Automated Module Interface Upgrade

László Lövei

Department of Programming Languages and Compilers
Faculty of Informatics
Eötvös Loránd University

Erlang Workshop 2009
Outline

1 Motivation
   Regular expression library
   Generic interface migration

2 Implementation
   Data flow analysis
   Change descriptions
   Prototype experiences
Regexp Upgrade Example

```erlang
find(Str) ->
    case regexp:match(Str, ?RE) of
        {match, Start, Len} ->
            strings:substr(Str, Start, Len);
        nomatch ->
            ""
    end.
```
Regexp Upgrade Example

```erlang
find(Str) ->
    case regexp_match(Str, ?RE) of
        {match, Start, Len} ->
            strings:substr(Str, Start, Len);
        nomatch ->
            ""
    end.

regexp_match(S, R) ->
    case re:run(S, R, [{capture,first}]) of
        {match, [{St, Ln}]} -> {match, St+1, Ln};
        nomatch -> nomatch
    end.
```
find(Str) ->
case \texttt{case re:run}(Str, \texttt{?RE}, [{\{capture,first\}@student\}]) of
    {match, [{St, Ln}]} -> {match, St+1, Ln};
    nomatch -> nomatch
end of
{match, Start, Len} ->
    strings:substr(Str, Start, Len);
nomatch ->
    ""
end.
find(Str) ->
  case re:run(Str, ?RE, [{capture,first}]) of
    {match, [{Start, Len}]} ->
      strings:substr(Str, Start+1, Len);
    nomatch ->
      ""
  end.
Regexp Upgrade Example

```erlang
find(Str) ->
    case re:run(Str, ?RE, [{capture,first}]) of
        {match, [{Start, Len}]} ->
            strings:substr(Str, Start+1, Len);
        nomatch ->
            ""
    end.
```

Return value changes are **propagated** to the place of usage
Generic Interface Migration

- Same functionality
  - Arguments and return values have the same information
- Incompatible interfaces
  - Data is restructured or slightly modified
- Many simple library changes could be supported by an automated generic migration tool

```erlang
case dict:find(K,S) of
  {ok, Val} -> Val;
  error -> throw(miss)
end
case gb_trees:lookup(K,S) of
  {value, Val} -> Val;
  none -> throw(miss)
end
```
Data Flow Analysis

\[
\text{find}(\text{Key}, [\{\text{Key}, \text{Val}\} | \_]) \rightarrow \text{Val};
\]

\[
\text{find}(\text{Key}, [\_ | \text{Tail}]) \rightarrow \text{find}(\text{Key}, \text{Tail}).
\]

\[
f() \rightarrow \text{find}(a, [\{a,1\}]).
\]
Data Flow Analysis

Direct edges: variables

\[
f\text{(Key, [{Key, Val}|_])} \rightarrow \text{Val};
\]

\[
f\text{(Key, [_|Tail])} \rightarrow f\text{(Key, Tail)}.
\]

\[
f() \rightarrow f\text{(a, [{a,1}])}.
\]
Data Flow Analysis

Direct edges: function calls

\[
\text{find}(\text{Key}, [\{\text{Key}, \text{Val}\}|\_]) \rightarrow \text{Val};
\]

\[
\text{find}(\text{Key}, [\_|\text{Tail}]) \rightarrow \text{find}(\text{Key}, \text{Tail}).
\]

\[
f() \rightarrow \text{find}(a, [\{a,1\}]).
\]
Data Flow Analysis

Direct edges: function calls

\[\text{find(Key, [{Key, Val}]|_])} \rightarrow \text{Val};\]

\[\text{find(Key, [_|Tail])} \rightarrow \text{find(Key, Tail)}.\]

\[f() \rightarrow \text{find(a, [{a,1}])}.\]
Data Flow Analysis

Direct edges: function calls

\[
\text{find}(\text{Key}, [\{\text{Key}, \text{Val}\} | \_]) \rightarrow \text{Val};
\]

\[
\text{find}(\text{Key}, [\_ | \text{Tail}]) \rightarrow \text{find}(\text{Key}, \text{Tail}).
\]

\[
f() \rightarrow \text{find}(\text{a}, [\{\text{a}, 1\}]).
\]
Data Flow Analysis

Direct edges: tuple selectors

\[ \text{find}(\text{Key}, [\{\text{Key}, \text{Val}\} | \_ ]) \rightarrow \text{Val}; \]

\[ \text{find}(\text{Key}, [\_ | \text{Tail}]) \rightarrow \text{find}(\text{Key}, \text{Tail}). \]

\[ f() \rightarrow \text{find}(a, [\{a,1\}]). \]
Data Flow Analysis

Direct edges: tuple constructors

\[
\text{find}(\text{Key}, [{\text{Key}, \text{Val}}|\_]) \rightarrow \text{Val};
\]

\[
\text{find}(\text{Key}, [\_|\text{Tail}]) \rightarrow \text{find}(\text{Key}, \text{Tail}).
\]

\[
f() \rightarrow \text{find}(\text{a}, [{\text{a}, 1}]).
\]
Data Flow Analysis

Direct edges: list element selectors

\[
\text{find}(\text{Key}, \ [{\text{Key}, \text{Val}}|\_\_]) \rightarrow \text{Val};
\]

\[
\text{find}(\text{Key}, \ [\_\_|\text{Tail}]) \rightarrow \text{find}(\text{Key}, \text{Tail}).
\]

\[
f() \rightarrow \text{find}(a, \ [{a,1}]).
\]
Data Flow Analysis

Direct edges: list constructors

\[
\text{find(Key, [{Key, Val}|_])} \rightarrow \text{Val};
\]

\[
\text{find(Key, [_|Tail])} \rightarrow \text{find(Key, Tail)}.
\]

\[
f() \rightarrow \text{find(a, [{a,1}])}.
\]
Data Flow Analysis

Reaching: transitivity

\[ \text{find(Key, [{Key, Val}|_])} \rightarrow \text{Val}; \]

\[ \text{find(Key, [_|Tail])} \rightarrow \text{find(Key, Tail)}. \]

\[ f() \rightarrow \text{find(a, [{a,1}])}. \]
Data Flow Analysis

Reaching: list construction and selection

\[
\text{find}(\text{Key}, [{\text{Key}, \text{Val}}|\_]) \rightarrow \text{Val};
\]

\[
\text{find}(\text{Key}, [\_|\text{Tail}]) \rightarrow \text{find}(\text{Key}, \text{Tail}).
\]

\[
f() \rightarrow \text{find}(a, [{a,1}]).
\]
Data Flow Analysis

Reaching: tuple construction and selection

\[ \text{find}(\text{Key}, \text{Val}) \rightarrow \text{Val}; \]

\[ \text{find}(\text{Key}, \text{Tail}) \rightarrow \text{find}(\text{Key}, \text{Tail}). \]

\[ f() \rightarrow \text{find}(\text{a}, \text{Val}); \]
Data Flow Analysis

Propagation: finding the end of data flow paths

\[
\text{find}(\text{Key}, \{\{\text{Key}, \text{Val}\}\mid _\}\}) \rightarrow \text{Val};
\]

\[
\text{find}(\text{Key}, [\_\mid \text{Tail}]) \rightarrow \text{find}(\text{Key}, \text{Tail}).
\]

\[
f() \rightarrow \text{find}(a, \{a, 1\}).
\]
Describing Changes

- Only simple changes: the same data is available...
- ...in a different structure...
  - Data patterns describe the new and the old structure:
    
    \[
    \begin{align*}
    \{\text{match, St, Len}\} &\mapsto \{\text{match, [{\text{decr}(\text{St}), \text{Len}]}}\}
    \\
    \text{nomatch} &\mapsto \text{nomatch}
    \end{align*}
    \]

- ...or in a slightly modified form
  - Compensations are provided by simple expressions:
    - \(\text{decr}(\text{Old} \mapsto \text{New}): \text{Old-1}\)
    - \(\text{decr}(\text{New} \mapsto \text{Old}): \text{New+1}\)

- These are sufficient to upgrade the \texttt{regexp} module calls
Prototype Implementation

- RefactorErl infrastructure is used
  - Semantic analysis
  - Syntax tree-based transformations
- Linear time and space complexity
  - Direct graph: about same size as the syntax tree
  - Reaching computation: breadth-first walk limited to the affected graph components
  - Usually a module is transformed in one step
Summary

Data structure refactoring for module interface migration

- Simple but powerful change descriptions
- Change propagation by data flow analysis