs) = span \ (\#p) \ xs
```

```
delete_a \ p \ \langle l, r \rangle_a
| member \ p \ l = \langle r, delete \ p \ l \rangle_a
| member \ p \ r = \langle (head \ l) : (delete \ p \ r), tail \ l \rangle_a
| otherwise = \langle l, r \rangle_a
```

Both functions have linear complexity.
Complexity

\[ \text{insert } p \ xs = \begin{cases} \text{if } \text{member } p \ xs \\ \text{then } xs \\ \text{else } p : xs \end{cases} \]

\[ \text{insert}_a p \ \langle l, r \rangle_a = \begin{cases} \text{if } \text{member}_a p \ \langle l, r \rangle_a \\ \text{then } \langle l, r \rangle_a \\ \text{else } \langle p : r, l \rangle_a \end{cases} \]

Again, these functions have linear complexity.
"Obfuscating Set Representations"

The paper looks at three representations:

- Unordered with duplicates
- Unordered without duplicates
- Strictly-increasing

Proofs and derivations of `delete` and `insert` are given for the other representations.

Also, another split is considered.
At the beginning, it was stated we obfuscate data-types directly so that we can exploit properties of the data-type.

Suppose that we want to split a matrix and we want to develop a transpose operation for the split matrix.

Suppose we flatten the matrix to an array and then split this array.
Using arrays means that we lose the "shape" of the matrix and so we have difficulty in constructing a transpose operation.

Using matrices directly:

\[
\begin{pmatrix}
A & B \\
C & D
\end{pmatrix}^T =
\begin{pmatrix}
A^T & C^T \\
B^T & D^T
\end{pmatrix}
\]
Conclusions

We have seen that using data-types and functional programming, we can

- derive obfuscations
- prove correctness

Our operations make little change to the complexity

Have to keep split secret
Future Work

Possible areas for future work

- Other obfuscations
- Other data-types (matrices, trees)
- Automation
- Obfuscation definition