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

Why is obfuscation needed?

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

$$\texttt{split}([\,]) = \langle [\,],[\,] \rangle_a$$

$$\texttt{split}([p]) = \langle [p],[\,] \rangle_a$$

$$\texttt{split}(p{:}q{:}xs) = \langle p : l, q : r \rangle_a$$

$$\quad \texttt{where } \langle l, r \rangle_{\mathrm{a}} = \texttt{split}(xs)$$

$$\texttt{unsplit } \langle [\,], [\,] \rangle_a = [\,]$$

$$\texttt{unsplit } \langle [p], [\,] \rangle_a = [p]$$

$$\texttt{unsplit } \langle p : l, q : r \rangle_a =$$

$$\quad p : q : \texttt{unsplit}(\langle l, r \rangle_a)$$

Derivations

`split . op = sp_op . split`

Data

Refinement

$$
\begin{tikzcd}
xs \arrow[r, "\mathsf{op}"] \arrow[d, "\mathsf{split}"'] & \mathsf{op}\ (xs) \arrow[d, "\mathsf{split}"'] \\
\langle l, r \rangle \arrow[r, "\mathsf{sp\_op}"] & \mathsf{sp\_op}\ (\langle l, r \rangle)
\end{tikzcd}
$$

```
sp_op =  split  .   op   . unsplit
op    = unsplit . sp_op . split
```

List operations

$$p : \langle l, r\rangle_a = \langle p : r, l\rangle_a$$

$$\begin{aligned}
&\langle l_0, r_0\rangle_a +\!+ \ \langle l_1, r_1\rangle_a \\
&\quad |\ |l_0| == |r_0| \qquad = \ \langle l_0 +\!+\ l_1, r_0 +\!+\ r_1\rangle_a \\
&\quad |\ \text{otherwise} \qquad = \ \langle l_0 +\!+\ r_1, r_0 +\!+\ l_1\rangle_a
\end{aligned}$$

`:` and ++ distribute over `split`

Unordered Lists without Duplicates

```
member p xs = or(map (==p) xs)

insert p xs = if member p xs then xs
                             else p:xs

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

Deriving Delete

Deriving the delete operation for split lists

$\texttt{delete}_{\texttt{a}}\ p = \texttt{split}.\texttt{delete}\ p.\texttt{unsplit}$

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

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) $(p \in l)$ 1

Suppose that $l_j = p$

Let $(ly, lz) = \texttt{span}\ (\texttt{/=}p)\ l$

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

$(ry, rz) = \texttt{splitAt}\ |ly|\ r$

$$= ([r_0, r_1, \ldots, r_{j-1}], [r_j, \ldots, r_{n'}])$$

Case (a) $(p \in l)$ 2

$\texttt{delete}_a\ p\ \langle l, r\rangle_a$

= {derivation equation}

$\texttt{split}(\texttt{delete}\ p\ (\texttt{unsplit}\ \langle l, r\rangle_a))$

={definition of $\texttt{unsplit}$}

$\texttt{split}(\texttt{delete}\ \ p\ [l_0, r_0, l_1, \ldots])$

={definition of $\texttt{delete}$, $l_j = p$}

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

Case (a) $(p \in l)$ 3

$\texttt{split}([l_0, r_0, \ldots, r_{j-1}] \mathbin{+\!\!+} [r_j, \ldots])$

$=\{\texttt{split}$ distributes over ++$\}$

$\texttt{split}([l_0, r_0, \ldots]) \mathbin{+\!\!+} \texttt{split}([r_j, \ldots])$

$=\{$definition of $\texttt{split}\}$

$\langle [l_0, \ldots], [r_0, \ldots] \rangle_a \mathbin{+\!\!+} \langle [r_j, \ldots, r_{n'}], [l_{j+1}, \ldots, l_n] \rangle_a$

$=\{$earlier definitions$\}$

$\langle ly, ry \rangle_a \mathbin{+\!\!+} \langle rz, \texttt{tail}\ lz \rangle_a$

Case (a) $(p \in l)$ 4

$$\langle ly, ry \rangle_a \mathbin{++} \langle rz, \texttt{tail}\ lz \rangle_a$$

$= \{$definition of $++$, $|ly| = |ry|\}$

$$\langle ly \mathbin{++} rz, ry \mathbin{++} \texttt{tail}\ lz \rangle_a$$

We cannot simplify this further,

but as lists are unordered:

$\langle ly, ry \rangle_a$ is equivalent to $\langle ry, ly \rangle_a$

Case (a) $(p \in l)$ 5

$\langle ry, ly \rangle_a \; +\!\!+ \; \langle rz, \texttt{tail}\; lz \rangle_a$

={definition of ++}

$\langle ry +\!\!+ rz, ly +\!\!+ \texttt{tail}\; lz \rangle_a$

={definitions}

$\langle r, \texttt{delete}\; p\; l \rangle_a$ □

Case (b) $(p \in r)$ 1

$\texttt{delete}_a \; p \, \langle l, r \rangle_a$

$= \{ \, l = (\texttt{head}\; l) \texttt{:} (\texttt{tail}\; l), \;\; l \neq [\;] \}$

$\texttt{delete}_a \; p \, \langle (\texttt{head}\; l) \texttt{:} \; (\texttt{tail}\; l), r \rangle_a$

$= \{\text{definition of :}\}$

$\texttt{delete}_a \; p \, ((\texttt{head}\; l) \texttt{:} \; \langle r, \texttt{tail}\; l \, \rangle_a)$

$= \{\texttt{head} \;\; l \neq p\}$

$(\texttt{head} \;\; l) \texttt{:} (\texttt{delete}_a \;\; p \; \langle r, \; \texttt{tail} \;\; l \, \rangle_a)$

Case (b) $(p \in r)$ 2

$(\texttt{head}\ l):(\texttt{delete}_a\ p\ \langle r, \texttt{tail}\ l\rangle_a)$

$=$ {previous definition of $\texttt{delete}_a$}

$(\texttt{head}\ l):(\langle \texttt{tail}\ l, \texttt{delete}\ p\ r\rangle_a)$

={definition of :}

$\langle(\texttt{head}\ l):(\texttt{delete}\ p\ r), \texttt{tail}\ l\rangle_a$ $\square$

Finally

(c) $p \notin l$ and $p \notin r$

$\texttt{delete}_a\ p \langle l, r \rangle_a = \langle l, r \rangle_a$ □

Final definition

$\texttt{delete}_a\ p \langle l, r \rangle_a$

$\quad |\ \texttt{member}\ p\, l\ = \langle r, \texttt{delete}\, p\, l \rangle_a$

$\quad |\ \texttt{member}\ p\, r\ =$

$\qquad\qquad \langle (\texttt{head}\ l) : (\texttt{delete}\ p\ r), \texttt{tail}\ l \rangle_a$

$\quad |\ \texttt{otherwise}\ = \langle l, r \rangle_a$

Insert operation

$$\mathtt{insert}_a\ p\ \langle l, r\rangle_a =$$
$$\quad \mathtt{if}\ \mathtt{member}_a\ p\ \langle l, r\rangle_a$$
$$\quad \mathtt{then}\ \langle l, r\rangle_a$$
$$\quad \mathtt{else}\ \langle p : r, l\rangle_a$$

We will now prove that

$$\mathtt{insert}\ p = \mathtt{unsplit} . (\mathtt{insert}_a\ p) . \mathtt{split}$$

Proof 1

Case for $p \in xs$ is trivial.

Otherwise, suppose that:

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

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

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

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

Proof 2

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

$=$ {definition of $\texttt{insert}_a$}

$\texttt{unsplit}(\langle p : r, l\rangle_a)$

$=$ {definition of :}

$\texttt{unsplit}(p : \langle l, r\rangle_a)$

$=$ {property of :}

$p : \texttt{unsplit}(\langle l, r\rangle_a)$

Proof 3

$$p:\texttt{unsplit}(\langle l, r\rangle_a)$$

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

$$p:xs$$

$$=\{\text{definition of } \texttt{insert}\}$$

$$\texttt{insert}\ p\ \ xs \qquad \square$$

Complexity 1

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

$\texttt{delete}_{\texttt{a}}\ p\ \langle l, r\rangle_a$

$\ \ |\ \texttt{member}\ p\ l = \langle r, \texttt{delete}\ p\ l\ \rangle_a$

$\ \ |\ \texttt{member}\ p\ r = \langle (\texttt{head}\ l):(\texttt{delete}\ p\ r), \texttt{tail}\ l\ \rangle_a$

$\ \ |\ \texttt{otherwise} = \langle l, r\rangle_a$

Both functions have linear complexity.

Complexity 2

$$\texttt{insert}\ p\ xs\ =\ \texttt{if member}\ p\ xs\ \texttt{then}\ xs\ \texttt{else}\ p\texttt{:}xs$$

$$\texttt{insert}_a\ p\ \langle l, r\rangle_a = \texttt{if member}_a\ p\ \langle l, r\rangle_a\ \texttt{then}\ \langle l, r\rangle_a\ \texttt{else}\ \langle p\texttt{:}r, l\rangle_a$$

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.

Matrices 1

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.

Matrices 2

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