In Search of a Map: Using Program Slicing to Discover Potential Parallelism in Recursive Functions

Adam D. Barwell and Kevin Hammond
School of Computer Science
University of St Andrews, UK
{adb23,kh8}@st-andrews.ac.uk

Abstract

Recursion schemes, such as the well-known `map`, can be used as loci of potential parallelism, where schemes are replaced with an equivalent parallel implementation. This paper formalises a novel technique, using *program slicing*, that automatically and statically identifies computations in recursive functions that can be lifted out of the function and then potentially performed in parallel. We define a new program slicing algorithm, build a prototype implementation, and demonstrate its use on 12 Haskell examples, including benchmarks from the NoFib suite and functions from the standard Haskell Prelude. In all cases, we obtain the expected results in terms of finding potential parallelism. Moreover, we have tested our prototype against synthetic benchmarks, and found that our prototype has quadratic time complexity. For the NoFib benchmark examples we demonstrate that relative parallel speedups can be obtained (up to 32.93× the sequential performance on 56 hyperthreaded cores).

CCS Concepts • **Software and its engineering** → **Automated static analysis**; **Patterns**; *Parallel programming languages*; *Functional languages*;

Keywords Program Slicing, Recursion Schemes, Pattern Discovery, Parallelism

1 Introduction

Traditional parallelisation techniques commonly consist of low-level primitives and libraries. They require the programmer to manually introduce and manage complex constructs, e.g. threads, communication, and synchronisation. This results in a tedious, difficult, and often error-prone process [22]. Structured parallelisation techniques, such as *algorithmic skeletons*, instead present instances of high-level composable patterns to the programmer [11]. They allow the parallel structure of a program to be expressed as an instance of a pattern, while abstracting over low-level primitives and libraries. Despite the simplified interface, introducing skeletons

can still be a non-trivial task, particularly with large legacy applications [3]. This often requires significant knowledge of the code base, language, and parallelism itself [6]. One common problem arises when control or data dependencies inhibit, or *obstruct*, the introduction of parallelism. This paper introduces a new technique to minimise such *obstructive* dependencies using *program shaping*, so enabling the automatic introduction of algorithmic skeletons.

Recursion schemes describe how data structures can be traversed or constructed [31]. In functional languages, recursion schemes are often implemented as *higher-order functions* such as `map` or `fold`. Moreover, since algorithmic skeletons can be seen as parallel implementations of higher-order functions, it is possible to use recursion schemes as *loci* of potential parallelism [7]. For example, a call to a standard `map` can be replaced with a call to an equivalent parallel implementation, provided that all mapped computations are independent [35]. Unfortunately, there is no guarantee that higher-order functions will be used in all possible instances by the programmer, or that the best higher-order function will be used. It follows that if these higher-order functions can be discovered automatically, then more options for introducing parallelism become available to the programmer.

In [4], we presented a high-level description of how program slicing might be used to discover `map` operations in recursive functions, and demonstrated that those `map` operations can be used as loci of potential parallelism. In this paper, we give a more formal presentation of the approach presented in [4], placing greater emphasis on our classification technique, and apply that technique to a wider range of examples. Moreover, we define a novel *program slicing* [34] algorithm to inspect how the values of variables change between recursive calls. As a basis for our analysis, we define a strict, uncurried, and higher-order expression language. We have implemented our approach in Erlang, and demonstrated its use on 12 examples, including ones derived from the NoFib benchmark suite [32] and functions from the Haskell Prelude. Our prototype correctly discovers all possible mappable operations in our examples. Experimental results on some synthetic benchmarks suggest that our prototype has quadratic time behaviour. Moreover, we show that good relative speedups are possible through the (semi-)automatic introduction of parallel map operations. We achieve a maximum of 32.93× speedup on a 56-core hyperthreaded experimental testbed machine for the matrix multiplication benchmark.

2 Illustrative Example

We illustrate our approach using an example from the the *NoFib* suite, `sumeuler`, that calculates Euler's totient function for a list of integers and sums the results.

```
1  sumeuler :: [Int] -> Int
2  sumeuler xs₀@[] = 0
3  sumeuler xs₀@(x:xs) = euler x + (sumeuler xs)
```

Here, we have unfolded (in the transformational sense) the definition given in the NoFib suite for demonstration purposes. Since this definition of sumeuler is explicitly recursive, parallelising the function definition requires the programmer to know how sumeuler traverses its arguments. This includes understanding how the computations that are performed on the (elements of the) input list affect: *i)* how the input is traversed; and *ii)* the result of sumeuler itself. To take advantage of recursion schemes as loci of potential parallelism, the above definition of sumeuler can be rewritten as an instance of a composition of a foldr and a map,

```
1 sumeuler :: [Int] -> Int
2 sumeuler xs₀ = foldr (+) 0 (map euler xs₀)
```

Multiple parallelisations of, e.g., map are possible. We can use, for example, parList from the Haskell Strategies library [37], parMap from the Par Monad [30], or GPUs via the Accelerate library [10]. The parallel map skeleton can be used to safely plug-replace any standard use of map *provided that all the mapped computations are independent*. In the case of sumeuler, for example, we might use the standard Haskell Strategies library,

```
1 sumeuler :: [Int] -> Int
2 sumeuler xs₀ = foldr (+) 0 (map euler xs₀)
3                         `using` parList rdeepseq
```

where the parallel map skeleton, parList, is parameterised with rdeepseq, a nested strategy that forces each element of the result to be fully evaluated. By cleanly separating the functional definition of the program from its evaluation strategy, it is easy to introduce alternative parallelisations, while still ensuring that the parallel version is functionally equivalent to the original definition. For example, the parallelisation can be further tuned to improve speedups, perhaps by *chunking xs₀* [7], and by taking advantage of the associativity of (+) to sum the individual chunks before returning the sum of all the chunks.

```
1 sumeuler :: [[Int]] -> Int
2 sumeuler xs₀ = sum (map (sum . map euler) xs₀)
3                         `using` parList rdeepseq
4   where sum = foldr (+) 0
```

Fig. 1 shows the raw speedups that we obtain for this program for the interval $[1, n]$ for varying sizes of n on our 56-core hyperthreaded experimental machine, *corryvreckan*. We obtain a maximum speedup of 30.50 for $n = 50000$ on 48 hyperthreaded cores.

In order to take advantage of skeletons, recursion schemes must be used *explicitly*. When recursion schemes are *implicit*, e.g. as in sumeuler, they must first be discovered either manually or automatically. To automatically discover the computations that may be performed as part of a map, we inspect the application expressions that are not recursive calls, hereafter termed *operations*, in the definition of sumeuler. Whether an operation may be lifted into a map depends on two things: *i)* how the arguments to the recursive function differ between the stages of the recursion; and *ii)* the structure of the arguments in the operation. We determine how the values of arguments change between recursive calls using *program slicing*. This is a technique that is used to extract all the program statements that, depending on the algorithm used, *may influence*, or *may be influenced by*, a given statement from the same program, called the *slicing criterion* [34]. A slice is often calculated
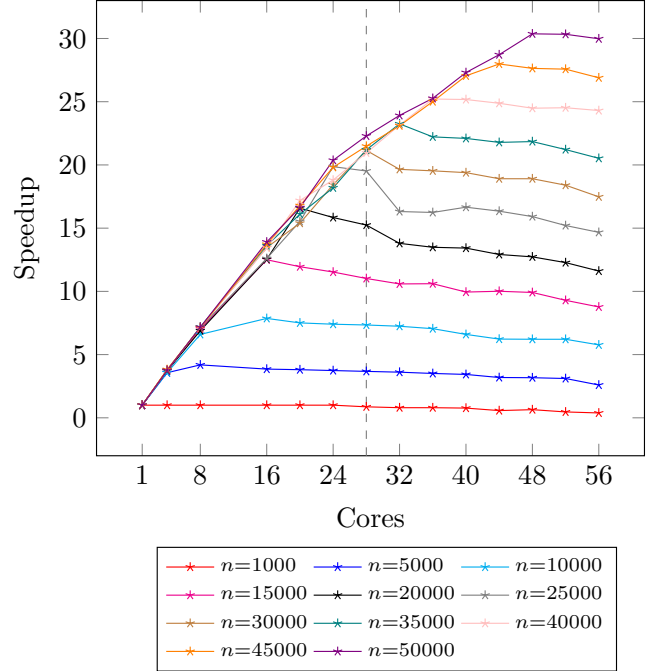
Figure 1. sumeuler, speedups on *corryvreckan* using reported mutator (MUT) time, dashed line shows extent of physical cores.

by inspecting the control and data dependencies of a program. Slicing the explicitly recursive definition of sumeuler, taking x as the criterion, produces:

```
1 sumeuler xs₀(x:_) = euler x + undefined
```

Both the empty-list clause and the recursive call, (sumeuler xs), are removed because they are not influenced by the value of x. Slicing might therefore be thought of as a *narrowing of focus* on only those parts of the program or expression that are of interest, and with all the irrelevant parts stripped away [4]. For the purposes of our approach, we can further narrow the slice to highlight how the values of the arguments change between recursive calls. Our slice only needs to be a set of variables, where inclusion in the set indicates *use*, and an annotated inclusion in the set indicates *update*. For example, the slice of sumeuler, with criterion xs_0, {x, xs}, indicates that both x and xs are *used*; in (euler x) and (sumeuler xs) respectively. The slice also indicates that xs_0 itself is *not* considered to be *updated*. We consider a variable to be *updated*, when its value is *significantly changed*. For a list, we do not consider recursively traversing a list in this way a significant change. We formalise this intuition in Defns. 4.1 and 4.3.

Given such a slice, we now need to decide whether that argument can be used within a map, where each *use* is independent of all other *uses*. If the argument is *used* and *not updated*, e.g. as with xs_0 in sumeuler, then it can also occur safely in a map operation. Conversely, if the argument is *updated* but *not used*, e.g. x in,

```
1 f x ys₀[] = 42
2 f x ys₀(y:ys) = f y ys
```

then it can similarly be updated independently of each input. In either of these cases, we consider the argument to be *clean*. However, when an argument is both *used and updated*, then it indicates that

$$\text{BOOL}_1 \frac{}{\Gamma \vdash true : bool} \quad \text{BOOL}_2 \frac{}{\Gamma \vdash false : bool} \quad \text{INT} \frac{}{\Gamma \vdash \mathbb{Z} : int} \quad \text{VAR} \frac{}{\Gamma \cup \{x : \tau\} \vdash x : \tau} \quad \text{LIST}_1 \frac{}{\Gamma \vdash nil_\tau : list \ \tau}$$

$$\text{LIST}_2 \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : list \ \tau}{\Gamma \vdash cons_\tau \ e_1 \ e_2 : list \ \tau} \quad \text{CASE} \frac{\Gamma \vdash xs : list \ \tau_1 \quad \Gamma \vdash y : \tau_1 \quad \Gamma \vdash ys : list \ \tau_1 \quad \Gamma \vdash e_1 : \tau_2 \quad \Gamma \cup \{y : \tau_1, ys : list \ \tau_1\} \vdash e_2 : \tau_2}{\Gamma \vdash case \ xs \ of \ nil_{\tau_1} \rightarrow e_1, \ cons_{\tau_1} \ (y, \ ys) \rightarrow e_2 : \tau_2} \quad \text{APP} \frac{\Gamma \vdash e_0 : \vec{\tau} \rightarrow \tau_m \quad \Gamma \vdash \vec{e} : \vec{\tau}}{\Gamma \vdash e_0 \ \vec{e} : \tau_m}$$

$$\text{FUN} \frac{\Gamma \cup \{\vec{x} : \vec{\tau}\} \vdash e : \tau_m}{\Gamma \vdash \lambda \ \vec{x} \rightarrow e : \vec{\tau} \rightarrow \tau_m} \quad \text{FIX} \frac{\Gamma \vdash e : (((\tau_2, \ldots, \tau_n) \rightarrow \tau_m), \tau_2, \ldots, \tau_n) \rightarrow \tau_m}{\Gamma \vdash fix \ e : (\tau_2, \ldots, \tau_n) \rightarrow \tau_m}$$

Figure 2. Typing judgements for E, determining simple types in T.

the usage and update of the variable is *not* independent of the other stages of the recursion. Any operation in which an argument is both *used* and *updated cannot* be safely lifted into a map. We therefore consider such an argument to be *tainted*. In sumeuler, xs_0 is the only argument, and it is *used* but *not updated*. It is therefore classified as *clean*, along with its case-defined variables, x and xs.

Having determined whether each argument in our recursive function is *clean* or *tainted*, we can then determine for each operation in the same recursive function whether that operation is independent of successive recursive calls. Such an operation is classified as *unobstructive* when none of its arguments use a tainted variable or include a recursive call. For example, sumeuler has two operations:

1. (euler x); and
2. ((euler x) + (sumeuler xs)).

Since x is classified as *clean*, (euler x) is classified to be *unobstructive*. Conversely, since the second argument to (+) in the second operation is a recursive call, the addition operation is classified as *obstructive*. We can therefore lift (euler x) into a map operation, perhaps by refactoring the original source:

```
1 sumeuler :: [Int] -> Int
2 sumeuler xs0 = sumeuler' (map euler xs0)
3
4 sumeuler' ys0[] = 0
5 sumeuler' ys0(y:ys) = y + ys
```

Here sumeuler' is functionally equivalent to sum, so our final form is:

```
1 sumeuler :: [Int] -> Int
2 sumeuler xs0 = sum (map euler xs0)
```

This can then be easily parallelised using a parallel map skeleton, as seen above.

3 Preliminaries and Assumptions

We illustrate our approach over the simple expression language, E, given in Fig. 3. E is a simple, strict, functional language. Its terms form a common subset of functional languages: boolean constants, true and false; integer constants, $z \in \mathbb{Z}$; variables, *var*; list constructors, nil_τ and $cons_\tau \ e \ e$; case discrimination on lists, case *var* of $nil_\tau \rightarrow e$, $cons_\tau \ var \ var \rightarrow e$; function application, $e \ \vec{e}$; lambda expressions, $\lambda \ \vec{var} \rightarrow e$; and fixpoints, fix e. Constructors are restricted here to cons-lists for simplicity and clarity of presentation, but the approach is extensible to other types, given a definition of *variable update* (Def. 4.1) for that type. Similarly, the approach can be extended to arbitrary types, given that a definition of variable update can be derived for arbitrary constructors. Vector

$$
\begin{array}{lll}
e \in E & ::= & true \\
& | & false \\
& | & \mathbb{Z} \\
& | & var \\
& | & nil_\tau \\
& | & cons_\tau \ e \ e \\
& | & case \ var \ of \ nil_\tau \rightarrow e, \ cons_\tau \ var \ var \rightarrow e \\
& | & e \ \vec{e} \\
& | & \lambda \ \vec{var} \rightarrow e \\
& | & fix \ e
\end{array}
$$

Figure 3. The expression language, E, that forms a base for our analysis.

notation, $\vec{e}$, refers to a non-empty tuple: $\vec{e} \equiv (e_1, \ldots, e_n), n \geq 1$. Tuples are purely meta-syntactic, and are used for notational clarity. In order to simplify our presentation, list constructors and case discriminators are annotated with the (monomorphic) type of the list elements, τ. The corresponding type language, T:

$$
\begin{array}{lll}
\tau \in T & ::= & bool \\
& | & int \\
& | & list \ \tau \\
& | & \vec{\tau} \rightarrow \tau
\end{array}
$$

The typing judgements in Fig. 2 then determine the *well-formedness* of expressions in E with regard to their monomorphic types in T. A statement, $s \in S$, is an assignment.

$$
\begin{array}{lll}
s \in S & ::= & \textbf{def} \ var = \lambda \ \vec{var} \rightarrow e \\
& | & \textbf{def} \ var = fix \ e
\end{array}
$$

Statements appear only at the top level of a program. A variable may be bound either to a function application or to a fixpoint expression. These bindings are then in scope for the duration of the program. A program $p \in P$ is a series of statements.

$$
\begin{array}{lll}
p \in P & ::= & s \\
& | & s \ ; \ p
\end{array}
$$

P can be thought of as an intermediate representation to which, e.g., Haskell or Erlang programs are compiled, similar to Core Haskell. For example, the Haskell definition of sumeuler,

```
1 sumeuler [] = 0
2 sumeuler (x:xs) = euler x + (sumeuler xs)
```

can be translated into the term in P given in Fig. 4. In this paper, we will use Haskell syntax for our examples in order to improve readability. All our examples can be translated into P following a similar principle to the above example. Our approach will inspect only the

$$\textbf{def } sumeuler = \text{fix } \lambda\,(f, xs_0) \;\rightarrow$$
$$\text{case } xs_0 \text{ of}$$
$$\text{nil}_\pi \rightarrow 0,$$
$$\text{cons}_\pi\,(x,\;xs) \rightarrow plus(euler(x), f\,(xs))$$

Figure 4. Definition of `sumeuler` in E.

code provided and does not presume to predict possible compiler optimisations, e.g. fusions or worker-wrapper transformations. When P is used as an intermediate representation, our technique can be applied either before or after such compiler optimisations.

Where pattern matching is used, we will use as-patterns to indicate the implicit list variables; e.g.

```
1 sumeuler xs₀@[] = 0
2 sumeuler xs₀@(x:xs) = euler x + (sumeuler xs)
```

For clarity, all the variables in our examples will be consistent across function clauses. All variables are assumed to be unique under α-conversion, at both the statement and expression levels. Type environments, Γ, are defined to be a set of bindings of variables to types:

$$\Gamma \quad \in \quad \{\, var : T \,\}$$

As usual, all values in the domain of Γ are assumed to be unique, and $\Gamma(x)$ denotes the type τ of a variable x in Γ, such that $\exists \tau \in T, (x : \tau) \in \Gamma$. For a given program p, the *program environment*, Γ_p, contains all the variables in p.

Definition 3.1 (Program Environment, Γ_p). Given some program p and the set of variables $X \subseteq var$ that occur (either free *or* bound) in p, we define an environment Γ_p to be a set such that $\forall x \in X, \exists \tau \in T, x : \tau \in \Gamma_p$

The above `sumeuler` definition, for example, has the Γ_p:

$$\Gamma_p = \{xs_0 : list\ int,\ x : int,\ xs : list\ int,$$
$$f : (list\ int) \rightarrow int,\ euler : (int) \rightarrow int, \dots\}$$

We omit the variables and types of *euler* for clarity. In the remainder of this paper, we will assume that for all given variables $x \in \Gamma_p$.

It is useful to define the notion of *subexpressions* in E, since subexpressions are a key element in both slicing and classification definitions.

Definition 3.2 (Subexpression). Given two expressions e, e', say that e' is a subexpression of e (denoted $e' \ll e$) when

$$\frac{e' = e}{e' \ll e} \qquad \frac{e' \ll e_1 \lor e' \ll e_2}{e' \ll \text{cons}_\tau\ e_1\ e_2}$$

$$\frac{e' \ll e_1 \lor e' \ll e_2}{e' \ll \text{case } x \text{ of nil}_\tau \rightarrow e_1,\ \text{cons}_\tau\ x'\ x'' \rightarrow e_2}$$

$$\frac{\exists i \in [0, n], e' \ll e_i}{e' \ll e_0\ \vec{e}} \qquad \frac{e' \ll e}{e' \ll \lambda\,\vec{x} \rightarrow e} \qquad \frac{e' \ll e}{e' \ll \text{fix } e}$$

Subexpressions form a partial order relation.

We will refer to any application that is a subexpression of a fixpoint and that is not a recursive call as an *operation*. For example, the fixpoint expression in Fig. 4, has two operations: *plus* and *euler*. The application subexpression, $f\,(xs)$ is *not* an operation because it is a recursive call.

Functions are introduced using a λ-expression. They are always pure, are uncurried, and are never partially applied. They may, however, be higher-order. Any recursive (function) definitions are always introduced using an explicit fixpoint expression, e.g.:

$$\text{fix}\,(\lambda\,(f, xs) \;\rightarrow\; f(xs)).$$

The form of recursion is not otherwise restricted; general recursive forms are allowed, for example.

Lists are defined to be an ordered collection of elements, where those elements are accessed *via* case-expressions. As shown by the typing rules of Fig. 3, case discrimination is restricted to lists of some type τ. We assume the existence of a built-in functions (e.g. *if, eq, plus*) for discrimination and operations on integers and booleans. Case-expressions can be extended to other types, given an additional check on the type of the discriminated variable in the relevant definitions. To simplify our presentation, we limit case-expressions here. In the non-nil branch of a case-expression, new variables are bound respectively to the first element in the list (i.e. the head) and to the remainder of the list (i.e. the tail). A corresponding Reachability Relation is defined for each case-expression.

Definition 3.3 (Reachability Relation). Given a program p, a program environment, Γ_p, and a case-expression in p, $e = $ case xs_0 of nil$_\tau \rightarrow e_1$, cons$_\tau$ x $xs \rightarrow e_2$, we say that $x \triangleleft_p xs_0$ and $xs \triangleleft_p xs_0$. The transitive closure of the reachability relation is defined such that $z \triangleleft_p^+ xs_0$ when $\exists y, z \triangleleft_p^+ y \land y \triangleleft_p xs_0$. The *reflexive*-transitive closure of the reachability relation is defined such that $y \triangleleft_p^* xs_0$ when $\exists y, y \triangleleft_p^+ xs_0 \lor y = xs_0$.

For example, $x \triangleleft_p xs_0$, $xs \triangleleft_p xs_0$, and $xs_0 \triangleleft_p^* xs_0$ all hold for the case expression in `sumeuler`. The Reachability Relation will be used to calculate the *program slice* for a given expression and variable. It is also useful to know when a *cons*-expression reconstructs xs_0 (e.g. (x:xs) in `sumeuler`), so that xs_0 can be included in the slice. The syntactic equivalence relation for lists is used to detect such list reconstructions.

Definition 3.4 (Syntactic Equivalence for Lists). Given some program p; argument x of type *list* τ, and a *cons*-expression $e = $ cons$_\tau$ e' e'', we say that e is syntactically equivalent to x, denoted $e \equiv x$, when $\exists y, \exists ys, e' = y \land e'' \equiv ys \land y \triangleleft_p x \land ys \triangleleft_p x$.

For example, where xs_0 is case-split in `sumeuler` into x and xs, xs_0 is syntactically equivalent to a *cons*-expression with x as the first argument and xs as the second argument; i.e. $xs_0 \equiv $ cons$_\tau$ x xs.

4 Determining Obstructiveness

Each operation is inspected to determine whether that operation is *obstructive*. An *obstructive* operation is an operation that has either: *i)* any arguments whose subexpressions contain a recursive call; or *ii)* any arguments that are both *used* in the body of the fixpoint expression and whose value is changed *significantly* in any recursive call. All other operations are classified as *unobstructive*. So, for example, (euler x) in `sumeuler` is classified as *unobstructive* because euler is a unary function that takes only the head of the list as its argument. Conversely, the infix addition operation in `sumeuler` is classified as *obstructive* because it takes the result of a recursive call as an argument. Obstructiveness is defined formally in Def. 4.6. The nature of *significance* is defined below.

4.1 Variable Usage and Update

From the intuitive definition of obstructiveness above, all variables in Γ_p for some fixpoint expression, $e = \text{fix}\,\lambda\,(f, x_1, \ldots, x_n) \to e'$, must be first inspected to classify operations in e as (un)obstructive. Each variable $x \in \Gamma_p$ is classified as *used*, *updated*, or both, in e.

Although all the variables in P are immutable, and each function application binds new values to each of its arguments, the value of some variable x is considered to be (potentially) *changed* in e when x is the ith argument to f and there exists some recursive call to f in e' or any of its subexpressions. That is, the differences in the value of a specific argument between successive recursive calls are considered to be changes to the corresponding bound variables. We extend this notion with the concept of *significant* change. Intuitively, a change in value is considered *significant* when for some recursive call, $f(e_1, \ldots, e_n)$, the ith argument, e_i, is not: *i)* a variable that is reachable from x; *ii)* syntactically equivalent to x; nor *iii)* a cons-expression that prepends an expression to x (or its syntactic equivalent) that contains no subexpression that is a variable reachable from x. Variables whose value is significantly changed in this way are classified as *updated*.

Definition 4.1 (Variable Update). Given a program p, a program environment Γ_p, a fixpoint expression $e = \text{fix}\,(\lambda\,(f, x_1, \ldots, x_n) \to e')$ in p, and an argument x, where x is the ith argument to f, x is considered to be updated in e (denoted $\bar{x} \sim_x e$) when there exists a recursive call, $f(e_1, \ldots, e_n) \ll e'$ such that $\neg(e_i \mathbin{\prec_p^*} x) \wedge (e_i \not\equiv x) \wedge \neg(e_i = \text{cons}_\tau\, e_k\, e_l \wedge e_l \equiv x \wedge \nexists y \ll e_k, y \mathbin{\prec_p^*} x)$

Usage is a simpler concept: a variable is considered to be emphused when it occurs as a subexpression of a fixpoint expression. There is one exception to this: when a subexpression is the ith argument to a recursive call then it is not considered to be *used*. We first define the notion of *variable-usage escapement*. This is to avoid the *used* classification of x, and ultimately the membership of x in the slice, when e_i is a x itself or when prepending to x. For example, it avoids the erroneous classification of x in:

```
1 f a x = if a < 5
2         then f (a+1) x
3         else f (a-1) a
```

Here, in Line 3, x is correctly classified as being *updated*. In Line 2, x is not updated, but it should also not be considered to be *used*, since x is necessary as an argument to the recursive call in order to retain the value of x.

Definition 4.2 (Variable-Usage Escapement). For some fixpoint expression $e = \text{fix}\,(\lambda\,(f, x_1, \ldots, x_n) \to e')$, $\forall f(e_1, \ldots, e_n) \ll e'$, and for some $i \in [1, n]$, given x, the ith argument to f, we substitute the special symbol ε for e_i whenever $e_i = x$ or $e_i \equiv x$. When $e_i = \text{cons}_\tau\, e_k\, e_l \wedge e_l \equiv x \wedge \nexists y \ll e_k, y \mathbin{\prec_p^*} x$ holds, we substitute ε for e_l. Variable-usage escapement of e with respect to x is denoted $e \setminus_\varepsilon x$.

In the above definition of f, for example, the x in Line 2 would be substituted for ε.

```
1 f a x = if a < 5
2         then f (a+1) ε
3         else f (a-1) a
```

We can now define variable *usage*.

Definition 4.3 (Variable Usage). Given some fixpoint expression $e = \text{fix}\,(\lambda\,(f, x_1, \ldots, x_n) \to e')$, and a variable x where x is the ith argument to f, a variable y is considered to be used in e (denoted $y \sim_x e$) when $y \mathbin{\prec_p^*} x$ and y exists as a subexpression to the variable-usage escaped e; i.e. $y \ll (e \setminus_\varepsilon x)$.

To illustrate this, consider the following functions.

```
1 f a xs0@[] = a
2 f a xs0@(x:xs) = f x xs
3
4
5 g xs0@(x:xs) ys0@(y:ys) = g (x:xs) (x:ys)
6 g xs0 ys0 = xs0
7
8 h xs0@(x:xs) ys0@(y:ys) = h (y:(x:xs)) (y:(y:ys))
9 h xs0 ys0 = xs0
```

For f, the value of a is *updated* since $\neg(\text{x} \mathbin{\prec_p^*} \text{a})$. However, the list argument is *not updated* since xs is reachable from xs_0; i.e. $\text{xs} \mathbin{\prec_p^*} xs_0$. All variables, excluding xs_0, in the program environment of f are considered to be *used*: a is used in Line 1; x in the first argument of the recursive call; and xs in the second argument of the recursive call. For g, xs_0 is *not updated* since (x:xs) is syntactically equivalent to xs_0; i.e. $\text{cons}_\tau\, \text{x}\, \text{xs} \equiv xs_0$. However, ys_0 is *updated* since (x:ys) is not syntactically equivalent to ys_0. All variables apart from y and xs are considered to be *used* in g: xs_0 is used both in Line 6 and as the semantically equivalent first argument to the recursive call; x and ys are used in the second argument of the recursive call. Finally, for h, xs_0 is *not* updated, since y (which is not reachable from xs_0) is prepended to xs_0. However, ys_0 is *updated* since y is prepended to ys_0 *and* y is reachable from ys_0. y, xs_0, and ys_0 are all considered to be *used* in h: xs_0 in Line 9 and the *cons* in the first argument of the recursive call; ys_0 in the *cons* in the second argument of the recursive call; y in the *cons* of both arguments of the recursive call.

4.2 Slicing Algorithm

Intuitively, a slice $\Sigma^{e|x}$ of an expression e with criterion x is a set of variables that indicates whether a variable y is *used* in e and whether x is *updated* in e, denoted by the annotation $\bar{x}$. A slice can be used to categorise variables, and ultimately to determine the obstructiveness of operations.

Definition 4.4 (Slice). Given some program p, the program environment Γ_p, a fixpoint expression $e = \text{fix}\,\lambda\,(f, x_1, \ldots, x_n) \to e'$, and a variable x, where x is the ith argument to f, we say that the slice of e with criterion x, denoted $\Sigma^{e|x}$, is the set of variables such that $\forall y \in \Sigma^{e|x}, (y \sim_x e) \vee (y = \bar{x} \wedge \bar{x} \sim_x e)$.

To illustrate the slicing relation, recall the definition of sumeuler,

```
1 sumeuler xs0@[] = 0
2 sumeuler xs0@(x:xs) = euler x + (sumeuler xs)
```

Considering each variable in sumeuler: both $\text{x} \in \Sigma^{\text{sumeuler}|xs_0}$ and $\text{xs} \in \Sigma^{\text{sumeuler}|xs_0}$ hold since both x and xs are reachable from xs_0 (i.e. $\text{x} \mathbin{\prec_p^*} xs_0$ and $\text{xs} \mathbin{\prec_p^*} xs_0$) and both x and xs are *used* in Line 2; conversely, *neither* $xs_0 \in \Sigma^{\text{sumeuler}|x}$ *nor* $\bar{xs}_0 \in \Sigma^{\text{sumeuler}|xs_0}$ hold, since xs_0 is not considered *used*, and xs_0 is not considered *updated*, in sumeuler. For the definition of sum:

$$\text{BOOL}_1 \frac{}{\Gamma_p, f, x, i \vdash \text{true} : \emptyset} \quad \text{BOOL}_2 \frac{}{\Gamma_p, f, x, i \vdash \text{false} : \emptyset} \quad \text{INT} \frac{}{\Gamma_p, f, x, i \vdash \mathbb{Z} : \emptyset} \quad \text{VAR}_1 \frac{y \vartriangleleft_p^* x}{\Gamma_p, f, x, i \vdash y : \{y\}} \quad \text{VAR}_2 \frac{\neg(y \vartriangleleft_p^* x)}{\Gamma_p, f, x, i \vdash y : \emptyset}$$

$$\text{LST}_1 \frac{}{\Gamma_p, f, x, i \vdash \text{nil} : \emptyset} \quad \text{LST}_2 \frac{\text{cons}_\tau \, y \, ys \equiv x}{\Gamma_p, f, x, i \vdash \text{cons}_\tau \, y \, ys : \{x\}} \quad \text{LST}_3 \frac{\Gamma_p, f, x, i \vdash e_1 : \Sigma_1 \quad \Gamma_p, f, x, i \vdash e_2 : \Sigma_2 \quad \text{cons}_\tau \, e_1 \, e_2 \not\equiv x}{\Gamma_p, f, x, i \vdash \text{cons}_\tau \, e_1 \, e_2 : \Sigma_1 \cup \Sigma_2}$$

$$\text{CASE} \frac{\Gamma_p, f, x, i \vdash e_1 : \Sigma_1 \quad \Gamma_p, f, x, i \vdash e_2 : \Sigma_2}{\Gamma_p, f, x, i \vdash \text{case } ys \text{ of nil}_\tau \rightarrow e_1, \; \text{cons}_\tau \, z \, zs \rightarrow e_2 : \Sigma_1 \cup \Sigma_2}$$

$$\text{REC-APP} \frac{\begin{array}{c} \forall j \in [1, i) \cup (i, n], \; \Gamma_p, f, x, i \vdash e_j : \Sigma_j \\ (((e_i = x \vee e_i \equiv x) \rightarrow \Sigma_i = \emptyset) \vee ((e_i \vartriangleleft_p^+ x) \rightarrow \Sigma_i = \{e_i\}) \vee \\ ((\exists e_k, \exists e_l, \, e_i = \text{cons}_\tau \, e_k \, e_l \wedge (\nexists v, (v \ll e_k \wedge v \vartriangleleft_p^* x)) \wedge e_l \equiv x) \rightarrow \Sigma_i = \emptyset) \vee \\ (\neg(e_i \vartriangleleft_p^* x \vee (\exists e_k, \exists e_l, \, e_i = \text{cons}_\tau \, e_k \, e_l \wedge (\nexists v, (v \ll e_k \wedge v \vartriangleleft_p^* x)) \wedge e_l \equiv x) \vee e_i \equiv x) \wedge \Gamma_p, f, x, i \vdash e_i : \Sigma \rightarrow \Sigma_i = \{\bar{x}\} \cup \Sigma)) \end{array}}{\Gamma_p, f, x, i \vdash f(e_1, \ldots, e_n) : \bigcup_{j \in [1, n]} \Sigma_j}$$

$$\text{APP} \frac{\forall j \in [0, n], \; \Gamma_p, f, x, i \vdash e_j : \Sigma_j \quad e_0 \neq f}{\Gamma_p, f, x, i \vdash e_0(e_1, \ldots, e_n) : \bigcup_{j=0}^n \Sigma_j} \quad \text{FUN} \frac{\Gamma_p, f, x, i \vdash e : \Sigma}{\Gamma_p, f, x, i \vdash \lambda(y) \rightarrow e : \Sigma} \quad \text{FIX} \frac{\Gamma_p, f, x, i \vdash e : \Sigma}{\Gamma_p, f, x, i \vdash \text{fix } e : \Sigma}$$

Figure 5. Inference rules to calculate the slice $\Sigma^{e|x}$ for an expression e with criterion x.

```
1 sum x ys0@[] = x
2 sum x ys0@(y:ys) = sum (x+y) ys
```

$x \in \Sigma^{\text{sum}|x}$ holds since x is *used* in both Lines 1 and 2. $\bar{x} \in \Sigma^{\text{sum}|x}$ also holds since x is *updated* in the recursive call in Line 2 by the (x+y) operation. As with sumeuler, both $y \in \Sigma^{\text{sum}|ys_0}$ and $ys \in \Sigma^{\text{sum}|ys_0}$ hold since both y and ys are considered to be *used* in Line 2. Conversely, *neither* $ys_0 \in \Sigma^{\text{sum}|ys_0}$ *nor* $\bar{ys_0}$ hold since ys_0 is *not* considered to be *used* or *updated* in sum. The slices of sumeuler and sum are:

$$\begin{aligned} \Sigma^{\text{sumeuler}|xs_0} &= \{x, xs\} \\ \Sigma^{\text{sum}|x} &= \{x, \bar{x}\} \\ \Sigma^{\text{sum}|ys_0} &= \{y, ys\} \end{aligned}$$

The slice $\Sigma^{e|x}$ is calculated using the inference rules from Fig. 5. For clarity of notation, we will write $\Sigma^{e|x}$ as Σ since neither e nor x can be changed within a slicing operation. Showing that $\Sigma = \Sigma^{e|x}$ remains future work. Judgements are in the form:

$$\Gamma_p, f, x, i \vdash e : \Sigma$$

where Γ_p is the environment for some program p; f is the name of the fixpoint function being sliced; x is the slicing criterion that is declared as the ith argument of f; e is the expression in p that is being sliced; and Σ is the resulting slice. For literal expressions and variables that are not reachable from x, e produces an empty slice, represented by the rules BOOL$_1$, BOOL$_2$, INT, VAR$_2$, and LST$_1$. A variable that is reachable from x, as a usage of x, produces the slice containing that variable (VAR$_1$). Cons-expressions that are syntactically equivalent to x produce the slice containing x itself (LST$_2$). All other expressions that are not recursive calls produce the union of the slices of their subexpressions, as stated in the rules LST$_3$, CASE, APP, FUN, and FIX. Finally, REC-APP determines whether x is updated in a recursive call, producing the appropriate slice. REC-APP has two main premises that: *i)* slice all subexpressions that are passed to f, apart from the ith argument; and *ii)* determines whether x is updated. The second premise is a disjunction of four implications: *i)* when e_i is either x itself, or is syntactically equivalent to x; *ii)* when e_i is a variable that is defined via case-discrimination on x; *iii)* when x is prepended by some expression e_k that does not contain a subexpression that is a variable that is *reachable* from x; or *iv)* when x is considered *updated*. In the first case, the slice, Σ_i, for the ith argument to f, e_i, is the empty set since the argument preserves the value of x for the next recursive call. In the second case, Σ_i is the singleton set containing e_i itself since the value of x is changed, and a case-derived variable is *used*, but x is not considered to be *updated*. In the third case, Σ_i is the slice of the subexpression that is prepended to x. Finally, in the fourth case, Σ_i is the slice of e_i and x is considered to be *updated*.

We conjecture that the slicing algorithm in Fig. 5 is sound with respect to the slicing definition, Def. 4.4, such that our algorithm produces a slice whose members are only those variables that are considered to be *used* or *updated* in e with respect to x. Similarly, we conjecture that slices are unique for each e and x. We defer the proofs of soundness and uniqueness to future work.

4.3 Classifying Variables

Variables can be classified as: *global*, *clean*, or *tainted*. Global variables are those that are in scope in e, but which are declared and bound outside of e. They may be *used*, but cannot be *updated* during the evaluation of e; they are therefore treated as literals. Variables that are classified as either *clean* or *tainted* are ones that are either defined in the fixpoint function (i.e. $x_1, \ldots, x_n$), or in case-subexpressions of e', where whenever $\forall v, w \in \Gamma_p$ such that $v \vartriangleleft_p^+ w$, it follows that v has the same classification as w.

Definition 4.5 (Variable Taint). Given a program p, program environment Γ_p, a fixpoint expression $e = \text{fix}(\lambda(f, x_1, \ldots, x_n) \rightarrow e')$ in p, a variable x, and a slice $\Sigma^{e|x}$ of e with criterion x, then x is classified as tainted when $\bar{x} \in \Sigma^{e|x} \wedge x \in \Sigma^{e|x}$. The variable is otherwise classified as clean.

4.4 Classifying Operations

Once all the arguments of e are classified as *global*, *clean*, or *tainted*, we can proceed to classify the non-recursive application subexpressions of e, i.e. the *operations* in e. Those *operations* that take one or more variables that are classified as *tainted* are themselves classified as *obstructive*. All other *operations* are classified as *unobstructive*.

Example	RFs	Ops	RCs	Args	Unobstructive Ops		Obstructive Ops		Time	σ
					Actual	Found	Actual	Found	(μs)	(μs)
Data.List	18	22	20	31	5	5	17	17	6797.22	409.14
elem	1	2	1	2	1	1	1	1	176.28	26.51
list-ackermann	1	0	3	2	0	0	0	0	196.46	20.65
gcd	1	3	1	2	0	0	3	3	178.46	48.93
hanoi	1	5	1	1	0	0	5	5	156.16	39.50
k-means	4	15	5	14	4	4	11	11	646.30	60.90
quicksort	3	6	5	5	2	2	4	4	350.94	17.98
swaterman	2	10	2	10	0	0	10	10	420.92	18.44
sumeuler	14	47	14	25	16	16	31	31	2171.36	63.95
matmult	8	23	7	16	6	6	17	17	1195.62	63.23
queens	2	11	2	6	2	2	9	9	487.04	24.48
sudoku	5	9	5	8	3	3	6	6	519.66	38.68

Figure 6. Examples run through prototype implementation. Times are an average of 50 runs.

Definition 4.6 (Operation Obstructiveness). Given some program p, some program environment Γ_p, and some fixpoint expression $e = \text{fix}\,(\lambda\,(f, x_1, \ldots, x_n) \rightarrow e')$ in p, an operation $g\,(e_1, \ldots, e_m) \ll e'$, when $g \neq f$, is classified as *obstructive* when $\exists i \in [1, m]$ such that $f\,(\vec{e_f}) \ll e_i \vee \exists y, y \ll e_i$ where y is classified as *tainted*. The operation is classified as *unobstructive* otherwise.

5 Examples

In this section, we demonstrate the use of our approach on 12 examples, including *sumeuler*, *matrix multiplication*, *n-queens*, and the *Smith-Waterman* algorithm. We give parallel speedup results for *sumeuler*, *matrix multiplication*, and *n-queens*. Our source code is available at https://adb23.host.cs.st-andrews.ac.uk/fhpc17-parallel.zip. Our results show that our approach can discover map operations that lead to real performance improvements. Our speedup results are given as an average of five runs on *corryvreckan*, a 2.6GHz Intel Xeon E5-2690 v4 machine with 28 physical cores and 256GB of RAM. This machine allows *turbo boost* up to 3.6GHz, and supports automatic dynamic frequency scaling between 1.2–3.6GHz. Speedups are given in terms of the reported *mutator* time, i.e. the amount of time that is spent solely on executing the program. This gives an indication of real parallel performance and not, e.g., any scheduling or garbage collection costs. We use a separate, standard desktop machine, *neptune*, to test our prototype implementation. *neptune* is a 2.7GHz Intel Core i5 machine with 8GB of RAM, running Mac OS X 10.11.6 and Erlang 19.2.3. The parallel speedup results, including speedup graphs in Figs. 1, 8, and 9 are reiterated from [4], where we expand upon the results reported here. sumeuler, queens, and matmult have all been compiled using GHC 7.6.3 on Scientific Linux version 3.10.0. We compiled the examples using the flags: -feager-blackholing, -threaded, -rtsopts, and -O. We found these gave the best general parallel performance.

We have implemented our inference system in Fig. 5 in Erlang, providing a parser for our expression language and a classifier for operations. For a given program p, we first produce the environment. Then for each fix expression $e = \text{fix}\,\lambda\,(f, \vec{x}) \rightarrow e'$ in p, we slice e with respect to each argument $x \in \vec{x}$. Each x is classified as either *clean* or *tainted*. This is then used to classify each operation in e either *obstructive* or *unobstructive*. Our prototype
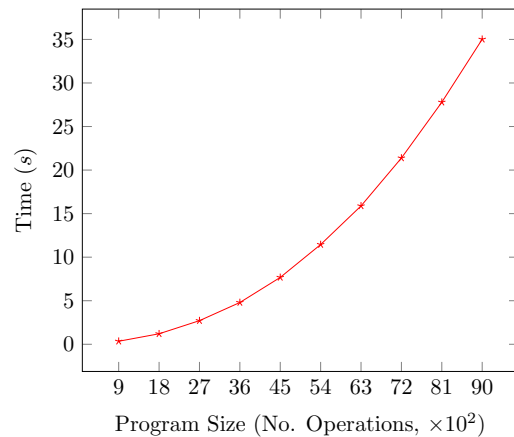

Figure 7. Prototype execution times for a program with varying sizes of input. Our implementation is quadratic.

implementation can be found at https://adb23.host.cs.st-andrews.ac.uk/fhpc17-artefact.zip.

The NoFib benchmarks have all been translated from Haskell; if-expressions have been translated as application expressions, and so count as operations. Our other examples, including the 18 functions from the Haskell Prelude Data.List library, may not all benefit from parallelisation, but serve as a demonstration of our approach. The 18 examples chosen from Data.List are: and, or, append, foldl, foldr, init, intersperse, last, length, map, maximum, replicate, reverse, scan, heads, subsequences, tails and transpose. These are a representative subset of the functions in the library; the remaining functions are similar to these. All our (translated) example code can be found at https://adb23.host.cs.st-andrews.ac.uk/fhpc17-examples.zip.

We give an overview of our results in Fig. 6, where gcd refers to the greatest common denominator function, hanoi refers to the Towers of Hanoi puzzle, and swaterman refers to the Smith-Waterman algorithm. The list-ackermann example is a reimplementation of the standard Ackermann function that traverses lists.

```
1 list-ackermann as₀@[] bs₀ = (1:bs₀)
2 list-ackermann as₀@(a:as) bs₀@[] =
3   list-ackermann as [1]
4 list-ackermann as₀@(a:as) bs₀@(b:bs) =
5   list-ackermann as (list-ackermann a bs)
```

Our results show that all operations are correctly classified. For each example, we give: the number of recursive functions that it contains (RFs), the total number of operations (Ops), the total number of recursive calls (RCs), the total number of arguments to all recursive functions (Args), the expected number of obstructive and unobstructive operations (Actual), those that are found by our prototype (Found), the average execution time for our prototype of 50 runs on *neptune* for that example (Time), and the standard deviation (σ) of those times.

As a synthetic benchmark, we have also applied our prototype to programs with varying input sizes, measured in the number of operations, by duplicating the translated SumEuler module *m* times. As the translated SumEuler has 9 operations, the total number of operations is $n = m \times 9$. Fig. 7 shows the average time taken of five runs, in seconds, by our prototype on *neptune*. Our classifier takes a minimum of $0.35s$ for $n = 900$, with a standard deviation of $0.03s$, and a maximum of $35.02s$ for $n = 9000$, with a standard deviation of $0.58s$. Our prototype implementation runs in quadratic time with respect to *n*. The total time taken to classify each of our examples on *neptune* are all low, as shown in Fig. 6. The longest our prototype takes to classify one of the examples in Fig. 6 is $6.80ms$.

5.1 Sumeuler

Recall that sumeuler applies Euler's totient function to a list of integers and sums the result.

```
1 sumeuler xs₀@[] = 0
2 sumeuler xs₀@(x:xs) = euler x + (sumeuler xs)
```

We first slice sumeuler for its only argument, xs_0, $\Sigma^{\text{sumeuler}|xs_0}$ = {x,xs}. As before, xs_0 is classified as *clean* since both case-split variables of xs_0 (x and xs) are used in the body of sumeuler, and xs_0 itself is not updated. Two operations exist as subexpressions to sumeuler: (euler x) and the application of (+) in Line 2. (euler x) takes a single clean argument, i.e. x, and is classified as *unobstructive*. Conversely, (+) takes two arguments, where the second argument comprises a recursive call, and is classified as *obstructive*. (euler x) can be lifted into a map operation, and sumeuler is rewritten to introduce chunking and parallelism using the Strategies library. We are also able to take advantage of the associativity of (+), summing each chunk before summing the result of all chunks.

In [4] we executed sumeuler for $n = 1,000$ and between $5,000$ and $50,000$ at intervals of $5,000$, with a chunk size of 500. Fig. 1 gives the speedups for sumeuler using mutator time. We achieve maximum speedups of 30.50 for $n = 50,000$ on 48 virtual cores. Sequentially, when $n = 50,000$ sumeuler has an average runtime of $154.33s$ ($\sigma = 0.25s$). We achieve good speedups in all cases. Moreover, sumeuler demonstrates good scalability as n increases. Fig. 1 shows that for varying n, speedups reach a limit, likely due to a lack of work.

Our translation of the NoFib example code comprises 14 recursive functions over three Haskell modules. Within those 14 recursive functions, there are a total of 47 operations, 16 of which are classified *unobstructive* and the remaining 31 of which are classified
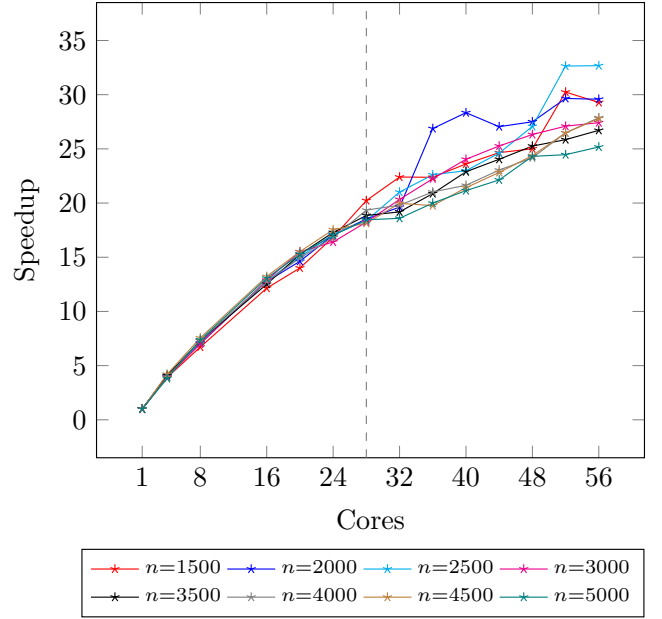
Figure 8. matmult, speedups on *corryvreckan* using reported mutator (MUT) time, dashed line shows extent of physical cores.

as *obstructive*. As with all the NoFib benchmarks, the high number of recursive functions is representative of the original duplication in the underlying Haskell modules.

5.2 Matrix Multiplication

The NoFib benchmark uses the following implementation of matrix multiplication.

```
1 matmult m1 m2 = multMatricesTr m1 (transpose m2))
2
3 multMatricesTr [] m2 = []
4 multMatricesTr (r:rs) m2 =
5   f m2 r : multMatricesTr rs m2
6
7 f cs₀@[] row = []
8 f cs₀@(c:cs) row = prodEscalar2 row c : f cs row
```

Here, transpose transposes a matrix and prodEscalar2 calculates the dot product of two lists. Since matmult is not a fix expression, we slice for all the arguments to multMatricesTr.

$$\Sigma^{\text{multMatricesTr}|rs_0} = \{r,rs\} \quad \Sigma^{\text{multMatricesTr}|m2} = \{mt\}$$

As with our other examples, both of the arguments are *used* but not *updated* and are therefore classified as *clean*. The sole operation, (f m2 r), is classified as *unobstructive*. Slicing for all arguments of f,

$$\Sigma^{\text{f}|cs_0} = \{c,cs\} \quad \Sigma^{\text{f}|row} = \{row\}$$

we can see the same pattern: all the arguments are classified as *clean*, and the sole operation, (prodEscalar2 row c), is classified as *unobstructive*. Both multMatricesTr, and f can be rewritten as map operations. Parallelism can be introduced in matmult by applying a custom strategy to the call to multMatricesTr that chunks matrices into blocks, rather than along rows or columns. In [4] we executed matmult for *n* between 1000 and 5000 at intervals

of 500, with the chunk size set to 20. Fig. 8 gives speedups for
`matmult` using mutator time, achieving maximum a speedup of
32.93 for $n = 2500$ on 52 hyper-threaded cores. Sequentially, when
$n = 2500$ `matmult` takes an average of $226.58s$, with a standard
deviation of $7.49s$. `matmult` scales well for both n and number of
cores. On hyperthreaded cores we observe some erratic behaviour
with curious superlinear speedups. This is possibly due to caching
effects.

Interestingly, the definition of f might be unfolded (in the trans-
formational sense) in `multMatricesTr`; e.g.

```
1  multMatricesTr xs₀@[] r m2 =
2    []
3  multMatricesTr xs₀@(x:xs) rr@[] m2 =
4    multMatricesTr m2 x m2 : multMatricesTr xs [] m2
5  multMatricesTr xs₀@(x:xs) rr m2 =
6    prodEscalar2 rr x : multMatricesTr xs rr m2
```

Here, `multMatricesTr` is now a fix-expression, case-splitting on
its first argument, xs_0. Slicing for its arguments, gives

$$\Sigma^{\text{multMatricesTr}|xs_0} = \{x, xs, \bar{xs_0}\}$$
$$\Sigma^{\text{multMatricesTr}|rr} = \{rr, \bar{rr}\}$$
$$\Sigma^{\text{multMatricesTr}|m2} = \{m2\}$$

While xs_0 is considered to be updated, due to m2 being passed as
the first argument in the recursive call in line Line 4, both xs_0
and m2 are considered to be *clean*. Conversely, rr is considered
to be *tainted*. (`prodEscalar2 rr x`), now the sole operation, is
therefore classified as *obstructive*, meaning that no `map` operations
can be introduced.

This is a correct result since `multMatricesTr` traverses *two* lists,
where both the lists are passed as the first argument. The standard
definition of `map`, for example, does not allow this behaviour, and
any attempt at introducing a `map` operation would result in a func-
tion that is not functionally equivalent to `multMatricesTr`. This
limitation arises again in `queens` with fused mutually recursive
functions. A map is only discoverable when the inspected function
recurses over one or more data structures and where at least one
argument is structurally smaller for each recursive call.

Our translation of the NoFib benchmark code comprises 8 recur-
sive functions over 2 Haskell modules. Within those 8 recursive
functions there are 23 operations, of which 6 are classified as *unob-
structive* and the remaining 17 are classified as *obstructive*.

5.3 N-Queens

We use the following implementation of the queens problem,

```
1  queens nq = gen 0 []
2
3  gen nq n b
4    | n >= nq = [b]
5    | otherwise = genloop nq n (gennext [b])
6
7  genloop nq n bs₀@[] = []
8  genloop nq n bs₀@(b:bs) =
9    gen nq (n+1) b ++ genloop nq n bs
```

where nq is the number of queens and gennext calculates the posi-
tion of a queen on a board. We slice for all arguments of genloop.

$$\Sigma^{\text{queens}|nq} = \{nq\} \quad \Sigma^{\text{queens}|n} = \{n\} \quad \Sigma^{\text{queens}|bs_0} = \{b, bs\}$$

All three arguments, nq, n, and bs_0, are classified as *clean* since
both nq and n are *used* but not *updated*; and both b and bs are
used but bs_0 is neither *used* nor *updated*. genloop has three subex-
pressions: *i)* (n+1) in the call to gen; *ii)* (gen nq (n+1) b) in the
first argument to (++); and *iii)* the top-level (++). Both (n+1) and
(gen nq (n+1) b) are classified as *unobstructive* since only *clean*
variables occur in the expressions passed to (+) and gen respec-
tively. We additionally note that (gen nq (n+1) b) contains the
unobstructive operation (n+1) as a subexpression. Hypothetically,
if (n+1) had been classified as *obstructive*, then (gen nq (n+1) b)
must also be classified as *obstructive*. While no *tainted* variables
occur in the arguments to the append operation, its second argu-
ment is a recursive call, which results in an *obstructive* classifica-
tion. Although it is possible to lift either *unobstructive* operations
into a map operation, we choose to lift the operation which does
not occur as a subexpression of any *unobstructive* operation, i.e.
(gen nq (n+1) b), in order to maximise the amount of work that
is done in parallel.

```
1  genloop nq n bs₀ =
2    genloop' nq n (map (gen nq (n+1)) bs₀)
3
4  genloop' cs₀@[] = []
5  genloop' cs₀@(c:cs) = c ++ genloop' cs
```

Similarly to f and `multMatricesTr` in `matmult`, we observe that
gen and genloop could be merged into a single definition.

```
1  gen 0 b xs₀@[] = [b]
2  gen n b xs₀@[] = []
3  gen n b xs₀@(x:xs) =
4    (gen (n-1) x (gennext [x])) ++ (gen n b xs)
```

This definition eliminates the mutual recursion of gen and genloop,
where xs_0 traverses two different lists and n acts as an additional
bound on the recursion. As before, our technique will (correctly)
classify all operations as *obstructive*. In this example, it is the recur-
sive call in the first argument to (++) that introduces an update to
all variables.

Our translation of the queens NoFib implementation comprises
two recursive functions with 11 operations, and 2 recursive calls.
Despite the high number of operations, there are only two unob-
structive operations in the translation. The majority of operations
are found in the definition of safe.

```
1  safe x d ys₀@[] = True
2  safe x d ys₀@(q:l) =
3    x /= q && x /= q+d && x /= q-d && safe x (d+1) l
```

Slicing `safe` for all arguments,

$$\Sigma^{\text{safe}|x} = \{x\} \quad \Sigma^{\text{safe}|d} = \{d, \bar{d}\} \quad \Sigma^{\text{safe}|ys_0} = \{q, l\}$$

we classify x and ys_0 as *clean* and d as *tainted*. Here, the *tainted*
classification of d results in *obstructive* classifications of all but one
infix inequality operations. Hypothetically, if d had been classified
as *clean*, we observe that how the infix (&&) operations are compiled
can affect the classifications of operations in safe. Parsing (&&)
as either left- or right-associative can produce different numbers
of *obstructive* operations. Here, parsing (&&) as left-associative
minimises the number of *obstructive* operations; i.e. when,

```
1  (x /= q && x /= q+d) && (x /= q-d && safe x (d+1) l)
```
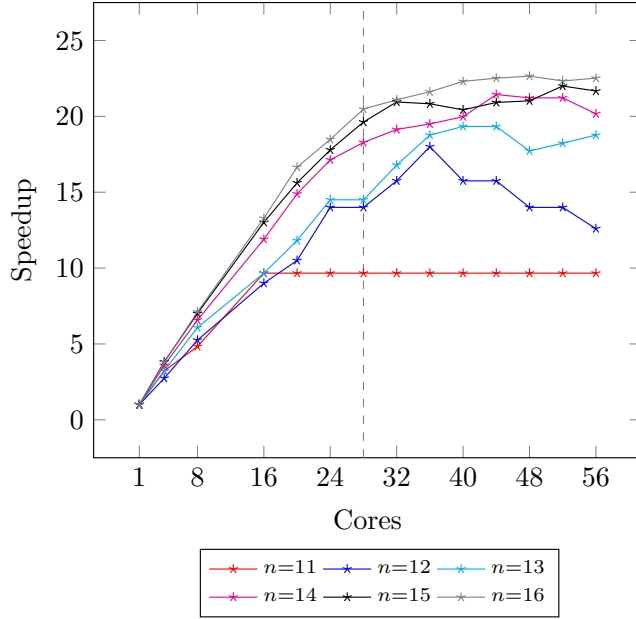
Figure 9. queens, speedups on *corryvreckan* using reported mutator (MUT) time, dashed line shows extent of physical cores.

only the topmost (&&) and its second argument are classified as *obstructive*. Conversely, parsing (&&) as right-associative *maximises* the number of *obstructive* operations; i.e. when,

```
1 x /= q && (x /= q+d && (x /= q-d && safe x (d+1) l))
```

all (&&) operations are classified as *obstructive*.

The queens example is parallelised by applying a strategy from the Strategies library to the map. To ensure that parallelism is worthwhile, the specific strategy used (parallel or sequential) is passed in as an argument. A threshold argument is then added to gen to control the depth to which the map operation is performed in parallel. We executed queens for *n* ranging from 11 to 16, with a threshold depth of 2. Fig. 9 show speedups for queens in terms of mutator time. We achieve maximum speedups of 22.65 for $n = 16$ on 48 hyperthreaded cores. Sequentially, when $n = 16$, queens takes an average of $717.66s$ ($\sigma = 0.25s$). When $n = 11$, speedups plateau before the physical cores are exhausted, likely due to a lack of work, as we also observed for sumeuler. When $14 \le n \le 16$, queens scales well until 28 cores; when hyperthreading is enabled, we see some further improvement in speedup, albeit at reduced rate when compared with the results for physical cores. For $n = 13$ and $n = 14$, we see intermediate scalability continue between 28 and 40 hyperthreaded cores. This may be due to the lack of chunking of bs_0, resulting in inefficient use of cores up to 28 cores, with hyperthreading saturating the hardware. For $n = 13$ and $n = 14$, we observe a similar effect to $n = 11$, but at a higher level due to increased availability of work.

5.4 Smith-Waterman

The Smith-Waterman algorithm compares the similarity of two strings. The algorithm was originally designed for the comparison of nucleotides [36]. It has two stages: populating a matrix, and backtracking over the matrix to find the shortest 'distance' between the

two strings. The code below concerns part of the matrix population stage. It traverses the matrix m, updating each cell with the result of h, where h (implementation omitted) calculates the maximum similarity score between the two strings a and b for the current row r and column c and updates the cell with that score.

```
1 tcs r c a b m =
2   if c > (length m)
3   then m
4   else tcs r (c+1) a b (h r c a b m)
5
6 trs r c a b m = if r > (length m)
7                 then m
8                 else trs (r+1) c a b (tcs r c a b m)
9
10 traverse m a b = trs 1 1 a b m
```

As before, we calculate a slice for each argument to each function, classifying those arguments using the slice.

tcs:

$$
\begin{aligned}
\Sigma^{tcs|r} &= \{r\} &\text{(clean)} \\
\Sigma^{tcs|c} &= \{c, \bar{c}\} &\text{(tainted)} \\
\Sigma^{tcs|a} &= \{a\} &\text{(clean)} \\
\Sigma^{tcs|b} &= \{b\} &\text{(clean)} \\
\Sigma^{tcs|m} &= \{m, \bar{m}\} &\text{(tainted)}
\end{aligned}
$$

trs:

$$
\begin{aligned}
\Sigma^{trs|r} &= \{r, \bar{r}\} &\text{(tainted)} \\
\Sigma^{trs|c} &= \{c\} &\text{(clean)} \\
\Sigma^{trs|a} &= \{a\} &\text{(clean)} \\
\Sigma^{trs|b} &= \{b\} &\text{(clean)} \\
\Sigma^{trs|m} &= \{m, \bar{m}\} &\text{(tainted)}
\end{aligned}
$$

Using these classifications, we can classify the operations within the recursive functions.

tcs: As both c and m are classified as *tainted*, all operations in tcs (i.e. length, and the less-than and addition operators) are classified to be *obstructive*.

trs: Analogous to tcs, and as both r and m are classified as *tainted*, all operations in trs are classified to be *obstructive*.

No refactoring can be applied to the Smith-Waterman implementation. However, the Smith-Waterman algorithm is an example of *wavefront parallelism* whereby cells of the matrix are divided into groups which can be calculated in parallel. As our classification looks only for independence across an entire data structure, at present this would be undetected. We will consider this in future work.

6 Related Work

Program slicing was first introduced by Mark Weiser in 1981 [38] for imperative, procedureless programs. Slicing has since seen numerous extensions and adaptations for a range of fields, e.g. testing and debugging [34]. Elsewhere, intra-function slicing has also been used to introduce patterns for concurrency in Erlang code [27]. We ourselves have used slicing to transform the data-types of Erlang functions in a slice, in order to take advantage of properties, e.g. to reduce copying overheads when sending between processes [2]. Similar to our own approach, Ahn and Han [1], use program slicing to categorise function parameters for detecting loci of parallelism.

Their approach is limited to first-order languages, whereas our approach is also suitable for higher-order languages. It is designed to only inspect recursive functions that can be transformed into a form where each constructor has exactly one clause, and where functions have only a single argument. In contrast, our approach inspects functions in any form, and so maximises the number of components that are available for parallelisation. Even if the introduced maps are eventually deemed not to be worth parallelising, the more components that are available for inspection, the more configurations of parallelism can be considered. Moreover, map operations, and recursion schemes in general, can encapsulate properties, e.g. independence of operation or type of output, that can be used for other purposes; e.g. termination checking. Affine types [33] and single-threadedness [20] both have their foundations in linear logic, wherein objects must be used exactly once. Affine types weaken this constraint by allowing objects to be used at most once. There is some superficial similarity with our technique, such that if an input can be used at most once within a recursive function then it follows that all operations that use that input can be performed independently of successive recursive calls. Single-threadedness shares a similar goal, but places greater emphasis on exposing mutation to state without sacrificing referential transparency. Again, this is superficially similar to our approach in the exposition of how an input changes between two points. However both of these approaches depend upon a specific language or system framework. In contrast, our approach is fully generic. We additionally note that our system allows a weaker sense of usage (in that an input can be used more than once), whilst still enabling us to reason about the independence of operations.

Structured parallelism approaches, such as those used here, have been shown to be an effective means to introduce and manipulate parallelism [9, 15, 16, 25, 29]. Pattern discovery for parallelism has primarily focussed on approaches for imperative programs by analysing dependencies and machine learning techniques [1, 6, 14, 21]. The polyhedral model, in particular, uses control and data dependencies to reorder statements in code. One limitation of the polyhedral model is that it requires the AST to be translated into a specific linear-algebraic form. However, not all statements can be translated to this form [5]. Alternative automatic approaches derive parallel implementations from small programs or specifications, commonly exploiting the Bird Meertens formalism [13, 17, 19, 24, 26]. These approaches use algorithmic structures, e.g. *list homomorphisms* [18] and *hylomorphisms* [9], which are amenable to *divide-and-conquer* skeletons. These approaches require operations to be translated into specific types which can be easily parallelised [13, 26]. Similar to our own approach, Hu, Takeichi, and Iwasaki's Diffusion [24] uses program transformation techniques to rewrite explicitly recursive functions into compositions of recursion schemes. Instead of using slicing to determine the operations that can be lifted into scheme fixpoint functions, Hu et al. use the the Bird Meertens formalisms to systematically reason about the behaviour of the function. This approach has been shown to extend to types other than lists, and to schemes other than maps. While our approach can in principle be extended to other schemes, the slicing approach requires that the schemes are encoded in terms of variable usage and update. The diffusion approach is, however, reliant on external normalisation processes that must be proven to be sound. Another limiting factor can be found in the applicability of the language to which diffusion is

applied: since it relies upon the Bird Meertens formalism, it may be difficult to extend the approach, including the necessary normalisation processes, to complex languages such as C++. Our approach may prove simpler to extend to such languages, providing that state is explicit and variable usage and update are properly defined. One exception to the program calculation approaches is Cook's approach, which uses *higher-order unification* to discover and introduce common sequential patterns [12]. This approach is, however, limited in that it uses a proof assistant as an intermediate representation, and further requires that functions have a structure that mirrors the structure of the pattern to be discovered. More generally, program calculation and unification approaches are limited by the information that they are able to infer from the source code. *Program shaping* approaches [3] that include pattern discovery techniques can potentially find more patterns by refactoring the source code prior to pattern discovery. Developed from early work on a fold/unfold transformation system by Burstall and Darlington [8], refactoring techniques change the structure of a program while preserving functional equivalence [7]. Refactoring tools automate this transformation process and exist for many languages [7, 28] and IDEs [23]. Recent work on refactoring for parallelism has demonstrated that a refactoring approach can aid in the introduction of skeletons, but also that refactorings can *enable* the introduction of parallelism [2, 3, 6, 7]. While these approaches can take advantage of programmer knowledge, e.g. the associativity of operations, they also require the programmer to know when and where to apply the refactorings. The approach presented here places no restriction or requirement on the structure of recursive functions, and can be included as part of a program shaping workflow to guide the application of refactorings. Moreover, our approach also allows the programmer to apply transformations that would be unreachable in fully automatic approaches, such as those described above.

7 Conclusions and Future Work

In this paper, we have investigated the use of program slicing to discover mappable, and by extension potentially parallelisable, computations in recursive functions. We have defined a novel program slicing algorithm that inspects how the value of a variable is used and changes across recursive calls. We have provided formal definitions for all our concepts. We have implemented our approach in Erlang, producing a parser and prototype classifier. We have applied our prototype to 12 examples, including benchmarks from the NoFib suite and functions from the Haskell Prelude. Our prototype successfully discovers all the maps in our examples. We have also applied our prototype to synthetic benchmarks, showing that our prototype has quadratic time complexity. We have provided speedup results for three of our examples: *sumeuler*, *matrix multiplication*, and *n-queens*. We achieve speedups of up to 32.93 on our 56-core hyperthreaded experimental machine, so demonstrating the effectiveness of using map operations as loci of potential parallelism.

For future work, we intend to address the main limitations of our approach, including discovering map operations in functions that traverse multiple data structures of different types. Relatedly, we also intend to expand our technique to inspect how inputs can be divided into sub-groups of inputs that can be performed in parallel, as in, e.g., wavefront parallelism. We also intend to expand

our property checking to other parallel structures, e.g. divide-and-conquer and feedback patterns [9]. Since our technique requires any new pattern to be encoded in terms of how values of variables change between recursive calls, we intend to consider how this translation might be simplified. Similarly, we intend to develop the means to translate from full languages, such as Haskell or Erlang, to our expression language. Finally, since our technique is, in principle, language-independent, we also intend to adapt and apply our technique to imperative languages; such as C++ or Java.

Acknowledgments

References

[1] Joonseon Ahn and Taisook Han. 2000. An Analytical Method for Parallelization of Recursive Functions. *Parallel Processing Letters* 10, 1 (2000), 87–98.

[2] Adam D. Barwell, Christopher Brown, David Castro, and Kevin Hammond. 2016. Towards Semi-automatic Data-type Translation for Parallelism in Erlang. In *Proceedings of the 15th International Workshop on Erlang (Erlang 2016)*. 60–61.

[3] Adam D. Barwell, Christopher Brown, Kevin Hammond, Wojciech Turek, and Aleksander Byrski. 2016. Using Program Shaping and Algorithmic Skeletons to Parallelise an Evolutionary Multi-Agent System in Erlang. *Computing and Informatics* 35, 4 (2016), 792–818.

[4] Adam D. Barwell and Kevin Hammond. 2017. Obliterating Obstructions: Discovering Potential Parallelism by Slicing for Dependencies in Recursive Functions. *International Journal of Parallel Programming* (2017). In submission.

[5] Cedric Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proc. of the 13th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT '04)*. 7–16.

[6] István Bozó, Viktoria Fordós, Zoltán Horvath, Melinda Tóth, Dániel Horpácsi, Tamás Kozsik, Judit Köszegi, Adam Barwell, Christopher Brown, and Kevin Hammond. 2014. Discovering Parallel Pattern Candidates in Erlang. In *Proc. of the 13th ACM SIGPLAN Workshop on Erlang (Erlang '14)*. 13–23.

[7] Christopher Brown, Marco Danelutto, Kevin Hammond, Peter Kilpatrick, and Archibald Elliott. 2013. Cost-Directed Refactoring for Parallel Erlang Programs. *International Journal of Parallel Programming* (2013), 1–19.

[8] R. M. Burstall and John Darlington. 1977. A Transformation System for Developing Recursive Programs. *J. ACM* 24, 1 (Jan. 1977), 44–67.

[9] David Castro, Kevin Hammond, and Susmit Sarkar. 2016. Farms, Pipes, Streams and Reforestation: Reasoning About Structured Parallel Processes Using Types and Hylomorphisms. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. 4–17.

[10] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell Array Codes with Multicore GPUs. In *Proc. of the 6th Workshop on Declarative Aspects of Multicore Programming*. 3–14.

[11] Murray I. Cole. 1988. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*. Ph.D. Dissertation.

[12] A. Cook, A. Ireland, G. Michaelson, and N. Scaife. 2005. Discovering Applications of Higher Order Functions Through Proof Planning. *Form. Asp. Comput.* 17, 1 (May 2005), 38–57.

[13] Michael Dever and G. W. Hamilton. 2014. AutoPar: Automatic Parallelization of Functional Programs. In *2014 4th Int. Valentin Turchin Workshop on Metacomputation (META 2014)*. 11–25.

[14] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele. 2005. Design pattern mining enhanced by machine learning. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. 295–304.

[15] F. Gava and F. Loulergue. 2005. A static analysis for Bulk Synchronous Parallel ML to avoid parallel nesting. *Future Generation Computer Systems* 21, 5 (2005), 665 – 671.

[16] Louis Gesbert, Frédéric Gava, Frédéric Loulergue, and Frédéric Dabrowski. 2010. Bulk synchronous parallel ML with exceptions. *Future Generation Computer Systems* 26, 3 (2010), 486 – 490.

[17] Alfons Geser and Sergei Gorlatch. 1999. Parallelizing Functional Programs by Generalization. *J. Functional Programming* 9, 06 (1999), 649–673.

[18] Jeremy Gibbons. 1996. The Third Homomorphism Theorem. *Journal of Functional Programming (JFP)* 6, 4 (1996), 657–665.

[19] Sergei Gorlatch. 1999. Extracting and Implementing List Homomorphisms in Parallel Program Development. *Science of Computer Programming* 33, 1 (1999).

[20] J. C. Guzman and P. Hudak. 1990. Single-threaded polymorphic lambda calculus. In *Logic in Computer Science*. 333–343.

[21] Clemens Hammacher, Kevin Streit, Sebastian Hack, and Andreas Zeller. 2009. Profiling Java Programs for Parallelism. In *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering (IWMSE '09)*. 49–55.

[22] Kevin Hammond, Marco Aldinucci, Christopher Brown, Francesco Cesarini, Marco Danelutto, Horacio González-Vélez, Peter Kilpatrick, Rainer Keller, Michael Rossbory, and Gilad Shainer. 2013. The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. In *Formal Methods for Components and Objects*. Lecture Notes in Computer Science, Vol. 7542. Springer Berlin Heidelberg, 218–236.

[23] Steve Holzner. 2004. *Eclipse: A Java Developer's Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

[24] Zhenjiang Hu, Masato Takeichi, and Hideya Iwasaki. 1999. Diffusion: Calculating Efficient Parallel Programs. In *Proc. PEPM'99*. 85–94.

[25] V. Janjic, C. Brown, and K. Hammond. 2016. Lapedo: Hybrid Skeletons for Programming Heterogeneous Multicore Machines in Erlang. In *Proc. ParCo 2015: Intl. Conf. on Parallel Computing*. 181–195.

[26] Venkatesh Kannan and Geoff W. Hamilton. 2016. Program Transformation to Identify Parallel Skeletons. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016*. 486–494.

[27] Huiqing Li and Simon Thompson. 2015. Safe Concurrency Introduction Through Slicing. In *PEPM '15*. 103–113.

[28] Huiqing Li, Simon J. Thompson, George Orösz, and Melinda Tóth. 2008. Refactoring with wrangler, updated: data and process refactorings, and integration with eclipse. In *Proc. of the 7th ACM SIGPLAN Workshop on Erlang*. 61–72.

[29] Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K. Aswad, and Phil Trinder. 2010. Seq No More: Better Strategies for Parallel Haskell. In *Proc. of the 3rd ACM Haskell Symp. on Haskell (Haskell '10)*. 91–102.

[30] Simon Marlow, Ryan Newton, and Simon Peyton Jones. 2011. A Monad for Deterministic Parallelism. In *Proc. of the 4th ACM Symposium on Haskell (Haskell '11)*. 71–82.

[31] Erik Meijer, Maarten Fokkinga, and Ross Paterson. 1991. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*. 124–144.

[32] Will Partain. 1993. The nofib Benchmark Suite of Haskell Programs. In *Proc. of the 1992 Glasgow Workshop on Functional Programming*, John Launchbury and Patrick Sansom (Eds.).

[33] Benjamin C. Pierce. 2004. *Advanced Topics in Types and Programming Languages*. The MIT Press.

[34] Martin P. Robillard. 2008. Topology Analysis of Software Dependencies. *ACM Trans. Softw. Eng. Methodol.* 17, 4, Article 18 (Aug. 2008), 36 pages.

[35] Norman Scaife, Susumi Horiguchi, Greg Michaelson, and Paul Bristow. 2005. A parallel SML compiler based on algorithmic skeletons. *Journal of Functional Programming* 15, 4 (2005), 615–650.

[36] Temple F Smith and Michael S Waterman. 1981. Identification of common molecular subsequences. *Journal of molecular biology* 147, 1 (1981), 195–197.

[37] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. 1998. Algorithm + Strategy = Parallelism. *J. Funct. Program.* 8, 1, 23–60.

[38] Mark Weiser. 1981. Program Slicing. In *Proc. of the 5th Intl. Conf. on Software Engineering (ICSE '81)*. 439–449.