Automated Module Interface Upgrade *

László Lövei

Department of Programming Languages and Compilers
Eötvös Loránd University, Budapest, Hungary
lovei@inf.elte.hu

Abstract

During the lifetime of a software product the interface of some used library modules might change in such a way that the new interface is no longer compatible with the old one. This paper proposes a generic interface migration schema to automatically transform the software in the case of such an incompatible change. The solution is based on refactoring techniques and data flow analysis, and makes use of a formal description of the differences between the old and the new interfaces. The approach is illustrated with a real-life example.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—Restructuring, reverse engineering, and reengineering

General Terms Design, Languages

Keywords Erlang, module interface change, interface transformation, refactoring

1. Introduction

Library modules may evolve during the lifetime of a software product. The usual approach is to keep library interfaces compatible with previous versions, but sometimes it makes sense to develop a new, richer, but incompatible interface for a library, which provides enhanced functionality, and gets rid of obsolete features. This paper proposes an approach to transform the code of a software product when such an incompatible interface change occurs. Our assumption is that the new interface retains the functionality of the old one, but makes it available in a different way. We show that by defining a generic interface migration schema, refactoring techniques combined with data flow analysis can help the automatic adaptation of a software product to an upgraded library interface.

Our approach will be illustrated by the real-life case of migrating from the Erlang/OTP `regexp` library to the novel, Perl compatible regular expression library, `re`. One of the interesting points is that characters are indexed from 1 in `regexp`, but from 0 in `re`.

RefactorErl [**?**] is a refactoring tool for Erlang. It turns out that this tool provides a convenient infrastructure to develop program transformations supporting our proposed generic interface migration schema. RefactorErl represents Erlang programs as *program graphs*. A program graph contains lexical, syntactic and semantic nodes and edges. The AST of the represented program is a subgraph of the program graph. Technically, the program graph has nodes labelled with the non-terminals of the abstract syntax, nodes that correspond to tokens either explicitly present in the represented program or generated by macro expansion, and nodes that describe semantical information, such as the binding structure of variables. The edges of the program graph are directed, labelled, and for each node the outgoing syntactic edges are ordered (as in an AST). RefactorErl supports a powerful query language to collect information by traversing the program graph, syntax based transformations (transformation that manipulate the AST) and semantic analysis plug-ins that automatically restore the consistency of the semantic information in a program graph after a transformation.

Our proposal to the interface migration problem is the following. First the differences between the old and the new interface have to be described with *change descriptions* written in an Erlang-like language. Then the program to be transformed is analyzed – in this paper we assume that the relevant syntactic and semantic information such as the binding structure or the function call graph is already available, e.g. by using the facilities of RefactorErl. Our goal is to find those parts of the program where transformations are the best to perform, e.g. where the transformations result in the smallest degradation of readability of the code. This is achieved by finding (using data flow analysis) the expressions affected by calls to functions from the library to be upgraded, and determining the very last points in the data flow where the necessary transformations can be safely applied.

The rest of the paper is structured as follows. In Section 2 a motivating example is given (the upgrade of the regular expression library). Section 3 introduces an abstract model of Erlang in which the interface migration problem can be studied. Section 4 describes how the necessary data flow analysis proceeds. Section 5 explains the way transformations are performed. Section 6 sketches some implementation issues. Section 7 presents related work, and Section 8 concludes the paper.

2. Interface upgrade example

In the following, we will use a real-life motivating example. Erlang/OTP contains an old regular expression library, called `regexp`, which has been made obsolete by a new, Perl compatible regular expression library called `re`. The new library covers almost every functionality of the old one, but there are small incompatibilities between them. For example, characters are indexed starting with 1 in the old library, and starting with 0 in the new one.

Here we show three old interface functions, and explain the details of migrating them to the new interface. The rest of the old interface is either compatible with the new one or has the same issues that are shown here. For details, refer to [**?**].

* Supported by TECH_08_A2-SZOMIN08, ELTE IKKK, and Ericsson Hungary

Erlang '09, September 5, 2009, Edinburgh, Scotland, UK.

2.1 The *match* function

The `regexp:match` function finds the first, longest match of a regular expression in a string. A simple example how this function is used:

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

The function returns the starting index and the length of the match, or the atom `nomatch`. The same functionality is available using the `re:run` function, but its interface has a number of incompatibilities:

- By default, the return value contains the whole match and every subpattern match. This can be turned off with an option.
- The return value contains a list of match tuples, which always has a length of one after turning off subpattern matches.
- `regexp:match` starts indexing at one, but `re:run` starts indexing at zero.

These are small changes that can be compensated easily in the example by passing the necessary options to `re:run`, rewriting the pattern to the new structure, and adding 1 to the returned index:

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

Unfortunately, the return value of the `regexp:match` function can be used in other ways as well, which may require more complicated compensations. For example, the return value may be returned directly from a function, and in turn, used in many places:

```
word(Str) ->
   regexp:match(Str, "[a-zA-Z]+").

read_word() ->
   Line = read_line(),
   {match, S, L} = word(Line),
   string:substr(Line, S, L).

word_len(Str) ->
   {match, _, Len} = word(Str),
   Len.
```

In this example, the desired solution is to modify the pattern matching constructs used in functions `read_word` and `word_len`, as opposed to converting the return value of function `word`. This example involves only function calls, but generally data flow analysis has to be applied to find every code part that uses the return value of `regexp:match`.

Another possibility is that the return value is used in an expression that is not a pattern matching construct. There are some cases that can be handled in a specific way, for example `regexp:match(S,R) /= nomatch` can simply be rewritten to `re:run(S,R) /= nomatch`.

But in worse cases, the function call itself must be replaced with a `case` construct that restores the original return value. In general, this should be avoided as it leads to illegible code. For example, `tuple_to_list(regexp:match(S,R))` would look like this after such a transformation:

```
tuple_to_list(
  case re:run(Str, RE, [{capture,first}]) of
      {match, [{S, L}]} -> {match, S, L};
      nomatch -> nomatch
  end)
```

2.2 The *matches* function

The `regexp:matches` function finds all non-overlapping matches of a regular expression in a string. This functionality is provided by the same `re:run` function as previously, just the `global` option should also be passed to specify global matching. The basic problems and solutions are the same as in the previous case, but there is a complication: the return value contains a list that should be updated element-by-element. Also, in case of no match the old function returns an empty list, while the new one uses the `nomatch` atom.

A typical example of usage:

```
case regexp:matches(Str, RE) of
   {match, []} ->
      throw(no_match);
   {match, Matches} ->
      [string:substr(Str, S, L) ||
         {S, L} <- Matches]
end
```

The equivalent of this expression is easy to construct by hand, but automated transformation is much harder even in this simple example:

```
case re:run(Str, RE, [global,{capture,first}]) of
   nomatch ->
      throw(no_match);
   {match, Matches} ->
      [string:substr(Str, S+1, L) ||
         [{S, L}] <- Matches]
end
```

Data flow analysis is necessary to find the pattern `{S,L}` in the list comprehension that must be turned into a one-element list, and the usage of `S` that must be incremented. As a list is returned, which does not have a fixed length, a function call or a list comprehension is very likely to be used together with this function, so identifying where the list and its elements are used is essential here.

2.3 The *gsub* function

The `regexp:gsub` function substitutes every substring matching a regular expression in a string. The `re:replace` function can do the same (some options are needed), but this time the new return value contains less information than the old one: instead of a tuple, only the result of the substitution is returned, and the number of replacements provided by the old interface is completely missing from the new one. Fortunately it is rarely needed (that's why it has been removed), so a typical example is not too hard to migrate:

```
{ok, Result, _Count} = regexp:gsub(Str, RE, Repl)
```

If there are no references later to variable `_Count`, the same code with module `re`:

```
Result = re:replace(Str, RE, Repl,
                    [{return, list}, global])
```

If the missing value is used in the code, there is no generic solution for the migration, such a code must be rewritten by hand. Our task here is to detect this situation, and notify the user without actually changing the code.

V ::= variables (including _, the underscore pattern)
A ::= atoms
I ::= integers
K ::= A | I | other constants (e.g. strings, floats)
P ::= K | V | {P,...,P} | [P,...,P|P]
E ::= K | V | {E,...,E} | [E,...,E|E] | [E||P<-E] |
P = E | $E \circ E$ | (E) | E(E,...,E) |
case E of
P -> E,...,E;
⋮
P -> E,...,E
end |
fun
(P,...,P) -> E,...,E;
⋮
(P,...,P) -> E,...,E
end
F ::= A(P,...,P) -> E,...,E;
⋮
A(P,...,P) -> E,...,E.

Figure 1. The used Erlang syntax subset

3. Model of Erlang programs

In the following discussion we do not consider the whole Erlang language. For the sake of simplicity, we omit some of the language elements, but the framework described here is capable of supporting the full Erlang syntax. The static graph construction rules of the missing syntactic constructs are easily added to those in Fig. **??**, and no other additions are needed to support them.

The syntax of the Erlang subset that we use is defined in Fig. **??** is used. The simplifications are the following:

- Irrelevant language constructs are left out, for example, the results are completely independent of the module structure and the attributes of modules. Instead of a set of modules, we consider Erlang programs that consist of a set of named function definitions (F in Fig. **??**).
- There are missing expression types which can be handled in the same way as one of the presented expressions, like records, which are analogous to tuples indexed with field names.
- The syntax contains further simplifications in the remaining expression types, for example, there are no guard expressions, because they would not be handled differently that other expressions.
- Finally, there are language elements that could be considered relevant, but analysis of industrial code shows that they are so rarely used in conjunction with the problem stated here that it does not pay off to support them. These include processes, message passing, and exceptions, as all of them would require control flow analysis in addition to the data flow analysis presented here.

We assume that the code to be transformed is available as an abstract syntax tree. The nodes of the syntax tree represent one of the above rules, they are uniquely identified, and attached to the corresponding source code part. Transformations will be expressed as rewriting parts of the syntax tree by providing the replacement structure for some of the nodes. This model is not elaborated further here, RefactorErl [**?**] and Wrangler [**?**] is capable of doing such transformations.

3.1 Semantic information

Information not available directly from the abstract syntax tree is also necessary for the analyses and transformations presented here. The information is based on standard Erlang semantics [**?**], so the exact definition of these concepts is omitted here. The presentation is also independent of how this information is represented or computed.

Internal function calls: for every function call expression that uses a constant function name, we need to know if the program contains the definition of the referred function. In this case, we call this an *internal function call*. The semantic model must make the function definition accessible.

External function calls: function calls that use a constant function name, but there is no corresponding function definition in the program, are called *external function calls*. These include built in functions (some of which are handled specially) and functions defined in libraries, but program parts which should not be modified during the transformation fall into this category too.

Variable bindings: every variable in a program must have at least one occurrence that binds the variable, and it is probably used elsewhere in the code, although there may be more bindings, and there may be no usages. Bindings are always patterns, but some patterns only use the value of the variable; expressions may only use the variable. Variables have a name, but there may be different variables with the same name in different scopes; they must be distinguished. The needed semantic information is the set of bindings and usages for any given variable occurrence.

4. Data flow graph

Data flow analysis is used to find expressions and patterns that are affected by changing a particular data in the program. The goal of data flow analysis is to inspect possible paths where data can travel during the execution of the program, and follow these paths to see what expressions can a particular data reach. The possible data flow paths are represented by a *data flow graph*.

The nodes of the graph ($n \in \mathcal{N} = \mathcal{E} \cup \mathcal{P}$) are the expressions ($e \in \mathcal{E}$) and patterns ($p \in \mathcal{P}$) of the program. Its directed edges represent single steps of data transfer. There are different kinds of data transfers, they are identified by edge labels. We use the notation $n_1 \xrightarrow{l} n_2$ for an edge from node n_1 to node n_2 with label l.

4.1 Direct flow information

Data flow graph edges represent direct data flow, later on we will derive information from these edges by traversing the direct graph. Data flow usually means copying data; when operations are used on data, it cannot be followed in general. In our interface upgrade problem when a modified data is used in a non-copying operation, we must convert it to the original value to preserve the original meaning of the program. The only exception is packing data into a compound term and later unpacking it: such an operation preserves the original data, so we can think of it as simple copying.

Based on these principles, the following kinds of edges are used in the graph to distinguish between different types of data flow steps:

Flow edges: $n_1 \xrightarrow{f} n_2$ means that the result of n_2 can be a copy of the result of n_1.

Constructor edges: $n_1 \xrightarrow{c_i} n_2$ means that the result of n_2 can be a compound value that contains n_1 at element i. In case of tuples, element labels are natural numbers, meaning the i^{th}

	Expressions	Direct graph edges
(a)	p is a binding n is a usage of the same variable	$p \xrightarrow{f} n$
(b)	e_0: $p = e$	$e \xrightarrow{f} p$ $e \xrightarrow{f} e_0$
(c)	p_0: $p_1 = p_2$	$p_0 \xrightarrow{f} p_1$ $p_0 \xrightarrow{f} p_2$
(d)	e_0: $e_1 \circ e_2$	$e_1 \xrightarrow{d} e_0$ $e_2 \xrightarrow{d} e_0$
(e)	e_0: (e)	$e \xrightarrow{f} e_0$
(f)	e_0: $\{e_1, \ldots, e_n\}$	$e_1 \xrightarrow{c_1} e_0, \ldots, e_n \xrightarrow{c_n} e_0$
(g)	p_0: $\{p_1, \ldots, p_n\}$	$p_0 \xrightarrow{s_1} p_1, \ldots, p_0 \xrightarrow{s_n} p_n$
(h)	e_0: $[e_1, \ldots, e_n \vert e_{n+1}]$	$e_1 \xrightarrow{c_e} e_0, \ldots, e_n \xrightarrow{c_e} e_0$ $e_{n+1} \xrightarrow{f} e_0$
(i)	e_0: $[e_1 \vert\vert p \leftarrow e_2]$	$e_1 \xrightarrow{c_e} e_0$ $e_2 \xrightarrow{s_e} p$
(j)	p_0: $[p_1, \ldots, p_n \vert p_{n+1}]$	$p_0 \xrightarrow{s_e} p_1, \ldots, p_0 \xrightarrow{s_e} p_n$ $p_0 \xrightarrow{f} p_{n+1}$
(k)	e_0: $\mathsf{hd}(e_1)$	$e_0 \xrightarrow{s_e} e_1$

	Expressions	Direct graph edges
(l)	e_0: $\mathsf{tl}(e_1)$	$e_0 \xrightarrow{f} e_1$
(m)	I is constant, e_0: $\mathsf{element}(I, e_1)$	$e_0 \xrightarrow{s_I} e_1$
(n)	e_0: $\mathsf{case}\ e\ \mathsf{of}$ $p_1 \rightarrow e_1^1, \ldots, e_{l_1}^1;$ $\vdots$ $p_n \rightarrow e_1^n, \ldots, e_{l_n}^n;$ end	$e \xrightarrow{f} p_1, \ldots, e \xrightarrow{f} p_n$ $e_{l_1}^1 \xrightarrow{f} e_0, \ldots, e_{l_n}^n \xrightarrow{f} e_0$
(o)	e_0: $\mathsf{f}(e_1, \ldots, e_n)$ $\mathsf{f/n}$: $\mathsf{f}(p_1^1, \ldots, p_n^1) \rightarrow$ $e_1^1, \ldots, e_{l_1}^1;$ $\vdots$ $\mathsf{f}(p_1^m, \ldots, p_n^m) \rightarrow$ $e_1^m, \ldots, e_{l_m}^m.$	$e_1 \xrightarrow{f} p_1^1, \ldots, e_1 \xrightarrow{f} p_1^m$ $\vdots$ $e_n \xrightarrow{f} p_n^1, \ldots, e_n \xrightarrow{f} p_n^m$ $e_{l_1}^1 \xrightarrow{f} e_0, \ldots, e_{l_m}^m \xrightarrow{f} e_0$
(p)	e_0: $e(e_1, \ldots, e_n)$ e is not constant, or $\mathsf{e/n}$ undefined	$e_1 \xrightarrow{d} e_0, \ldots, e_n \xrightarrow{d} e_0$

Figure 2. Static data flow edge generation rules

element of the tuple. In case of lists, element label e is used for list elements (we cannot usually track their indexes).

Selector edges: $n_1 \xrightarrow{s_i} n_2$ means that the result of n_2 can be element i of n_1. Element labels have the same meaning as in constructor edges.

Dependency edges: $n_1 \xrightarrow{d} n_2$ means that the result of n_2 can directly depend on the value of n_1. This edge label is used when data flow cannot be followed, but data usage must be represented in the graph.

4.2 Static graph building rules

Most edges of the full data flow graph can be constructed based on the syntax tree and static semantic information. Every expression has a set of associated data flow edges, and the static graph is constructed by taking all of the edges generated by the expressions of the program.

In the following, graph edge construction rules are explained for every expression type. The summary of these rules is shown in Fig. **??**.

Variable occurrences. Variable patterns in Erlang may bind a value to the variable, or may use the value; variable expressions always use the value. The value of a variable never changes during its life. This means that the only kind of data flow through a variable is copying the value from its binding occurrences to its usages. This is represented by creating flow edges from every binding occurrence to every usage occurrence (Fig. **??**.**??**).

Expression results. There are expressions that return some of their subexpression's results without modifying it. An example is the `case` expression, which executes one of its clauses, and returns the result of the last expression from that clause. This is represented by flow edges from the last expressions of the clauses to the `case` expression itself (Fig. **??**.**??**). Similar flow edges appear for match expressions and parentheses (Figs. **??**.**??** and **??**.**??**), except that there are no multiple clauses.

An alias pattern, which has the same syntax as a match expression, is a special construction that flows data in the opposite direction. Such a pattern means that two patterns are matched on the same value at the same time, which is represented by flow edges from the alias pattern to its subpatterns (Fig. **??**.**??**).

Pattern matching expressions. During pattern matching, data is compared with a pattern, and free variables in the pattern are bound to values. Data is simply copied, which is represented by flow edges from the matched expression to every matched pattern. In case of simple match expressions there is only a single pattern (Fig. **??**.**??**), but `case` expressions use multiple patterns, and each of them gets a flow edge (Fig. **??**.**??**).

Compound data handling. Compound data structures (tuples and lists) preserve data, so putting a value in a tuple, copying the tuple, selecting the value from the tuple is the same as copying the value. To detect these copies, we use constructor and selector edges to connect the flow of embedded data and the flow of its container.

Constructor and selector edges are indexed to make a note of the position where the embedded data is stored. Only constructors and selectors with the same index are considered to be part of the same data flow. Tuple indexes are natural numbers, which enables a precise tracking of tuple elements. Typical use of lists makes tracking of list element indexes useless, so in this case only list elements are distinguished. The "tail" of a list do not need special handling, as almost every element of a list appears in its tail; approximating this by representing a list tail as a copy of the original list is good enough for our purposes.

Patterns are selectors, so they generate selector edges to their subexpressions (Figs. **??**.**??** and **??**.**??**). Expressions with the same syntax are constructor expressions with constructor edges

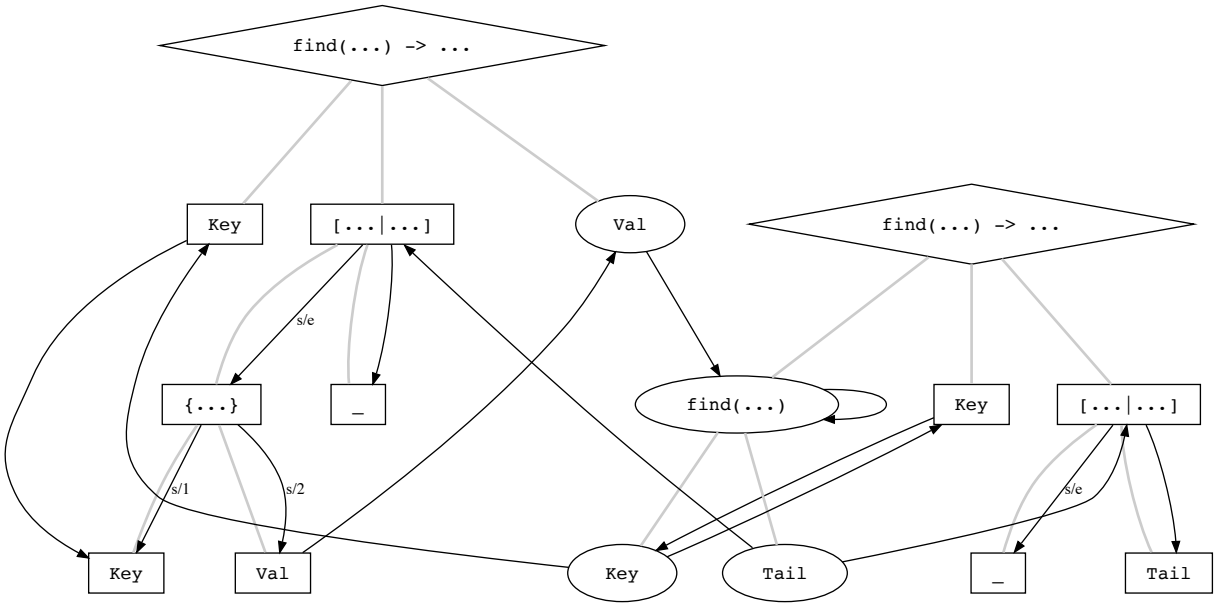

Figure 3. Data flow graph example.

(Figs. **??**.**??** and **??**.**??**). Selector expressions are built-in functions, they generate selector edges (Figs. **??**.**??**, **??**.**??**, and **??**.**??**).

List comprehensions are also selecting and constructing lists. The expression that provides the list values has a list element constructor edge to the whole list comprehension expression. The generator part, which means a pattern match for every element on its list expressions, produces a list element selector edge from the generator list expression to the generator pattern (Fig. **??**.**??**).

Function calls. We have two essentially distinct case of function calls. First, calls to external functions cannot be represented in the flow graph, but they must be taken into account: such a construct means that data can get out of our hands, and it can be used in program parts that we cannot transform. We represent this situation by dependency edges from the function arguments to the function call expression (Fig. **??**.**??**). Such edges signal a dead end, where data flow continues, but cannot be followed (as opposed to a node with no edges starting from it, which means there is no further data flow from that node). The same representation is used for expressions (usually operators) that do some computation with their arguments: data flow continues, but in an undefined way (Fig. **??**.**??**).

On the other hand, functions with a known definition should use that definition in the flow graph. We don't even want to know that there is a function call involved, so the representation is very similar to a `case` expression, except that there are multiple arguments, therefore multiple pattern matches. Every clause of the function is considered, arguments from the function call are matched on the patterns in the function definition, and the possible results of the function are returned to the call expression (Fig. **??**.**??**).

Example. As an illustration of these graph building rules, the data flow graph of the following simple function is presented in Fig. **??**:

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

The syntax tree of the function is represented by grey lines. The diamond-shaped nodes are the function's clauses, these are not part of the data flow graph. Boxes are patterns, ovals are expressions, and the black arrows are data flow edges. Most of the edges are labelled with `f`, these labels are omitted from the drawing to make it clearer.

Rules applied to create this graph include variable usage rules, element selection for list and tuple patterns, and a function call rule. The last one is responsible for the edges between the two syntax tree parts and the looping edges that appear because of the recursive call.

4.3 Derived flow information

The static data flow graph represents only a part of every possible data flow in a program, because there are many dynamic constructs in Erlang. However, many of these constructs' data flow information can be approximated starting from the static graph.

Here we show the basic techniques to derive useful data flow information from the direct data flow graph. This information can be used to approximate the behaviour of some dynamic language constructs. Furthermore, the same techniques will be used to define the interface upgrade transformation.

4.3.1 Derivation techniques

The so-called "techniques" of this section are in fact simple sets or relations which are useful, and easy to calculate based on the flow graph.

Reaching. The most important relation we will use describes which nodes are connected by multiple-step data flow paths. We say that the value of node n_1 *reaches* node n_2, if during the evaluation of the program a value associated with n_1 may be passed to n_2. The notation $n_1 \rightsquigarrow n_2$ will be used for this concept.

The $\rightsquigarrow$ relation relation can be computed from the direct flow graph by following paths set out by edges. Simple flow edges are simply followed; dependency edges break the data flow. Constructor and selector edges are handled specially: when a data is packed into a structure, the compound data is tracked, and the corresponding unpacked nodes (pointed by selector edges with the same index as the constructor edge) are the next steps on the path. This com-

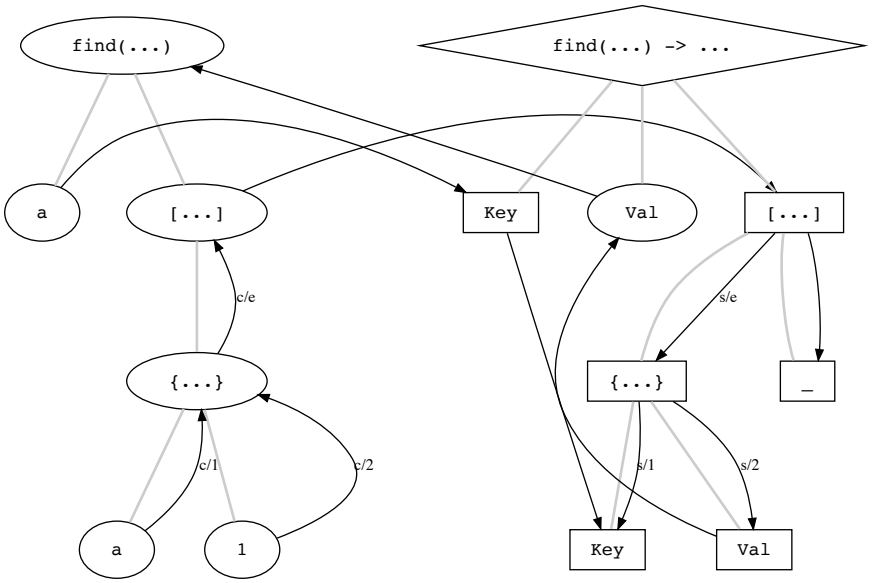

Figure 4. Data flow graph example.

putation involves graph traversals on f-labelled edges, executed recursively on c_i-labelled edges.

The explanation above can be formulated by defining $\rightsquigarrow$ as the minimal relation that satisfies the following rules:

1. $n \rightsquigarrow n$
2. $n_1 \rightsquigarrow n_2 \wedge n_2 \xrightarrow{f} n_3 \Rightarrow n_1 \rightsquigarrow n_3$
3. $n_1 \xrightarrow{c_i} n_2 \wedge n_2 \rightsquigarrow n_3 \wedge n_3 \xrightarrow{s_i} n_4 \Rightarrow n_1 \rightsquigarrow n_4$

Example. Fig. **??** shows the graph of the first clause of the previous example's `find` function and expression `find(a, [{a, 1}])`. Using the above reaching definition, we can deduce that the value of expression `1` can be returned from the `find` function call:

1. $[\{\mathsf{a}, 1\}] \xrightarrow{f} [\{\mathsf{Key}, \mathsf{Val}\}, \_] \Rightarrow [\{\mathsf{a}, 1\}] \rightsquigarrow [\{\mathsf{Key}, \mathsf{Val}\}, \_]$
2. $\{\mathsf{a}, 1\} \xrightarrow{c_e} [\{\mathsf{a}, 1\}] \rightsquigarrow [\{\mathsf{Key}, \mathsf{Val}\}, \_] \xrightarrow{s_e} \{\mathsf{Key}, \mathsf{Val}\}$
   $\Rightarrow \{\mathsf{a}, 1\} \rightsquigarrow \{\mathsf{Key}, \mathsf{Val}\}$
3. $1 \xrightarrow{c_2} \{\mathsf{a}, 1\} \rightsquigarrow \{\mathsf{Key}, \mathsf{Val}^p\} \xrightarrow{s_2} \mathsf{Val}^p \Rightarrow 1 \rightsquigarrow \mathsf{Val}^p$
4. $1 \rightsquigarrow \mathsf{Val}^p \xrightarrow{f} \mathsf{Val}^e \Rightarrow 1 \rightsquigarrow \mathsf{Val}^e$
5. $1 \rightsquigarrow \mathsf{Val}^e \xrightarrow{f} \mathsf{find}(...) \Rightarrow 1 \rightsquigarrow \mathsf{find}(...)$

One-step flow information. The following simple notation will be useful in other definitions. Starting from a given node, they describe the next (or previous) nodes directly accessible with a given edge type in the flow graph:

$$\begin{aligned} \mathrm{next}(l, n) &= \{n_n \mid n \xrightarrow{l} n_n\} \\ \mathrm{prev}(l, n) &= \{n_p \mid n_p \xrightarrow{l} n\} \end{aligned}$$

The input set of a node contains its direct predecessors on copying flow paths, taking data constructions into account. It's like taking one step backward on the $\rightsquigarrow$ relation.

$$\begin{aligned} \mathrm{input}(n) = \; & \mathrm{prev}(f, n) \cup \\ & \{n_1 \mid n_1 \xrightarrow{c_i} n_2 \wedge n_2 \rightsquigarrow n_3 \wedge n_3 \xrightarrow{s_i} n\} \end{aligned}$$

4.3.2 Dynamic graph building

Data flow graph building rules introduced so far have all been static rules, which means they don't depend on run time data values, only on syntactic structures which are available at compile time. While our experiences show that these rules cover most of the interface upgrading situations which appear in practise, it should be pointed out that this data flow graph can be expanded to support some dynamic construct as well.

Our goal with this work is to create a simple framework for data flow analysis to support refactoring, and introducing sophisticated control flow analysis is not desired here. The static data flow graph itself contains enough information to make data flow analysis of some dynamic constructs possible.

Constant propagation. The static direct data flow graph can be iteratively extended with direct, but dynamic flow edges. Dynamic constructs use run-time values to control data flow, but in some cases these values can be derived from constants in the code.

The set of possible origins of an expression's value can be computed using the data flow graph, as we have all the static data flow paths:

$$\mathrm{source}(n) = \{n_1 \mid n_1 \rightsquigarrow n \wedge \nexists n_2 : n_2 \neq n_1 \wedge n_2 \rightsquigarrow n_1\}$$

This set may contain patterns, which means the information is not complete, or expressions that compute the value in any way. When $\mathrm{source}(n)$ contains only constant expressions, we have the finite set of the possible values of n.

Dynamic control flow. A dynamic function call is a function application where the function name is not an atom constant. In this case, the expression in place of the name must evaluate to a function object. If we have a dynamic function call $e_0(e_1, \ldots, e_n)$, then $\mathrm{source}(e_0)$ contains the function expressions that provide that function objects that may be called. This means that the same graph edges can be entered into the graph as in case of a static function call, using the function expression instead of the function definition. Extending the direct graph with these edges gives the same result as 0CFA [**?**].

Dynamic indexing. If we have the expression $\mathsf{element}(e_1, e_2)$ where e_1 is not an integer constant, the possible index values may be obtained from constant propagation. If $\mathrm{source}(e_1)$ contains only integer constant expressions, these integers can be used in the same way as in case of directly used integer constants.

5. Transformation

The goal of the transformation is to change calls of old interface functions to new ones. The interface change affects how the function is called and what is returned from the call.

Our experience shows that changes in function arguments are usually less complex. This is probably because the arguments themselves are usually less complex than the return values. However, if function arguments had to be transformed, the same approach could be used for them as for the return values. Still, in our main examples the only changes are some extra constant arguments.

In the following, we concentrate on the compensation of the changes in the return value of the function. The goal of the compensation is to modify existing code that expects the old return value to work with the new return value. The trivial compensation is to insert a run time conversion on the return value of the new function that produces the old value, and the unmodified old code will work. However, the approach presented here can provide a much better result.

5.1 Compensation spreading by data flow tracking

The run time conversion of the modified function's return value decomposes the returned data structure, and builds the old structure from the components. The old value is then probably decomposed again, and the same components are used somewhere. The idea is to skip the data structure conversion, and update the decomposition of the old value to work with the new structure.

To do this, we need to find code parts where the modified data can flow to. Ideally, data is copied through some expressions, and then a pattern is matched on it, so only the pattern should be updated. However, there are constructs that not only copy the modified data, but use it in an uncontrolled way. When an expression's value may be used in such a situation, compensation must be done on that expression at run time.

In the following, we formalise how to calculate the set of expressions that simply copy the modified data and should be left intact, and the set of expressions and patterns that terminate these data flow paths and should be transformed.

1. Let's start from a set of source graph nodes ($\mathcal{S}$). This set contains the function calls which return a modified value, or other data flow path start nodes which contain modified data. First, calculate the set of nodes that may get their value from this set:

$$\mathrm{reach}(\mathcal{S}) = \{n \in \mathcal{N} \mid \mathcal{S} \rightsquigarrow n\}$$

2. Next, we must leave out every node from this set that may get its value from somewhere else, because it means that the node may receive values from unmodified sources, so it must not be transformed:

$$\mathrm{strict}(\mathcal{S}) = \mathcal{S} \cup \{n \in \mathrm{reach}(\mathcal{S}) \mid \mathrm{input}(n) \subseteq \mathrm{reach}(\mathcal{S}) \wedge \mathrm{prev}(d, n) = \emptyset\}$$

   Element of $\mathcal{S}$ are explicitly included in this set, because they are the starting point of changes, so they are known to be modified, but the other conditions obviously do not hold for them.

3. We say that a node is *unsafe* if its value is used in an uncontrolled way, that is, copied to a node outside $\mathrm{strict}(\mathcal{S})$, put into a compound value, or used in an unspecified way. Nodes that safely copy their value are the following:

$$\mathrm{safe}(\mathcal{S}) = \{n \in \mathrm{strict}(\mathcal{S}) \mid \mathrm{next}(f, n) \subseteq \mathrm{strict}(\mathcal{S}) \wedge \mathrm{next}(d, n) = \emptyset \wedge \mathrm{next}(c_i, n) = \emptyset\}$$

4. Now we have to follow those flow paths which safely copy a modified value. These paths are terminated by non-safe nodes. We say that nodes found on these paths are *safely reachable*. Nodes that are safely reachable from set $\mathcal{S}$ are denoted with $\mathcal{S} \rightsquigarrow_s n$, and this is the minimal set that satisfies the following conditions:

   (a) $n \in \mathcal{S} \wedge n \in \mathrm{strict}(\mathcal{S}) \Rightarrow \mathcal{S} \rightsquigarrow_s n$

   (b) $\mathcal{S} \rightsquigarrow_s n_1 \wedge n_1 \in \mathrm{safe}(\mathcal{S}) \wedge n_1 \xrightarrow{f} n_2 \Rightarrow \mathcal{S} \rightsquigarrow_s n_2$

   Note that $\mathcal{S} \rightsquigarrow_s n$ implies $n \in \mathrm{strict}(\mathcal{S})$.

5. Finally we have arrived at the scope of transformations. We modify the code so that safely reachable nodes will get the new data, and non-safe nodes will return the original data. This means that safe expressions, that only copy their value, remain unchanged; safely reachable patterns must be updated to reflect the structure of the new data; and finally, the return value of safely reachable but non-safe expressions must be converted at run time to the old form. Formally, the nodes to be transformed:

$$\begin{aligned} \mathrm{trf}_p(\mathcal{S}) &= \{p \in \mathcal{P} \mid \mathcal{S} \rightsquigarrow_s p\} \\ \mathrm{trf}_e(\mathcal{S}) &= \{e \in \mathcal{E} \mid \mathcal{S} \rightsquigarrow_s e, e \notin \mathrm{safe}(\mathcal{S})\} \end{aligned}$$

Example. Let's have a look at how these concepts work on real code. We will use the `find` function shown on Fig. **??** and the function call show on Fig. **??**, and a simple new function:

```
find(Key, [{Key, Val}|_]) -> Val;
find(Key, [_|Tail])       -> find(Key, Tail).
f()    -> find(a, [{a, 1}]).
dbl(X) -> 2*X.
```

As we have seen before, the value of expression `1` reaches the `find` function call in `f()`. If you check the conditions, this call is an element of $\mathrm{strict}(\mathtt{1})$, because it cannot get its value from anywhere else; and it is in $\mathrm{safe}(\mathtt{1})$, because its value is not used in an uncontrolled way (in fact, it's not used at all in this code).

Let's extend this code with the expression `dbl(f())`. Now the value of `1` can reach the call to `f()`, and variable `X` in `dbl` (both the pattern and the expression). These are elements of the $\mathrm{strict}(\mathtt{1})$ set, but expression `X` in the body of `dbl` is unsafe: it is used in expression `2*X` in an uncontrolled way. This is represented by edge $\mathtt{X} \xrightarrow{d} 2 * \mathtt{X}$ in the graph, therefore $\mathrm{next}(d, \mathtt{X})$ is not empty, which violates the conditions of safe. In this case, $\mathrm{trf}_p(\mathtt{1})$ contains patterns `Val` in `find` and `X` in `dbl`, and $\mathrm{trf}_e(\mathtt{1})$ contains expression `X`.

If there is another call to `dbl`, e.g. we extend the code with `dbl(3)`, the situation changes: $\mathrm{input}(\mathtt{X})$ (for pattern `X` in `dbl`) now includes expression `3`, so `X` is not in the strict set anymore. This means that `f()` in `dbl(f())` is not safe, because its value is used in a non-strict expression, so it will become the only element of $\mathrm{trf}_e(\mathtt{1})$, and $\mathrm{trf}_p(\mathtt{1})$ now excludes `X` (as `X` in `dbl` is not safely reachable anymore).

5.2 Change descriptions

The goal of this section is to provide a formal description of data changes and a mechanism to reflect these changes in affected code. Changes are specified by a set of *change descriptions*. A change description can be applied to an expression or a pattern that works with the old data, and it results in two things: a new expression or pattern that works with the new data, and a (possibly empty) set of

$P^c ::= K \mid V \mid \{P^c, \ldots, P^c\} \mid \texttt{[}P^c, \ldots, P^c \texttt{|} P^c\texttt{]}$
$E^c ::= K \mid V \mid \{E^c, \ldots, E^c\} \mid \texttt{[}E^c, \ldots, E^c \texttt{|} E^c\texttt{]} \mid$
$A(V) \mid \texttt{map(}A\texttt{,}V\texttt{)}$

Figure 5. Syntax of change patterns and expressions

induced transformations. Descriptions sometimes need to refer to each other, so each of them has a unique name.

Applying changes to a pattern means rewriting the pattern itself. This approach puts a restriction on possible changes, but changing only patterns leads to a much better result than changing expressions. This is because applying a change to an expression means doing run time data conversion, which is usually less readable and possibly slower.

Complete transformations are specified by a change description cd and a set $\mathcal{S}$ of source nodes. The transformation is executed by applying cd on $\mathrm{trf}_e(\mathcal{S})$ and $\mathrm{trf}_p(\mathcal{S})$; this may induce a set of other transformations on nodes. The source nodes of the same transformations are joined together, and the transformation is applied on the joined set.

5.2.1 Structural change descriptions

Changes in the data structures returned by a function can be conveniently handled through the pattern matching mechanism. When the changed data is used in a pattern matching expression, the pattern can usually be updated; when a compensating expression must be generated, it can be a `case` expression that decomposes the new data, and constructs the old one from the pieces.

Structural changes can be described quite naturally by providing the conversion from the old data structure to the new one, using Erlang patterns and data constructors, just as if we would implement a run time converter function. We will shortly see that such a description can be used to generate every kind of compensation that we need during the application of a change description.

A structural change description is formulated by a set of change patterns and change expressions (see Fig. **??**). Change patterns are usual Erlang patterns, change expressions may additionally contain function call expressions (these refer to other change descriptions).

There are some restrictions on variables. We will use these descriptions to generate conversions between the old and new structures in both ways, so every variable must occur both in the change pattern and in the change expression exactly once. When a piece of data cannot be mapped to anything on the other side, it should be marked with the underscore pattern.

Matching patterns. When transforming a pattern, we need to find out which change patterns to apply. This decision is made based on the concept of *matching* between patterns. We use $\mathrm{subst}(p)$ to denote the set of substitution functions that map the free variables of pattern p to arbitrary patterns. The application of a substitution to p means replacing the variables in p with their mappings.

We say that pattern p_1 matches pattern p_2 when there is a substitution $s \in \mathrm{subst}(p_2)$ that satisfies $s(p_2) = p_1$. In this case, s maps subpatterns of p_1 to the free variables of p_2.

Example. A very simple interface change is migrating code that uses the `gb_trees` module to represent dictionaries with balanced trees to use the `dict` module instead, which represent dictionaries as hash tables. The changes in the return value of the `lookup` function are described this way:

$$\{\texttt{value}, \texttt{V}\} \mapsto \{\texttt{ok}, \texttt{V}\}, \quad \texttt{none} \mapsto \texttt{error}$$

Substitutions of the first change pattern, elements of the substitution set $\mathrm{subst}(\{\texttt{value}, \texttt{V}\})$, map `V` to patterns. Examples of patterns that match `{value, V}` are `{value, X}`, where the substitution is $\texttt{V} \mapsto \texttt{X}$, and `{value, [1,2]}`, where the substitution is $\texttt{V} \mapsto [\texttt{1}, \texttt{2}]$.

Pattern transformation. When pattern p matches a change pattern cp, its transformation is straightforward. The new pattern for the data is given in the corresponding change expression ce, and we have the substitution function $s \in \mathrm{subst}(cp)$ so that $p = s(cp)$. All we have to do is apply the substitution s to ce, and replace the original pattern p with the result, $s(ce)$.

Applying a substitution to a change expression is similar to change patterns, that is, we replace the variables with their mappings in s. For example, let's transform the following code using the change description from the previous example:

```
case lookup(Key, Store) of
   {value, 0} -> inf;
   {value, N} -> 1/N;
   none       -> 0
end
```

Let n be the `lookup` function call node. $\mathrm{trf}_e(\{n\})$ is empty, $\mathrm{trf}_p(\{n\})$ contains the patterns of the `case` expression. Each of these pattern match a change pattern:

- `{value, 0}` matches `{value, V}`, the substitution is $\texttt{V} \mapsto \texttt{0}$. The result of applying this substitution to the corresponding change expression `{ok, V}` is `{ok, 0}`.
- `{value, N}` is similar, it should be changed to `{ok, N}`.
- `none` is exactly the same as the second change pattern, so it should be replaced with `error`.

After replacing the patterns (and the `lookup` function call), the result is the following:

```
case find(Key, Store) of
   {ok, 0} -> inf;
   {ok, N} -> 1/N;
   error   -> 0
end
```

Change description references. There is a change expression construct that has to be dealt with: references to other change descriptions of the form $\texttt{name}(v)$. This reference is substituted just as if only the variable was there, but this construct induces a new transformation. As we noted earlier, $s(v)$ refers to a subpattern of p, which is substituted into the place of the reference, and the induced transformation is applying the referred change description on the graph node of this subpattern.

For example, look at the following change description, where `lookup` is the name of the previous example's change description:

$$\{\texttt{A}, \texttt{B}\} \mapsto \{\texttt{lookup(B)}, \texttt{A}\}$$

Let's transform the expression

```
{X, {value, Y}} = f()
```

by changing the return value of `f()` with the above change description. $\mathrm{trf}_e(\{\texttt{f()}\})$ is empty, $\mathrm{trf}_p(\{\texttt{f()}\})$ contains only the pattern of the match expression, which matches the change pattern $\{\texttt{A}, \texttt{B}\}$ using the substitution $\texttt{A} \mapsto \texttt{X}$, $\texttt{B} \mapsto \{\texttt{value}, \texttt{Y}\}$.

The first step is replacing the pattern with the substituted change expression, ignoring the `lookup` reference:

```
{{value, Y}, X} = f()
```

The second step is applying the induced transformation, which is `lookup` on `{value, Y}`. Repeating the same procedure as previously, the final result is:

```
{{ok, Y}, X} = f()
```

Generalised patterns. When a pattern does not match any of the change patterns, it is much harder to find a solution. Usually this means that the set of change descriptions is incomplete, in that case a warning should be given about the pattern that could not be transformed.

There is an exception: when a change pattern matches the pattern, there is a possibility for compensation. Consider the following change description: {ok, {A, B}} ↦ {ok, A, B}. How to transform the pattern in the following code?

```
{ok, Tup} = read(),
process(Tup).
```

In this case, the change pattern matches {ok, Tup}, the substitution is Tup ↦ {A,B}. A solution is to replace the pattern with the change expression, and replace the occurrences of the change pattern's variables with their substitution:

```
{ok, A, B} = read(),
process({A,B}).
```

When a change description reference occurs in the change expressions, the generated replacements should contain the variable compensated by the referred change description (as an expression). Note that this solution generates new variables in the program, which may introduce name clashes that must be resolved by renaming the variables in the change expression.

Multiple matches. Consider the following change description:

$$\{\texttt{ok}, []\} \mapsto \texttt{none}, \quad \{\texttt{ok}, \texttt{Lst}\} \mapsto \texttt{Lst}$$

In this situation the old, homogeneous data structure is replaced by at least two different constructs. This means that some of the patterns used in the old code cannot be used in the new code, because the same pattern cannot describe the various new structures.

In this case our approach is to duplicate the code that contains the pattern. In a `case` expression, problematic clauses are duplicated; in case of match expressions, the expression is turned into a `case` expression with one clause, which is then duplicated.

This situation can be recognised by a pattern that matches multiple change patterns. Every matching change pattern should get its own duplicate of the original code, and each duplicate is transformed according to one of the change expressions.

For example, let's transform the following expression by changing the return value of `f()` according to change description above:

```
case f() of
   {ok, L} -> length(L);
   error   -> 0
end
```

There are two possible ways to transform the first pattern: it is a generalisation of the first change pattern, and it matches the second change pattern. First the corresponding clause is duplicated:

```
case f() of
   {ok, L} -> length(L);
   {ok, L} -> length(L);
   error   -> 0
end
```

Then the first duplicate is transformed using the first change expression, and the second duplicate using the second change expression:

```
case f() of
   none  -> length([]);
   L     -> length(L);
   error -> 0
end
```

Transforming expressions. Expression transformation means that the return value of the expression should be converted at run time from the new structure to the original. This can easily be done by putting the expression into a `case` construct which uses patterns to decompose the return value according to the new structure, and rebuild the old structure using the components.

The patterns of the `case` expression should be generated from the change expressions. This means leaving the change description references out, only leaving their variables in the pattern; these variables will be converted at their usage places.

The results for the patterns are generated from the change patterns (these describe the original structure). The patterns are simply copied, except variables that have a change description reference in the change expressions: these are converted as expressions by the referred change description.

Note that the generated expression contains new variables, which may clash with existing variables names in the program; in this case, they should be renamed consistently.

Example. For an expression e, the following converter expression is generated from the `lookup` change description:

```
case e of
   {value, V} -> {ok, V};
   none       -> error
end
```

List element handling. There is a special change description reference with syntax `map(cd, Var)`, where `cd` is a change description name. We use this syntax to denote an element-wise application of a change description on a list. The meaning of this operator is simple, but cannot be described by the structural change elements introduced so far.

When such a change description reference is applied on a pattern, it makes no direct syntactic changes, because the outer data structure is unchanged: both the old and the new data is a list. Only the elements of the list are changed, this is reflected with a number of induced transformations:

- When the pattern has the form $[p_1, \ldots, p_n \,|\, p_{n+1}]$, the induced transformations are `cd` on p_1,. . ., and p_i, and the same mapping transformation on p_{n+1}.
- Other patterns are not affected.

When this transformation is applied on an expression, it may be directly transformed, based on its type:

- $[e_1, \ldots, e_n \,|\, e_{n+1}]$ is converted by inducing transformation `cd` on e_1, . . ., and e_n, and the same mapping transformation on e_{n+1}.
- `[`e_1 `||` p `<-` e_2`]` is converted by inducing transformation `cd` on e_1.
- Any other expression e is enclosed in the following compensation expression: `[case El of ... end || El <-` e`]`, where the clauses of the `case` expression are the same as described in the transformation of expressions.

5.2.2 Non-structural changes

In case of non-structural changes, like incrementing a value by one, we take a quite different approach. Instead of generating pattern and expression compensations from a common description, we provide two compensating expressions, one that converts old values to new values (used in updating patterns), and another that converts new values to old values (used on expression return values).

Such change descriptions are supposed to be used on atomic data. In case of patterns the compensation must result in a constant, so it makes sense to restrict the possible compensations to

side effect free expressions that can be evaluated during the transformation.

The syntactic representation of a non-structural change description is a pair of Erlang expressions. Both expressions may contain only one variable, that will be replaced with an expression. The first expression describes how to convert an old value to a new value, the second is the inverse of the first one.

Application on a constant pattern thus done by evaluation the expression on the constant, and replacing the original constant with the result. Variable patterns are not modified, their usages will be transformed (a safely reachable variable pattern is always safe itself).

Application on an expression is simply done by substituting the expression into the compensation expression.

Example. Transform the call to `f` in the following code using the non-structural change description (`Old-1`, `New+1`):

```
case f() of
   0 -> 0;
   N -> 1/N
end
```

trf_p(`f()`) contains the patterns of `case`, trf_e(`f()`) contains variable `N` in `1/N`. Pattern `0` is a constant, so it is substituted into `Old-1` and evaluated; pattern `N` is a variable, so it is not changed; expression `N` is substituted into `New+1`. The result is:

```
case f() of
   -1 -> 0;
   N -> 1/(N+1)
end
```

5.3 Change descriptions for the `regexp` module upgrade

Finally, the complete change description set that specifies the transformations proposed in Sec. **??** is provided here. They use all the features described in Sec. **??**, which demonstrates that this rule set, in spite of being minimalistic, is strong enough to support real world applications.

When changing the calls to interface functions of module `regexp`, the new call expressions are to be transformed using the change description with the same name as the function. Applying these change descriptions on the examples in Sec. **??**, the result is the solution proposed there.

match:

```
{match, St, Len}  ↦  {match, [{decr(St), Len}]}
         nomatch  ↦  nomatch
```

matches:

```
{match, []}  ↦  nomatch
{match, Ms}  ↦  {match, map(match_elem, Ms)}
```

match_elem: `{St, Len}` ↦ `[{decr(St), Len}]`

decr: (`Old-1`, `New+1`)

gsub: `{ok, Result, _}` ↦ `Result`

6. Implementation experiences

The main question about the real-world applicability of the presented approach is how large the data flow graph will be, and what the computational cost of the $\rightsquigarrow$ and $\rightsquigarrow_s$ relations is.

Experimental implementation in the RefactorErl system shows that the size of the presented static data flow graph is comparable to the size of the syntax tree. A syntax based transformation tool has to handle data structures of that size, so this should not be an issue.

The calculation of the relation based on the graph is a more delicate problem. These relations can be computed by either an iterative algorithm that finds new relation elements based on previously found elements and rules, or recursively applied graph traversals.

The former approach requires storage of the whole $\rightsquigarrow$ and $\rightsquigarrow_s$ relations, which is much more expensive than the direct flow graph, and its calculation time is proportional to its size. The latter approach with recursively called full-blown graph traversals seems to be no better.

Fortunately, using the graph traversal approach we don't have to calculate the whole $\rightsquigarrow$ and $\rightsquigarrow_s$ relations. In case of a large module, the data flow graph falls apart into many isolated components, because most data flow paths are not interconnected. The relations to be computed obviously cannot cross these graph component boundaries, which means we only have to compute them over the affected flow graph components.

In fact, we don't even compute the relations themselves, only the sets that are used during the transformation: reach, strict, safe, trf_p, and trf_e. These sets can be computed using graph traversal, starting from the initially changed node set, and the results will automatically restricted to the affected graph components, as the traversals won't cross component boundaries. This solution ensures that the run time cost of the transformation is proportional to the size of the affected code parts.

7. Related work

Restructuring software code while maintaining its consistency, known as refactoring, is a well known topic [**?**]. Tool support for refactoring Erlang programs exists for some years now [**?**, **?**], but no support has been provided for automated data structure refactoring based on change descriptions. A scripted refactoring framework is built by Verbaere [**?**], targeting generic refactoring implementation for object oriented languages, it provides syntax-based manipulations opposed to declarative style descriptions used by this work.

Data flow analysis for functional languages has been studied by Shivers [**?**], but this work and its followers like [**?**] target optimising compilers with flow analysis. For Erlang programs, data flow analysis was used by Dialyzer for type inference, but that approach has been dropped in favour of success typing [**?**]. Control flow and data flow analysis has been successfully applied to improve testing of Erlang programs by Widera [**?**].

8. Conclusions and future work

A refactoring-based generic approach to introducing incompatible module interface changes to existing source code has been presented. The main contribution of this work is an automatic transformation mechanism that uses a simple, intuitive change description schema to describe interface changes.

Transformation is done applying data flow analysis based on simple data flow graphs. This is a cost-effective technique that produces a good static approximation of data flow paths in a program, and it is applicable for other data structure refactorings as well [**?**].

A complete real-life example of migrating regular expression module calls shows that while the presented approach is not capable of handling very complicated changes, it is still useful in practise.

An obvious area of necessary improvement is support for more language elements. The complete Erlang syntax can easily be supported by defining direct data flow edges for the missing constructs, this is straightforward to do in the same way as in Fig. **??**.

A more complicated work that may improve the scope of data flow analysis is real support for those language features that require control flow analysis as well. These include processes and message passing, and exceptions. Analysing the process structure

and matching send and receive expressions is a really interesting, but hard problem.

Finally, it is also possible to extend the scope of the transformation by defining new change description schemes or improving the current ones. This can be done based on experiences with the current system, or by studying other interface migration cases which could be useful in practise.

References

[1] Armstrong, J.: Programming Erlang, Software for a Concurrent World. Pragmatic Bookshelf, 2007.

[2] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[3] Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Víg, A., Nagy, T., Tóth, M., and Király, R.: Building a refactoring tool for Erlang. Proceedings of the Workshop on Advanced Software Development Tools and Techniques, Paphos, Cyprus, 2008.

[4] Jagannathan, S. and Weeks, S.: A Unified Treatment of Flow Analysis in Higher-Order Languages Proceedings of the 22nd ACM Symposium on Principles of Programming Languages, 393–407, San Francisco, California 1995

[5] Li, H. and Thompson, S.: Tool Support for Refactoring Functional Programs. Proceedings of the Second ACM SIGPLAN Workshop on Refactoring Tools, Nashville, Tennessee, USA, 2008.

[6] Lindahl, T. and Sagonas, K.: Practical type inference based on success typings. Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, 167–178, Venice, Italy, 2006.

[7] Lövei, L., Horváth, Z., Kozsik, T., and Király, R.: Introducing Records by Refactoring. Proceedings of the 6th ACM SIGPLAN Erlang Workshop, 18–28, Freiburg, Germany, 2007.

[8] Muchnick, S. S. Advanced Compiler Design and Implementation. Morgan Kauffmann Publishers, 1997.

[9] Shivers, O.: Control-Flow Analysis of Higher-Order Languages. PhD thesis, Carnegie Mellon University, 1991.

[10] STDLIB Reference Manual. `http://www.erlang.org/doc/apps/stdlib/`

[11] Verbaere, M., Ettinger, R., de Moor, O.: JunGL: a scripting language for refactoring. Proceedings of the 28th International Conference on Software Engineering, 172–181, Shanghai, China, 2006.

[12] Widera, M.: Flow Graphs for Testing Sequential Erlang Programs. Proceedings of the ACM SIGPLAN 2004 Erlang Workshop, 48–53, Snowbird, Utah, USA, 2004.