

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

Regex Upgrade Example

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

Regex Upgrade Example

```
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.
```

Regex Upgrade Example

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

Regex Upgrade Example

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

Regex Upgrade Example

```
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

```
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

```
find(Key, [{Key, Val}|_]) -> Val;
```


```
find(Key, [_|Tail]) -> find(Key, Tail).
```

```
f() -> find(a, [{a,1}]).
```

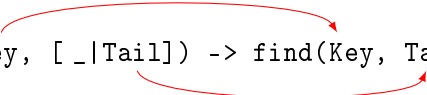
Data Flow Analysis

Direct edges: variables

```
find(Key, [{Key, Val}|_ ]) -> Val;
```

A red curved arrow points from the variable 'Key' in the function signature to the variable 'Val' in the return type.

```
find(Key, [_|Tail]) -> find(Key, Tail).
```

Two red curved arrows illustrate data flow: one from 'Tail' in the function argument to 'Tail' in the function call, and another from 'Tail' in the function call to 'Tail' in the function argument.

```
f() -> find(a, [{a,1}]).
```

Data Flow Analysis

Direct edges: function calls

```
find(Key, [{Key, Val}|_]) -> Val;  
find(Key, [_|Tail]) -> find(Key, Tail).  
f() -> find(a, [{a,1}]).
```

The diagram shows three function definitions. Red arrows indicate direct edges (calls) between the functions. Grey arrows indicate return values being passed back to callers.

- Red arrow from the second `find` call to the first `find` definition.
- Red arrow from the `f()` call to the second `find` definition.
- Red arrow from the `find` call inside `f()` to the second `find` definition.
- Grey arrow from the first `find` definition to the `find` call inside `f()`.
- Grey arrow from the second `find` definition to the `find` call inside the second `find` definition.

Data Flow Analysis

Direct edges: function calls

```
find(Key, [{Key, Val}|_]) -> Val;  
find(Key, [_|Tail]) -> find(Key, Tail).  
f() -> find(a, [{a,1}]).
```

The diagram illustrates data flow analysis for function calls. It shows three function definitions:

- `find(Key, [{Key, Val}|_]) -> Val;`
- `find(Key, [_|Tail]) -> find(Key, Tail).`
- `f() -> find(a, [{a,1}]).`

Red arrows indicate direct edges from function calls to their definitions:

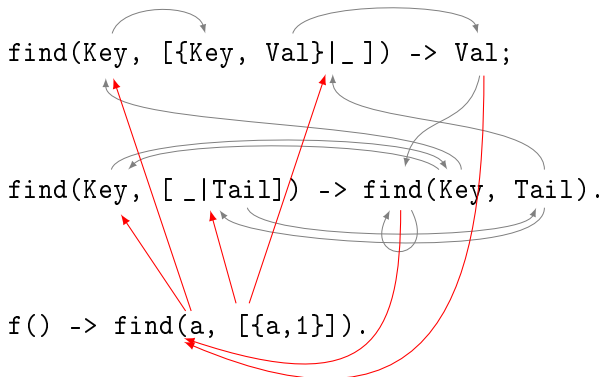
- From the `find` call in the second line to the `find` definition in the first line.
- From the `find` call in the second line to the `find` definition in the second line.
- From the `find` call in the third line to the `find` definition in the second line.

Grey arrows show the flow of arguments and return values:

- From the `Key` argument in the second line to the `Key` parameter in the first line.
- From the `[_|Tail]` argument in the second line to the `[_]` parameter in the first line.
- From the `Key` argument in the second line to the `Key` parameter in the second line.
- From the `[_|Tail]` argument in the second line to the `[_]` parameter in the second line.
- From the `find` call in the second line to the `find` call in the third line.
- From the `find` call in the third line to the `find` call in the second line.

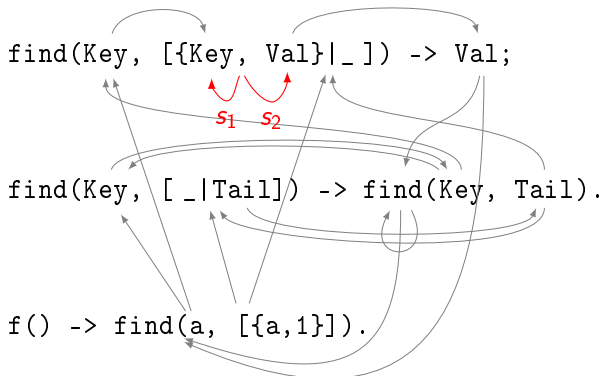
Data Flow Analysis

Direct edges: function calls



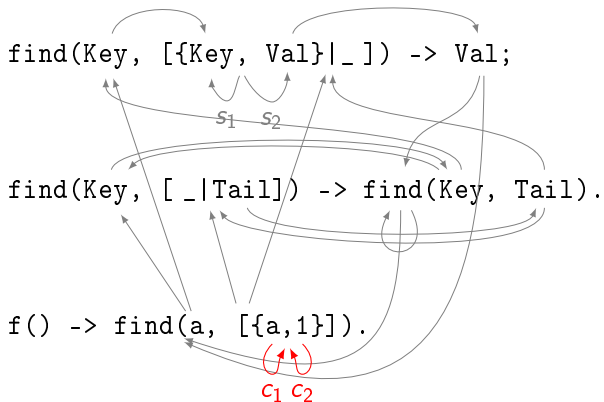
Data Flow Analysis

Direct edges: tuple selectors



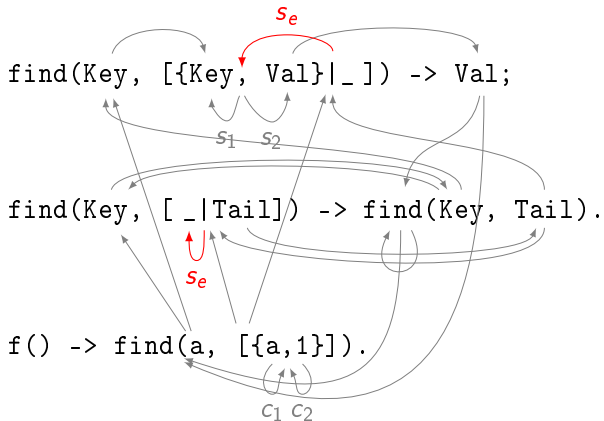
Data Flow Analysis

Direct edges: tuple constructors



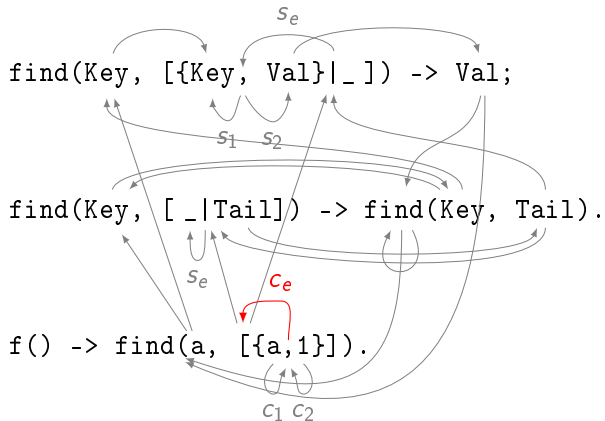
Data Flow Analysis

Direct edges: list element selectors



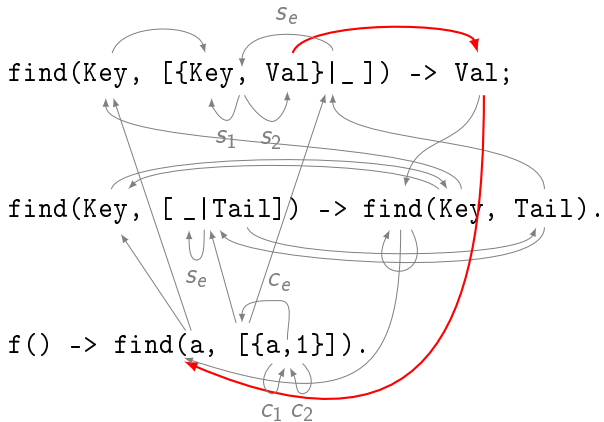
Data Flow Analysis

Direct edges: list constructors



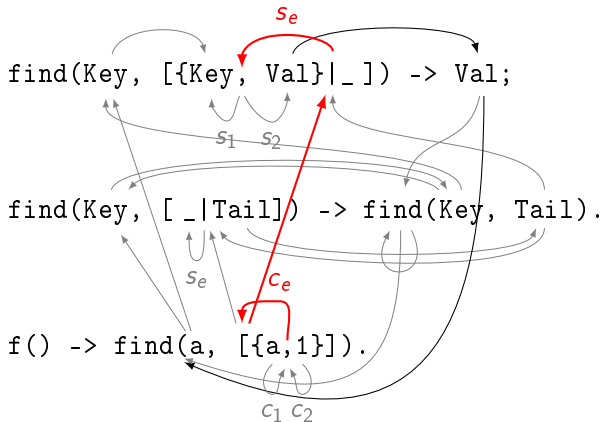
Data Flow Analysis

Reaching: transitivity



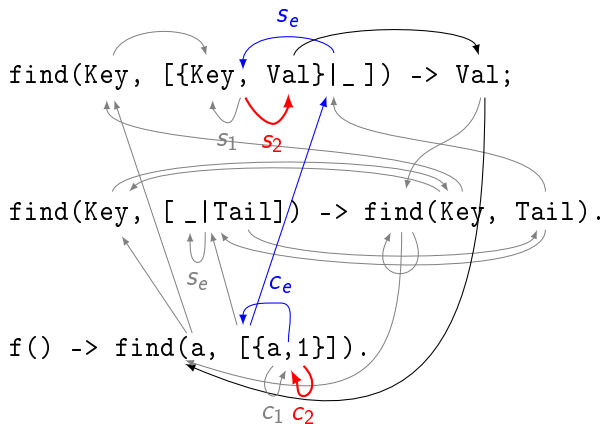
Data Flow Analysis

Reaching: list construction and selection



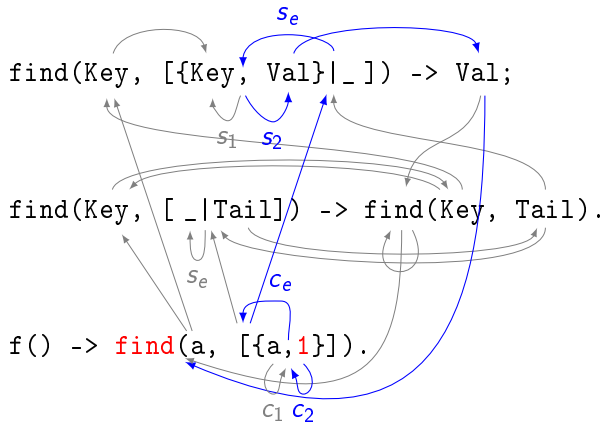
Data Flow Analysis

Reaching: tuple construction and selection



Data Flow Analysis

Propagation: finding the end of data flow paths



Describing Changes

- Only simple changes: the same data is available...
- ...in a different structure...
 - Data patterns describe the new and the old structure:
$$\begin{array}{l} \{\text{match}, \text{St}, \text{Len}\} \mapsto \{\text{match}, [\{\text{decr}(\text{St}), \text{Len}\}]\} \\ \text{nomatch} \qquad \qquad \mapsto \text{nomatch} \end{array}$$
- ...or in a slightly modified form
 - Compensations are provided by simple expressions:
$$\begin{array}{l} \text{decr}(\text{Old} \mapsto \text{New}): \text{Old}-1 \\ \text{decr}(\text{New} \mapsto \text{Old}): \text{New}+1 \end{array}$$
- These are sufficient to upgrade the `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