Introducing Records by Refactoring in Erlang Programs*

László Lövei, Zoltán Horváth, Tamás Kozsik, Roland Király

Department of Programming Languages and Compilers, Eötvös Loránd University, Budapest, Hungary
{lovei,hz,kto,kiralyroland}@inf.elte.hu

Abstract. This paper focuses on introducing a new transformation to our existing model for refactoring Erlang programs. The goal of the transformation is to introduce a new abstraction level in data representation by substituting a group of related data with a record. Using record types enhances the legibility of the source code, makes further development easier, and makes programs less error prone by providing better possibilities for both compile time and run time checks.
Erlang is a dynamically typed language, and many of its semantical rules are also dynamic. Therefore the main challenge in this research is to ensure the safety of statically performed refactoring steps. The paper analyses the possibilities of transforming tuples to records via examples. There is a strong industrial demand for such a transformation in refactoring legacy code.

1 Introduction

The phrase "refactoring" stands for program transformations that preserve the meaning of programs [1]. Such transformations are often applied in order to improve the quality of program code: make it more readable, satisfy coding conventions, prepare it for further development etc. Simple refactorings are used by developers almost every day; they rename variables, introduce new arguments to functions, or create new functions from duplicated code. The object-oriented paradigm is especially well suited for refactoring-supported programming. In this area refactoring has already appeared in programming methodologies [2] and it is used heavily in the industry.

In old-fashioned programming environments, refactoring steps have been applied manually by the programmer. This requires the application of systematic changes on the program text, which is hard to accomplish. It is also very error-prone, even for simple transformations like renaming a variable. If we use the standard search&replace function found in every text editor, false replacements are likely to occur. Semantical analysis is required to correctly identify the occurrences of a variable, which is not provided by a simple editor. Furthermore, the

* Supported by GVOP-3.2.2-2004-07-0005/3.0 ELTE IKKK, Ericsson Hungary, ELTE CNL and OMAA-ÖAU 66öu2.

renaming of a variable should be avoided if its new name conflicts with another variable.

However, it *is* possible to perform most refactoring steps in an automated way with an appropriate software tool—a tool that is aware of the syntactic and semantical rules of the programming language in use. Such tools exist for many programming languages [3], and modern programming environments often incorporate such capabilities. Refactoring in functional languages is not really wide-spread yet, but there are many ongoing researches on the topic. For the functional programmer, the only full-featured refactoring tool is HaRe [6], which provides refactoring capabilities for Haskell programs within the editors Emacs and VIM. A prototype of a refactoring tool for Clean is also available [7]. The work presented in this paper has an approach similar to that of that prototype tool.

The paper [8] has described the design highlights of a refactoring tool for the functional programming language Erlang (and for the Erlang/OTP environment). The focus is on the method of extracting semantical information from programs written in a rather dynamic language. A novelty of the presented approach is that the strong dynamic nature of Erlang programs is handled by static analysis, and that programs are represented, stored and manipulated in a relational database. This feature makes it possible to express refactoring steps in a fairly compact and comprehensible way.

In this paper a new kind of refactoring is explored using the same approach. The goal is to transform tuples into records, which is a kind of type transformation. As there are no compile time type enforcements in Erlang, it is a real challenge to automatically discover every part of a program that should be modified. After some introductory notes on Erlang refactoring, the problems and possible solutions of this refactoring are presented using a reasonably simple example.

2 Refactoring in Erlang

Erlang/OTP [9] is a functional programming language and environment developed by Ericsson, designed for building concurrent and distributed fault-tolerant systems with soft real-time characteristics (like telecommunication systems). The core Erlang language consists of simple functional constructs extended with message passing to handle concurrency, and OTP is a set of design principles and libraries that support building fault-tolerant systems [10]. The language has a very strong dynamic nature that partly comes from concurrency and partly from dynamic language features.

From the refactoring point of view, the most important characteristic of a programming language is the extent of semantical information available by static analysis. As Erlang is a functional language, most language constructs can be analysed easily. Side effects are restricted to message passing and built-in

functions[1], variables are assigned a value only once in their life, and the code is organised into modules with explicit interface definitions and static export and import lists. An unusual feature (at least in a functional language) is that variables are not typed statically, they can have a value of any data type, but even that does not make the life of a refactor tool much harder.

On the other hand, the remaining few constructs offer a real challenge to static analysis. An example of this is matching corresponding message send and receive instructions. A destination of a message can be a process ID or a registered name, which are bound to function code at run time. Data flow analysis might help in discovering these relations, but it is a hard research topic in itself.

Another kind of problem is the possibility of running dynamically created code. The most prominent example of this is the `erl_eval` standard module, which contains functions that evaluate Erlang code constructed at run time. This functionality is clearly out of the scope of a static refactoring tool, but there are other constructs similar to this that are widely used and should be covered, like the `spawn` function that starts the evaluation of a function in a new process (and the function name and arguments might be constructed at run time), or `apply` function that calls a function (with the same problems). The normal function call syntax has some run time features too: variables are allowed instead of static module or function names.

It is very important to exactly define what part of the language is covered by our refactoring tool. Due to the simple syntax and relatively small number of constructs of Erlang, full syntactical coverage of the language is feasible, which is a key point in real life usability. However, semantical coverage seems to be achievable only for a lesser extent. A third aspect besides the syntax and semantics of the core language is library coverage: `spawn` and `apply` could be handled just like any other function call, but special support for them seems useful. OTP libraries fall in the same category.

In the following, the extent of the language features that are already supported by the tool is outlined.

Transformations on variables. One of the most simple refactorings is the *rename variable* transformation. The only semantical information that is necessary for it is the *scope* and *visibility* of the variables. The exact rules for them are given in the form of input and output contexts for every language construct [11], which is hard to follow and not really helpful in defining the conditions of a refactoring, so we have created a more suitable definition which is given in [8].

As it turned out while investigating other refactorings, variables play the central role in the static semantics of expressions. Every refactoring modifies expressions in some way, and most of them are concerned about variables, so it was important to make this semantical information available in an efficient way to the implementation of almost every refactoring, and this paper makes use of these concepts too.

[1] Built-in functions, or BIFs, are functions that are implemented in the runtime system.

Refactoring functions. The next simple transformation is *rename function*, which seems similar to *rename variable*, but different problems arise when dealing with it. Function visibility is much simpler than variable visibility: a named function is either exported, in this case it is visible from every module, or not exported, and then it is visible only in the defining module. There are no function name hiding, embedded functions or any other kinds of complication. Here the real problem comes from the already mentioned dynamic language constructs. While a variable can only be used statically, a function name can be constructed dynamically, and functions can be called using built-in functions.

Many of the function-related refactorings rely on finding every place of call for a given function. Obviously, it is impossible to statically determine the place of every dynamic call, but there are semi-dynamically constructed calls that are possible to find. The analysis described here heavily depends on this concept, refer to [8] for details.

Properties of subexpressions. The last set of transformations that we have investigated concern subexpressions. Such refactorings are *generalisation* when a subexpression of a function is replaced with an argument, *merge subexpression duplicates* when two or more instances of a subexpression are replaced with a variable that stores the value of the subexpression, and *extract function* when a subexpression is substituted with a call to a new function that uses the subexpression as body. These refactorings depend on variable semantics, but sometimes use other concepts as well: a particular subexpression is side effect free or not, which variables does it depend on. However, these concepts are not used in this paper.

3 Introducing records

This paper focuses on introducing a new transformation to our existing model for refactoring Erlang programs. The goal of the transformation is to introduce a new abstraction level in data representation by substituting a group of related data with a record. Using record types enhances the legibility of the source code, makes further development easier, and makes programs less error prone by providing better possibilities for both compile time and run time checks. Consider, for instance, a function that computes the product of two complex numbers. Complex numbers can be represented as pairs of real numbers: the real part and the imaginary part. The function below takes two arguments, both of them are tuples representing complex numbers, and the result is also a tuple representing the product. Note that the variable names `Re1`, `Re2`, `Im1` and `Im2` reflect the purpose of the variables: they serve as the real parts and imaginary parts of complex numbers.

```
mul( {Re1,Im1}, {Re2,Im2} ) -> {Re1*Re2-Im1*Im2, Re1*Im2+Im1*Re2}.
```

Besides the above Cartesian representation of complex numbers, the polar representation is also widely used – each complex number can be specified by its

polar coordinates: its absolute value and its argument. This representation can also be programmed in Erlang as a tuple of two real numbers. Multiplication on polar representation of complex numbers can be implemented in the following way.

```
mulPolar( {R1,Phi1}, {R2,Phi2} ) -> {R1*R2, Phi1+Phi2}.
```

Using both representations in the same Erlang program is rather error-prone. Only the variable names used in tuple patterns reflect the representation, and neither the compiler nor the runtime system can force the legal use of the multiplication functions. No syntactic or semantical rules forbid, for example, the application of `mul` on complex numbers in polar representation. However, turning the tuples into records can help. One can distinguish the two representations by introducing two record types, `cart` and `polar`. (Similar distinction between the two representations of complex numbers was made in [12].)

```
-record(cart,{re,im}).
-record(polar,{r,phi}).
```

The arguments of the multiplication functions should be adjusted. (Note that `mul` is renamed to `mulCart`.)

```
mulCart(  #cart{re=Re1,im=Im1}, #cart{re=Re2,im=Im2} )
   -> #cart{re=Re1*Re2-Im1*Im2, im=Re1*Im2+Im1*Re2}.
mulPolar( #polar{r=R1,phi=Phi1}, #polar{r=R2,phi=Phi2} )
   -> #polar{r=R1*R2, phi=Phi1+Phi2}.
```

The explicit separation of the two representations and the possibility to pattern-match on the structure of the arguments of a function make it possible to prepare a polymorphic multiplication function, one that accepts complex numbers in both representations.

```
mulPolar( #polar{r=R1,phi=Phi1}, #polar{r=R2,phi=Phi2} )
   -> #polar{r=R1*R2, phi=Phi1+Phi2}.
mul( C1, C2 )
   -> mulCart( toCart(C1), toCart(C2) ).
toCart(C=#cart{}) -> C;
toCart(#polar{r=R,phi=Phi}) -> #cart{re=R*cos(Phi),im=R*sin(Phi)}.
```

3.1 Motivating example

In this example we will work on the source code of a time-based property storage server process. The process manipulates two sets of properties, always working with one of them at a time. One set is used at peak time, the other is at off-peak time, and an external signal is used to switch between them. Client processes can read or modify the value of a property based on an arbitrary key value.

The usual Erlang implementation of such a process is shown in Fig. 1. The module `propserv` consists of two main parts: exported interface functions for

```
-module(propserv).
-export([start/1, next/0, set/2, get/1]).

%% Interface functions
start(Time) when (Time == peak) or (Time == offpeak) ->
    register(server, spawn(fun () -> init(Time) end)).
next() ->
    server ! {next}.
get(Key) ->
    server ! {get, self(), Key},
    receive {reply, Value} -> Value end.
set(Key, Value) ->
    server ! {set, Key, Value}.

%% Server implementation
init(Time) ->
    loop(Time, empty(), empty()).
loop(Time, StP, StOP) ->
    receive
        {next} ->
            do_next(Time, StP, StOP);
        {get, From, Key} ->
            do_get(Time, StP, StOP, From, Key);
        {set, Key, Value} ->
            do_set(Time, StP, StOP, Key, Value)
    end.

do_next(peak, StP, StOP) ->
    loop(offpeak, StP, StOP);
do_next(offpeak, StP, StOP) ->
    loop(peak, StP, StOP).

do_get(peak, StP, StOP, From, Key) ->
    get_value(From, StP, Key),
    loop(peak, StP, StOP);
do_get(offpeak, StP, StOP, From, Key) ->
    get_value(From, StOP, Key),
    loop(offpeak, StP, StOP);

do_set(peak, StP, StOP, Key, Value) ->
    NewSt = set_value(StP, Key, Value),
    loop(peak, NewSt, StOP);
do_set(offpeak, StP, StOP, Key, Value) ->
    NewSt = set_value(StOP, Key, Value),
    loop(offpeak, StP, NewSt).
```

Fig. 1. Source code of the time based property storage server.

every operation that hide the communication protocols between the clients and the server, and internal functions that implement the server process itself. The source code of the `empty`, `set_value`, and `get_value` functions are omitted, they manipulate storages of key-value pairs. These lower level functions are not affected by the example transformations. Interface functions also remain unchanged, we will focus on improving the server implementation by introducing a record representation for the internal state of the server.

The state of this server has three components: the current time (represented by the constants `peak` and `offpeak`) and a storage object for both times. In this naïve implementation, each component is represented by a variable, and a naming convention is used to express their relationship: the storage object for peak time is called `StP`, `StOP` is the storage for off-peak time, and the current time is referred to as `Time`.

There are obvious disadvantages of this approach. While naming conventions are useful, they are not enforced by the compiler, and provide only an ad hoc connection between the arguments of the functions. The greatest problem arises when the state should be extended with a new component: every single place in the source code where the state is accessed should be updated, which is a very tedious and error-prone task. Records in Erlang are meant to solve these issues.

In the following, a step-by-step description of a refactoring is given, where we transform this program to store its state in a record. The most important steps can be automated based on semantical analysis, thus they can be supported by a refactoring tool.

User input. The whole transformation process cannot be automated, it would require the detection of which variables are used for representing the state of the process and which are not. This is an intuitive task that should be carried out by an experienced developer; all we can do is to minimize her manual work and automate most of the syntactical changes.

The preparation of the input source code involves grouping components of the state together. Grouping can be done by making tuples of related data, and this is usually done anyway, because this is the only way to pass around more that one piece of data together, *except* in function arguments, as it is the case in our example. Fortunately, there is a simple refactoring that helps in this task, it is called *tuple function arguments*. There may be situations, where it is not enough, because the state components in the arguments of a function are scattered, or not in the proper order, but reordering function arguments can be supported by a refactoring too.

The preparation phase of our example is shown in Fig. 2. Every involved function should be refactored to contain the state components in a tuple argument[2]. The figure does not show every single change, only a sample, every other function and call is modified the same way – the same applies to the subsequent steps.

[2] Note that this is much less work than manually converting every involved *expression* in the code!

After the preparation, the real user input to the *introduce record* refactoring is a position of one tuple, for example, the argument of the `loop` function. The record type to be created must be given a name and a set of field names, this is the other part of the input. In the example, we use `state` as record name and `time`, `stP`, and `stOP` as field names.

Original code

```
init(Time) -> loop(Time, empty(), empty()).
    loop(Time, StP, StOP) ->
        receive
            {set, Key, Val} ->
                do_set(Time, StP, StOP, Key, Value);
        end.
do_set(peak, StP, StOP, Key, Value) ->
    NewSt = set_value(StP, Key, Value),
    loop(peak, StP, StOP);
```

→

Tuple arguments

```
init(Time) -> loop( { Time, empty(), empty() } ).
    loop( { Time, StP, StOP } ) ->
        receive
            {set, Key, Val} ->
                do_set( { Time, StP, StOP } , Key, Value)
    end.
do_set( { peak, StP, StOP } , Key, Value) ->
    NewSt = set_value(StP, Key, Value),
    loop( { peak, NewSt, StOP } );
```

Fig. 2. Tuple Function Arguments

Converting directly affected expressions. Syntactically the first argument of the `loop` function is a *tuple pattern*. Its role is to decompose the actual argument of the function that should be a 3-tuple, and to store the components in the given variables. It should be transformed into a *record pattern*: its role is just the same, only it applies to record data, and decomposes based on field names instead of positions in the tuple. To preserve the meaning of the program, every tuple that is passed as the first argument of the `loop` function, must be transformed to a record. The first arguments of the `loop` function calls are the *directly affected* expressions of this transformation: the minimal compensation for transforming the `loop` function is the transformation of these expressions.

In our example, this transformation is easy, because the directly affected expressions are all tuple constructors (which is not a surprise, they were created as the result of the preparation phase). They are just simply changed to record constructors, and the resulting program code (shown in Fig. 3) works. A record

definition is naturally introduced at the beginning of the module, but no other changes are required.

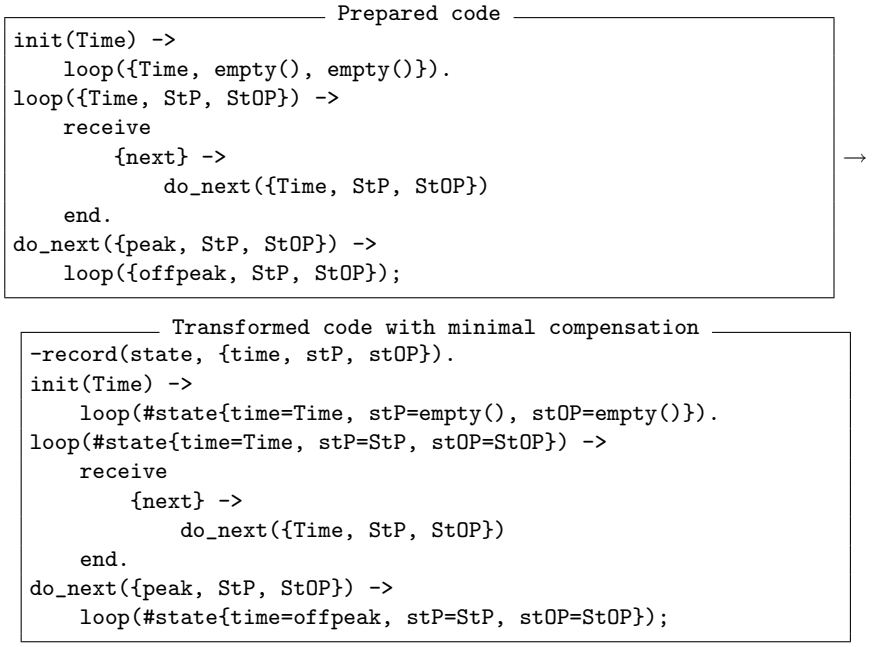

Fig. 3. 1st step: convert expressions matched against the pattern.

Of course, there may be cases when the actual argument of a transformed function is not a tuple expression. In that case, the trivial solution is to introduce a runtime conversion that creates records from tuples, and a more complicated approach is to propagate the conversion when it is possible, e.g. when the actual argument is a variable, the expression bound to the variable could be converted to a record expression.

Finding derived expression. The minimal compensation described above works, but does not yield very useful results. It is not enough to transform the directly affected expressions in the program, indirectly affected parts should be discovered automatically. A possible way towards this is the propagation of the record introduction through variables, extending the set of directly affected places in the program. While this is a limited possibility, it can be useful.

In our example, another kind of indirect link exists between expressions, which can be found by statical analysis. Let's have a look at the `loop` function! The variables `Time`, `StP`, and `StOP` are known to be originating from the respective record fields, and in the function body, new tuples are constructed using these variables as the actual arguments of `do_next`, `do_get` and `do_set`.

It is easy to recognise automatically that every component of these tuples comes from a `state` record, therefore, the tuple itself could be transformed into a `state` record. The result of that transformation is shown in Fig. 4, with the directly affected patterns transformed.

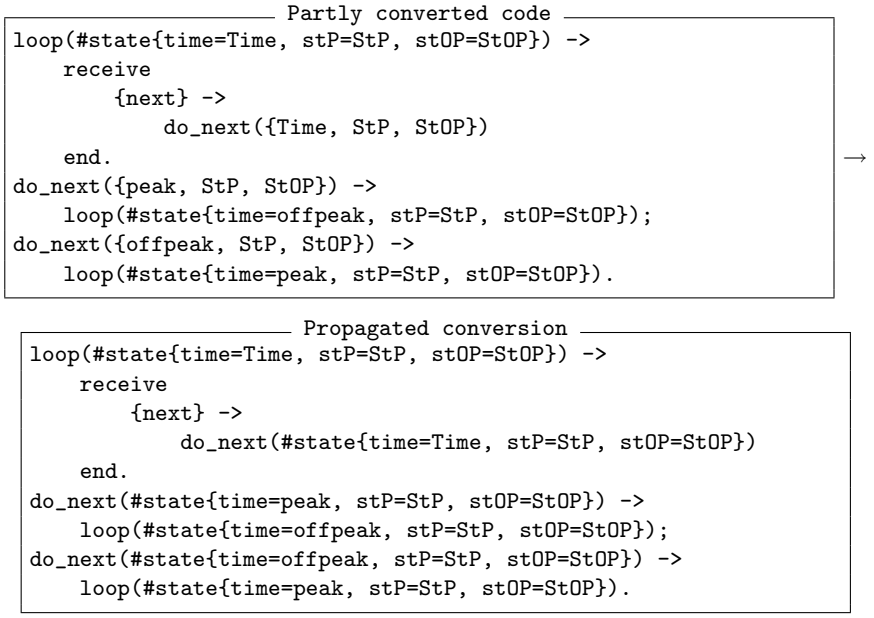

Fig. 4. 2nd step: find derived expressions and convert matching patterns

Transforming a tuple is easy again in the example, because of the same reason as before: the tuples come from a previous refactoring. The same question remains as before: what if the affected patterns are not tuple patterns? Well, if the pattern is a simple variable, we can propagate the transformation (that is, convert every expression that uses the variable), but in every other case, the transformation is not feasible. When we think of this transformation as an automatically iterated step of a complex refactoring, this means that the record expression must be converted back to its tuple form at runtime.

Record updates. So far we have seen several rules that can be applied iteratively: transformation of patterns, then directly affected expressions, then derived expressions, and finally directly affected patterns, and we are back at the beginning. Combined with propagation through variables, most of the tuples that share common data with the originally selected tuple can be transformed into the desired record.

While the resulting code works just the same as before and uses records, we still haven't yet reached the stage where using records is actually more useful

than using tuples. Recall that one of the main advantages of records is that they can be extended with new components without changing existing code. However, we can rely on this possibility only if the code we write is independent of the non-yet-existing record fields. In our example this means that every state transition should leave the unaffected fields intact.

The code that we generated so far clearly fails to achieve this: every state transition decomposes the state using pattern matching and builds a new record using a mixture of original and modified field values. Using *record updates* instead of record constructors would yield the same result, the only difference is that fields introduced later would be preserved by the update syntax.

Code using constructors

```
loop(#state{time=Time, stP=StP, stOP=StOP}) ->
    receive
        {next} ->
            do_next(#state{time=Time, stP=StP, stOP=StOP})
    end.
do_get(#state{time=peak, stP=StP, stOP=StOP}, From, Key) ->
    get_value(From, StP, Key),
    loop(#state{time=peak, stP=StP, stOP=StOP});
do_set(#state{time=peak, stP=StP, stOP=StOP}, Key, Value) ->
    NewSt = set_value(StP, Key, Value),
    loop(#state{time=peak, stP=NewSt, stOP=StOP});
```

→

Code using updates

```
loop(St=#state{time=Time, stP=StP, stOP=StOP}) ->
    receive
        {next} ->
            do_next(St)
    end.
do_get(St=#state{time=peak, stP=StP, stOP=StOP},From,Key)->
    get_value(From, StP, Key),
    loop(St);
do_set(St=#state{time=peak, stP=StP, stOP=StOP},Key,Value)->
    NewSt = set_value(StP, Key, Value),
    loop(St#state{stP=NewSt});
```

Fig. 5. 3rd step: utilize record updates

Fig. 5. shows the result of the next simple transformation. A simple change that makes using record updates possible is the introduction of a variable name for every record pattern (the name of the variable is `St` in the example). After this change, every record constructor expression can be transformed into a record update expression that is based on the newly introduced variable. The list of updated fields could be the same as the list of created fields in the constructor expression, but it is easy to mark the fields that have the same value as in the

pattern. These fields can be left out of the update, sometimes leaving the update empty, which means that the argument of the function was simply copied.

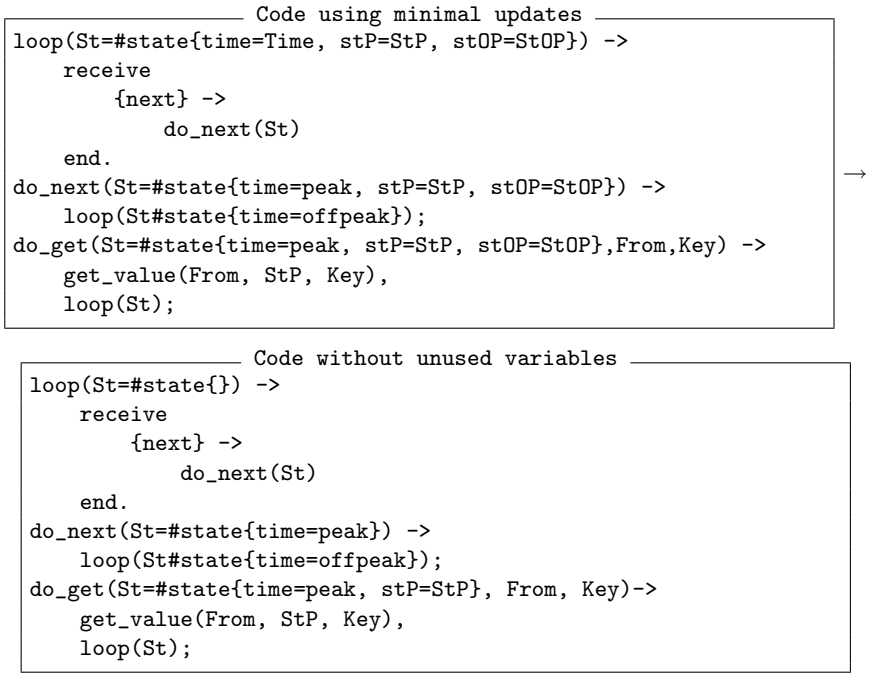

Fig. 6. 4th step: remove unused variables from patterns

Syntactical cleanup. The resulting code after the previous steps is good enough for our primary purpose: the state of the example code is transformed from a tuple representation into a record, and the state transition functions preserve every field of this record, even those that do not yet exist. There is only one more issue to be solved which is presented here for completeness. If we compiled the result of Fig. 5, we would get a couple of warning messages from the compiler about unused variables – these were introduced when we left out unnecessary parts of records updates. It helps to leave out these unused variables of the patterns, the result is much clearer, see Fig. 6. Note that the record pattern itself must not be removed, even if it becomes empty: it represents a type guard condition in the pattern.

3.2 Transformation parameters

Before we can go into the details of how to perform an *introduce record* refactoring, we need to determine what should be the input data that can be used

as a starting point. It is natural to demand the name of the record and the names of the record fields as input, and we also need the starting point of the transformation, but when should we accept the input as valid?

The starting point must be some kind of tuple occurrence. This can be guaranteed by accepting only a tuple skeleton construct as input (e.g. using the position of the opening brace in the tuple syntax). A tuple skeleton has a fixed number of elements, so it is easy to check another important condition: the number of record fields must be the same as the number of tuple elements.

Introduction of records involves using names, which always carries the possibility of introducing name conflicts. Fortunately, this is not the case here: it can always make sense to reuse existing records names. Even if some fields of the existing records are unused, the result of the transformation will be correct. Introduction of new field names to an existing record works too, without changing the meaning of existing code.

One more possible parameter needs to be mentioned here. When an exported function is refactored, every call to it should be updated, which means calls from other modules too. Now, the record definition must be available at every place of call, and inter-module record definitions are best to be put in a header file. The name of the header file should be supplied by the user too.

3.3 Transformation rules

In the following a semi-algorithmic description of the refactoring is given. A set of iteratively applied steps ensure that every affected code part is updated. There are basic steps that update a specific language construct or generate direct compensations of these updates, and there are propagation steps that find indirectly affected code parts. Propagation is not necessary for behaviour preservation, but highly increases the practical usefulness of the refactoring. First the description of basic steps are given.

Pattern transformation. The minimal compensation for the rewriting of a pattern is the rewriting of every expression that is matched against the pattern. The rewritten pattern must provide the same variable bindings with the same values as the original[3] There are three issues here: how to rewrite a pattern, how to find the expressions that are to be rewritten, and how to rewrite them.

In this paper we only deal with top-level patterns. Embedded patterns might be supported in the future too, but they have much more complex compensations. Furthermore, there are language constructs with multiple clauses that use pattern matching: function definitions, `case`, `try`, and `fun` expressions. Some of these patterns may have a form different from a tuple, and they may be tuples with a length different from the converted one. These patterns are not affected by the tuple to record conversion, but every clause with a tuple pattern of the same

[3] Except of course when the original value was a tuple that is being converted into a record. When that tuple is bound to a variable, propagation is used to solve the situation.

length as the converted one should be updated: its pattern must be rewritten the same way as the originally selected one.

When a top-level pattern is a tuple skeleton, it has the following syntactic form:

$$\{p_1, p_2, \ldots, p_n\}.$$

The rewritten form that uses the record name "`rec`" and field names "f_1", "f_2", ..., "f_n" is the following:

$$\texttt{\#rec}\{f_1 = p_1, f_2 = p_2, \ldots, f_n = p_n\}.$$

The side condition of this transformation is that the pattern is a top-level pattern. The compensation is that every matched expression is transformed to the same record (see below).

When the whole pattern is just a variable, the transformation should be propagated through this variable. The case of an as-pattern (which has the form `Var =` $\{p_1, \ldots, p_n\}$) combines the two previous cases: the tuple part should be transformed as a standalone tuple pattern, and the transformation should be propagated through the variable as well.

Finally, any other kind of pattern stands for a non-tuple pattern, so it cannot be transformed at all. Tuples with a size different from the originally converted tuple fall into this category as well. In a branching construct, this means that the non-matching branches are left intact; non-branching constructs establish a dead end for propagation.

Finding matched expressions. Direct compensations require a simple analysis of data flow in the program. Data flow in Erlang is based on pattern matching: when the result of an expression is calculated, the result (or parts of the decomposed result) is bound to variables by a pattern. When a pattern is modified, every expression that creates values for that pattern has to be modified accordingly, and when an expression is modified, every pattern that receives a value from that expression has to be modified too.

These connections between patterns and expressions can be tracked based on the following rules:

- A function call expression matches its actual arguments against the referenced function's formal arguments. The function might have many clauses, in that case every clause introduces a possible pattern match.
- Expressions of the form `case expr of` and `try expr of` match `expr` against the patterns in their clauses.
- Pattern match expressions of the form `pattern = expr` match `expr` against `pattern`.

There are some constructs that use pattern matching as well, but we cannot support them due to the lack of statical analysis:

- A `fun` expression is very much like a defined function, but it is usually called using variables. None of these constructs are supported right now.

– Message sending and receiving (`!` and `receive` expressions) is not supported, just like `throw` and the `catch` part of `try` expressions.
– List comprehensions have the same problems as embedded tuple patterns, so they are ruled out right now.

Expression transformation. The tuple result of an expression can always be turned into a record, because if all else fails, a runtime conversion could be applied to the result. However, that should be done only as a last resort to keep the code working, the main goal of the refactoring is the introduction of records with record syntax. The compensation of the transformation depends on how we use result of the expression. We should try to propagate the changes until a matching pattern is found, then that pattern should be transformed too. If the propagation stops in a chain of expressions, the solution is the insertion of a reverse runtime conversion.

A tuple skeleton expression can be rewritten into a record expression in the same way as a tuple pattern. It has the following syntactic form:

$$\{e_1, e_2, \ldots, e_n\}.$$

The rewritten form that uses the record name "`rec`" and field names "f_1", "f_2", ..., "f_n" is the following:

$$\texttt{\#rec}\{f_1 = e_1, f_2 = e_2, \ldots, f_n = e_n\}.$$

This rewriting can always be applied, and the compensation is to transform the expressions that use the result of the rewritten expressions. If this is not possible, the result itself should be converted using a runtime conversion function.

Expressions of every other type only pass around the result, so they should be handled by propagation. Special exceptions are built-in functions that deal with tuples:

– `element(`k`,expr)` returns the k^{th} element of the tuple, it can be replaced with `expr#rec.`f_k when k is a constant.
– `is_tuple(expr)` is a type test function which has an equivalent for records: `is_record(rec, expr)`.

When propagation fails, a runtime conversion should be made to the result of the last expression that has successfully been transformed. We use `fun` expressions (these create anonymous functions, also called lambda expressions) to handle this situation, and bind these conversion functions to macros to keep source code somewhat readable. The conversion functions for a record named "`rec`" with field names "f_1", "f_2", ..., "f_n" are the following:

$$\texttt{fun(}\{V_1, V_2, \ldots, V_n\}\texttt{) -> \#rec}\{f_1 = V_1, f_2 = V_2, \ldots, f_n = V_n\}\texttt{ end}$$

converts a tuple to the record, and

$$\texttt{fun(\#rec}\{f_1 = V_1, f_2 = V_2, \ldots, f_n = V_n\}\texttt{) -> }\{V_1, V_2, \ldots, V_n\}\texttt{ end}$$

does the reverse conversion. These macros are placed right next to the record definition (maybe in a header file), and during later refactoring they can be recognised and removed if used on each other.

3.4 Propagation rules

So far we have seen transformations based on direct connections between code parts. The idea of propagation is that there are constructs that simply pass data around regardless of its type, and therefore they can be left intact while they help discovering indirectly affected code parts. The prime example of such a construct is a variable. When we transform a tuple that is bound to a variable, only the variable is affected directly. A possible compensation would be to insert runtime conversions to every occurrence of the variable. But instead, we can trigger the continuation of the transformation to the code parts that are directly affected by variable occurrences, and are indirectly affected by the binding. In fact, we trace the flow of data through a variable, and transform everything that uses the transformed data.

This kind of propagation has two directions. We can follow the flow of data, and starting from a tuple constructor, we can find the places where that data is used, or we can go in the opposite direction: trace where a piece of data comes from, and find the corresponding tuple constructor. These two kinds of propagations use the same rules, only the direction of traversal is changing.

Record propagation. The information that a piece of data should be transformed from a tuple to a record can be propagated by the following constructs:

- When a variable as a whole pattern is transformed, its every occurrence should be transformed too. When a variable expression as a whole is transformed, the occurrences and the pattern that bound the variable should be defined too.
- When a block expression as a whole is transformed, the last expression of its body should be transformed too (and vice versa). The same applies to a parenthesised expressions and its only subexpression.
- When a branching expression (`if`, `case`, `receive`, and `try`) as a whole is transformed, the last expression of every clause should be transformed too. When the last expression of one clause is transformed, every other clause's last expression and the expression as a whole should be transformed.
- When a function call expression as a whole is transformed, the last expression of the referred function's every clause should be transformed too. When the last expression of a function clause is transformed, every other clause's last expression and every function call that refers to the function should be transformed too.

Field propagation. The role of a tuple pattern is the decomposition of a tuple data. When a field of a tuple has to be changed, the only way to go is to decompose the tuple and re-compose a new tuple using some of the original components and some new components. When we transform the original tuple to a record, it would be nice to recognise this kind of recomposition, and transform the recomposing tuple constructor into a record constructor, partly in the hope that it will be possible to turn it later into a record update expression.

It is easy to recognise that to find these constructs we need a mechanism similar to record propagation. The flow of data need to be tracked, but instead of the whole tuple, only an element of the tuple is tracked - using the same propagation rules as for the whole tuple. Using this technique, we can find every tuple constructor that uses data that originates from a record field. When the tuple has the same size as the originally converted one, and every element that comes from a record field has the same position as the corresponding element in the original tuple, we have reason to believe that the newly discovered tuple plays the same role in the code as the introduced record, so the record transformation should be propagated to it.

4 Related work

Refactoring was first recognised as a distinct programming technique of its own in Fowler's refactoring bible [1] which addressed a wide range of refactorings for object-oriented software providing examples in Java. Most research activities in this field focus on object-oriented environments, an exhaustive survey on the existing techniques and formalisms is [13].

Tool support for refactoring was first provided by the refactoring browser for Smalltalk [14]. Many tools are available for Java, sometimes embedded into a development environment (e.g. Eclipse, JFactor, IntelliJ Idea, Together-J etc.), and some for C# (ReSharper, C# Refactory) and C++ (SlickEdit, Ref++). These tools support various kinds of renamings, extracting or inlining code, and manipulating the class hierarchy. There is a good summary of the available tools and a catalog of well-known refactorings at [3].

Refactoring in functional languages has received much less attention. Haskell was the first functional language to gain tool support for refactoring, and so far the Haskell Refactorer prototype [6] is the only functional refactorer software that is actually usable in practice. Refactoring functional programs using database representation first appeared in [15] for the Clean language, and a standalone prototype is available [7] from this research.

Refactoring Erlang programs is a joint research with the University of Kent, building on experiences with Haskell and Clean. While we are sharing ideas and experiences, they are investigating a completely different implementation approach using traversals on annotated abstract syntax trees [16].

5 Conclusion

This paper analyses the Erlang programming language with respect to introduction of records via case studies. The investigation is based on our existing model of refactoring that already supports several simpler transformations ("rename variable", "rename function", "reorder function arguments", "tuple function arguments", "extract function", "eliminate variable", and "merge subexpression duplicates"). The main problem is to express a static program transformation and statically computable conditions in a dynamic environment like

Erlang/OTP. The article presents how to compose a complex transformation from simple steps on a running example. The model uses our results on applying static analysis methods instead of the dynamic semantical rules of Erlang. It is recognised that a statically computed data flow graph is needed to support fully automated transformation of complex source code.

References

1. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
2. Beck, K.: Extreme Programming Explained. Addison-Wesley (1999)
3. : Martin Fowler's refactoring site. (http://www.refactoring.com/)
4. : Eclipse Project homepage. (http://www.eclipse.org/)
5. : C# Refactory homepage. (http://www.xtreme-simplicity.net/)
6. Li, H., Thompson, S., Reinke, C.: The Haskell Refactorer, HaRe, and its API. Electronic Notes in Theoretical Computer Science **141**(4) (2005) 29–34
7. Szabó-Nacsa, R., Diviánszky, P., Horváth, Z.: Prototype environment for refactoring Clean programs. In: The Fourth Conference of PhD Students in Computer Science (CSCS 2004), Vol. of extended abstracts, Szeged, Hungary (2004) 113 (full paper: http://aszt.inf.elte.hu/~fun_ver/,10 pages).
8. Lövei, L., Horváth, Z., Kozsik, T., Víg, A., Nagy, T.: Refactoring erlang programs. to appear in Periodica Polytechnica – Electrical Engineering (2007) 19 pages.
9. Armstrong, J., Virding, R., Williams, M., Wikstrom, C.: Concurrent Programming in Erlang. Prentice Hall (1996)
10. Armstrong, J.: Making reliable distributed systems in the presence of software errors. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden (2003)
11. Barklund, J., Virding, R.: Erlang Reference Manual. (1999) Available from http://www.erlang.org/download/erl_spec47.ps.gz.
12. Kozsik, T.: Tutorial on subtype marks. In Horváth, Z., ed.: Proceedings of the Central-European Functional Programming School (CEFP'05). Volume 4164 of LNCS., Springer-Verlag (2006) 191–222
13. Mens, T., Tourwe, T.: A survey of software refactoring. IEEE Transactions on Software Engineering **30**(2) (2004) 126–139
14. Roberts, D., Brant, J., Johnson, R.: A refactoring tool for Smalltalk. Theory and Practice of Object Systems (TAPOS) **3**(4) (1997) 253–263
15. Diviánszky, P., Szabó-Nacsa, R., Horváth, Z.: Refactoring via database representation. In Csőke et al., L., ed.: The Sixth International Conference on Applied Informatics (ICAI 2004), Eger, Hungary (2004) 129–135
16. Li, H., Thompson, S., Lövei, L., Horváth, Z., Kozsik, T., Víg, A., Nagy, T.: Refactoring Erlang programs. In: The Proceedings of 12th International Erlang/OTP User Conference, Stockholm, Sweden (2006) http://www.erlang.se/euc/06/, 10 pages.