Abstract
This paper considers how to use refactorings to semi-automatically transform Erlang programs to use alternative data structures as part of the parallelisation process. Although it has not previously been widely studied, the correct choice of data structure can make a significant difference to the overall performance and parallel speedups that can be obtained. A key contribution of the paper is that we introduce a new recursive descent refactoring technique that transforms a specific instance of a data structure and any operations affected by the initial transformation, descending into and refactoring any programmer-defined functions. We illustrate our approach by showing how widely-used generic list operations can be easily transformed to Erlang-specific binary or ETS (Erlang Term Storage) data structures or vice-versa, using a programmer-in-the-loop semi-automatic refactoring process. Our examples show that, on a 24-core multicore machine, we can achieve speedups of up to 24 times the sequential performance in some cases by choosing the correct data structure. The wrong choice of data structure can reduce performance by 30% or more, or, in the worst case, even cause complete system failure.

1. Introduction
This paper studies the problem of introducing alternative data structures to support a programmer-in-the-loop parallelisation approach. The correct choice of data structure can make a significant difference to the parallel performance or even make parallelism infeasible. Automated support for data structure transformations makes it easier to produce the best parallelisation for some given program. In this paper, we consider three widely-used Erlang data structures: lists, binary structures and ETS (Erlang Term Storage) tables, showing how transformations may be easily introduced under programmer control. Our approach integrates with the Wrangler refactoring tool for Erlang, and uses the advanced Skel [7] skeleton library for Erlang. This library has previously been shown to give good parallelisations for a number of applications, including a multi-agent system [3] where we have achieved speedups of up to 142.44 on a 61-core machine with 244 threads.

1.1 Novel Contributions
The main contributions of this paper are:
• We show that alternative choices of data structures can have a significant impact on parallel performance and speedup – the wrong choice may even lead to complete failure;
• We introduce new refactorings for translating generic lists into functionally equivalent Erlang-specific binary and ETS data structures, including automatically translating common operations over those lists;
• We develop a new notion of recursive descent refactoring that applies a refactoring repeatedly, and use this to systematically transform a complete program to use the most appropriate data structures; and
• We give three examples of standard benchmark programs in Erlang that illustrate the use of our refactoring for the Skel parallel skeleton library, showing that we are able to deliver speedups over the original sequential program of around 24 on a 24-core multicore machine.

In general, ETS tables deliver the best parallel performance for the examples that we have considered. However, our results show that simple lists may deliver similar performance to the use of ETS tables, and better performance than using binary structures. This means that we cannot blindly choose to implement a single optimisation as part of the compilation process. Our approach also allows the use of new (possibly user-defined) data structures and other transformations in future, giving a high level of flexibility and generality.

2. Algorithmic Skeletons and the Skel Library
Algorithmic skeletons [10, 11] abstract commonly-used patterns of parallel computation, communication, and interaction into parameterised templates. A recent survey of algorithmic skeleton approaches can be found at [16]. In a functional language, such as Erlang, skeletons can be defined as higher-order functions that can be instantiated with specific user code to give some concrete parallel behaviour. For example, we might define a parallel map skeleton, whose functionality is identical to a standard map function, but which creates a number of Erlang processes (worker processes) that evaluate each element of the map in parallel. For example, we can merge each element of a list of image pairs, Xs, by mapping over them:

\[
\text{lists:map(fun(X) -> readImage(\text{convertMerge}(X)) end, Xs)}
\]

As each pair of images in Xs is independent of each other image pair, we can use a parallel map to perform the merging in parallel. As defined above, this data parallel approach will introduce one worker process per list element. This will often mean that each
worker process only performs a small amount of work (it is fine-grained). However, the skeleton approach allows more efficient implementations to be used where appropriate, provided that they are functionally equivalent to the original map. For example, a worker process may deal with multiple map operations in a single call, so increasing the granularity of the worker processes. This will usually improve performance on a modern CPU.

Using a skeleton approach allows the programmer to adopt a top-down structured approach to parallel programming, where skeletons are composed to give the overall parallel structure of the program. This gives a flexible semi-implicit approach to parallelisation, where parallelism is exposed to the programmer only through the choice of skeleton and perhaps through some specific behavioural parameters (e.g. the number of parallel processes to be created, or how the individual elements of the parallel list are to be grouped to reduce communication costs). Details such as communication, task creation, task or data migration, scheduling etc. are embedded within the skeleton implementation, which may then be tailored to a specific parallel architecture or class of architectures. This offers an improved level of portability over typical low-level approaches. However, it will still be necessary to tune behavioural parameters in particular cases, and it may even be necessary to alter the parallel structure to deal with varying architectures (especially where an application targets different classes of architecture).

2.1 The Skel library of Erlang Skeletons

The Skel library \[7\] defines a small set of classical skeletons for Erlang. Each of the skeletons that we will consider operates over a stream of independent input values, producing a corresponding stream of independent results. Because each skeleton is defined as a streaming operation, one skeleton can be freely substituted for another provided that they both have equivalent types. The same property also allows simple composition and nesting of skeletons. This paper will consider the following subset of the Skel skeletons:

- \texttt{func} is a primitive wrapper skeleton that applies a function atomically to each value in the input stream, producing a stream of results.

- The parallel pipeline skeleton, \texttt{pipe}, applies each stage of the pipeline in turn to the input stream, passing the result of each stage to the next stage. Parallelism arises from the fact that each stage of the pipeline is evaluated by a separate process; different stages of the pipeline may therefore evaluate their inputs in parallel (Figure 2).

  - The parallel task farm skeleton, \texttt{farm}, applies a function to all elements of the input stream. A number of workers are created to allow the parallel execution of multiple applications (Figure 1).

For example, the map operation
\[
\texttt{lists:map}(\text{fun } m:f/1, Xs)
\]
can be transformed into a Skel farm with \texttt{Nw} workers:
\[
\texttt{skel:farm}(\text{fun } m:f/1, \texttt{Nw}, Xs)
\]
where the top-level \texttt{farm} function is syntactic sugar for the generic top-level function:
\[
\texttt{skel:do}((\text{farm}, \{\text{func}, \text{fun } m:f/1\}, \texttt{Nw}), Xs)
\]
The \texttt{skel:do} operation instantiates the skeleton given by its first parameter (here, the tuple \{\texttt{farm}, \{\text{func}, \text{fun } m:f/1\}, \texttt{Nw}\}) and applies it to the inputs defined in its second parameter (here, the list \texttt{Xs}). Each skeleton is represented by a tuple with at least two elements: the first element is an atom that identifies the skeleton, and the second and subsequent elements are the parameters to the skeleton. In our example, the skeleton is a \texttt{farm} over the embedded \texttt{func} skeleton, \{\text{func}, \text{fun } m:f/1\}, which lifts the user-defined Erlang function \text{m:f/1} into a streaming skeleton. The farm has \texttt{Nw} workers. The \texttt{skel:do} operation creates the required Erlang worker processes, links their inputs and outputs correctly, etc.

3. Refactoring for Parallelism

Refactoring involves rewriting program source according to some pre-defined rules. In contrast to general program transformations, refactoring focuses on purely structural changes rather than on changes to program functionality, and it is generally applied semi-automatically, i.e. under programmer direction, rather than fully automatically. This allows programmer knowledge about e.g. safety properties to be exploited, and so permits a wider range of possible transformation than could be applied automatically. Typical refactoring include variable renaming (changes all instances of a variable that are in scope to a new name), parameter adding (introduces a new parameter to a function definition and updates all relevant calls to that function with a placeholder), function extraction...
For each relevant statement:

- Send
- Tuple Constructor
- List Constructor
- Pattern Match
- Case Expression
- Wrapper/Library Function
- Programmer-Defined Function

Program (Before) Program (After)

Send

Tuple Constructor

List Constructor

Pattern Match

Case Expression

Wrapper/Library Function

Programmer-Defined Function

Descend into Function

Converter

Figure 3. Recursive Descent Refactoring Approach

For each relevant statement:

- Convert to Binary
- ETS table equivalent

Program (Before) Program (After)

$s_1$

$s_2$

$s_3$

$s_4$

$s_5$

Figure 4. Conceptual result of applying recursive descent refactoring.

4. Recursive Descent Refactoring

In order to transform lists into binaries or ETS tables, we need to define a composite refactoring. Our recursive descent refactoring consists of a setup phase, converting the input list $xs$, and a recursive phase that inspects each statement in the program, refactoring those that are relevant to $xs$ (Fig. 3). The refactoring is designed to convert the code block that contains $xs$, expanding the set of terms that it looks for as $xs$ is manipulated, and duplicating and converting any programmer-defined functions that are invoked with $xs$ as argument. This results in an “island” of refactored functions whose interface(s) remain the same before and after refactoring (Fig 3).

Although we focus here on translating lists to binaries or ETS tables, the approach can be extended to either the reverse translation, or to translation to other data-types.

Lists are a primitive data-type in Erlang that can contain potentially infinite elements of any type. Lists are copied when they passed between processes, and can be slow when randomly accessing elements. Erlang binaries have a similar syntax to lists, but have more numerous options during construction and pattern matching, where the type and size of individual elements can be specified. Binaries that are larger than 64 bytes are not copied between process heaps, but instead are passed by reference. They are also useful, and sometimes necessary, when interfacing both with GPUs and with other programming languages. Finally, ETS tables are global, mutable term storage providing constant access time to the contained data. There are four types of ETS table and a range of construction options. The following sections present and describe the algorithm of our recursive descent refactoring as it applies to both binaries and ETS tables (Sec. 4.1). We discuss how the criteria set of relevant variables should grow (Sec. 4.2). We highlight the extra step needed as part of the ETS translation (Sec. 4.3). We describe our approach to translate relevant pattern matches for both binary and ETS translations (Sec. 4.4). Finally, we describe our approach to converting and reverting lists to and from binary and ETS, and defining a library of functions equivalent to a subset of the standard list operations (Sec. 4.5).

4.1 General Approach

The programmer selects a variable $xs$ assigned to a list defined in some pattern match, i.e. in a function declaration, a case statement, a receive statement or an irrefutable pattern match. We first use function extraction to lift the statements in the body of the function, case clause, receive clause, or in the case of an irrefutable pattern match. We then use the FastFlow skeleton library, OpenMP and Intel’s Thread Building Blocks, embedding our refactorings in the Eclipse IDE.

convertMerge((Xs, Ys, F1, F2, Name)) ->
  Xs_p = lists:map(
    fun(L) ->
      removeAlpha(L, F1)
    end, Xs),
  Ys_p = lists:map(
    fun(L2) ->
      removeAlpha(L2, F2)
    end, Ys),
  WhiteR = lists:map(
    fun(Col) ->
      convertToWhite(Col)
    end, Xs_p).
  Result = lists:zipwith(
    fun(L1,L2) ->
      mergeTwo(L1, L2)
    end, WhiteR, Ys_p),
  {Result, length(Xs), Name}).

Here, both Xs and Ys are variables bound to lists in a pattern match. Assume we pick Xs, we therefore use the standard function extrac-
tion refactoring over the body of \texttt{convertMerge/1}, and apply the converter function to \texttt{Xs}, resulting in:
\[
\text{convertMerge}((\texttt{Xs}, \texttt{Ys}, \texttt{F1}, \texttt{F2}, \texttt{Name})) 
\rightarrow \text{f(} \text{convert}(\texttt{Xs}), \texttt{Ys}, \texttt{F1}, \texttt{F2}, \texttt{Name})).
\]

\[
f(\texttt{Xs}, \texttt{Ys}, \texttt{F1}, \texttt{F2}, \texttt{Name}) 
\rightarrow \text{Xs}_p = \text{lists:}\text{map}(\text{fun}(L) \rightarrow \text{removeAlpha}(L, \texttt{F1})) \text{end, Xs)}, \text{Ys}_p = \text{lists:}\text{map}(\text{fun}(L2) \rightarrow \text{removeAlpha}(L2, \texttt{F2})) \text{end, Ys)}, \text{WhiteR} = \text{lists:}\text{map}(\text{fun}(\text{Col}) \rightarrow \text{convertToWhite}(\text{Col}) \text{end, Xs}), \text{Result} = \text{lists:}\text{zipwith}(\text{fun} (\text{L1}, \text{L2}) \rightarrow \text{mergeTwo}(\text{L1}, \text{L2})) \text{end, WhiteR, Ys}_p), \text{Result, length(Xs), Name}).
\]

Where new names are required, we have the option of either automatically generating those required names or requesting them of the programmer. For the purposes of this paper we assume the former. Thus, in our above example, the name of \texttt{f} is assumed to be automatically generated and not currently in scope. Whilst we will refer to variables by their names in the following description, as variables may be shadowed, we will use the unique variable identifiers when our criteria set of target variables to avoid refactoring erroneously.

**Definition of terms.** We first define a number of terms we will use as part of the refactoring. A **statement** is defined to be an expression ending in a comma (,), semi-colon (;), full stop (.), or the end keyword (end). A list of one or more statements comprising a **clause**, where we refer to the last statement in the list the **return statement**. An **expression** may be a function description, an anonymous function, a literal value, a variable, a list or tuple constructor, an irrefutable pattern match, a case expression, an if expression, a block expression, or a send (!) or receive expression. An expression that contains one or more other expressions is said to contain **subexpressions**. Where an expression may be a subexpression of another transitively, we refer to those intransitively reachable subexpressions as **immediate subexpressions**. A fold or loop over a list of statements refers to the depth-first traversal of the AST representation of those statements.

### 4.1.1 Refactoring \texttt{f} (Alg. \ref{alg:top_level})

Algorithm \ref{alg:top_level} describes the process by which we refactor \texttt{f}, where \(\Gamma\) is the program environment, and \(\Sigma\) the criterion set of variables relevant to the refactoring (currently the singleton set of \texttt{xs}). We fold over the statements in the body of \texttt{f} (lines 2–3) until the return statement of \texttt{f} is reached. Upon discovery of a variable in \(\Sigma\), e.g. \texttt{xs}, we determine whether \texttt{xs} is at the statement or subexpression level. When \texttt{xs} is a return statement, we apply the revert function (line 7) and when otherwise a statement, we do nothing. The revert function returns the binary/ETS representation of \texttt{xs} to a list representation, and is described further in Sec. 4.3. When \texttt{xs} is a subexpression, we inspect the subexpression \(e_0\) for which \texttt{xs} is an immediate subexpression (hereinafter \(e_0 \text{ contains } e\)). (Line 10.) Algorithm \ref{alg:top_level} describes the process by which \(e_0\) is inspected and refactored. \(e_0\) can have one of six forms:

1. a send expression (lines 3–5),
2. a tuple constructor (lines 6–8),
3. a list constructor (lines 9–15),
4. an irrefutable pattern match (lines 16–22),
5. a case-expression (lines 23–26), or
6. a function application (lines 27–44).

**Send Expression.** When \(e_0\) is a send expression, \texttt{xs} is the message being sent. Without further information where \texttt{xs} will always be sent, we apply the revert function.

**Tuple Constructor.** Similarly, when \(e_0\) is a tuple constructor, and \texttt{xs} one of the elements of the tuple, we apply the reverter. As we do not keep track of the location of \texttt{xs} when \texttt{xs} is nested within an arbitrary subexpression of \(e_0\) we must revert \texttt{xs} here. In future versions of the refactoring, it is desirable to keep track of this information, allowing for more effective and more complex refactorings.

**List Constructor.** When \(e_0\) is a list constructor, \texttt{xs} will either be in the head or the tail position. When in the head position, i.e. being appended to some list, we invoke the revert function (lines 9–11). When in tail position, i.e. being prepended to, we transform the constructor into an equivalent append operation with \texttt{xs} on the right (lines 12–14). As a prepend operation we determine whether any expression containing \(e_0\), should also be a subject of the refactoring (line 14). We describe this process in Sec. 4.4.

**Irrefutable Pattern Match.** When \(e_0\) is an irrefutable pattern match, \texttt{xs} will either be on the left- or right-hand side of the equals sign. When on the left-hand side, we again invoke the revert function on \texttt{xs} (lines 16–18). When on the right-hand side (lines 12–14), potentially binding new variables, the left-hand side is restricted to three generic forms:

1. the empty list, [],
2. the simple list, \([x_1, \ldots, x_n]\), and
3. the split list, \([x_1, \ldots, x_n, y]\).

As the simple list form is syntactic sugar for the split list (where \(y = []\)), we consider only empty and split (hereinafter ‘non-empty’) list cases. In both cases we transform the pattern match expression into an equivalent expression for the target data-type, adding the relevant bound variables to \(\Sigma\). This process is described in Sec. 4.4. When \texttt{xs} is on the right-hand side we again decide
Algorithm: Loop($\Gamma, \Sigma, e_0, e, S_i$)

```
1 switch $e_0$ do
2   case $\nu ! e$ do
3   | Apply Revert
4   end
5   case $\{w\} \land e \in w$ do
6   | Apply Revert
7   end
8   case $[w_1 | w_2] \land e \in w_1$ do
9   | Apply Revert
10  end
11  case $[w_1 | w_2] \land e \in w_2$ do
12  | Apply Cons;
13  | ConsiderContaining($\Gamma, \Sigma, e_0, e, S_i$)
14 end
15 case $e = w$ do
16  | Apply Revert
17 end
18 case $p = e$ do
19  | Apply Pattern Match Translation;
20  | ConsiderContaining($\Gamma, \Sigma, e_0, e, S_i$)
21 end
22 case case $e$ do
23  | Apply Pattern Match Translation;
24  | ConsiderContaining($\Gamma, \Sigma, e_0, e, S_i$)
25 end
26 case case $m : g(args)$ do
27   if isPrimOrLib($m, g, \text{Length}(args)$) then
28     | Apply Translate Primitive/Library Function;
29     | ConsiderContaining($\Gamma, \Sigma, e_0, e, S_i$)
30   else
31     $MFA = \{m, g, \text{Length}(args),\}
32     \text{PosInList}(\text{args}, e);$
33     if $MFA \in \Delta$ then
34       $\Delta = \Delta \cup \{MFA\};$
35       $MFA' = \text{Duplicate and Invoke MFA};$
36       $\delta = \delta \cup \{\text{MFA} \mapsto \text{MFA}'\};$
37       $gdef = \text{FunDef}(MFA');$
38       $\text{RDR}(\Gamma, i, \text{CorrespondingArg}(gdef, \text{args}, e), gdef)$
39     end
40   else
41     $e = e_0;$
42     $e_0 = \text{ContainsAsSubExpr}(e_0);$  
43 end
44 end
45```

Algorithm 2: Inner Recursive Descent Refactoring

whether any expression containing $e_0$ should also be refactored, and any relevant variables added to $\Sigma$.

Case Expression. When $e_0$ is a case expression, $Xs$ must be the conditional clause. This results in an inspection and transformation of the case clause patterns using the same approach as the process under irrefutable pattern matches when $Xs$ is on the right-hand side. We again therefore transform the case expression into an equivalent expression for the target data-type, adding the relevant bound variables to $\Sigma$.

```
Algorithm: ConsiderContaining($\Gamma, \Sigma, e_0, e, S_i$)

1 if $e_0 \neq S_i$ then
2   $e = e_0;$
3   $e_0 = \text{ContainsAsSubExpr}(e_0);$  
4 switch $e$ do
5   case $m : g(args)$ do
6     $MFA = \{m, g, \text{Length}(args),\}
7     \text{PosInList}(\text{args}, e);$
8     if isPrimOrLib($MFA$) then
9       | $\text{ConsiderContaining}(\Gamma, \Sigma, e_0, e, S_i);$
10      else
11        | $\text{ConsiderContaining}(\Gamma, \Sigma, e_0, e, S_i);$
12      end
13 end
14 else
15   | $\text{Loop}(\Gamma, \Sigma \cup \text{BoundListVars}(e), e_0, e, S_i);$
16 end
17 end
18```

Algorithm 3: Continuation Decision Function for Algorithm 1

**Function Application.** When $e_0$ is a function application, that function will be either a primitive or library function (lines 28–31), or a programmer-defined function (lines 31–43). When a primitive or library function, we substitute that function for the target data-type equivalent as described in Sec. 4.3. In the latter case, we find and duplicate that function definition, assigning the duplicate a new name, and refactor the duplicated function. A new $\Sigma$ is derived from that function from the position of $x$s in the argument list. For example, when $e$ is $\text{hd}(Xs)$, it could be replaced with $\text{etslists:hd}(Xs)$. Alternatively, were $e_0 \text{ m:g}(Xs)$, the $m:g/1$ would be duplicated, provided a new name, and refactored. Assume the definition of $m:g/1$ is

$g(Ys) \rightarrow \text{hd}(Ys)$.

Here $\Sigma$ would become the singleton set of $Ys$, and $m:g/1$ the subject of the refactoring, ultimately producing:

$g_1(Ys) \rightarrow \text{etshists:hd}(Ys)$.

Where the original invocation of $m:g/1$ would be changed to invoke the new function in $f$. The function duplication is to ensure that any other functions that invoke $m:g/1$ would not then fail due to the refactoring. As part of future work, we intend to expand this approach to consider the function callgraph of the program to more intelligently determine which functions need to be duplicated and how they should be duplicated. The refactoring maintains a list of seen programmer-defined functions to avoid cycles and unnecessary duplications.

4.2 Bounding the Scope of the Refactoring

As variables are immutable in Erlang all list operations produce a new list. Transforming a single variable is therefore of little practical utility. It is instead desirable to refactor successive containing expressions (ultimately until the statement level is reached), and expand $\Sigma$, where appropriate. For example, in

$Ys = \text{lists:map}(m:g/1, Xs)$

where we are initially concerned with $Xs$ (i.e. $e$), the subexpression $\text{lists:map}(m:g/1, Xs)$ (i.e. $e_0$) would be refactored to the
binary or ETS equivalent map. This map will produce either an updated ETS table or a new binary. As such it is desirable to also consider the irrefutable pattern match that is the assignment to \( Y_s \), and adding \( Y_s \) to \( \Sigma \) for future refactoring. As the assignment to \( Y_s \) is at the statement level, we go no further, instead continue folding over the rest of the statements. Conversely, given

\[ Z = \text{lists:max}(Y_s) \]

and having refactored lists:max/1 to the binary or ETS equivalent, we would not also refactor the assignment to \( Z \) as lists:max/1 does not produce a new or updated version of \( Y_s \).

Algorithm 3 describes the process used to determine whether the expression containing the refactored \( e_0 \) should itself, if it exists, be refactored. With \( e_0 \) now refactored, it becomes our criterion expression \( e \), and \( e_0 \) the expression containing the former \( e_0 \) should the former \( e_0 \) not be a statement (lines 2–4). Unlike in Algorithm 3, we consider the form of \( e \). When \( e \) is any non-function application form, we refactor \( e_0 \) (lines 14–16). When \( e \) is a primitive or library function application, we determine whether to continue based on the classification of that function (lines 8–10). When \( e \) is a programmer-defined function, we determine whether to continue based on the return statement of that function (lines 10–12). For example, given the statement \( g(X_s) \) in a function, and the definitions

\[
g(X_s) \rightarrow \\
\h(X_s).
\]

we first inspect the return statement of \( g/1 \), which here is an invocation to the programmer-defined function \( h/1 \). The return statement of \( h/1 \) is the primitive operation \( h/1 \) which we would refactored. As \( h/1 \) is not a member of the classification of functions which we know to continue, \( h/1 \) inherits this classification, as does \( g/1 \). We do not simply determine whether to refactor \( e_0 \) based on type of the function, as a function may nevertheless return a list, but not a list belonging to \( \Sigma \).

To determine whether \( e_0 \) is refactored when \( e \) is a primitive or library function application, we will define six categories of function: 1. inspection/single-element-lookup, 2. addition, 3. removal, 4. creation, 5. map, and 6. fold. Inspection/single-element-lookup includes functions that select one element or reason about the elements in the list, e.g. \( \text{hd}(1), \text{lists:nth}(2), \text{lists:all}(2) \). Addition includes functions that add to the list, increasing its length, e.g. \( \text{lists:append}/2 \) and \( \text{lists:concat}/2 \). Removal does the opposite of addition serving to decrease the number of elements in the list, e.g. \( \text{tl}(1), \text{lists:subtract}/2, \text{lists:filter}/2 \). Creation includes those functions whose primary purpose is to construct a list in some way, e.g. \( \text{lists:duplicate}/2 \). Map includes those functions that act over every element in a list, changing neither its length nor its ordering, but potentially the value of the elements themselves; e.g. \( \text{lists:map}/2, \text{lists:foreach}/2 \). Lastly, fold includes the remainder of the functions, i.e. functions that arbitrarily act over the length of the list with the possibility to change it into a range of things, e.g. \( \text{lists:foldr}/3 \). Full classification listings for primitive and lists operations can be found alongside our full list of implementations of equivalent binary and ETS functions (Sec. 3.5). If \( e \) is a member of inspection/single-element-lookup, creation, or fold categories, the \( e_0 \) is not refactored. \( e_0 \) is refactored otherwise.

Whilst \( e_0 \) may take any of eight possible forms: 1. a send expression, 2. a tuple constructor, 3. a list constructor, 4. an irrefutable pattern match, 5. a case-expression, 6. an if expression, 7. a receive expression, or 8. a function application, only those forms of \( e \) which can potentially pass on an updated criterion need be considered. Those forms of \( e \) which are not considered all require forms of \( e \) which do not pass on an updated criterion. Send returns an error or \( \text{ok} \). Tuple constructors return a tuple with any nested criteria reverted to list form. If expressions require \( e \) to either be as part of a pattern match, which requires a boolean variable, or as a return statement of the clause, which we handle in Algorithm 3. Receive expressions are similar to if expressions, but allow only valid guard functions in the pattern match, a subset of function applications.

4.3 Additional Requirements for ETS Table Translation

ETS tables are shared mutable term storage. Addition, removal, and map operations therefore make direct and permanent changes to the transformed list \textit{in situ}. This can result in potentially incorrect results should \( x_s \), or some derivative of \( x_s \), be used multiple times in the function body. It is therefore necessary to copy the stored list in the relevant state when the list is used multiple times in the same clause. This may be solved by determining separate ‘traces’ of use, and copying the state of the list when a trace branches. Practically, these traces can be produced in a separate pass, and the AST annotated where copies should be made and mapping of lists to copies. An alternate implementation could use a lookahead system at each addition, removal, and map operation to determine whether a copy need be made. To return to the convertMerge/1 example, we note \( X_s \) is used twice, and therefore should be copied, producing:

\[
f(X_s, Y_s, F_1, F_2, \text{Name}) \rightarrow \\
X_s_1 = \text{etslists:copytable}(X_s), \\
X_s_p = \text{etslists:map}( \\
g(X_s) \rightarrow \\
\text{hd}(X_s). \\
\h(X_s) \rightarrow \\
\text{removeAlpha}(L, F_1) \\
\text{convertToWhite}(C), \\
\text{mergeTwo}(L_1, L_2) \rightarrow \\
\text{etslists:length}(X_s_1, \text{Name}).
\]

We note that in this example, the map operations applied to \( x_s \) and its derivatives do not alter the length of the original list. It is therefore unnecessary to duplicate the list. In future we intend to extend the inspection of usage to more intelligently determine which combinations of functions require a new copy.

4.4 Pattern Match Translation

Recall that pattern matching over lists is limited to two general forms: 1. the empty list, and 2. the non-empty list. Where these patterns can be matched as argument to a function clause, as part of the left-hand side of an irrefutable pattern match, in a case expression clause, or in an if expression clause. All four have unique aspects to their translation, but also share a core approach to the handling of patterns for both ETS tables and binary targets.

General Approach for ETS Tables. We first match on size, then bind or match on individual elements derived from the expected patterns. Given the empty list we match on the size of the table being 0, for example. Similarly, given a pattern expecting \([1,2,3,4,X]\), we first check the size of the table to be 5, then the first four elements of that list 1 to 4 respectively, and lastly bind the fifth element to \( X \). For patterns where the tail of the list is
bound to a variable, e.g. [Hd | T1], we check for the minimum size required by the pattern, here 1, and bind or check the matched elements and tail to the respective variables.

**General Approach for Binaries.** Our approach is determined by the translation scheme. When matching oniolists the built-in binary pattern matching is used directly; e.g. for \([1,2,3,4,B]\), we would match \(<0,1,2,3,4,B>\). For all other binaries, we first match on length of list, and then revert the requested elements into non-binary terms. When binding the tail of a list, we do not revert the tail.

**Translating If Expression Pattern Matching.** When matching a list as part of a function argument declaration, we move the pattern match into a case statement within the body of the function, collating clauses as necessary. For example, given the function

\[
g(\emptyset) ->
\begin{cases}
  \emptyset & \text{if } Xs = \emptyset \\
  g(T) & \text{else if } Xs = [5 | T] \\
  1 + g(T) & \text{else if } Xs = [H | T]
\end{cases}
\]

where we are refactoring the argument to \(g\), we would collapse all clauses matching on the first argument, and refactor according to the case approach.

\[
g(Xs) ->
\begin{cases}
  0 & \text{if } etslists:length(Xs) = 0 \\
  g(T) & \text{if } etslists:length(Xs) = 1 \\
  1 + g(T) & \text{else}
\end{cases}
\]

**Translating If Expression Pattern Matching.** We approach if expressions similarly, producing a series of nested case expressions where each clause is tested as the case condition and case clauses are set to true and false. Where case expressions allow arbitrary conditional expressions, if expressions provide no such conditional expression, and allow only valid guard expressions for pattern matching expressions in its clauses.

**Translating Irrefutable Pattern Matches.** We transform the code in situ, wrapping the transformed code into a block expression when more than one statement is produced. For example, given \([1,2,3,4,B|T] = Xs\), we would produce the block expression:

```erlang
begin
  1 = etslists:nth(1, Xs),
  2 = etslists:nth(2, Xs),
  3 = etslists:nth(3, Xs),
  4 = etslists:nth(4, Xs),
  B = etslists:nth(5, Xs),
  T = etslists:nthtail(5, Xs)
end
```

**Translating Case Expression Pattern Matching** We first lift each clause body into a function, then rebuild the case expression according to the approach described above, invoking one of the clause functions in each eventual branch. For example, the definition of \(g\) above using a case expression is translated into the following:

\[
g(Xs) ->
\begin{cases}
  case etslists:length(Xs) of
    0 -> g_1();
    _ ->
      case etslists:length(Xs) >= 1 of
        'true' ->
          case etslists:head(Xs) of
            5 -> g_2(etslists:tail(Xs));
            H ->
              g_3(H, etslists:tail(Xs))
          end
        end
  end
\end{cases}
\]

Where the case expression is split based on length of the list. For patterns that match a tail, it is necessary to check for lengths equal to or greater than the length of the head. Instead of nested case expressions, it is equally possible to define these using guards. In cases where multiple clauses expect a tail, and the length of the heads are also different, it is necessary to tighten the constraints on the length for those affected branches. Instead of defining top-level functions for each clause body, it is possible to assign them as anonymous functions to variables within the body of \(g\).

**4.5 Converter, Reverter, and Library Function Refactorings**

To support the recursive descent refactoring we need three additional elements for each target data-type: 1. a converter function, to safely transform an existing list data structure into the equivalent data structure of choice; 2. a reverter function, to safely transform a data structure in the target type to an equivalent representation as a list; and 3. a set of functions that are equivalent to the library list operations.

**Converter.** The converter must take at least a single list argument, and return an equivalent data structure representation in the target data-type. For ETS, we define the converter:

```
converter(Xs) ->
  Tab = ets:new(gen_name(), ['ordered_set']),
  insert(Tab, Xs),
  Tab.
```

```
insert(Tab, Xs) when is_list(Xs) ->
  lists:map(fun(X) -> ets:insert(Tab, X) end,
    Xs).
```

As lists have specific order, we use an ordered set type ETS table, providing each element in the original list an index, as this simplifies library operations. Tables are created private rather than public as our refactoring reverts to lists when sending a criterion list between processes. Similarly, tables are created unnamed, instead using their table ID, as the ID will be assigned in lieu of the represented list to \(\text{Tab}\) and any relevant derivatives.

Binaries provide a range of possible construction approaches. The simplest is the use of the built-in \texttt{list_to_binary/1} function, but this is limited to deep lists whose elements are integers in the range of \([0, 255]\). Conversely, Erlang’s bit syntax allows for representation and pattern matching against a variety of types and sizes, yet this requires more information than we can expect to have at time of refactoring. Therefore, a more generic approach must be taken, wherein we first attempt \texttt{list_to_binary/1}, and in case of failure, or a difference in length between binary and list, \texttt{term_to_binary/1}. Producing either a binary of integers in the range \([0, 255]\), or a list of binary elements.
converter(Xs) ->
  try
    case list_to_binary(Xs) of
      Xs_io when length(Xs) == size(Xs) -> Xs_io;
      _ -> error(badarg)
    end;
  catch
    error:badarg ->
      lists_map(fun term_to_binary/1, Xs)
  end.

Reverter. A reverter function is an effective inverse of the converter function, allowing us to generate an equivalent list data structure from a data structure in the target data-type. This is useful when a function returns the target list, or the result of any operations performed on that list, but also in target-equivalent library functions or primitive operations which require a mid-function transformation. The reverter function should take at least one argument, the data structure to be transformed, and return its list equivalent. For ETS, we define the reverter:

to_list(Table) ->
  Sn = fun(X) -> element(2, X) end,
  lists_map(Sn, ets:tab2list(Table)).

Here, we fetch the table contents as a list, stripping each element of its index. As we are using an ordered set type ETS table, no ordering of the list is necessary. For binaries, we define the reverter:

to_list(Xs) when is_binary(Xs) ->
  binary_to_list(Xs);
  to_list(Xs) when is_list(Xs) ->
    lists_map(fun binary_to_term/1, Xs).

As we have two possible approaches for binary translation, we must account for those two representations in reverting to list. Here, both clauses call their respective inverse operations over the passed binary or individual term.

Target-Equivalent Library Function Refactoring. The standard Erlang lists library offers 72 operations, and the standard erlang module a further 12 list-specific operations. Where binaries and ETS tables have their own modules, these are not direct mirrors of lists, with directly equivalent operations and interfaces. Where ets has ets:foldl/3 and ets:foldr/3, but no ets:map/2, and binary has neither fold nor map operations, for example.

In translating between data-types we must consider how each of the 84 functions can be converted to functions that act with equivalent behaviour over the target data-type. A simple approach to this is to first devise a library which exposes a set of functions of equivalent behaviour and interface to lists and relevant primitive operations. This enables primitive/library functions to be swapped for their direct equivalents, without needing to duplicate boilerplate code. For example, given lists:nth/2, we might define

nth(N, Xs) ->
  element(2, hd(ets:lookup(Xs, N))).

for ETS tables, and

tnth(1, <<X, _/binary>>) ->
  X;
ntnth(N, <<X, Xs/binary>>) ->
  nth(N-1, Xs);
nth(N, Xs) ->
  binary_to_term(lists:nth(N, Xs)).

for binaries. ETS’s nth/2 acts as a simple wrapper to the equivalent ets:lookup/2 function, automatically extracting the singleton list result and result body, as we know the form and content of the result. For the binary version, when our translated binary list is a list of binaries, we fetch and translate the element using lists:nth/2 itself, and the single term reverter function. When our binary list is a binary, we recurse over the binary using Erlang bit syntax. A similar approach can be taken for map and fold operations.

Within the context of the recursive descent refactoring, all three of these functions can be directly replaced, requiring no further conditions. Those functions which involve another list, e.g. lists:append/2, the equivalent function and refactoring must also decide how to use the second list. For lists:append/2, we might define an ETS equivalent:

append(Xs, Ys) when is_binary(Xs) ->
  Xs;
append([], Ys) ->
  Ys;
append(Xs, Ys) when is_list(Ys) ->
  N = ets:info(Xs, ’size’) + 1,
  lists_map(fun(X) -> ets:insert(Xs, X) end,
    lists_zip(
      lists:seq(0, N - erlang:length(Ys) - 1),
      Ys));
append(Xs, Ys) when is_list(Xs) ->
  lists_map(fun(X) -> ets:lookup(Xs, X) end,
    lists_zip(
      Xs, Ys)).

appendl(Xs, Ys) when is_binary(Xs) ->
  Xs;
appendl([], Ys) ->
  Ys;
appendl(Xs, Ys) when is_list(Ys) ->
  N = ets:info(Xs, ’size’) + 1,
  lists_map(fun(X) -> ets:insert(Xs, X) end,
    lists_zip(
      lists:seq(0, N - erlang:length(Ys) - 1),
      Ys));
appendl(Xs, Ys) when is_list(Xs) ->
  lists_map(fun(X) -> ets:lookup(Xs, X) end,
    lists_zip(
      Xs, Ys)).

appendr(Xs, Ys) when is_binary(Xs) ->
  Xs;
appendr([], Ys) ->
  Ys;
appendr(Xs, Ys) when is_list(Ys) ->
  N = ets:info(Xs, ’size’) + 1,
  lists_map(fun(X) -> ets:insert(Xs, X) end,
    lists_zip(
      lists:seq(0, N - erlang:length(Ys) - 1),
      Ys));
appendr(Xs, Ys) when is_list(Xs) ->
  lists_map(fun(X) -> ets:lookup(Xs, X) end,
    lists_zip(
      Xs, Ys)).

appendr(Xs, Ys) when is_list(Ys) ->
  N = ets:info(Xs, ’size’) + 1,
  lists_map(fun(X) -> ets:insert(Xs, X) end,
    lists_zip(
      lists:seq(0, N - erlang:length(Ys) - 1),
      Ys));
appendr(Xs, Ys) when is_list(Xs) ->
  lists_map(fun(X) -> ets:lookup(Xs, X) end,
    lists_zip(
      Xs, Ys)).
must split the definition, using the function refactoring to determine which to use. An additional definition where both list arguments are expected to be translated can also be defined, and a possible refactoring result. In convertMerge/1, for example, we might choose to translate both list arguments, requiring equivalent versions of lists:zip/2 to take both ETS tables or binaries. It is similarly possible that different target representations be chosen when refactoring the list arguments, requiring a further definition handling this case. Again, we leave exploration of the possible translation equivalents to future work.

We provide the full list of our equivalent implemented functions, and full list of primitive and library function classifications here:

https://github.com/adbarwell/typetl

We additionally note that whilst strings are lists of characters, and therefore list functions are applicable, and strings come with their own library of functions, we do not consider them here.

5. Examples

We have chosen to use three examples to illustrate our approach. All our experiments have been performed on Titanic, a 2.3GHz AMD Opteron 6176 machine with 24 physical cores and 32 GB of RAM at the University of Pisa, and each experiment repeated 10 times and our plots below presenting the average value of all runs.

5.1 Image Merge

Image Merge takes a list of pairs of images (each represented as a two-dimensional list), and merges each pair. The main computation is defined by the convertMerge/1 function.

\[
\begin{align*}
\text{convertMerge}((Xs, Ys, F1, F2, Name)) & \rightarrow \\
& \text{Xs}_p = \text{lists:map}( \\
& \quad \text{fun}(L) \rightarrow \\
& \quad \text{removeAlpha}(L, F1) \\
& \quad \text{end, Xs}), \\
& \text{Ys}_p = \text{lists:map}( \\
& \quad \text{fun}(L2) \rightarrow \\
& \quad \text{removeAlpha}(L2, F2) \\
& \quad \text{end, Ys}), \\
& \text{WhiteR} = \text{lists:map}( \\
& \quad \text{fun}(Col) \rightarrow \\
& \quad \text{convertToWhite}(Col) \\
& \quad \text{end, Xs}_p), \\
& \text{Result} = \text{lists:zipwith}( \\
& \quad \text{fun}(L1, L2) \rightarrow \\
& \quad \text{mergeTwo}(L1, L2) \\
& \quad \text{end, WhiteR, Ys}_p), \\
& \{\text{to_list(Result), etslists:length(Xs_1), Name}\}.
\end{align*}
\]

There are two refactoring opportunities. Both Xs and Ys are lists. We might choose either, or both, of these to translate into either an ETS or binary equivalent. The code that results if Xs is translated to an ETS table can be found in Section 4.4. When performed as a combination of a task farm and a two-stage pipeline, transmitting images between processes can result in significant memory usage. Indeed, this problem presented itself during testing, causing Titanic to crash when more than four cores were used to merge 100 pairs of images. This alone provides significant motivation to use our recursive descent refactoring to translate both Xs and Ys.

\[
\begin{align*}
\text{convertMerge}((Xs, Ys, F1, F2, Name)) & \rightarrow \\
& \text{f(convert}(Xs), \text{convert}(Ys), F1, F2, Name).
\end{align*}
\]

\[
\begin{align*}
\text{f}(Xs, Ys, F1, F2, Name) & \rightarrow \\
& \text{Xs}_1 = \text{etslists:copytable}(Xs), \\
& \text{Xs}_p = \text{etslists:map}( \\
& \quad \text{fun}(L) \rightarrow \\
& \quad \text{removeAlpha}(L, F1)
\end{align*}
\]

Whilst Xs and Ys have both been translated to ETS tables, this alone will not solve the problem of copying overheads. As the problem lies in the sending of images between processes, here within the farm and pipeline, the benefits of the binaries/ETS representations will be felt across all functions. When merging 100 pairs of 1024x1024 images, we observe that both the binary and ETS representations avoid the excessive memory usage of the list representation. Fig. 5 gives speedups for varying number of cores on Titanic, against the original list sequential version. We observe a maximum speedup of 12.2 for the ETS representation, and 11.1 for binary representation, where the original list sequential version takes on average 600, 211, 668.4µs, or 10.0035 minutes. The slight advantage for the ETS version could be because the built-in fold operation, which the ETS map operation uses, is more efficient than the recursive function that is defined for binaries.

5.2 Matrix Multiplication

We use a relatively naive imperative-style algorithm for matrix multiplication that calculates the dot product for each cell in a matrix from two square matrices. As in our Image Merge example, we select both source matrices, A and B, in the top-level function multiply/2.

\[
\begin{align*}
\text{multiply}(A, B) & \rightarrow \\
& \text{mul(\text{gen_coords(length}(A)), A, B)}.
\end{align*}
\]

As \text{length}/1 returns an integer, and is a member of the inspection category, the refactoring does not descend into \text{gen_coords}/1, merely translating \text{length}/1. The refactoring does however duplicate and descend into \text{mul}/3.
mul(IJs, A, B) ->
  lists:map(fun({I, J}) ->
    mul_1(I, J, length(A), A, B, 0)
  end, IJs).

Here, the map itself is not transformed as it takes IJs as its argument. Within the map function however, both A and B are used again, and are similarly refactored.

mul_1(_, _, 0, _, _, R) ->
r;

mul_1(I, J, K, A, B, Sum) ->
  mul_1(I, J, K-1, A, B, R);

lookup(I, K, A) * lookup(K, J, B) + Sum).

Within mul_1/6, which is a fold, the refactoring again descends into lookup/3. This has the expected definition:

lookup(I, J, M) ->
  lists:nth(J, lists:nth(I, M)).

It is here that the refactoring reaches its maximum depth, transforming the inner invocation of lists:nth/2 to the target equivalent. The outer lists:nth/2 is not changed since the converter function does not recursively convert deep lists, and the library function lists:nth/2 is a member of the single-element-lookup category. As the return statement of lookup/3 and by extension both mul_1/6 and mul_3/3, is not relevant to the refactoring, it determines that nothing more needs to be transformed, resulting in the following code for ETS tables:

multiply(A, B) ->
  multiply_(converter(A), converter(B)).

multiply_(A, B) ->
  mul_(gen_coords(etslists:length(A)), A, B).

mul_(IJs, A, B) ->
  lists:map(fun({I, J}) ->
    {I, J, mul_1(I, J, etslists:length(A), A, B, 0)}
  end, IJs).

mul_1(_, _, 0, _, _, R) ->
r;

mul_1(I, J, K, A, B, Sum) ->
  mul_1(I, J, K-1, A, B, R);

lookup_(I, K, A) * lookup_(K, J, B) + Sum).

lookup_(I, J, M) ->
  lists:nth(J, etslists:nth(I, M)).

We note that the above code does not produce a matrix in the same format as A and B, but a matrix can be constructed using a function that we omit here. This function can itself be refactored in a similar manner if so desired. In parallel versions, a farm is used in place of the map within mul/1.3. We also take the extra step of applying the Introduce Chunking refactoring described in [7], and included in Wrangler, to avoid excessive communications overhead.

When applied to matrices of 100 integers square, we observe maximum speedups of 16.5, 14.8, and 24.3 for lists, binaries, and ETS representations respectively when compared against the original sequential list version which takes on average 7.336, 790.9μs, or 7.337 seconds (Fig. 6). Unlike Image Merge, whilst the copying overhead from lists is the likely cause of the relatively low speedups, it does not cause Titanic to crash at higher core usage. Binaries provide a slight improvement, likely a cause of the reduced copying overhead. ETS tables, however, show superlinear speedups over the sequential execution. This is likely a result of both the avoidance of copying relatively large amounts of data between processes, and the constant-time random-access functionality offered by ETS tables, reducing the time spent extracting data from the tables when compared with the linear time access of both list and binary representations.

5.3 N-Body

The N-Body simulation predicts the movement of bodies over a period of time according to net forces exerted upon them, usually celestial bodies and their respective gravitational forces [24]. The Barnes-Hut algorithm [2] is an approximation technique to solve the N-Body problem, comprising the creation of a quad-tree or oct-tree to approximate net forces and the update of locations and velocities of bodies in the system. In implementing Barnes-Hut, we define the function particle_update/4.

particle_update(Ps, Ps) ->
  NPs = lists:map(fun({X, Y, V_x, V_y, M} = P) ->
    {A_x, A_y} = acceleration(P, Tree),
    update_loc(X, V_x, T, A_x),
    update_loc(Y, V_y, T, A_y),
    update_vel(A_x, V_x),
    update_vel(A_y, V_y),
    M
  end, Ps),
  particle_update(NPs, C, T, N-1).

The first argument, Ps, is a list, and the subject of our refactoring. As Ps is the pattern match differentiating between function clauses, the refactoring does not transform this to use a case expression. As the non-recursive branch has Ps as its return statement, we apply the reverter. In the recursive branch, we change the map operation for its target equivalent (again, here we use ETS for illustrative purposes). In the return statement, i.e. the recursive call, the refactoring knows we have seen particle_update/4 with a refactored list as first argument, and has a reverter as its return statement, so changes the invocation to its refactored version. As build_tree/2 has not been seen by the refactoring, and takes Ps as argument, it is refactored.

build_tree(C, Ps) ->
  M = total_mass(Ps),
  Tree = build_tree(C, Ps),
  Ps = lists:map(fun({X, Y, V_x, V_y, M} = P) ->
    {A_x, A_y} = acceleration(P, Tree),
    update_loc(X, V_x, T, A_x),
    update_loc(Y, V_y, T, A_y),
    update_vel(A_x, V_x),
    update_vel(A_y, V_y),
    M
  end, Ps),
  particle_update(NPs, C, T, N-1).
CP = centre_point(Ps, M),
build_tree_1(C, N, CP, Ps).

Containing three functions, each taking Ps as argument, the refactoring descends into each.

total_mass(Ps) ->
lists:foldr(
    fun(_, _, _, _, M, M_acc) ->
        M_acc + M
    end, 0, Ps).

Each of the three functions within build_tree/2 are similar folds acting over Ps. The refactoring converts the fold operation into the target equivalent, and as the return statement of each of the three functions, the refactoring does not continue their results. We omit the full definitions of centre_point/2 and build_tree_1/4 because of this similarity. No further statements require refactoring, resulting in the following code:

particle_update(Ps, C, T, N) ->
    particle_update(converter(Ps), C, T, N).

particle_update_(Ps, _, _, 0) ->
    to_list(Ps);
particle_update_(Ps, C, T, N) ->
    Tree = build_tree_(C, Ps),
    NPs = etslists:map(
        fun({X, Y, V_x, V_y, M} = P) ->
            {A_x, A_y} =
            acceleration(P, Tree),
            {update_loc(X, V_x, T, A_x),
             update_loc(Y, V_y, T, A_y),
             update_vel(A_x, V_x),
             update_vel(A_y, V_y),
             M}
        end, Ps),
    particle_update_(NPs, C, T, N-1).

build_tree_(C, Ps) ->
    M = total_mass_(Ps),
    CP = centre_point_(Ps, M),
    build_tree_1_(C, N, CP, Ps).

total_mass_(Ps) ->
    etslists:foldr(
        fun(_, _, _, _, M, M_acc) ->
            M_acc + M
        end, 0, Ps).

We observe maximum speedups of 18.8, 17.2, and 19.0, for list, binary, and ETS table representations respectively, when applied to 20,000 bodies and compared against the original sequential list implementation (Fig. 7). Unlike Image Merge and Matrix Multiplication, there is little difference between the three versions. Where list and ETS representations of bodies present no significant detriment, the binary representation suffers likely a result of repeatedly accessing and translating individual elements between binary and standard Erlang terms. The lack of advantage for ETS tables over lists here is likely caused by the nature of the operation: whereas in, e.g., Matrix Multiplication lots of random-access lookup was required, a task ETS tables are well suited for. N-Body consists of folding or mapping over data. This example demonstrates that a change of type does not guarantee better performance. In such circumstances, the programmer might keep the refactored code, or undo the refactoring, itself an automatic process, requiring no further input from the programmer.

6. Related Work

Numerous and varied approaches have previously been proposed to simplify the introduction and management of parallelism [10, 14, 20, 26]. While fully automatic approaches benefit from requiring programmer involvement, they are also limited by the language(s) they work on, and by the transformations they can perform (i.e. they may sometimes perform no optimisation at all). Other approaches hide low-level parallel mechanics from the programmer, and have requirements for their introduction that are unlikely to be met without first restructuring the program [3]. The existing tools and techniques that are designed to simplify this restructuring primarily focus on involving the programmer in decision making, whilst performing menial tasks automatically. These include interactive parallelisation tools such as [3], and previous refactoring approaches applied to parallelism [4, 7]. Recent work on refactoring for parallelism has demonstrated that a refactoring approach can not only aid in the introduction of skeletons over equivalent sequential operations but also that refactoring can enable the introduction of parallelism [3, 4, 21]. Current demonstrations of this approach have been limited to refactorings that focus on rearranging and abstracting patterns from existing code. Even when the program types themselves are targets for the refactoring [19], a change of data-type for the data being manipulated is not considered. Elsewhere, refactoring approaches have been applied to library migration, i.e. where code bases must be changed to reflect a change in library interface, in both Erlang and imperative languages [1, 13, 25].

Whilst this is instructive in implementing interfaces for binary and ETS tables for list equivalent functions, unlike the work considered here, these approaches are also not concerned with fundamental changes in data type. The few explorations of applying refactoring techniques to the translation of data types that have been carried out to date focus only on the abstraction of primitive data types and arrays into objects for imperative Object-Oriented languages [15], or on the abstraction of tuples into records in Erlang [23]. Both systems are useful when abstracting and introducing structure to programs, and are thus instructive in determining what expressions are relevant to a data-type translation refactoring, but do not actually change the now-nested data-types as we have done.
7. Conclusions and Future Work

The correct choice of data structure can make a significant difference to parallel performance. To date, this has been generally a manual process. In this paper we have introduced a new recursive descent refactoring that is designed to automatically translate Erlang list structures and operations to equivalent binary and ETS forms. We have applied our approach to three standard benchmark programs implemented using the Skel parallel skeleton library for Erlang. For these examples, our results show that ETS tables deliver the best parallel performance, but that lists can often be as effective, provided they do not exceed memory limits. Whilst our results might therefore lead us to conclude there is little reason to use binaries, and that all lists should potentially be translated into ETS tables, this would be premature. As demonstrated in [18], binaries are required when interfacing with other languages and hardware, e.g. OpenCL and GPUs. Furthermore, both lists and binaries can be passed across distributed systems with less administrative overhead than ETS tables, nor do they present a natural bottleneck with high frequency accesses across processes as with ETS tables. We therefore instead conclude that the correct choice of data-type is highly dependent upon the specific parallel program and its context, and should not be left to a blind optimisation process as part of a compilation phase.

As future work, we intend to: i) expand our system to handle nested instances of the translated data structure without needing to revert; ii) use a function callgraph to avoid unnecessary duplication of functions; and iii) explore how the use of static and/or dynamic analysis techniques can be used to further inform our refactoring, enabling more idiomatic translations and so more efficient use of target data structures. We also intend to expand our set of target data-types, including useful structures that do not currently have an implementation in the Erlang standard library. We will similarly expand our set of examples to include, e.g., the EMAS system in [3], to further explore the effect of data-type translation on parallel performance. Finally, we intend to formally verify the soundness of our refactorings, perhaps using a similar approach to [2].

References