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
{adb23,kh8}@st-andrews.ac.uk

Abstract
Recursion schemes, such as the well-known map, can be used as loci of potential parallelism, where, e.g., map is 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 potentially performed in parallel. We define a new program slicing algorithm, built a prototype implementation, and demonstrated 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, demonstrating 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).

ACM Reference format:
DOI: 10.1145/nnnnnn.nn.nn

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 foldr. 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 otherwise 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 to produce relative speedups. 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 Er-lang, 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 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.

2 Illustrative Example
We illustrate our approach using an example from the NoFib suite, ssumeuler, that calculates Euler’s totient function for a list of integers and sums the results.

\[
\text{sumeuler} :: \text{[Int]} \rightarrow \text{Int}
\]
\[
\text{sumeuler \text{x}@[0]} = 0
\]
\[
\text{sumeuler \text{x}@[\text{x}]} \circ (\text{x} : \text{xs}) = \text{euler x} + (\text{sumeuler xs})
\]

Here, we have unfolded (in the transformational sense) the definition of ssumeuler to a standard map, representing the map function as a higher-order function. The definition of ssumeuler is explicitly parallelising the function definition requires the programmer to know how ssumeuler traverses its arguments. This includes understanding how the computations performed on the (elements of the) input list affect: i) how the input is traversed; and ii) the result of ssumeuler itself. To take advantage of recursion schemes as loci of potential parallelism, the above definition of ssumeuler can be rewritten as an instance of a composition of a foldr and a map,
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 hyper-threaded experimental machine, corryvreckan. We obtain a maximum speedup of 30.50 for \(n = 50000\) on 48 hyperthreaded parallel 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 \textit{operations}, in the definition of \textit{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 \textit{program slicing}. This is a technique that is used to extract all the program statements that, depending on the algorithm used, \textit{may influence}, or \textit{may be influenced by}, a given statement from the same program, called the \textit{slicing criterion} \cite{34}. A slice is often calculated by inspecting the control and data dependencies of a program. Slicing the explicitly recursive definition of \textit{sumeuler}, taking \(x\) as the criterion, produces:

\begin{verbatim}
sumeuler x0 = foldr (+) 0 (map euler x0)
\end{verbatim}

Both the empty-list clause and the recursive call, \((\textit{sumeuler } x)\), are removed because they are not influenced by the value of \(x\). Slicing might therefore be thought of as a \textit{narrowing of focus} on only those parts of the program or expression that are of interest, and with all the irrelevant parts stripped away \cite{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 \textit{use}, and an annotated inclusion in the set indicates \textit{update}. For example, the slice of \textit{sumeuler}, with criterion \(x0\), \((x, x)\), indicates that both \(x\) and \(x0\) are \textit{used}; in \((\textit{euler } x)\) and \((\textit{sumeuler } x)\) respectively. The slice also indicates that \(x0\) itself is not considered to be \textit{updated}. We consider a variable to be \textit{updated}, when its value is \textit{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 \textit{used} and not \textit{updated}, e.g. as with \(x0\) in \textit{sumeuler}, then it can also occur safely in a map operation. Conversely, if the argument is \textit{updated} but not \textit{used}, e.g. \(x\) in,

\begin{verbatim}
f x y0 [] = 42
f x y0 (y:ys) = f y ys
\end{verbatim}

then it can similarly be updated independently of each input. In either of these cases, we consider the argument to be \textit{clean}. However, when an argument is both \textit{used} and \textit{updated}, then it indicates that the usage and update of the variable is not independent of the other stages of the recursion. Any operation in which an argument that is both \textit{used} and \textit{updated} cannot be safely lifted into a map. We therefore consider such an argument to be \textit{tainted}. In \textit{sumeuler}, \(x0\) is the only argument, and it is \textit{used} but not \textit{updated}. It is therefore classified as \textit{clean}, along with its case-defined variables, \(x\) and \(xs\).

Figure 1. \textit{sumeuler}, speedups on corryvreckan using reported mutator (MIT) time, dashed line shows extent of physical cores.
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. \((\text{euler } x);\) and
2. \(((\text{euler } x) + (\text{sumeuler } xs))\).

Since \(x\) is classified as clean, \((\text{euler } x)\) 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 \((\text{euler } x)\) into a map operation, perhaps by refactoring the original source:

\[
\begin{align*}
\text{sumeuler} & : \int \rightarrow \int \\
\text{sumeuler } x_0 &= \text{sumeuler'}(\text{map euler } x_0)
\end{align*}
\]

Here \text{sumeuler'} is functionally equivalent to \text{sum}, so our final form is:

\[
\begin{align*}
\text{sumeuler} & : \int \rightarrow \int \\
\text{sumeuler } x_0 &= \text{sum}(\text{map euler } x_0)
\end{align*}
\]

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

### 3 Preliminaries and Assumptions

We illustrate our approach over the simple expression language, \(E\).

\[
e \in E ::=
\begin{align*}
& \text{true} \\
& \text{false} \\
& \int \\
& \text{var} \\
& \text{nil}_\tau \\
& \text{cons}_\tau e e \\
& \text{case} \text{var} \text{of} \text{nil}_\tau \rightarrow e, \text{cons}_\tau \text{var} \rightarrow e \\
& e \hat{\tau} \\
& \lambda \text{var} \rightarrow e \\
& \text{fix} e
\end{align*}
\]

\(E\) 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, \(\text{var}\); list constructors, \(\text{nil}\) and \(\text{cons}\); case discrimination on lists, \(\text{case} \text{var} \text{of} \text{nil} \rightarrow e, \text{cons} \text{var} \rightarrow e\); function application, \(e \hat{\tau}\); lambda expressions, \(\lambda \text{var} \rightarrow e\); and fixpoints, \(\text{fix} e\). 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 notation, e.g., \(\hat{\tau}\), refers to a non-empty tuple: \(\hat{\tau} \equiv (\tau_1, \ldots, \tau_n), n \geq 1\). Tuples are purely meta- syntactic, and are used for clarity of notation. In order to simplify our presentation, list constructors and case discriminators are annotated with the (monomorphic) type of the list elements, \(\tau\). The corresponding type language, \(T\), is shown below.

\[
\tau \in T ::= \text{bool} \mid \int \mid \text{list } \tau \mid \hat{\tau} \rightarrow \tau
\]

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.

\[
s \in S ::= \text{def } \text{var} = \lambda \text{var} \rightarrow e \\
& \mid \text{def } \text{var} = \text{fix } e
\]

**Figure 2.** Typing judgements for \(E\), determining simple types in \(T\).

**Figure 3.** Definition of \text{sumeuler} in \(E\).
We will refer to any application that is a subexpression of a fixpoint expression in Fig. 3, case discrimination is restricted to lists of some type \( \tau \). We assume the existence of a built-in functions (e.g. \( \text{if}, \text{eq}, \text{plus} \)) 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. We limit case-expressions here to simplify our presentation. 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, \( \Gamma_p \), and a case-expression in \( p, e = \text{case } x_0 \text{ of nil}_r \rightarrow e_1, \text{cons}_r x x \rightarrow e_2 \), we say that \( x \approx_p x_0 \) and \( x s \approx_p x_0 \).

The transitive closure of the reachability relation is defined such that \( z \approx^*_p x_0 \) when \( \exists y, z \approx^*_p y \land y \approx_p x_0 \). The reflexive-transitive closure of the reachability relation is defined such that \( y \approx^*_p x \) when \( \exists y, y \approx^*_p x \land y \approx_p x \).

For example, \( x \approx_p x_0, x s \approx_p x_0, \) and \( x_0 \approx^*_p x \) all hold for the case expression in \( \text{sumeuler} \). 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 \( x_0 \) (e.g. \((x:xs)\) in \( \text{sumeuler} \)), so that \( x_0 \) 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 a program \( p \); argument \( x \) of type list \( \tau \), and a cons-expression \( e = \text{cons}_r x e' \), we say that \( e \) is syntactically equivalent to \( x \), denoted \( e \equiv_p x \), when \( \exists y, e' = y \land e'' \equiv_p y s \approx_p x \).

For example, where \( x \approx_p x_0, x s \approx_p x_0, \) and \( x_0 \approx^*_p x \) all hold for the case expression in \( \text{sumeuler} \). 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 \( x_0 \) (e.g. \((x:xs)\) in \( \text{sumeuler} \)), so that \( x_0 \) can be included in the slice. The syntactic equivalence relation for lists is used to detect such list reconstructions.

### 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, \( \text{euler} x \) in \( \text{sumeuler} \) is classified as unobstructive because euler is a unary function that takes only the head of the list. Conversely, the infix addition operation in \( \text{sumeuler} \) 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 \( \Gamma_p \) for some fixpoint expression, \( e = \text{fix } \lambda (f, x_1, \ldots, x_n) \rightarrow e' \), 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 \( i^{\text{th}} \) 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 significantly changed. Intuitively, a change in value is considered significant when for
some recursive call, $f(e_1, \ldots, e_n)$, the $i^{th}$ 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 \textit{updated}.

\textbf{Definition 4.1 (Variable Update).} Given a program $p$, a program environment $\Gamma_p$, a fixpoint expression $e = \text{fix} (\lambda (f, x_1, \ldots, x_n) \rightarrow e')$ in $p$, and an argument $x$, where $x$ is the $i^{th}$ argument to $f$, $x$ is considered to be updated in $e$ (denoted $\sim_x e$) when there exists a recursive call, $f(e_1, \ldots, e_n) \ll e'$ such that $\neg (e_i \neq x) \wedge (e_i \equiv x) \wedge e \equiv e_i \wedge y \equiv e_i \wedge f e_i \equiv e_i \wedge f_j y \equiv e_i \wedge f_j y \equiv e_i x,$

Usage is a simpler concept: a variable is considered used when it occurs as a subexpression of a fixpoint expression. There is one exception to this: when a subexpression is the $i^{th}$ argument to a recursive call. We first define the notion of \textit{variable-usage escape-ment}. 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:

\begin{verbatim}
1  f a x = if a < 5
2   then f (a+1) x
3   else f (a-1) a
\end{verbatim}

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

\textbf{Definition 4.2 (Variable-Usage Escape-ment).} For some fixpoint expression $e = \text{fix} (\lambda (f, x_1, \ldots, x_n) \rightarrow e')$, and for some $i \in [1,n]$, given $x$, the $i^{th}$ argument to $f$, we substitute the special symbol $e_i$ for $e_i$ whenever $e_i = x$ or $e_i \equiv x$. When $e_i = \text{cons} e_k e_l e_i \equiv e_i \equiv x \wedge x \equiv e_k \wedge f_k x \equiv e_l$, we substitute $e_l$ for $e_i$. Variable-usage escape-ment of $e$ with respect to $x$ is denoted $e \setminus x$.

In the above definition of $\mathcal{f}$, for example, the $x$ in Line 2 would be substituted for $x$.

\begin{verbatim}
1  f a x = if a < 5
2   then f (a+1) x
3   else f (a-1) a
\end{verbatim}

We can now define variable \textit{usage}.

\textbf{Definition 4.3 (Variable Usage).} Given some fixpoint expression $e = \text{fix} (\lambda (f, x_1, \ldots, x_n) \rightarrow e')$, and a variable $x$ where $x$ is the $i^{th}$ argument to $f$, a variable $y$ is considered to be used in $e$ (denoted $y \sim_x e$) when $y \neq x$ and $y$ exists as a subexpression to the variable-usage escape $e_i$, i.e. $y \equiv (e \setminus x)$.

To illustrate this, consider the following functions:

\begin{verbatim}
1  f a x0@[x] = a
2  f a x0@[x]:xs = f x xs
3  g x0@[x]:xs y0@[y]:ys = g (x:xs) (y:ys)
4  g x0 y0 = x0
5
6
7
\end{verbatim}

For $\mathcal{f}$, the value of $a$ is \textit{updated} since $\neg e$ is reachable from $x_0$; i.e. $x_0 \neq p x_0$. All variables, excluding $x_0$, in the program environment of $\mathcal{f}$ are considered to be \textit{used}: a is used in Line 1; $x$ in the first argument of the recursive call; and $x$ is used in the second argument of the recursive call. For $g$, $x_0$ is \textit{not updated} since $(x_0 : xs)$ is syntactically equivalent to $x_0$; i.e. $\text{cons} x_0 \equiv x_0$. However, $y_0$ is \textit{updated} since $(x_0 : y)$ is not syntactically equivalent to $y_0$. All variables apart from $y$ and $x_0$ are considered to be used in $g$: $x_0$ is used both in Line 5 and as the semantically equivalent first argument to the recursive call; and $y$ and $x_0$ are used in the second argument of the recursive call. Finally, for $h$, $x_0$ is \textit{not updated}, since $y$ (which is not reachable from $x_0$) is prepended to $x_0$. However, $y_0$ is \textit{updated} since $y$ is prepended to $y_0$ and $y$ is reachable from $y_0$, $y$, $x_0$, and $y_0$ are all considered to be used in $h$: $x_0$ in Line 9 and the $\text{cons}$ in the first argument of the recursive call; $y_0$ in the $\text{cons}$ in the second argument of the recursive call; $y$ in the $\text{cons}$ of both arguments of the recursive call.

\subsection{4.2 Slicing Algorithm}

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

\textbf{Definition 4.4 (Slice).} Given some program $p$, a program environment $\Gamma_p$, a fixpoint expression $e = \text{fix} (\lambda (f, x_1, \ldots, x_n) \rightarrow e')$, and a variable $x$, where $x$ is the $i^{th}$ argument to $f$, we say that the slice of $e$ with criterion $x$, denoted $\Sigma_{\mathcal{E} \setminus x}$, is the set of variables such that $\forall y \in \Sigma_{\mathcal{E} \setminus x}, (y \sim_x e) \lor (y \neq x \wedge (y \sim_x e) \lor (y \neq x \wedge (y \sim_x e))$.\)

To illustrate the slicing relation, recall the definition of \textit{sameuler},

\begin{verbatim}
1  sameuler x0@[x] = 0
2  sameuler x0@[x]:xs = euler x + (sameuler xs)
\end{verbatim}

Considering each variable in \textit{sameuler}: both $x \in \Sigma_{\text{sameuler}\setminus x_0}$ and $x_0 \in \Sigma_{\text{sameuler}\setminus x_0}$ hold since both $x$ and $x_0$ are reachable from $x_0$ (i.e. $x \neq p x_0$ and $x \neq p x_0$) and both $x$ and $x_0$ are \textit{used} in Line 2; conversely, \textit{neither} $x_0 \in \Sigma_{\text{sameuler}\setminus x}$ nor $x_0 \in \Sigma_{\text{sameuler}\setminus x}$ hold, since $x_0$ is not considered \textit{used}, and $x_0$ is not considered \textit{updated}, in \textit{sameuler}. For the definition of $\textit{sum}$:

\begin{verbatim}
1  sum x y0@[x] = x
2  sum x y0@[y]:ys = (sum (x+y) y)
\end{verbatim}

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

\begin{verbatim}
1  \Sigma_{\text{sameuler}\setminus x_0} = \{x, x_0\}
2  \Sigma_{\text{sum}} = \{x, \bar{x}\}
3  \Sigma_{\text{sum}} = \{y, \bar{y}\}
\end{verbatim}
The slice $\Sigma^e|x$ is calculated using the inference rules from Fig. 4. For clarity of notation, we will write $\Sigma^e|x$ as $\Sigma$ 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 $\Gamma_p$ 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 $i$th argument of $f$; $e$ is the expression in $p$ that is being sliced; and $\Sigma$ is the resulting slice. For literal expressions and variables that are not reachable from $x$, $e$ produces an empty slice, represented by the rules BOOL1, BOOL2, INT, VAR2, and LST1. A variable that is reachable from $x$, as a usage of $x$, produces the slice containing that variable (VAR3). Cons-expressions that are syntactically equivalent to $x$ produce the slice containing $x$ itself (LST2). All other expressions that are not recursive calls produce the union of the slices of their subexpressions, as stated in the rules LST3, CASE, APP, FUN, and FIX. Finally, RECA-P determines whether $x$ is updated in a recursive call, producing the appropriate slice. RECA-P has two main premises that: i) slice all subexpressions that are passed to $f$, aside from the $i$th argument; and ii) determines whether $x$ is updated. The second premise is a disjunction of four implications: i) when $e_1$ is $x$ itself or syntactically equivalent to $x$; ii) when $e_1$ is a variable that is defined via case-discrimination on $x$; iii) when $x$ is prepended by some expression $e_0$ that does not contain a subexpression that is a variable reachable from $x$; and otherwise iv) $x$ is considered updated. In the first case, the slice, $\Sigma_i$, for the $i$th 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, $\Sigma_i$ 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, $\Sigma_i$ is the slice of the subexpression that is prepended to $x$. Finally, in the fourth case, $\Sigma_i$ is the slice of $e_i$ and $x$ is considered to be updated.

We conjecture that the slicing algorithm in Fig. 4 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 $e$, 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 $v \not\approx w$, it follows that $v$ has the same classification as $w$.

**Definition 4.5** (Variable Taint). Given a program $p$, program environment $\Gamma_p$, 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 $\exists x \in \Sigma^e|x \land x \in \Sigma^e|x$. The variable is classified as clean, otherwise.

### 4.4 Classifying Operations

Once all the arguments of $e$ are classified as **global**, **clean**, or **tainted**, we can proceed to classify those 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.

**Definition 4.6** (Operation Obstructiveness). Given some program $p$, some program environment $\Gamma_p$, and some fixpoint expression $e = \text{fix}(\lambda(f, x_1, \ldots, x_n) \rightarrow e')$ in $p$, an operation $g(e_1, \ldots, e_m) \not\approx e'$, when $g \not\approx f$, is classified as obstructive when $\exists i \in [1, m]$ such that $f(\varepsilon_i) \not\approx e_i \lor \forall y, y \not\approx e_i$ where $y$ is classified as tainted. The operation is classified as unobstructive otherwise.

### 5 Examples

In this section we demonstrate our approach on 12 examples, including **sameuler**, **matrix multiplication**, **n-queens**, and the **Smith-Waterman algorithm**. We give parallel speedup results for **sameuler**, **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. Speedups are 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 use reported mutator time, i.e. the amount of time spent solely on executing the program. This gives an indication of real parallel performance. Where we use corryvreckan for parallel speedup results, 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, 7, and 8 are reiterated from [4], where we expand upon the results reported here. semeuler, queens, and matmult have all been compiled using GHC 7.6.3 on Scientific Linux version 3.10. We compiled the examples using the flags: -fprof-blackholing, -fstack-check, -rtsopts, and -O. We found these gave the best general parallel performance.

We have implemented our inference system in Fig. 4 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 $e = \text{fix} \lambda (f, x) \rightarrow e'$ in p, we slice e with respect to each argument $x \in \lambda$. Each x is classified as either clean or tainted. This is then used to classify each operation in e either obstructive or unobstructive. Our implementation can be found at http://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.

<table>
<thead>
<tr>
<th>Example</th>
<th>RFs</th>
<th>Ops</th>
<th>RCs</th>
<th>Args</th>
<th>Unobstructive Ops</th>
<th>Obstructive Ops</th>
<th>Time (µs)</th>
<th>σ (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data.List</td>
<td>18</td>
<td>22</td>
<td>20</td>
<td>31</td>
<td>5</td>
<td>5</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>elem</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>list-ackermann</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>gcd</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>hanoi</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>k-means</td>
<td>4</td>
<td>15</td>
<td>5</td>
<td>14</td>
<td>4</td>
<td>4</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>quicksort</td>
<td>3</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>swaterman</td>
<td>2</td>
<td>10</td>
<td>2</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>ssumeuler</td>
<td>14</td>
<td>47</td>
<td>14</td>
<td>25</td>
<td>16</td>
<td>16</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>matmult</td>
<td>8</td>
<td>23</td>
<td>7</td>
<td>16</td>
<td>6</td>
<td>6</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>queens</td>
<td>2</td>
<td>11</td>
<td>2</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>sudoku</td>
<td>5</td>
<td>9</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 5. Examples run through prototype implementation. Times are an average of 50 runs.

![Figure 6](https://example.com/fhpc17-figure6.png)

Figure 6. Prototype execution times for a program with varying sizes of input. Our implementation is quadratic.

We give an overview of our results in Fig. 5, 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.

```haskell
list-ackermann as0@[] bs0 = (1:bs0)  
list-ackermann as0@(a:as) bs0@[] =  
list-ackermann as [1]  
list-ackermann as0@(a:as) bs0@(b:bs) =  
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 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 SunEuler module m times. As the translated SunEuler has 9 operations, the total number of operations is \( n = m \times 9 \). Fig. 6 shows the average time taken of five runs, in seconds, by our prototype on neptune. Our classifier takes a minimum of 0.33s 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. 5. The longest our prototype takes to classify one of the examples in Fig. 5 is 6.80ms.

### 5.1 SunEuler

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

```haskell
1. sunEuler \( \times x_0 \)\( = \emptyset \)
2. sunEuler \( x_0 \)\( = \text{euler} \ x + (\text{sunEuler} \ x) \)
```

We first slice sunEuler for its only argument, \( x_0 \), \( \Sigma \)\( \text{sunEuler} \mid x_0 = \{ x, x \} \). As before, \( x_0 \) is classified as clean since both case-split variables of \( x_0 \) and \( x \) are used in the body of sunEuler, and \( x_0 \) itself is not updated. Two operations exist as subexpressions to sunEuler: (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 sunEuler 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 sunEuler 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 sunEuler using mutator time. We achieve maximum speedups of 30.50 for \( n = 50000 \) on 48 virtual cores. Sequentially, when \( n = 50000 \) sunEuler has an average runtime of 154.33s (\( \sigma = 0.25s \)). We achieve good speedups in all cases. Moreover, sunEuler demonstrates good scalability as \( n \) increases. Fig. 1 shows that for varying \( n \), speedups reach a limit, likely due to 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 31 of which are classified as obstructive. As with all the NoFib benchmarks, the high number of recursive functions are representative of the original duplication in the underlying Haskell modules.

### 5.2 Matrix Multiplication

The NoFib benchmark uses the following implementation of matrix multiplication.

```haskell
1. matmult \( m1 \ m2 \) = multMatricesTr \ m1 \ (\text{transpose} \ m2))
2. multMatricesTr \ [\] \ m2 = \[
3. multMatricesTr \ (r:\text{rs}) \ m2 = \[
4. \text{f} \ m2 \ r : \text{multMatricesTr} \ rs \ m2
```

We can see the same pattern: all the arguments are classified clean, and the sole operation, \( (\text{prodEscalar2} \ \text{row} \ c : \text{f} \ \text{cs}) \), is classified as unobstructive. Both \( \text{multMatricesTr} \) and \( \text{f} \) can be rewritten as map operations. Parallelism can be introduced in \text{matmult} by applying a custom strategy to the call to \( \text{multMatricesTr} \) 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. 7 gives speedups for matmult using mutator time, achieving maximum speedup of 32.93 for \( n = 2500 \) on 52 hyper-threaded cores. Sequentially, when \( n = 2500 \) matmult takes an average of 226.08ms, with a standard deviation of 7.49s. matmult scales well for both \( n \) and number of cores. On hyper-threaded cores we observe some erratic behaviour with curious superlinear speedups. This is possibly due to caching effects.

Interestingly, the definition of \( \text{f} \) might be unfolded (in the transformational sense) in \text{multMatricesTr}; e.g.

```haskell
1. multMatricesTr \ x0[@\] \ r \ m2 = \[
2. \[
3. \text{multMatricesTr} \ x0\mid (x:\text{xs}) \ \text{rr[@]} \ m2 = \[
4. \text{multMatricesTr} \ m2 \ x \ m2 : \text{multMatricesTr} \ xs \ \text{[]} \ m2
```
5 multMatricesTr xs@(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, xs0. Slicing for its arguments, gives

\[
\begin{align*}
\Sigma_{\text{multMatricesTr}|xs0} &= \{x, xs, xs0\} \\
\Sigma_{\text{multMatricesTr}|rr} &= \{rr, rr\} \\
\Sigma_{\text{multMatricesTr}|m2} &= \{m2\}
\end{align*}
\]

While xs0 is considered to be updated, due to m2 being passed as the first argument in the recursive call in line 4, both xs0 and m2 are considered to be clean. Conversely, rr is considered to be tainted. (prodEscalar2 rr x), now the sole operation, is therefore classified obstructive, meaning 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 function 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 recursive functions over 2 Haskell modules. Within those 8 recursive functions there are 23 operations, of which 6 are classified as unobstructive and 17 are classified as obstructive.

5.3 N-Queens

We use the following implementation of the queens problem,

\[
\begin{align*}
\text{queens } nq &= \text{gen } 0 \ [] \\
\text{gen } nq & n b \\
| n > nq = nq = [b] \\
| \text{otherwise } = \text{genloop } n q (\text{gennext } [b]) \\
\text{genloop } n q n b s @[] = [] \\
\text{genloop } n q n b s @ (b:bs) = \\
\text{gen } n q (n+1) b ++ \text{genloop } n q n b s
\end{align*}
\]

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

\[
\Sigma_{\text{queens}|nq} = \{nq\} \quad \Sigma_{\text{queens}|n} = \{n\} \quad \Sigma_{\text{queens}|bs} = \{b, bs\}
\]

All three arguments, nq, n, and bs, are classified as clean since both nq and n are used but not updated; and both b and bs are used but bs0 is neither used nor updated. genloop has three subexpressions: 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 the expressions passed to (+) and gen respectively. 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 argument is a recursive call, which results in an obstructive classification. 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), to maximise the amount of work done in parallel.

\[
\begin{align*}
\text{genloop } n q n b s 0 &= \\
\text{genloop’ } n q n (\text{map } (\text{gen } n q (n+1)) b s 0) \\
\text{genloop’ } c s @[] = [] \\
\text{genloop’ } c s @ (c:cs) = c ++ \text{genloop’ } cs
\end{align*}
\]

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

\[
\begin{align*}
\text{gen } b x s @[] &= [b] \\
\text{gen } n b x s @[] &= [] \\
\text{gen } n b x s @ (x:xs) &= \\
\text{gen } (n-1) x (\text{gennext } [x]) ++ (\text{gen } n b x s)
\end{align*}
\]

This definition eliminates the mutual recursion of gen and genloop, where xs 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 recursive 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 number of operations in queens, there are only two unobstructive operations in the translation. The majority of operations are found in the definition of safe.

\[
\begin{align*}
\text{safe } x d g s @[] &= \text{True} \\
\text{safe } x d g s @ (q:1) &= \\
x /= q \&\& x /= q+d \&\& x /= q-d \&\& \text{safe } x (d+1)
\end{align*}
\]
we classify $x$ and $y_0$ as clean 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 complicate
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,

\[
(x /= q \&\& x /= q+d \&\& safe x (d+1) 1)
\]

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,

\[
(x /= q \&\& (x /= q+d \&\& safe x (d+1) 1))
\]

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. 8 show speedups for queens in terms of
mutator time. We achieve maximum speedups of 22.65x 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 in simuueler. When 14 $\leq n$ $\leq 16$, queens scales well
until 28 cores; when hyperthreading is enabled, we see some further
improvement in speedup, albeit at reduced rate when compared
with 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 enabling saturation
of the hardware. For $n = 13$ and $n = 14$, 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 popula-
tion 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.

\[
\text{tcs } r c a b m = \begin{cases}
\text{if } c > (\text{length } m) & \text{then } m \\
\text{else tcs } r (c+1) a b (h r c a b m) & \\
\text{trs } r c a b m = \begin{cases}
\text{if } r > (\text{length } m) & \text{then } m \\
\text{else trs } (r+1) c a b (\text{tcs } r c a b m)
\end{cases}
\end{cases}
\]

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

\[
\text{tcs: } \Sigma^{\text{tcs}} r = \{r\} \quad \text{(clean)} \\
\Sigma^{\text{tcs}} c = \{c, c\} \quad \text{(tainted)} \\
\Sigma^{\text{tcs}} a = \{a\} \quad \text{(clean)} \\
\Sigma^{\text{tcs}} b = \{b\} \quad \text{(clean)} \\
\Sigma^{\text{tcs}} m = \{m, m\} \quad \text{(tainted)}
\]

\[
\text{trs: } \Sigma^{\text{trs}} r = \{r, r\} \quad \text{(tainted)} \\
\Sigma^{\text{trs}} c = \{c\} \quad \text{(clean)} \\
\Sigma^{\text{trs}} a = \{a\} \quad \text{(clean)} \\
\Sigma^{\text{trs}} b = \{b\} \quad \text{(clean)} \\
\Sigma^{\text{trs}} m = \{m, m\} \quad \text{(tainted)}
\]

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

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

\text{trs: } Analogous to tcs, and as both $r$ and $m$ are classified $tainted$,
all operations in trs are classified to be obstructive.

No refactoring can be applied to the Smith-Waterman implementa-
tion. 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 goes undetected, but is part of 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,
wheras our approach is also suitable for higher-order languages.
It is designed to only inspect recursive functions that can be trans-
formed 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 to maximise the number
of components that are available for parallelisation. Even if the
introduced maps are eventually deemed not worth parallelising,
the more components available for inspection, the more configura-
tions of parallelism that 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 require a specific language or system to function. 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, and 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 operations that can be lifted into scheme fixpoint functions, Hu et al. use the the Bird Meertens formalism to systematically reason about the behaviour of the function. This approach has been shown to extend to types other than lists, and 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 sound, and which can be a limiting factor. 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 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 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 code. Program shaping approaches [3] that include pattern discovery techniques can potentially find more patterns by refactoring 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. 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 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 intend to expand our property checking to other parallel structures, e.g. divide-and-conquer and feedback patterns [9]. Relatedly, as 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 shift 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.

References

Submitted to FHPC, 2017, Oxford, UK

Adam D. Barwell and Kevin Hammond


