# Accepted Manuscript

Finding parallel functional pearls: Automatic parallel recursion scheme detection in Haskell functions *via* anti-unification

Adam D. Barwell, Christopher Brown, Kevin Hammond

PII: S0167-739X(17)31520-0
DOI: http://dx.doi.org/10.1016/j.future.2017.07.024
Reference: FUTURE 3554

To appear in: *Future Generation Computer Systems*

Received date : 16 December 2016
Revised date : 27 June 2017
Accepted date : 10 July 2017

Please cite this article as: A.D. Barwell, C. Brown, K. Hammond, Finding parallel functional pearls: Automatic parallel recursion scheme detection in Haskell functions *via* anti-unification, *Future Generation Computer Systems* (2017), http://dx.doi.org/10.1016/j.future.2017.07.024

# Finding Parallel Functional Pearls: Automatic Parallel Recursion Scheme Detection in Haskell Functions *via* Anti-Unification

Adam D. Barwell, Christopher Brown, and Kevin Hammond

*School of Computer Science, University of St Andrews, St Andrews, United Kingdom*
{adb23,cmb21,kh8}@st-andrews.ac.uk

**Abstract**

This paper describes a new technique for identifying potentially parallelisable code structures in functional programs. *Higher-order functions* enable simple and easily understood abstractions that can be used to implement a variety of common *recursion schemes*, such as *maps* and *folds* over traversable data structures. Many of these recursion schemes have natural parallel implementations in the form of *algorithmic skeletons*. This paper presents a technique that detects instances of potentially parallelisable recursion schemes in Haskell 98 functions. Unusually, we exploit *anti-unification* to expose these recursion schemes from source-level definitions whose structures match a recursion scheme, but which are not necessarily written directly in terms of *maps*, *folds*, etc. This allows us to automatically introduce parallelism, without requiring the programmer to structure their code *a priori* in terms of specific higher-order functions. We have implemented our approach in the Haskell refactoring tool, HaRe, and demonstrated its use on a range of common benchmarking examples. Using our technique, we show that recursion schemes can be easily detected, that parallel implementations can be easily introduced, and that we can achieve real parallel speedups (up to 23.79× the sequential performance on 28 physical cores, or 32.93× the sequential performance with hyper-threading enabled).

## 1. Introduction

Structured parallelism techniques, such as *algorithmic skeletons*, expose parallelism through suitable language-level constructs [1]. By exploiting a range of high-level, composable patterns that have natural parallel implementations, such as *maps* or *folds*, parallelism can be introduced quickly, easily and automatically. Unlike common lower-level approaches, such as OpenCL, the programmer does not need to explicitly deal with implementation details such as communication, task creation, scheduling, etc. [2]. Structured parallel code also avoids common and difficult-to-debug problems, such as race conditions and deadlocks, and allows the definition of simple, but effective, *cost models* that can accurately

predict parallel runtimes and speedups. There is a close, and long-observed, correspondence between structured approaches and functional programming. In particular, algorithmic skeletons can be seen as parallel implementations of *higher-order functions* (e.g. [3, 4]). There are further advantages to structured parallel approaches when writing parallel software for *heterogeneous* [5] or *high performance* parallel systems [6], which may involve massive numbers of processors, of possibly different types. Given a set of skeletons for the desired target architecture(s), the programmer need only be concerned with which skeletons to call, where they should be called, and what parameters should be given to them so that they give the best performance on the given architecture. Unfortunately, determining precisely where to introduce skeletons can be a non-trivial exercise, requiring significant programmer effort [2, 7, 8]. It follows that if instances of higher-order functions can be discovered automatically, and if cost information can be used to direct the choices of skeletons, parameters, etc., then parallel functional programs can be produced quickly, easily and efficiently. Such patterns occur frequently in real applications. For example, the *spectral* part of the *NoFib* suite of Haskell benchmarks [9] comprises a total of 48 programs. Manual inspection shows that at least 19 (39.6%) of these have one or more functions that could be rewritten in terms of *map* or *fold* patterns. This gives a potentially large corpus of easily parallelisable programs.

### *1.1. Novel Contributions*

This paper introduces a new technique for discovering instances of generalised higher-order functions (recursion schemes), using *anti-unification* [10]. As part of our technique, we define a new, specialised anti-unification algorithm, and demonstrate how pattern arguments can be derived from the substitutions that are inferred by the anti-unification process. Our anti-unification analysis and corresponding refactorings are implemented for Haskell in the HaRe refactoring tool. Our implementation allows the automatic discovery and refactoring of parallelisable recursion schemes, detecting instances that could otherwise be difficult to detect. Moreover, manually checking a large code base for possible instances of parallelisable patterns would be a time-consuming and error-prone task, even for an expert programmer. We have tested our prototype on four standard benchmark programs: *matrix multiplication*, *n-body*, *n-queens*, and *sumeuler*, demonstrating that it can successfully expose the underlying recursion schemes. The corresponding parallel implementations achieve maximum speedups of 32.93 (21.66 using physical cores) for matrix multiplication, 27.08 (23.46 using physical cores) for n-body, 22.65 (20.48 using physical cores) for n-queens, and 30.50 (22.24 using physical cores) for sumeuler, on a 28-core hyper-threaded experimental testbed machine. Our technique is not restricted to Haskell, or to a specific set of recursion schemes, but is, in principle, completely general. It can be applied to a wide range of recursion schemes/skeletons, and to a wide range of programming languages, including both functional languages, such as Erlang, Scala, CAML, or Clojure, and other languages such as C, C++, or Java. The only requirement is that the code structure matches the recursion schemes of interest, and that there are no unwanted side effects.

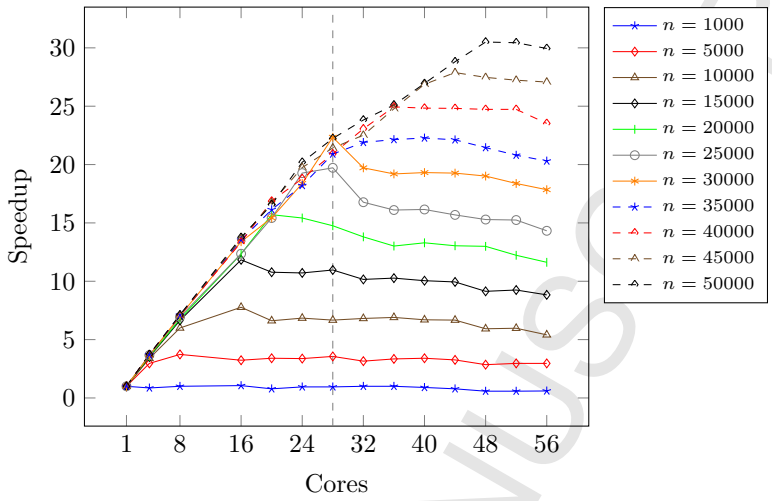


Figure 1: *sumeuler*, speedups on *corryvreckan*, a 28-core hyper-threaded Intel Xeon server, dashed line shows extent of physical cores.

### *1.2. Illustrative Example*

We illustrate our approach using an example from 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 xs = sum (map euler xs)
```

Here, `euler` is Euler's totient function, and `sum` sums the values in its argument, a list of integers. `sumeuler` can be easily parallelised using, e.g., the Haskell Strategies library:

```
1  sumeuler  :: [[Int]] -> Int
2  sumeuler xs =
3    sum (map (sum . map euler) xs `using` parList rdeepseq)
```

Here, `parList`, from the standard `Control.Parallel.Strategies` library, parallelises the `map` over `xs` without changing the original definition. It is parameterised on `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 ensuring that the parallel version is functionally equivalent to the original definition. Fig. 1 shows the raw speedups that we obtain for this program for varying sizes on `n` on our 28-core hyper-threaded experimental machine, *corryvreckan*. We obtain a maximum speedup of 30.50 for `n` = 50000 on 48 hyper-threaded cores. Other parallelisations are possible. Eden [11], the Par Monad [12], GPUs via the Accelerate library [13], etc., can all substitute for `parList` to introduce parallelism over the *map* operation. It is also possible

to rewrite `sumeuler` to use a direct *fold* (or *reduce*) operation. This applies an operation, `g` between each pair of elements in an input list, returning `q`, when the end of the input list is reached. So, for example, `foldr (+) 0 [1,2,3]` is expanded to `foldr (+) 0 ((:) 1 ((:) 2 ((:) 3 [])))`, where `(:)` is the binary list constructor function of type `a -> [a] -> [a]` and `[]` is the empty list. This is reduced to `(+) 1 ((+) 2 ((+) 3 0))`, the result of which is `6`, the sum of the elements in the input list.

```
1 sumeuler [Int] -> Int
2 sumeuler xs = foldr ((+) . euler) 0 xs
3    where foldr g q [] = q
4          foldr g q (w:ws) = g w (foldr g q ws)
```

This new definition of `sumeuler` can then be parallelised in an alternative way to the parallel map above, using e.g. a parallel implementation of `foldr`. Taking advantage of parallelism in `sumeuler` is therefore both *simple* and potentially *automatic* [14]. Moreover, alternative parallelisation approaches are possible. However, if `sumeuler` was defined without using either an explicit `map` or `foldr`, e.g. as shown below using direct recursion, then parallelisation would be less straightforward, and lower, or no, speedups might result.

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

Since this version of `sumeuler` is *implicitly* an instance of `foldr`, it is best to first restructure the definition so that it *explicitly* calls `foldr` before attempting parallelisation. In order to rewrite the inlined `sumeuler` as an explicit `foldr`, however, we need to derive concrete arguments to `foldr` that will yield a functionally equivalent definition. These arguments can be derived by inspecting the inlined definition of `sumeuler`, as shown in Fig. 2. To *automatically* derive the arguments to `foldr`, we inspect the definitions of `sumeuler` and `foldr` using *anti-unification*, which aims to find the *least general* generalisation between two terms. In Plotkin and Reynolds' original work [10, 15], anti-unification was defined for totally ordered terms, where terms consisted of variables, literals, and function application. More recent approaches to anti-unification have applied the technique to programming languages such as Haskell [16, 17], primarily for clone detection and elimination [18]. In these approaches, anti-unification compares two terms (expressions) to find their shared structure, producing an *anti-unifier* term (representing the shared structure), plus two sets of *substitutions* that enable the original term to be reconstructed from the anti-unifier. For example, anti-unifying the `[]` clause of `sumeuler` with `foldr` compares the unequal `0` and `z` expressions, producing the anti-unifier, `h`:

```
1 h g z [] = z
```

Where the anti-unified structures diverge, a variable is introduced in the anti-unifier that can be substituted for the original term. Substitutions, $\sigma$, have the

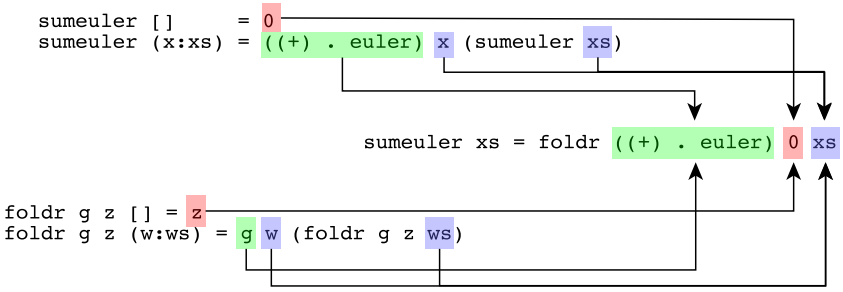


Figure 2: Rewriting `sumeuler` as an instance of `foldr`

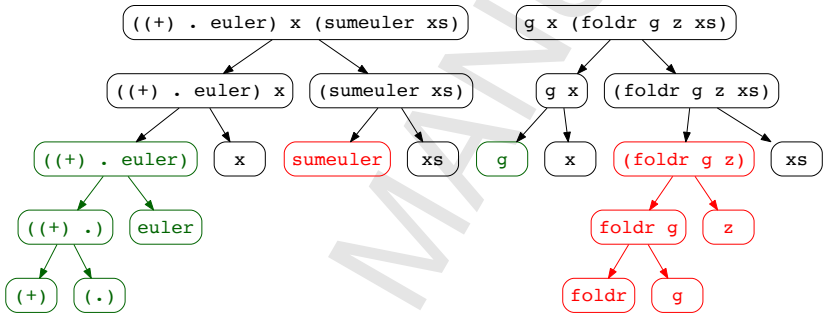


Figure 3: ASTs for the `(:)` branch of `sumeuler` (left) and `foldr` (right); shared structure in black, differing structure in dark green and red.

form $(v \mapsto t_i)$, where $v$ is a variable and $t_i$ is some term. Substitutions are applied to terms, written using postfix notation, e.g. $t\ \sigma$, and can be composed like functions. An applied substitution replaces all instances of $v$ with $t_i$ in $t$. For example, the substitutions for the `[]` clause of `sumeuler` are:

$$\sigma_1 = (\mathtt{z} \mapsto 0)$$

and the substitutions for the `[]` clause of `foldr` are $\varepsilon$, i.e. the identity substitution. Anti-unifying the respective `(:)` clauses of `sumeuler` and `foldr` produces the anti-unifier:

```
1 h g z (x:xs) = g x (α xs)
```

where $\alpha$ is a free variable. As shown in Fig. 3, the structures of the two clauses are very similar, consisting primarily of application expressions. Differences can be found at the leaves of `foldr`, highlighted in both dark green and red in Fig. 3. Since `x` and `xs` are in the same relative positions, they feature in `h`. As `((+) . euler)` $\neq$ `g` (highlighted in dark green), `g` is used to represent

the function applied to two arguments in `h`. Finally, since the recursive call prefixes `sumeuler` $\neq$ `(foldr g z)` (highlighted in red), the variable $\alpha$ is used in their place in `h`. This produces the substitutions: (`g` $\mapsto$ `((+) . euler)`) and ($\alpha \mapsto$ `sumeuler`) for `sumeuler`, and: ($\alpha \mapsto$ `(foldr g z)`) for `foldr`. Since it is possible to reconstruct the original terms by applying the relevant substitutions to an anti-unifier, when given the anti-unifier `h`:

```
h g z []     = z
h g z (x:xs) = g x (a xs)
```

we can, in principle, rewrite `sumeuler` and `foldr` in terms of `h` using substitutions as arguments:

```
sumeuler xs = h ((+) . euler) 0 xs

foldr g z xs = h g z xs
```

Furthermore, because `foldr` is *equivalent* to `h`, we conclude that `sumeuler` must be an instance of `foldr`. As both `sumeuler` and `foldr` can be rewritten in terms of `h`, it must be the case that `sumeuler` can be rewritten in terms of `foldr`. The substitutions for `sumeuler` inferred as part of anti-unification are valid as arguments to `foldr`, allowing `sumeuler` to be rewritten:

```
sumeuler xs = foldr ((+) . euler) 0 xs
```

Parallelism can now be introduced using a parallel implementation `foldr`, either manually or using e.g. a refactoring/rewriting tool (e.g. [19, 8]). Alternatively, the `foldr` operation can be split into its `map` and `foldr` (`sum`) components, perhaps by using the laws of hylomorphisms as in [4], producing:

```
sumeuler xs = foldr (+) 0 (map euler xs)
```

which is equivalent to our original definition of `sumeuler`:

```
sumeuler xs = sum (map euler xs)
```

This can then be parallelised using the parallel map that we originally showed.

## 2. Algorithmic Skeletons and Structured Parallelism

*Algorithmic skeletons* [3] abstract commonly-used patterns of parallel computation, communication and interaction into parameterised templates that can be used to implement high-level patterns of parallelism. There is a long-standing connection between the skeletons community and the functional programming community [20]. In the functional world, skeletons are implemented as *higher-order functions* that can be instantiated with specific user code to give some concrete parallel behaviour [2]. Using a skeleton approach allows the programmer to adopt a top-down *structured* approach to parallel programming, where skeletons are *composed* to give the overall parallel structure of the program. This

gives a flexible approach, where parallelism is exposed to the programmer only through the choice of skeleton and perhaps through some specific behavioural parameters (e.g., the number of parallel processes to be created, or how elements of a parallel list are to be grouped to reduce communication costs). Details of communication, task creation, task or data migration, scheduling, etc. are embedded within the skeleton implementation, which may be tailored to a specific parallel architecture or class of architectures [5].

Common skeletons include: *farm*, *reduce*, *feedback*, and *pipeline* [1]. A *farm* applies some function, `(f :: a -> b)`, to each element in a stream of inputs in parallel, where the number of processes created may be controlled by some parameter. The *farm* skeleton corresponds to the sequential *map* operation. The *reduce* skeleton takes an associative function, `(f :: a -> a -> a)`, and repeatedly applies it to a collection of two or more elements in parallel until one element, i.e. the result, remains. This corresponds to a *fold* when the fold fixpoint function is associative. The *feedback* skeleton enables looping operations for skeletons, taking some skeleton, `s`, to which inputs are applied repeatedly until some function, `(f :: a -> Bool)`, is satisfied. This corresponds to an *iterate*, where the result is the $n^{\text{th}}$ element of the produced list. Finally, the *pipeline* skeleton enables the composition of two or more skeletons, where parallelism arises from executing multiple pipeline stages in parallel. This corresponds to function composition `(.)`. Many implementations of these skeletons exist, e.g. [1, 7, 8, 12, 11, 19, 21, 22, 23], with some for multiple architectures such as GPUs, e.g. [5, 13, 24]. The choice of implementation is beyond the scope of this paper. In principle, however, this correspondence allows skeletons to be introduced over their sequential equivalents, introducing parallelism *for free* [4]. This paper takes full advantage of these correspondences. Our anti-unification technique therefore focuses on finding instances of recursion scheme implementations such as `map`, `foldr`, and `iterN`.

```
1  iterate :: (a -> a) -> a -> [a]
2  iterate f x =  x : iterate f (f x)
3  -- iterate ((+) 1) 1 = [1, 1+1, 1+2, 1+3,...]
4
5  iterN :: Int -> (a -> a) -> a -> a
6  iterN n f x = iterate f x !! n
7  -- iterN 1 ((+) 1) 1 = 2
8
9  map :: (a -> b) -> [a] -> [b]
10 map f [] = []
11 map f (x:xs) = f x : map f xs
12 -- map ((+) 1) [1,2,3,..] = [1+1,1+2,1+3,..]
13
14 foldr :: (a -> a -> a) -> a -> [a] -> a
15 foldr f z []     = z
16 foldr f z (x:xs) = f x (foldr f z xs)
17 -- foldr (+) 0 [1,2,3,4,5] = 1 + (2 + (3 + (4 + (5 + 0))))
```

| | | |
|---|---|---|
| $\mathcal{P}$ | = | Set of pattern implementations |
| `f` | = | Function to be transformed |
| `p` | = | A function in $\mathcal{P}$ |
| `h` | = | Result of the anti-unification of `f` and `p` |
| `f'` | = | `f` rewritten as an instance of `p` |
| $a_i$ | = | Argument of `f` or `p` being recursed over |
| $v_{a_i}$ | = | Variable declared in $a_i$ |
| $t, t_i$ | = | Terms for anti-unification |
| $\sigma, \sigma_i$ | = | Substitutions for anti-unification |
| $\varepsilon$ | = | The identity substitution |
| $\alpha$ | = | Hedge variables, found only in `h` |

Figure 4: Key to terms and notation.

These higher-order functions can be automatically introduced for suitable Haskell 98 data types, using e.g. the Haskell type class `Functor` for `map` pattern instances. We assume they are supplied in a library such as the Haskell standard Prelude. Here, `iterate` creates an `n`-element list where the first item is calculated by applying the function `f` to the second argument, `x`; the second item is computed by applying `f` to the previous result and so on. The example on Line 3 shows how `iterate` can be used to generate an infinite list of incrementing integers. For convenience, we also define `iterN`, which takes the $n^{\text{th}}$ element from the result of `iterate`. Line 7 illustrates this. `map` applies a given function (`((+) 1)`, which defines the function that adds 1 to its argument) to a list of elements. Line 12 shows how map can be used to apply the function `((+) 1)` to a list of integers, `[1,2,3,..]`, to produce a list of integers where each element is incremented by one. As discussed above, `foldr` reduces a list by replacing `(:)` with a binary right associative function. The result reduces the elements of the list by applying the binary function, `f`, pair-wise to the elements of the list. Line 17 shows how `foldr` can be used to sum a list. For the rest of this paper, we refer to these higher-order functions as *patterns*, and we denote these patterns collectively by $\mathcal{P}$ (see Figure 4).

While some of the above parallel patterns have additional requirements, such as associativity of operation for `fold`, proving these requirements is beyond the scope of this paper. Through the discovery of these recursion schemes, we are able to expose *components* available for potential parallelisation. Further property checking, such as those for associativity, and further transformations that adjust the configuration, such as chunking [25], may be necessary or worthwhile to produce the best speedups. By exposing components for parallelism, we expose properties, such as how a data structure is traversed, and are able to effectively guide the programmer to where any extra property checking is required and where parallelism might be introduced.

## 3. Detecting and Refactoring Pattern Instances for Haskell

This section gives an overview of our technique. Fig. 4 provides a key to our notation. Some function `f` that matches a pattern, `p`, in the set of patterns, $\mathcal{P}$, is rewritten to a new function, `f'`, that uses `p` and that is functionally equivalent to `f`. Our approach determines whether `f` is an instance of `p`, and if so, derives the arguments that are needed to make `f` and `p` functionally equivalent. This is achieved in two main stages: *i)* anti-unifying `f` and `p` to derive argument candidates; and *ii)* using the result of the anti-unification to determine if `f` is an instance of `p`, and if so, validating the argument candidates. Our approach compares two functions, and given a finite set of $\mathcal{P}$, can be applied repeatedly to discover a set of potential rewrites. When multiple rewrites are valid, we take the first, although other selection methods are also valid. We give our assumptions in Sec. 3.1; we define a specialised anti-unification algorithm in Sec. 3.2; and define the properties to check whether `f` can be rewritten as a call to `p`. We give definitions for all our concepts, and provide a proof of soundness for our anti-unification algorithm.

### *3.1. Preliminaries and Assumptions*

We illustrate our approach using the Programatica AST representation of the Haskell 98 standard [26]. Programatica is used as a basis for the Haskell refactorer, HaRe, which we extend to implement our approach. Our prototype implementation is available at `https://adb23.host.cs.st-andrews.ac.uk/hare.html`. Programatica represents a Haskell AST using a collection of 20 Haskell data types, comprising a total of 110 constructors [16]. These types are parameterised by the location of the syntactic elements they represent, and in the case of variables and constructors, where they are declared. For example, the expression 42 is represented in Programatica by `(Exp (HsLit loc (HsInt 42)))`. Here, `HsLit` indicates that 42 is an expression; `HsInt` indicates that it is an integer; and `loc` represents its location in the source code. We refer to the representation of Haskell *expressions* in Programatica as *terms*; i.e. given some expression $e$ and some term $t$, we say that $t$ is the Programatica representation of $e$ (denoted $[\![e]\!] = t$). In the above example, $[\![42]\!]$ = `(Exp (HsLit loc (HsInt 42)))`; i.e. `(Exp (HsLit loc (HsInt 42)))` is the *term* that represents the *expression* 42. There is a one-to-one mapping between expressions and terms. In the Programatica tool set, terms are values of the `HsExpI` type. We make the distinction between expressions and terms since terms can be easily generalised over using their constructors, as we do in Sec. 3.2.

Although in principle we anti-unify the Haskell functions `f` and `p`, we do so in a structured way that we define in Sec. 3.2. Beyond this, we do not need to consider (the representations of) arbitrary Haskell declarations or modules since we only anti-unify terms that form the right-hand side of like-equations in `f` and `p`. Since location information will mean that any two compared terms will *always* be unequal, we *discard location information.* Finally, and to simplify our presentation, we will omit the outer `Exp` constructor, the common `Hs` prefix of constructors, and any explicit specification of literal terms (e.g. integer or

string). For example, `(Exp (HsLit loc (HsInt 42)))` is instead recorded as `(Lit 42)`. Where variables and constructors are both represented as identifiers, we will instead record these using `Var` and `Con`, respectively. For example, the term of the variable expression `x` is `(Var x)`, and the term of the *cons* operator expression, `(:)`, is `(Con (:))`. While our approach works for all terms, here we only need to explicitly refer to: `Var` (representing variables), `Lit` (representing literals), and `App` (representing application expressions). All other terms fall under the generic constructor $C$, and lists of terms (e.g. $[t_1, \ldots, t_n]$) that used as arguments to $C$ to represent lists and tuples. Other common expressions found in functional languages, such as lambdas, fall under the general case since they are both unlikely to appear in recursion scheme implementations, and are otherwise safe to anti-unify due to the assumptions below.

*Assumptions.* In order to determine whether `f` is an instance of `p`, we will assume:

1. that variables are unique in `f` and `p`;
2. that no variables are free in `p`, and that all variables in `p` are declared as arguments to `p`;
3. that `f` and `p` recurse on the same type (denoted $\tau$);
4. that `f` and `p` only pattern match on the argument that is being traversed (denoted $a_i$, where $a_i$ is the $i^{\text{th}}$ argument to `f` and/or `p`);
5. that for every clause that matches a constructor of $\tau$ in `p`, there is a corresponding clause that matches the same constructor in `f`, and vice versa; and
6. that no parameter to $a_i$ (denoted $v_{a_i}$) occurs as part of any binding in an as-pattern, `let`-expression, or `where`-block.

Clauses in `p` are always anti-unified with like clauses in `f`, according to the constructor of $\tau$ matched for that clause, $C$. $a_i$ then acts as 'common ground' between `f` and `p`; i.e. both functions declare the parameters of the constructor in $a_i$ as arguments. This enables a check to ensure that the inferred substitutions are *sensible* (Def. 14). It is therefore useful to know which $v_{a_i}$ in `f` correspond to which $v_{a_i}$ in `p`. Despite this, since variables in `f` and `p` are assumed to be unique, it follows that no variable term in `f` will ever be syntactically equal to any variable term in `p`. We instead consider $v_{a_i}$ with the same position in `f` and `p` to be equivalent.

**Definition 1** (Shared Argument Equivalence)**.** *Given the argument of type $\tau$ that is pattern matched in both `f` and `p`, $a_i$, where in `f`, $a_i = (D\ v_1\ \ldots\ v_n)$, and in `p`, $a_i = (D\ w_1\ \ldots\ w_n)$, we say that each $v_i$ is equivalent to each $w_i$ (denoted $v_i \equiv w_i$); i.e. $\forall i \in [1, n]$, $v_i \equiv w_i$. Given two arbitrary $v_{a_i}$ in `f` and `p`, $v^{\mathtt{f}}_{a_i}$ and $v^{\mathtt{p}}_{a_i}$, we denote their equivalence by $v^{\mathtt{f}}_{a_i} \equiv v^{\mathtt{p}}_{a_i}$.*

In `sumeuler` and `foldr` from Sec. 1.2, for example, `x` and `xs` are $v_{a_i}$ in `sumeuler`, and `w` and `ws` are similarly $v_{a_i}$ in `foldr`. Here, both `x` $\equiv$ `w` and `xs` $\equiv$ `ws` hold, since `x` and `w` are the first arguments to their respective *cons* operations, and `xs` and `ws` are the second arguments.

Traditional anti-unification algorithms use syntactic equality to relate two terms, $t_1$ and $t_2$, their anti-unifier, $t$, and the substitutions $\sigma_1$ and $\sigma_2$. To take advantage of argument equivalence we must use a weaker form of syntactic equality. First, we define the binary relation $\sim$ over terms, a form of alpha equivalence.

**Definition 2** (Alpha Equivalent Terms)**.** *Given two variables, $v^{\mathtt{f}}_{a_i}$ and $v^{\mathtt{p}}_{a_i}$, we say that the term representations of $v^{\mathtt{f}}_{a_i}$ and $v^{\mathtt{p}}_{a_i}$ are equivalent (denoted $[\![v^{\mathtt{f}}_{a_i}]\!] \sim [\![v^{\mathtt{p}}_{a_i}]\!]$) when:*

$$\frac{v^{\mathtt{f}}_{a_i} \equiv v^{\mathtt{p}}_{a_i}}{\mathtt{Var}\ v^{\mathtt{f}}_{a_i} \sim \mathtt{Var}\ v^{\mathtt{p}}_{a_i}}$$

We then replace syntactic equality with a weaker form of equivalence using Def. 2.

**Definition 3** (Syntactic Equivalence)**.** *We define syntactic equivalence to be a binary relation over two terms, $t_1$ and $t_2$, denoted $t_1 \cong t_2$, where $\cong$ is the reflexive structural closure of $\sim$.*

For example, (`Lit 42`) $\cong$ (`Lit 42`) holds; $t_1 \cong t_2$ holds for all $t_1 = (C\ t_{11}\ \ldots\ t_n)$ and $t_2 = (C\ t_{21}\ \ldots\ t_{2n})$ when $\forall i \in [1, n]$, $t_{1i} \cong t_{2i}$; and finally, (`Var` $v^{\mathtt{f}}_{a_i}$) $\cong$ (`Var` $v^{\mathtt{p}}_{a_i}$) holds *only* when $v^{\mathtt{f}}_{a_i} \equiv v^{\mathtt{p}}_{a_i}$ is true. We note that for all other variables, $v, w$, (`Var` $v$) $\cong$ (`Var` $w$) *does not hold.*

Where $\tau$ is a product type, e.g. as in `zipWith`, we permit pattern matching on all arguments that represent the data structure(s) being traversed. All other arguments must be declared as simple variables. For example, given:

```
f1 a b c [] [] = []

f2 0 (b',b'') c [] = []
```

`f1` is permitted, but `f2` is not.

As-patterns, `let`-expressions, and `where`-blocks all enable occurrences of some $v_{a_i}$ to be aliased, or *obfuscated*, and thereby potentially obstruct argument derivation. Since our analysis inspects the syntax of `f` and `p`, such patterns and expressions can obfuscate the exact structure of `f`; i.e. lift potentially important information out of the main body of `f` and `p`. For example, given the definition,

```
f3 [] = []
f3 (x:xs) = g1 x : f3 xs where g1 = g2 xs
```

the fact that `xs` is passed to `g2` is obfuscated by the `where`-block, and could lead to an incorrect rewriting of `f3` as, e.g., a `map`.

We do not restrict the type of recursion; general recursive forms are allowed, for example. Partial definitions are also allowed. For example, both `zipWith`

and `zipWith1` are valid as implementations of the general *zipWith* recursion scheme:

```
zipWith g []     [] = []
zipWith g (x:xs) (y:ys) = g x y : zipWith xs ys

zipWith1 g1 g2 g3 []     [] = []
zipWith1 g1 g2 g3 (x:xs) [] = g1 x : zipWith1 xs []
zipWith1 g1 g2 g3 []     (y:ys) = g2 y : zipWith1 [] ys
zipWith1 g1 g2 g3 (x:xs) (y:ys) = g3 x y : zipWith1 xs ys
```

This permits more implementations of a given scheme, and so increases the likelihood of discovering an instance of a scheme in `f`.

Finally, and since our approach effectively requires that `f` has a similar syntactic structure to `p` as a result of the anti-unification, we permit any valid and finite normalisation procedure that rewrites some arbitrary `f0` to `f`, such that `f0` *is functionally equivalent to* `f`. Normalisation can be used, e.g., to ensure that any of the above assumptions are met. For example, consider assumption 6: as-patterns and definitions in `let`-expressions and `where`-blocks can be inlined (or *unfolded* in the transformational sense). For example, the `where`-block definition of `g1` in `f3` can be unfolded to produce:

```
f3 [] = []
f3 (x:xs) = g2 xs x : f3 xs
```

Alternatively, normalisation procedures can be used to *reshape* [2] functions into a form to allow, or simplify, the discovery of recursion schemes. For example, the definition,

```
f4 xs0 = case xs0 of
           [] -> 0
           (x:xs) -> x + f xs
```

can be rewritten to lift the case-split into a pattern match, allowing the function to be anti-unified against a `foldr`:

```
f4 [] = 0
f4 (x:xs) = x + f xs

foldr g z [] = z
foldr g z (x:xs) = g x (foldr g z xs)
```

More ambitious normalisation procedures may split a function, e.g. `f5`:

```
f5 []     z = g2 z
f5 (x:xs) z = f5 xs (g1 x z)
```

which can be can considered a *near fold*. Here, we can lift out the application of `g2` in the base case, and therefore expose a `foldl` instance:

```
f5 xs z = g2 (f5' xs z)

f5' [] z = z
f5' (x:xs) z = f5' xs (g1 x z)
```

Splitting of functions in this way is allowed provided that any such splitting is *finite* and *reduces the size of the split function*. For example, some `f0` cannot be split into the composition of `f0'` and `f0''`, where `f0'` or `f0''` is functionally equivalent to the identity operation. In line with the intention for `f` to be anti-unified against a set of recursion schemes, we do *not* require the normalisation procedure to be confluent.

### *3.2. Argument Derivation via Anti-Unification*

In order to derive arguments to express `f` as an instance of `p`, we must anti-unify `f` and `p`. We define anti-unification for two levels: *i)* at the term level for two arbitrary terms; and *ii)* at the function level where we anti-unify `f` and `p`. At the term level, the anti-unification of two terms, $t_1$ and $t_2$, (denoted $t_1 \triangleq t_2$) we obtain the triple $(t \times \sigma_1 \times \sigma_2)$, where $t$ is the anti-unifier with respect to the substitutions $\sigma_1$ and $\sigma_2$. At the function level, the anti-unification of `f` and `p` produces a list of triples that represents the results of anti-unification for each pair of clauses in `f` and `p`.

We define a *substitution* to be a function that takes a variable, $x$, and returns a term, $t_2$. When applied to a term, $t_1$, a substitution returns $t_1$ *but with all occurrences of* $x$ *replaced by* $t_2$.

**Definition 4** (Substitution)**.** *Given two terms,* $t_1$,$t_2$*, and a variable name,* $x$*, that occurs in* $t_1$*, substituting* $x$ *for* $t_2$ *in* $t_1$ *replaces all occurrences of* (`Var` $x$) *in* $t_1$ *with* $t_2$*. This substitution is denoted* $(x \mapsto t_2)$*; and the application of substitutions to terms is denoted* $t_1\,(x \mapsto t_2)$*, where* $(x \mapsto t_2)$ *is applied to* $t_1$*. We use* $\sigma$ *to refer to substitutions; e.g.* $\sigma = (x \mapsto t_2)$*.*

Here, we update the classical definition of substitution [15] to account for the Programatica representation. Substitutions are denoted as substituting a variable *name* for some term. When applied to a term, a substitution replaces all *variable terms that are parameterised by the given variable name* with another term. For example, the result of the substitution $\sigma$ applied the term $t$, $t\,\sigma$, is:

$$
\begin{aligned}
t &= \texttt{Var g} \\
\sigma &= (\texttt{g} \mapsto (\texttt{App (App (Var .) (Var +)) (Var euler)})) \\
t\,\sigma &= (\texttt{App (App (Var .) (Var +)) (Var euler)})
\end{aligned}
$$

Here, `g` is substituted for a term representing the expression `((+) . euler)`, from `sumeuler`. While the *name* of the variable being substituted is given in $\sigma$, it is the variable term parameterised by `g`, i.e. (`Var g`), in $t$ that is substituted. We call the variables in substitutions that are substituted for terms *hedge variables*. Uniqueness of variables means that all occurrences of a hedge variable in a term can be substituted safely.

**Definition 5** (Identity Substitution)**.** *The identity substitution, denoted $\varepsilon$, is defined such that for all terms $t$, $t = t\,\varepsilon$.*

Substitutions can be composed using an n-ary relation such that composition forms a rose tree of substitutions.

**Definition 6** (Substitution Composition)**.** *For all terms $t_0, \ldots, t_n$, and for all substitutions $\sigma_1, \ldots, \sigma_n$, where $t_0 = (C\,t_1\ \ldots\ t_n)$, there exists some substitution $\sigma_0 = (\!|\,\sigma_1, \ldots, \sigma_n\,|\!)$ such that $t_0\,\sigma_0 = (C\ (t_1\,\sigma_1)\ \ldots\ (t_n\,\sigma_n))$. Similarly, where $t_0 = [t_1, \ldots, t_n]$, there exists some substitution $\sigma_0 = (\!|\,\sigma_1, \ldots, \sigma_n\,|\!)$ such that $t_0\,\sigma_0 = [(t_1\,\sigma_1), \ldots, (t_n\,\sigma_n)]$.*

*For the function $f$, defined*

$$\begin{aligned} f\ v_{11}\ \ldots\ v_{1n} &= e_1 \\ &\vdots \\ f\ v_{21}\ \ldots\ v_{2n} &= e_n \end{aligned}$$

*and for the substitutions $\sigma_1, \ldots, \sigma_n$, there exists some substitution*

$$\sigma_0 = (\!|\,\sigma_1, \ldots, \sigma_n\,|\!)$$

*such that $[\![f]\!]\,\sigma_0$ is defined:*

$$\begin{aligned} f\ v_{11}\ \ldots\ v_{1n} &= e'_1 \\ &\vdots \\ f\ v_{21}\ \ldots\ v_{2n} &= e'_n \end{aligned}$$

*where*

$$\forall i \in [1, n],\ [\![e'_i]\!] = [\![e_i]\!]\,\sigma_i$$

Composed substitutions are *ordered*, such that when applied to some term $t_0$, the first composed substitution is applied to the first direct subterm to $t_0$, the second composed substitution is applied to the second direct subterm, and so on. Each composed substitution is only applied to its corresponding direct subterm in $t_0$, and does not interfere with any other subterm of $t_0$. For example, the expression `(g x x)` is represented by the term $t_0$,

$$t_0 = \texttt{App}\ (\texttt{App}\ (\texttt{Var g})\ (\texttt{Var x}))\ (\texttt{Var x})$$

and given the substitution, $\sigma_0$,

$$\sigma_0 = (\!|\,(\!|\,(\texttt{g} \mapsto \texttt{Var euler}), \varepsilon\,|\!), (\texttt{x} \mapsto \texttt{Var p})\,|\!)$$

when we apply $\sigma_0$ to $t_0$, the resulting term is:

$$t_0\,\sigma_0 = \texttt{App}\ (\texttt{App}\ (\texttt{Var euler})\ (\texttt{Var x}))\ (\texttt{Var p})$$

A composition of substitutions may be applied to either `f` or `p`, where each substitution applies to the right-hand side of each equation in order. We will refer to such compositions by $\sigma_{\texttt{f}}$ and $\sigma_{\texttt{p}}$. For example, given the function `sum`,

```
sum [] = []
sum (x:xs) = x + (sum xs)
```

and the substitution $\sigma_{\texttt{f}} = (\!|\sigma_1, \sigma_2|\!)$, applying $\sigma_{\texttt{f}}$ to `sum`, applies $\sigma_1$ to the term representation of `[]`, i.e. the right-hand side of the first equation, and $\sigma_2$ to the term representation of `(x + (sum xs))`, i.e. the right-hand side of the second equation.

Anti-unification computes the *anti-unifier* of two terms. In the literature, all hedge variables are *fresh*. Conversely, and since we aim to rewrite `f` in terms of `p`, we can use the variables declared in `p` as hedge variables. This allows us to easily derive expressions that can be passed to `p` in `f`. We refer to the set of hedge variables from `p` by $\mathcal{V}_{\texttt{p}}$. All hedge variables not in $\mathcal{V}_{\texttt{p}}$ are fresh hedge variables, which we denote by $\alpha$.

Another difference to traditional anti-unification algorithms can be found the concept of *unobstructive recursive prefixes*. As we are anti-unifying two different recursive functions, by definition they will diverge in at least one respect: their recursive calls. We extend this notion to *recursive prefixes*, where a recursive prefix is a variable expression comprising the name of the function, i.e. `f` or `p`, or an application expression that applies `f` or `p` to one or more expressions. For example, given the definition of `foldr`,

```
foldr g z [] = z
foldr g z (x:xs) = g x (foldr g z xs)
```

the recursive call `(foldr g z xs)`, has four recursive prefixes: *i)* `(foldr g z xs)`; *ii)* `(foldr g z)`; *iii)* `(foldr g)`; and *iv)* `foldr`. It is useful to further extend the notion of recursive prefixes with the concept of *unobstructiveness*. We define an *unobstructive recursive prefix* to be a recursive prefix for which the values of its arguments *do not change*. In the above example, all recursive prefixes, with the exception of `(foldr g z xs)`, are *unobstructive*.

**Definition 7** (Unobstructive Recursive Prefix)**.** *Given a recursive function $f$,*

$$f\ v_1\ \ldots\ v_n = e$$

*for all recursive calls in $e$,*

$$f\ e_1\ \ldots\ e_n$$

*we say that a recursive prefix is either:*

1. *the direct subexpression $f\ e_1\ \ldots e_m$, where $1 \leq m \leq n$; or*
2. *the expression $f$ itself.*

*We say that a recursive prefix is* unobstructive *when $\forall i \in [1, m],\ e_i = v_i$. An expression, $e'$ is denoted as being an unobstructive recursive prefix by membership of the set of unobstructive recursive prefixes for $f$; i.e. $e' \in \mathcal{R}_f$.*

To help ensure that as many instances of some scheme are found, we will also assume the existence of the identity operator, which is defined in Haskell as `id`.

**Definition 8** (The Identity Operation)**.** *We define id to be the lambda expression* `(\ x -> x)` *such that for all terms, t,* $(\texttt{App}\ (\texttt{Var}\ id\ )(t)) \cong t$.

Our anti-unification algorithm calculates the anti-unifier $t$ and substitutions $\sigma_1$ and $\sigma_2$ for the terms $t_1$ and $t_2$, such that $t_1 \cong t\,\sigma_1$ and $t_2 \cong t\,\sigma_2$ hold, denoted:

$$t_1 \triangleq t_2 = t \times \sigma_1 \times \sigma_2$$

Our rules in Fig 5 define how to infer $t$, $\sigma_1$, and $\sigma_2$ from $t_1$ and $t_2$. Rules have the form,

$$\frac{p}{\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} \cong (t) \begin{bmatrix} \sigma_1 \\ \sigma_2 \end{bmatrix}}$$

where $p$ denotes any additional assumptions necessary for $t_1 \cong t\,\sigma_1$ and $t_2 \cong t\,\sigma_2$ to hold. We use matrix notation as shorthand for the two equations; i.e.

$$\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} \cong (t) \begin{bmatrix} \sigma_1 \\ \sigma_2 \end{bmatrix} = t_1 \cong t\,\sigma_1 \wedge t_2 \cong t\,\sigma_2$$

When $t_1$ and $t_2$ are the same, i.e. $t_1 \cong t_2$, EQ holds. For all other cases when $t_2$ is a variable term, and is not $(\texttt{Var p})$, $t_2$ is used as the hedge variable in $t$ (VAR). When $t_2$ is an application term applying some function $u$ to a $v_{a_i}$, and $t_2$ is a variable term that is equivalent to the $v_{a_i}$ in $t_2$, then the application term is used in $t$, and $u$ is substituted for $id$ in $\sigma_1$ (ID). When $t_2$ is an unobstructive recursive prefix of `p`, RP holds. CONST states that when $t_1$ and $t_2$ are terms with the same constructor, $C$, then $t$ is a term with constructor $C$ and its direct subterms are the anti-unifiers of each of the respective direct subterms in $t_1$ and $t_2$. LIST is similar to CONST, but satisfies when the $t_1$ and $t_2$ are lists of terms, where $t_1$ and $t_2$ are the same length. In all other cases, OTHERWISE holds. For example, the terms, $t_1$ and $t_2$,

$$t_1 = \texttt{App}\ (\texttt{Var e})\ (\texttt{Lit 42})$$
$$t_2 = \texttt{App}\ (\texttt{Var g})\ (\texttt{Lit 108})$$

both apply some function to some literal, and have the anti-unifier and substitutions:

$$t = (\texttt{App}\ (\texttt{Var g})\ (\texttt{Var}\ \alpha))$$
$$\sigma_1 = (\!|\, (\texttt{g} \mapsto (\texttt{Var e})), (\alpha \mapsto \texttt{(Lit 42)})\,|\!)$$
$$\sigma_2 = (\!|\, \varepsilon, (\alpha \mapsto \texttt{(Lit 108)})\,|\!)$$

Here, $\sigma_1$ and $\sigma_2$ are compositions reflecting the two arguments to `App` (CONST). Since the first argument in both $t_1$ and $t_2$ are variable terms, but with different names, $t_2$ is chosen as the anti-unifier with the substitution for $\sigma_1$ replacing `g` with `e`, and the identity substitution for $\sigma_2$ (VAR). Conversely, the second argument differs in $t_1$ and $t_2$, and since `(Lit 108)` is not an identifier, a fresh hedge variable, $\alpha$, is used for $t$, with the corresponding substitutions replacing `(Var` $\alpha$`)` in `(Lit 42)` and `(Lit 108)` for $t_1$ and $t_2$, respectively (OTHERWISE).

$$\text{EQ}\ \frac{t_1 \cong t_2}{\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} \cong (t_2) \begin{bmatrix} \varepsilon \\ \varepsilon \end{bmatrix}}$$

$$\text{VAR}\ \frac{v \neq \mathtt{p}}{\begin{bmatrix} t_1 \\ \mathtt{Var}\ v \end{bmatrix} \cong (\mathtt{Var}\ v) \begin{bmatrix} (v \mapsto t_1) \\ \varepsilon \end{bmatrix}}$$

$$\text{ID}\ \frac{u \neq \mathtt{p} \qquad v^1_{a_i} \equiv v^2_{a_i}}{\begin{bmatrix} \mathtt{Var}\ v^{\mathtt{f}}_{a_i} \\ \mathtt{App}\ (\mathtt{Var}\ u)\ (\mathtt{Var}\ v^{\mathtt{p}}_{a_i}) \end{bmatrix} \cong (\mathtt{App}\ (\mathtt{Var}\ u)\ (\mathtt{Var}\ v^{\mathtt{p}}_{a_i})) \begin{bmatrix} (\!|(u \mapsto id), \varepsilon|\!) \\ \varepsilon \end{bmatrix}}$$

$$\text{RP}\ \frac{t_2 \in \mathcal{R}_{\mathtt{p}}}{\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} \cong (\mathtt{Var}\ \alpha) \begin{bmatrix} (\alpha \mapsto t_1) \\ (\alpha \mapsto t_2) \end{bmatrix}}$$

$$\text{CONST}\ \frac{\forall i \in [1, n],\ \begin{bmatrix} t_{1i} \\ t_{2i} \end{bmatrix} \cong (t_i) \begin{bmatrix} \sigma_{1i} \\ \sigma_{2i} \end{bmatrix}}{\begin{bmatrix} C\ t_{11}\ \ldots\ t_{1n} \\ C\ t_{21}\ \ldots\ t_{2n} \end{bmatrix} \cong (C\ t_1\ \ldots\ t_n) \begin{bmatrix} (\!|\sigma_{11}, \ldots, \sigma_{1n}|\!) \\ (\!|\sigma_{21}, \ldots, \sigma_{2n}|\!) \end{bmatrix}}$$

$$\text{LIST}\ \frac{\forall i \in [1, n],\ \begin{bmatrix} t_{1i} \\ t_{2i} \end{bmatrix} \cong (t_i) \begin{bmatrix} \sigma_{1i} \\ \sigma_{2i} \end{bmatrix}}{\begin{bmatrix} [t_{11}, \ldots, t_{1n}] \\ [t_{21}, \ldots, t_{2n}] \end{bmatrix} \cong ([t_1, \ldots, t_n]) \begin{bmatrix} (\!|\sigma_{11}, \ldots, \sigma_{1n}|\!) \\ (\!|\sigma_{21}, \ldots, \sigma_{2n}|\!) \end{bmatrix}}$$

$$\text{OTHERWISE}\ \frac{}{\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} \cong (\mathtt{Var}\ \alpha) \begin{bmatrix} (\alpha \mapsto t_1) \\ (\alpha \mapsto t_2) \end{bmatrix}}$$

Figure 5: Inference rules to calculate the anti-unifier $t$ for the terms $t_1$ and $t_2$.

As we are interested in finding instances of a pattern in a function, it is convenient to relax the *least general* property found in traditional anti-unification algorithms. Specifically, the RC rule in Fig. 5 can result in an anti-unifier that is not the least general generalisation of two terms. To illustrate this, consider the example of `elem` anti-unified against a `foldr`, where

```
1  elem a [] = False
2  elem a (x:xs) =
3    (\y ys -> if a == y then True else ys) x (elem a xs)
4
5  foldr g z [] = z
6  foldr g z (x:xs) = g x (foldr g z xs)
```

Here, the anti-unification of the recursive calls in the *cons*-clauses produces the term:

$$\texttt{App}\ (\texttt{Var}\ \alpha)\ (\texttt{Var}\ \texttt{xs})$$

since `(foldr g z)` is an unobstructive recursive prefix. In this case, the RC rule applies instead of CONST. CONST would otherwise apply since both `(foldr g z)` and `(elem a)` are application expressions, and would produce a less general generalisation than the result of applying the RC rule. Intuitively, the least general generalisation is the term $t$ that shares the closest structure to $t_1$ and $t_2$, and requires the least number of substitutions in $\sigma_1$ and $\sigma_2$. In the above example, the least general generalisation for the recursive call is:

$$\texttt{App}\ (\texttt{App}\ (\texttt{Var}\ \alpha)\ (\texttt{Var}\ \texttt{z}))\ (\texttt{Var}\ \texttt{xs})$$

The 'shortcuts' produced by application of the RC rule are *useful*. Unobstructive recursive prefixes are uninteresting, aside from their relative location, since they produce no valid *argument candidates* in $\sigma_{\texttt{f}}$ (Sec. 3.3). Moreover, without the RC rule, extra work would be necessary to derive and/or choose between argument candidates.

**Theorem 1** (Soundness of Anti-Unification Algorithm)**.** *Given the terms $t_1$ and $t_2$, we can find $t$, $\sigma_1$, and $\sigma_2$, such that $t_1 \cong t\,\sigma_1$ and $t_2 \cong t\,\sigma_2$ hold.*

We give the proof for the above soundness property in Appendix A.

Having defined anti-unification for terms, we can now define it for `f` and `p`. Given the functions `f` and `p`,

$$\begin{array}{ll} \texttt{f}\ v_{11}\ \ldots\ v_{1m} = e_1^{\texttt{f}} & \texttt{p}\ w_{11}\ \ldots\ w_{1l} = e_1^{\texttt{p}} \\ \vdots & \vdots \\ \texttt{f}\ v_{n1}\ \ldots\ v_{nm} = e_n^{\texttt{f}} & \texttt{p}\ w_{n1}\ \ldots\ w_{nl} = e_n^{\texttt{p}} \end{array}$$

where $v_{ij}$ and $w_{ij}$ are arguments to `f` and `p`, respectively, and we define the anti-unification of `f` and `p`, denoted `h`, to be the list:

$$\texttt{h} = \left[ [\![e_1^{\texttt{f}}]\!] \triangleq [\![e_1^{\texttt{p}}]\!], \ldots, [\![e_n^{\texttt{f}}]\!] \triangleq [\![e_n^{\texttt{p}}]\!] \right]$$

where each element, $\forall i \in [1, n]$, $e_i^{\texttt{h}}$, in `h` corresponds to the anti-unification of the $i^{\text{th}}$ clauses in `f` and `p`; and elements are the triple $(t_i \times \sigma_{1i} \times \sigma_{2i})$. The substitutions for each element in `h` can be composed, and we refer to the composition of substitutions for `f` as $\sigma_{\texttt{f}}$, and the composition of substitutions for `p` as $\sigma_{\texttt{p}}$; i.e.

$$\sigma_{\texttt{f}} = (\!|\, \sigma_{11}, \ldots, \sigma_{1n} \,|\!)$$
$$\sigma_{\texttt{p}} = (\!|\, \sigma_{21}, \ldots, \sigma_{2n} \,|\!)$$

`h` is, in principle, equivalent to a function with $n$ equations and where: *i)* the anti-unifier term in each element represents the right-hand side of its respective equation; and *ii)* the set of all hedge variables are the parameters to the function. For example, the functions `f` and `p`,

```
f x = x + 42
g y = y + 108
```

when anti-unified, produce:

$$\texttt{h} = [(\texttt{App}\ (\texttt{App}\ (\texttt{Var}\ \texttt{+})\ (\texttt{Var}\ \texttt{y}))\ (\texttt{Var}\ \texttt{z})) \times \{\!\{\{\!\{\varepsilon, (\texttt{y} \mapsto (\texttt{Var}\ \texttt{x}))\}\!\}, (\texttt{z} \mapsto (\texttt{Id}\ 42))\}\!\} \times \{\!\{\{\!\{\varepsilon, \varepsilon\}\!\}, (\texttt{z} \mapsto (\texttt{Id}\ 108))\}\!\}]$$

Here, `h` can be represented as:

```
h y z = y + z

f x = h x 42

g y = h y 102
```

where, `h` is the anti-unifier, and the substitutions are represented using standard function application. To simplify our presentation, in examples we confuse `h` for its equivalent function definition.

### *3.3. Deriving Pattern Arguments*

Given the anti-unifier `h` of `f` and `p`, with substitutions $\sigma_{\texttt{f}}$ and $\sigma_{\texttt{p}}$ respectively, we next derive the arguments to `p` that are needed to rewrite `f` as an instance of `p`. Our anti-unification algorithm is designed such that $\sigma_1$ provides *candidate* arguments. These candidates are considered *valid* when they adhere to three properties:

1. that `p` and `h` are equivalent (Def. 10);
2. that there does not exist a (sub-)substitution in $\sigma_{\texttt{f}}$ or $\sigma_{\texttt{p}}$ where a $v_{a_i}$ occurs as either a hedge variable or as a term (Def. 12); and
3. that all substitutions in $\sigma_{\texttt{f}}$ substituting for the same hedge variable that is derived from `p` must substitute for the same term (Def. 14).

*Equivalence of Pattern and Anti-Unifier.* Recall that we aim to rewrite `f` in terms of `p`. As stated in Sec. 3.2, the syntactic equivalence properties of the produced anti-unifier allow `f` to be rewritten as a call to the function representation of `h`. Furthermore, by equational reasoning [27], if `p` and `h` are *equivalent*, then `f` can be rewritten to replace the call to `h` with a call to `p`; i.e. `f` is an instance of `p`. Since anti-unification will *always* produce an anti-unifier and substitutions between any two arbitrary terms, the production of an anti-unifier *cannot* be used as a test of equivalence. We instead define an equivalence relation between `p`, `h`, $\sigma_{\texttt{f}}$, and $\sigma_{\texttt{p}}$. In order to do this, we first define equivalence between two terms $t_2$ and $t$ and substitutions $\sigma_1$ and $\sigma_2$ with respect to $t_1$.

**Definition 9** (Equivalence of Terms with Substitutions)**.** *Given the terms, $t$, $t_1$, $t_2$, and the substitutions $\sigma_1$ and $\sigma_2$, such that*

$$\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} \cong (t) \begin{bmatrix} \sigma_1 \\ \sigma_2 \end{bmatrix}$$

*we say that $t_2$ and $t$ are equivalent with respect to $\sigma_1$, $\sigma_2$, and $t_1$, (denoted $t_2 \equiv_{t_1} t \times \sigma_1 \times \sigma_2$) when:*

$$\text{EQ}\ \frac{t_2 \cong t}{t_2 \equiv_{t_1} t \times \varepsilon \times \varepsilon} \qquad \text{HEDGE}\ \frac{v \neq \mathtt{p} \quad v \in \mathcal{V}_\mathtt{p}}{(\mathtt{Var}\ v) \equiv_{t_1} (\mathtt{Var}\ v) \times (v \mapsto t_1) \times \varepsilon}$$

$$\text{FRESH}\ \frac{t_1 \in \mathcal{R}_\mathtt{f} \quad t_2 \in \mathcal{R}_\mathtt{p} \quad \alpha \notin \mathcal{V}_\mathtt{p}}{t_2 \equiv_{t_1} (\mathtt{Var}\ \alpha) \times (\alpha \mapsto t_1) \times (\alpha \mapsto t_2)}$$

$$\text{CONST}\ \frac{\forall i \in [1,n],\ t_{2i} \equiv_{t_1} t_i \times \sigma_{1i} \times \sigma_{2i}}{(C\ t_{21}\ \ldots\ t_{2n}) \equiv_{t_1} (C\ t_1\ \ldots\ t_n) \times (\!|\sigma_{11}, \ldots, \sigma_{1n}|\!) \times (\!|\sigma_{21}, \ldots, \sigma_{2n}|\!)}$$

$$\text{LIST}\ \frac{\forall i \in [1,n],\ t_{2i} \equiv_{t_1} t_i \times \sigma_{1i} \times \sigma_{2i}}{[t_{21}, \ldots, t_{2n}] \equiv_{t_1} [t_1, \ldots, t_n] \times (\!|\sigma_{11}, \ldots, \sigma_{1n}|\!) \times (\!|\sigma_{21}, \ldots, \sigma_{2n}|\!)}$$

Here, EQ states that when $t_2$ and $t$ are the same, and when $\sigma_1$ and $\sigma_2$ are both $\varepsilon$, then $t_2$ and $t$ are equivalent with respect to $\sigma_1$ and $\sigma_2$. When $t_2$ and $t$ are both variable terms, $t_2$ and $t$ are equivalent when the variables have the same name and when the the variable is a hedge variable derived from `p` (HEDGE). Similarly, when $t_2$ and $t$ are both fresh hedge variables, $\alpha$, then both terms substituted for $\alpha$ must be unobstructive recursive prefixes in their respective functions (FRESH). Finally, CONST and LIST state that terms consisting only of subterms are equivalent when all respective subterms are equivalent. The definition of equivalence for `p` and `h` then follows naturally.

**Definition 10** (Equivalence of Pattern and Anti-Unifier)**.** *Given two functions* `f` *and* `p`*, and the result of their anti-unification,* `h`*, containing the substitutions $\sigma_\mathtt{f}$ and $\sigma_\mathtt{p}$, we say that* `p` *and* `h` *are equivalent when for all clauses in* `p`*, $e_i^\mathtt{p}$, and elements in* `h`*, $e_i^\mathtt{h} = (t_i \times \sigma_{1i} \times \sigma_{2i})$, $[\![e_i^\mathtt{p}]\!] \equiv_{t_{1i}} e_i^\mathtt{h}$.*

*Absence of Shared Arguments in Substitutions.* As stated in Sec. 3.1, the variables, $v_{a_i}$, that are declared in $a_i$ are shared between `f` and `p`. As such, if `f` is an instance of `p`, $v_{a_i}$ must be used in `f` in an equivalent way to their use in `p`; i.e. their relative locations in the structure of `f` and `p` must be the same. For example, in `sumeuler` and `foldr`,

```
1 sumeuler [] = 0
2 sumeuler (x:xs) = ((+) . euler) x (sumeuler xs)
3
4 foldr g z [] = z
5 foldr g z (w:ws) = g w (foldr g z ws)
```

the heads of the matched lists, `x` in `sumeuler` and `w` in `foldr`, are both passed as the first argument to some function, `g`; and the tails of the matched lists, `xs` and `ws` respectively, are passed as the last argument to the recursive calls in `f` and `p`. While the equivalence of pattern and anti-unifier (Def. 10) ensures the structural equivalence of `p` and `h`, it does not inspect hedge variables derived from `p`. Our anti-unification rules in Fig. 5 means that no $v_{a_i}$ will occur as either a hedge variable or as a subterm in either $\sigma_{\texttt{f}}$ or $\sigma_{\texttt{p}}$ when all equivalent $v_{a_i}$ have the same relative positions in `f` and `p`. Conversely, the occurrence of some $v_{a_i}$ in $\sigma_{\texttt{f}}$ or $\sigma_{\texttt{p}}$ indicates that not all equivalent $v_{a_i}$ have the same relative positions in `f` and `p`.

**Definition 11** (Variable Occurrence in Terms)**.** *For all variables $v$, and for all terms, $t$, we say that $v$ occurs in $t$ (denoted $v \ll t$) when $t = (\texttt{Var}\ v)$ or when there exists a subterm, $t_i$, in $t$ such that $t_i = (\texttt{Var}\ v)$.*

**Definition 12** (Shared Argument Absence)**.** *Given the functions `f` and `p`, and their anti-unification `h`, containing the substitutions $\sigma_{\texttt{f}}$ and $\sigma_{\texttt{p}}$, we say that no $v_{a_i}$ occurs in $\sigma_{\texttt{f}}$ or $\sigma_{\texttt{p}}$ (denoted $v_{a_i} \notin (\!|\sigma_{\texttt{f}}, \sigma_{\texttt{p}}|\!)$) when, for all $v_{a_i}$:*

$$\frac{v \ncong v_{a_i} \quad (\texttt{Var}\ v_{a_i}) \not\ll t}{v_{a_i} \notin (v \mapsto t)} \qquad \frac{\forall i \in [1,n],\ v_{a_i} \notin \sigma_i}{v_{a_i} \notin (\!|\sigma_1, \ldots, \sigma_n|\!)}$$

*Substitution Uniqueness.* Substitution composition (Def. 6) ensures that the same hedge variable can be used to substitute for different terms without the problem of substitution interference. For example, consider some `f` that is anti-unified against `scanl`:

```
f a [] = [a]
f a (x:xs) = (+) a x : f ((-) a x) xs

scanl g z [] = [z]
scanl g z (x:xs) = g z x : scanl g (g z x) xs
```

Here, the result of the anti-unification of the two *cons*-clauses will include substitutions where `g` is substituted for both `(+)` and `(-)` in $\sigma_{\texttt{f}}$. In `scanl`, however, `g` is the same in both instances. We therefore also require that all substitutions that substitute for the same hedge variable in $\sigma_{\texttt{f}}$ are substituted for the same term.

**Definition 13** (Substitution Flattening)**.** *For all substitutions, $\sigma$, we say the flattening of $\sigma$ (denoted $\lfloor\sigma\rfloor$) is the list of substitutions, where:*

$$\frac{}{\lfloor(v \mapsto t)\rfloor = [(v \mapsto t)]} \qquad \frac{}{\lfloor(\!|\sigma_1, \ldots, \sigma_n|\!)\rfloor = \lfloor\sigma_1\rfloor + \cdots + \lfloor\sigma_n\rfloor}$$

*Here, $+$ appends two lists, and $[a]$ denotes the singleton list.*

A list of substitutions cannot be applied to a term, allowing us to safely expose any potential substitution interference across `f`.

**Definition 14** (Substitution Uniqueness)**.** *Given the functions* `f` *and* `p`*, and their anti-unification* `h`*. Given that* $\sigma_{\mathtt{f}}$ *is the composition of substitutions in* `h` *for the clauses of* `f`*. We say that substitutions are unique in* $\sigma_{\mathtt{f}}$ *when for all hedge variables,* $v \in \mathcal{V}_{\mathtt{p}}$*, and for all pairs of substitutions in* $\lfloor \sigma_{\mathtt{f}} \rfloor$*,* $(v \mapsto t_1)$ *and* $(v \mapsto t_2)$*,* $t_1 = t_2$ *holds.*

*Sufficiency of Validity Properties.* Given `f`, `p`, `h`, $\sigma_{\mathtt{f}}$ and $\sigma_{\mathtt{p}}$, the above three properties are sufficient to determine whether the candidate arguments in $\sigma_{\mathtt{f}}$ are valid. Equivalence of `p` and `h` (Def. 10) ensures that for all clauses in `p` and `h`, the terms for that clause, $t_2$ and $t$, are syntactically equivalent (Def. 3). Hedge variable terms have additional information in their respective substitutions, $\sigma_1$ and $\sigma_2$. There are two cases: fresh hedge variables, $\alpha$, and hedge variables derived from `p`, $v$. For $\alpha$, Def. 10 requires that both terms, $t_1$ and $t_2$, substituted for $\alpha$ are unobstructive recursive prefixes (Def. 7). No other terms are allowed. For $v$, two cases are possible: $v = v_{a_i}$ and otherwise. All equivalent $v_{a_i}$ in `f` and `p` must have the same relative locations to ensure that $a_i$ is traversed in the same way, and that no free variables occur in any argument candidate expression. The rules in Fig. 5 mean that the identity substitution is derived for all equivalent $v_{a_i}$ in the same relative position in `f` and `p`. For the case when a $v_{a_i}$ is not anti-unified with its equivalent, Shared Argument Absence (Def. 12) requires that no $v_{a_i}$ occurs in any substitution and so will fail in this case. For all other $v$, Def. 10 requires that $v$ will be in scope, due to the assumption that all variables that occur in `p` are in scope and are declared as parameters of `p`. Any updates to the value of $v$ must be reflected in `p`, e.g. such as in `foldl` and `scanl`. Finally, and since each argument may be instantiated only once for each call to `p`, Substitution Uniqueness (Def. 14) ensures that for all substitutions that substitute for the same hedge variable, the substituted terms are the same.

### *3.4. Deriving Arguments from Substitutions*

Given that `f`, `p`, and their anti-unification, `h`, adhere to Def. 10 (Equivalence of Pattern and Anti-Unifier), Def. 12 (Shared Argument Absence), and Def. 14 (Substitution Uniqueness), the arguments for `p` can be directly obtained from $\sigma_{\mathtt{f}}$. For `sumeuler`, given $\sigma_1$ for the `[]` clause:

$$\sigma_1 = (\mathtt{z} \mapsto (\mathtt{Lit}\ 0))$$

and $\sigma_1$ for the `(:)` clause:

$$\sigma_1 = \langle\!\langle\, (\mathtt{g} \mapsto (\mathtt{App}\ (\mathtt{App}\ (\mathtt{Var}\ .)\ (\mathtt{Var}\ +))\ (\mathtt{Var}\ \mathtt{euler}))), \varepsilon \,\rangle\!\rangle,\ \langle\!\langle\, (\alpha \mapsto (\mathtt{Var}\ \mathtt{sumeuler})), \varepsilon \,\rangle\!\rangle\, \rangle\!\rangle$$

`((+) . euler)` is passed as `g`; `0` is passed as `z`; and $a_i$ (i.e. `(x:xs)`) is passed as itself due to Shared Argument Absence and Argument Equivalence properties. Substitutions with fresh hedge variables are discarded. We can now refactor `f` in terms of `p` using the derived arguments above, to give:

```
1 sumeuler xs = foldr ((+) . euler) 0 xs
```

Which, as before, can be rewritten to give our original definition:

```
sumeuler xs = foldr (+) 0 (map euler xs)
```

*¡/¿Maximum Segment Sum.* Where `sumeuler` is a simple `foldr`, consider the less obvious example of maximum segment sum, as defined by Kannan [7]:

```
mss xs0 = mss2 0 0 xs0

mss2 y z [] = z
mss2 y z (x:xs) = mss2 (max y (x+z)) (max x (x+z)) xs
```

Here, `mss2` traverses over the list `xs0`. Since `mss2` does not return a list, we instead consider a fold pattern. Two possibilities are available: a `foldr` or a `foldl`. The standard Prelude definitions of these functions take only one initial/accumulator argument, and neither will produce a rewritten `mss2`. To solve this, `y` and `z` in `mss2` may be tupled as part of a normalisation process:

```
mss2 y z xs0 = fst (mss2' (y,z) xs0)

mss2' w [] = w
mss2' w (x:xs) =
  mss2' ((\(y1,z1) x -> (max y1 (x1+z1), max x1 (x1+z1))) w x) xs
```

When anti-unified against `foldr` and `foldl`, `mss2'` will produce the following anti-unifiers:

```
hr g z [] = z
hr g z (x:xs) = g x α

hl g z [] = z
hl g z (x:xs) = α (g z x) xs
```

where `hr` is the result of anti-unifying `mss2'` and `foldr`, and `hl` is the result of anti-unifying `mss2'` and `foldl`. For `hr`, `z` is substituted for `w`, `g` is substituted for `mss2'`, `x` is substituted for

```
(\(y1,z1) x1 -> (max y1 (x1+z1), max x1 (x1+z1))) w x)
```

and $\alpha$ is substituted for `xs` in `mss2'` and `(foldr g z xs)` in `foldr`. `hr` will not pass the validation properties, since: *i)* `x` and `xs` (i.e. $v_{a_i}$) both occur in substitutions; and *ii)* `hr` is not equivalent to `foldr`, in part because $\alpha$ is substituted for `(foldr g z xs)` which is not an unobstructive recursive prefix. Conversely, an instance of `foldl` can be found, and `f` rewritten:

```
mss2' w xs0 = foldl (\(y,z) x -> (max y (x+z), max x (x+z))) w xs0
```

An alternative and equally valid approach can be to anti-unify against a `foldl` that takes two accumulative variables; e.g.:

```
foldl g1 g2 y z [] = y
foldl g1 g2 y z (x:xs) = foldl g1 g2 (g1 y z x) (g2 y z x) xs
```

When anti-unified against `mss2'` in the form:

```
mss2' y z [] = y
mss2' y z (x:xs) =
  mss2' ((\y1 z1 x1 -> max y1 (x1+z1)) y z x)
        ((\y1 z1 x1 -> max x1 (x1+z1)) y z x)
        xs
```

the following anti-unifier is produced, and `mss2'` is rewritten:

```
h g1 g2 y z [] = y
h g1 g2 y z (x:xs) =
  α (g1 y z x) (g2 y z x) xs

mss2' y z xs0 =
  foldl (\y1 z1 x1 -> max y1 (x1+z1))
        (\y1 z1 x1 -> max x1 (x1+z1))
        xs0
```

where $\alpha$ is substituted for `mss2'` and `(foldl g1 g2)` in `mss2'` and `foldl`, respectively. This example demonstrates the increased importance of normalisation as functions become more complicated; a limitation we will explore in future work.

*Termination of Approach.* As an additional result, we observe that our approach will always terminate, given our assumptions, that the definitions of `f` and `p` are themselves finite, and $\mathcal{P}$ contains a finite number of scheme implementations. Our first-order anti-unification algorithm can be represented as a fold over two finite terms, and so will both always terminate and produce a finite result. Similarly, our validation properties will always terminate, since they too can be represented as a fold over finite substitutions and function definitions. Should the validation properties *not* hold for a given `f` and `p`, then the process terminates with no rewriting. Conversely, when the validation properties *hold* for a given `f` and `p`, the selection of arguments to `p` and rewriting of `f` will both always terminate. The selection of arguments can be represented as a search for a particular node of a finite tree. Similarly, the rewrite always constructs a single clause function whose body is an application expression that applies `p` to a finite number of arguments. Should our approach be extended by a normalisation procedure, as suggested in Sec. 3.1, we require and assume that the normalisation procedure is both finite and valid. Therefore, whether our approach is extended by a normalisation procedure or not, our approach will always terminate.

| $n$ | MUT | SD | Total | SD | Sparks | Heap | Residency |
|---|---|---|---|---|---|---|---|
| 1000 | 0.06 | 0.02 | 0.06 | 0.02 | 3 | 67795856 | 68256 |
| 5000 | 1.29 | 0.01 | 1.31 | 0.01 | 11 | 1958285432 | 214792 |
| 10000 | 5.34 | 0.03 | 5.39 | 0.03 | 21 | 8296937744 | 224584 |
| 15000 | 12.43 | 0.03 | 12.54 | 0.03 | 31 | 19280868336 | 498624 |
| 20000 | 22.82 | 0.08 | 23.03 | 0.07 | 41 | 35050884184 | 761720 |
| 25000 | 36.29 | 0.03 | 36.6 | 0.04 | 51 | 55705003000 | 1024160 |
| 30000 | 53.24 | 0.18 | 53.7 | 0.18 | 61 | 81319554136 | 1228304 |
| 35000 | 73.36 | 0.3 | 73.99 | 0.3 | 71 | 111956760792 | 1428304 |
| 40000 | 96.77 | 0.04 | 97.6 | 0.04 | 81 | 147666965368 | 1612232 |
| 45000 | 123.98 | 0.13 | 125.04 | 0.13 | 91 | 188497463480 | 1612232 |
| 50000 | 154.33 | 0.25 | 155.65 | 0.26 | 101 | 234487830776 | 1778184 |

Figure 6: Sequential mutator (`MUT`) and total (`Total`) times in seconds (with standard deviations), total number of sparks for parallel version, and heap allocation and residency in bytes for `sumeuler` on *corryvreckan*.

## 4. Examples

We have applied our approach to four examples from the *NoFib* suite [9] of Haskell benchmarks that are known to have *parallel* implementations. For each example we provide: performance and speedup results on our 28-core testbed machine; a brief description of the size and structure of the program; the number of explicit patterns that it contains; and a description of any other interesting functions/patterns. All four examples have been anti-unified against relevant `map`/`fold` patterns using our prototype anti-unification implementation.

*Experimental Setup.* All our examples have been executed on our experimental testbed machine, *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*. Using built-in *hyper-threading* support, it is possible to scale to 56 virtual cores, but this does not always give performance improvements. All four examples have been compiled using GHC 7.6.3 on Scientific Linux version 3.10.0. The original source programs have been taken from the repository at `git.haskell.org/nofib.git` (commit `ce4b36b`). We have not changed this code, apart from adding wrappers to aid profiling where this is necessary. Our code is available at `https://adb23.host.cs.st-andrews.ac.uk/nofib-parallel.zip`. We compiled the examples using `-O`, `-feager-blackholing`, `-threaded`, and `-rtsopts` flags, which we found gave the best general performance. Our measurements represent the average of three runs and are taken from the `-s` Haskell RTS option, which records execution statistics. Timing information is taken from two sources: *mutator time* (`MUT`) and *total time* (`Total`), both in seconds. The mutator time represents the CPU time that is spent purely on executing the program, and is thus a measure of pure parallel performance. The total time is the elapsed or

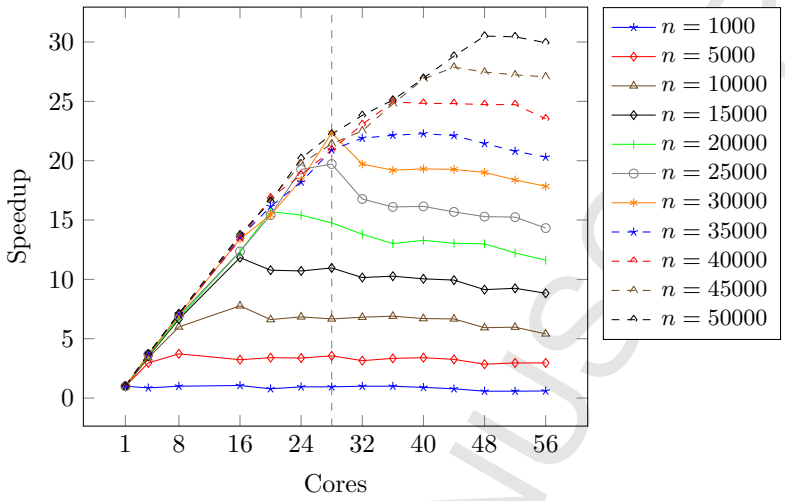


Figure 7: Speedups for `sumeuler` on *corryvreckan* using reported mutator (`MUT`, reduplicated from Sec. 1.2) times. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores.

real-time for the program as a whole, and also includes the additional system runtime overheads of initialisation, finalisation and garbage collection time.

### *4.1. Sumeuler*

As described in Sec. 1.2, `sumeuler` calculates Euler's totient function for a list of integers and sums the results. In the NoFib suite, `sumeuler` consists of three Haskell files: `ListAux` (41 lines), `SumEulerPrimes` (36 lines), and `SumEuler` (290 lines). Collectively, these introduce 31 functions, 12 of which have explicit *maps* or *folds* as part of their definitions. Our prototype implementation rediscovers all the implicit *maps* and *folds*. One interesting case is `primesIn`:

```
1 primesIn :: [Int] -> Int -> [Int]
2 primesIn ps@(p:rest) n
3          | p > n           = []
4          | n `mod` p == 0 = p:primesIn ps (n `div` p)
5          | otherwise       = primesIn rest n
```

`primesIn` is a `foldr` over an *infinite* list when the standard Prelude definition of `foldr` does not define behaviour for the empty list.

We executed `sumeuler` for $n = 1000, 5000$ and between 10000 and 50000 at intervals of 5000, with a chunk size of 500. Fig. 6 gives average sequential times (with standard deviations), the number of sparks, heap allocation, and residency for all $n$. All sparks are converted for all $n$. Fig. 7 and Fig. 8 give speedups for `sumeuler` using mutator and total time, achieving maximum speedups of 30.50 for $n = 50000$ on 48 virtual cores (mutator) and 20.92 for $n = 50000$ on virtual 48 cores (total). This clearly shows that `sumeuler` scales both with $n$ and with

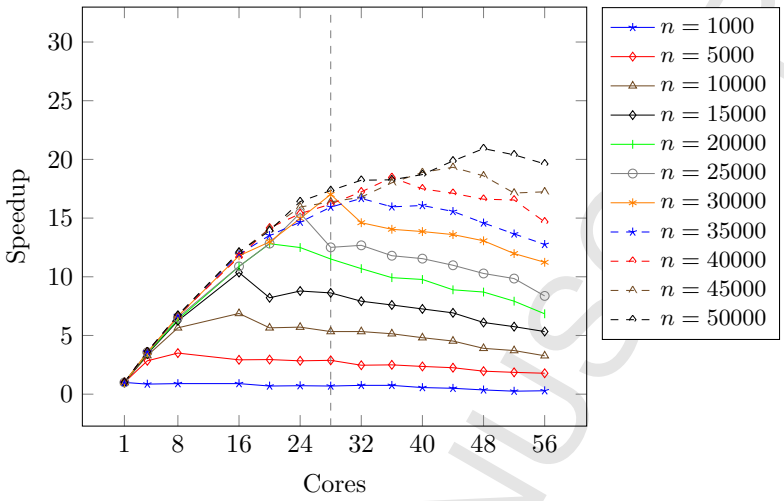


Figure 8: Speedups for `sumeuler` on *corryvreckan* using reported total (`Total`) times. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores.

| $n$ | MUT | SD | Total | SD | Sparks | Heap | Residency |
|---|---|---|---|---|---|---|---|
| 11 | 0.1 | 0.03 | 0.1 | 0.03 | 101 | 95722624 | 64088 |
| 12 | 0.42 | 0.08 | 0.43 | 0.08 | 122 | 569715376 | 1693112 |
| 13 | 2.13 | 0.02 | 2.19 | 0.03 | 145 | 3590068664 | 9854376 |
| 14 | 13.65 | 0 | 14.1 | 0.01 | 170 | 24108663032 | 59041312 |
| 15 | 96.36 | 0.17 | 103.11 | 0.18 | 197 | 171560553832 | 479706512 |
| 16 | 717.66 | 0.81 | 821.03 | 0.35 | 226 | 1293635766936 | 3097146864 |

Figure 9: Sequential mutator (`MUT`) and total (`Total`) times in seconds (with standard deviations), total number of sparks for parallel version, and heap allocation and residency in bytes, for `queens` on *corryvreckan*.

the number of cores. Total time speedups are more likely to *reduce* as further cores are added for a given $n$, since garbage collection time *increases* with more cores. The irregularity of the tasks in `sumeuler` is also likely to have contributed to this result. For this example, all of the potential parallelism (Haskell *sparks*) was converted into actual parallelism (Haskell ultra-lightweight *threads*). In general, however, the sophisticated *evaluate-and-die* mechanism will throttle excess parallelism, giving a better load-balance, and avoiding system overload when massive amounts of parallelism are present.

### *4.2. N-Queens*

The n-queens problem (`queens`) calculates the positions of $n$ queens on a $n \times n$ chess board such that no two queens threaten each other. In the NoFib suite, `queens` consists of one Haskell file, `Main`, that is 42 lines long and defines 6

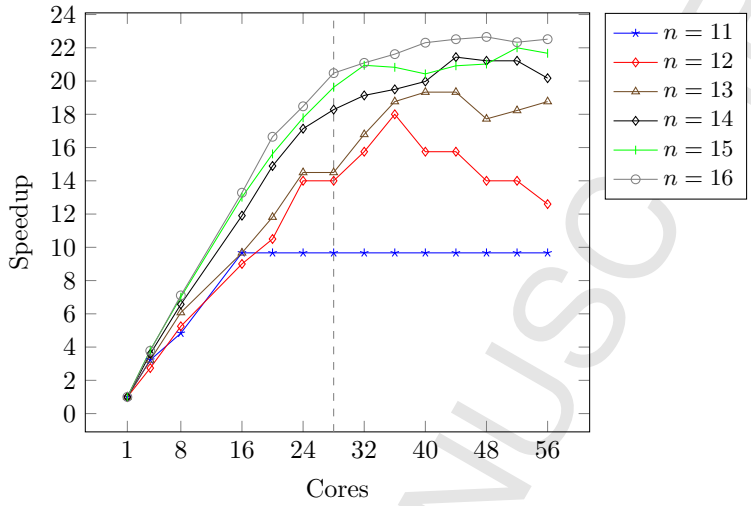


Figure 10: Speedups for `queens` on *corryvreckan* using reported mutator (`MUT`) time. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores.

functions. Two of these 6 functions have explicit maps and folds (including list comprehensions), and an explicit call to `iterate`. One other function, `safe`,

```
safe :: Int -> Int -> [Int] -> Bool
safe x d []     = True
safe x d (q:l) = x /= q && x /= q+d && x /= q-d && safe x (d+1) l
```

is an *implicit* `foldr` over lists:

```
safe :: Int -> Int -> [Int] -> Bool
safe x d qs = fst $ foldr f (True,d) qs
  where
    f q (l,d) = (x /= q && x /= q+d && x /= q-d && l, d+1)
```

Since `d` is updated between recursive calls, our prototype implementation will not currently detect this `foldr`[1]. By rewriting `safe` as a composition of an `unfold` and an (implicit) `foldr`, our prototype can detect the `foldr` over the tree generated by the `unfold`. `safe` may alternatively be considered an instance of a *zygomorphism* [28], it may therefore be worth investigating this pattern as part of our future work. Another function, `pargen`, enumerates the different possible locations of the queens on the board.

[1]This is because in the original definition of `safe`, `d` is updated between recursive calls meaning a standard `foldr` is not equivalent (Def. 10).

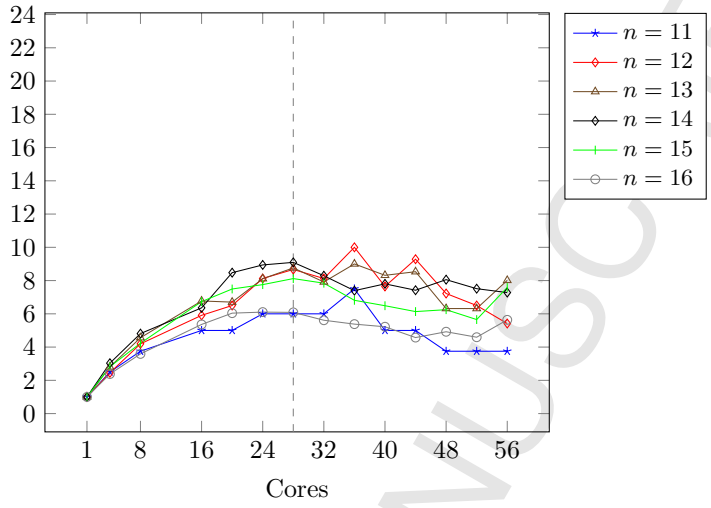


Figure 11: Speedups for `queens` on *corryvreckan* using reported total (`Total`) time. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores.

```
pargen strat n b
   n >= t = iterate gen [b] !! (nq - n)
   otherwise =
      concat (map (pargen strat (n+1)) (gen [b]) `using` strat)
```

This is defined as a sub-function of the top-level `queens` function. Here, `nq` denotes the number of queens that are to be searched for, and `t` is a threshold for controlling parallelism. The `strat` parameter is an evaluation strategy: either `evalList` or `parList` depending on whether sequential or parallel operation is chosen. `n` is used as a measure for determining whether the threshold, `t`, is reached; `b` accumulates results; and `gen` generates an individual result. We can unfold both `iterate` and `map` expressions as implicit `iterN` and `map` functions that are called by `pargen`:

```
pargen n b
   n >= t = pgI ([b],(nq-n))
   otherwise = concat $ pgM (gen [b])
    where
      pgI (rs,0) = rs
      pgI (rs,i) = pgI ((gen rs),i-1)

      pgM [] = []
      pgM (r:rs) = pargen' (n+1) r : pgM rs
```

Our prototype correctly (re)discovered instances of both `iterN` and `map` within `pgI` and `pgM`. We executed `queens` for $n$ ranging from 11 to 16, with `t` = 2. Fig. 9

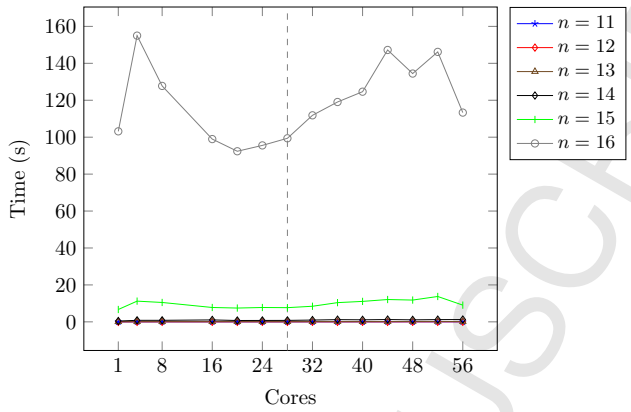


Figure 12: Garbage collection times for `queens` on *corryvreckan*. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores.

gives mutator and total times (with standard deviations) for sequential runs, and number of sparks, heap allocation size, and residency for each $n$. An average of 30% of sparks were converted on 4 cores, 65% on 8 cores, and 85% for all other cores. Fig. 10 and 11 shows speedups for `queens` in terms of mutator and total time. We achieve maximum speedups of 22.65 for $n = 16$ on 48 hyper-threaded cores and 10.00 for $n = 12$ on 36 hyper-threaded cores. When $n = 11$, the mutator time plateaus quickly and sharply, possibly due to lack of work. When $n = 12$, speedups reduce as the number of cores is increased, probably because there is insufficient work to offset increased overheads. Otherwise, mutator times plateau above 28 cores, probably because hyper-threading is not effective in this example. Unlike our other examples, total time speedups reveal an overall halving of performance gains as a consequence of increased garbage collection time. Fig. 12 shows that when $n = 15$ and $n = 16$, garbage collection times are heavily increased. This is probably due to repeated generation of lists.

### *4.3. N-Body*

`nbody` calculates the movement of bodies in $m$-dimensional space according to the physical forces between them. The classical representation of this problem is between celestial bodies, modelling the effects of gravity on their trajectories. In the NoFib suite, `nbody` consists of two Haskell files: `Future` (16 lines) and `nbody` (135 lines) which collectively define 8 functions. Two of these 8 functions have an explicit map or fold, and one, `loop`, is an *implicit* `iterN`. Our prototype discovers the maps and folds in both functions where they are explicitly used, from their inlined definitions. The `loop` sub-function of `compute` is defined as:

```
1 loop i ax ay az
2   i == end =  (# ax,ay,az #)
3   otherwise = loop (i+1) (ax+px) (ay+py) (az+pz)
4    where
```

| $n$ | MUT | SD | Total | SD | Sparks | Heap | Residency |
|---|---|---|---|---|---|---|---|
| 10000 | 2.06 | 0.06 | 2.07 | 0.06 | 40 | 6310152 | 1197488 |
| 25000 | 11.61 | 0.1 | 11.63 | 0.11 | 40 | 15670360 | 5216880 |
| 50000 | 46.93 | 0.13 | 46.98 | 0.13 | 40 | 31270520 | 12311984 |
| 75000 | 104.9 | 0.35 | 104.96 | 0.34 | 40 | 46870848 | 12547600 |
| 100000 | 187.47 | 0.1 | 187.57 | 0.11 | 40 | 62471232 | 25082544 |
| 250000 | 1164.34 | 0.68 | 1164.52 | 0.65 | 40 | 156226496 | 40215360 |

Figure 13: Sequential mutator (`MUT`) and total (`Total`) times in seconds (with standard deviations), total number of sparks for parallel version, and heap allocation and residency in bytes, for `nbody` on *corryvreckan*.

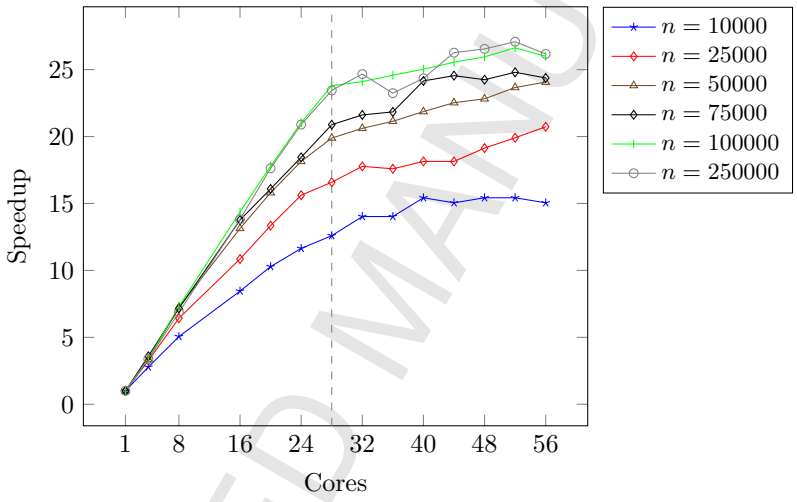


Figure 14: Speedups for `nbody` on *corryvreckan* using reported mutator (`MUT`) time. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores.

```
5           ( x,y,z ) = vector
6           ( x',y',z' ) = vecList Array.! i
7
8           (# dx,dy,dz #) = (# x'-x, y'-y, z'-z #)
9           eps = 0.005
10          distanceSq = dx^2 + dy^2 + dz^2 + eps
11          factor = 1/sqrt(distanceSq ^ 3)
12          (# px,py,pz #) =
13            (# factor * dx, factor * dy, factor *dz #)
```

where `i` and `n` control the number of iterations performed; `ax`, `ay`, `az` are the three-dimensional cartesian coordinates of the body being updated; and, ultimately, `px`, `py`, and `pz` are updates to those initial coordinates. Applying our prototype to `loop`, with guards converted to pattern matched clauses and `i` and

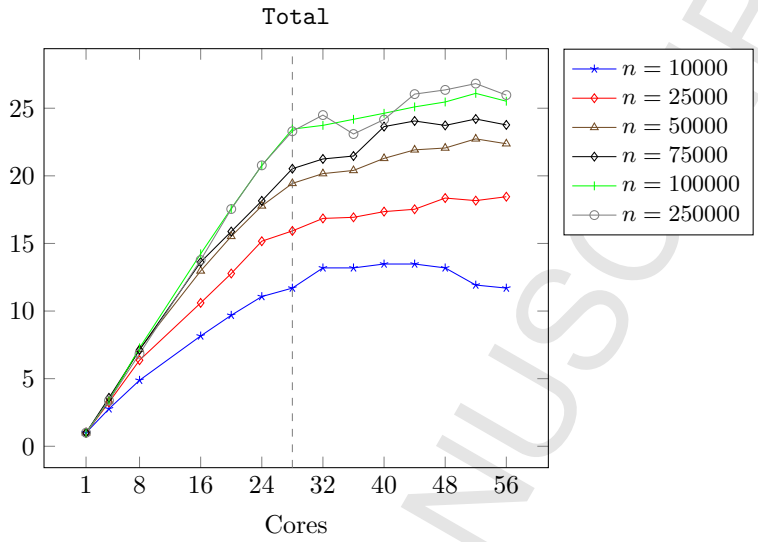


Figure 15: Speedups for `nbody` on *coryvreckan* using reported total (`Total`) time. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores.

`end` inverted such that `i` decreases to 0, our prototype discovers the `iterN`.

```
1 loop ((ax,ay,az),i) = iterN g ((ax,ay,az),i)
2   where
3     g (ax,ay,az) = let ... in ((ax+px),(ay+py),(az+pz))
```

We have elided the local definitions for clarity. We executed `nbody` for $n = \{10000, 250000\}$ and between 0 and 100000 at intervals of 25000. Fig. 13 gives the corresponding mutator and total times (with standard deviations), and the number of sparks (potential parallelism), heap allocation size, and residency for each $n$. An average of 97.5% of sparks were converted on 4 cores, and 100% of sparks were converted on all other numbers of cores. Fig. 14 and 15 give speedups for `nbody` using mutator and total time, achieving maximum speedups of 27.08 for $n = 250000$ on 52 hyper-threaded cores and 26.82 for $n = 250000$ on 52 hyper-threaded cores, respectively. There is little difference between mutator and total reported speedups for `nbody`, showing that garbage collection is not significant for this example. The example also demonstrates good scalability for both varying $n$ and number of cores. The growth in speedup does diminish after 28 cores, and the use of hyper-threading is again the most likely cause.

### *4.4. Matrix Multiplication*

Matrix multiplication (`matmult`) multiplies two matrices. The *NoFib* suite definition consists of two Haskell files: `ListAux` (41 lines) and `MatMult` (246 lines). These collectively define 42 functions, 11 of which feature explicit *maps* and *folds* (not including the specialised versions of `foldr`, e.g. `filter` or `sum`, that

| $n$ | MUT | SD | Total | SD | Sparks | Heap | Residency |
|---|---|---|---|---|---|---|---|
| 1000 | 13.15 | 1.59 | 13.37 | 1.58 | 18 | 317362136 | 44416 |
| 1500 | 56.09 | 5.12 | 56.33 | 5.14 | 27.5 | 713751520 | 44416 |
| 2000 | 131.72 | 14.07 | 131.96 | 14.07 | 38 | 1268624520 | 44416 |
| 2500 | 226.58 | 7.49 | 226.83 | 7.51 | 49.5 | 1835831694 | 44416 |
| 3000 | 402.13 | 2.71 | 402.36 | 2.69 | 62 | 1801405160 | 44416 |
| 3500 | 633.18 | 9.43 | 633.44 | 9.43 | 67 | 2451609160 | 44416 |
| 4000 | 946.42 | 12.1 | 946.67 | 12.08 | 80.5 | 3201845928 | 44416 |
| 4500 | 1344.39 | 16.09 | 1344.59 | 16.08 | 85.5 | 4052049928 | 44416 |
| 5000 | 1823.05 | 0.96 | 1823.31 | 0.96 | 100 | 5002253928 | 44416 |

Figure 16: Sequential mutator (`MUT`) and total (`Total`) times in seconds (with standard deviations), total number of sparks for parallel version, and heap allocation and residency in bytes, for `matmult` on *corryvreckan*.

are used extensively in this example). All the instances of explicit maps and folds are discovered by our prototype from their inlined definitions. Three functions, `shiftRight` and both `addProd` definitions, are implicit folds. `shiftRight` is defined as:

```
shiftRight c [] = []
shiftRight c (xs:xss) = (xs2++xs1):shiftRight (c-1) xss
  where (xs1,xs2) = splitAt c xs
```

and is an implicit `foldr`:

```
shiftRight c xs = fst $ foldr f ([],c) xs
  where
    g xs (xss,c) = let (xs1,xs2) = splitAt c xs
                   in ((xs2++xs1):xss, c-1)
```

As with `safe` from `queens` (Sec. 4.2), our prototype cannot discover the `foldr` in `shiftRight` because `c` is updated in the recursive call. The second implicit fold, `addProd`, is defined as:

```
addProd :: Vector -> Vector -> Int -> Int
addProd (v:vs) (w:ws) acc = addProd vs ws (acc + v*w)
addProd [] [] n = n
```

`addProd` can be viewed as a composition of a `zipWith` and a `foldl`, or simply as a `foldl`. Here, both `zipWith` and `foldl` implementations need not be total definitions, as mentioned in Sec. 3.1. We executed `matmult` for $n$ between 1000 and 5000 at intervals of 500, with chunk size set to 20. Two parallel versions are possible: *i.* performs the multiplication of each row in parallel, and *ii.* divides each matrix into blocks to be computed in parallel and joins their result. We have used both modes here. Whilst the original *NoFib* suite definition provides its own matrix generation function, we have adjusted this so that matrices are generated using:

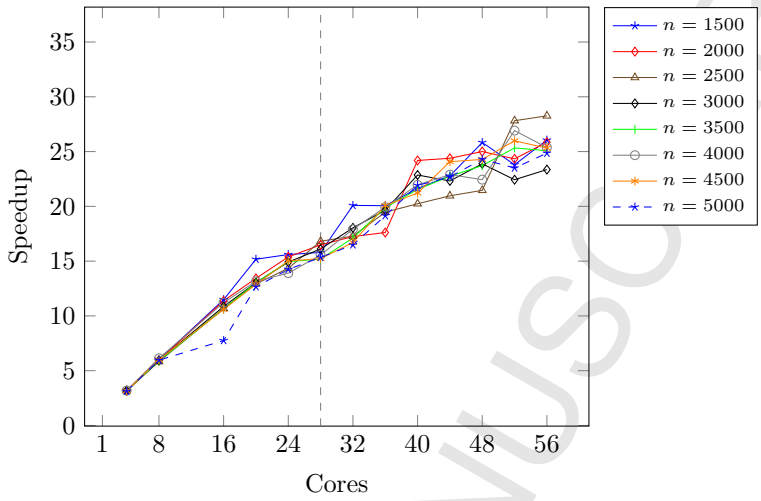


Figure 17: Speedups for `matmult` on *corryvreckan* using the row-wise parallelisation option and reported mutator (`MUT`) time. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores.

```
1 replicate n [1..n]
```

Fig. 16 reports sequential times (with standard deviations), and the number of sparks, heap allocation size, and residency for each $n$. The average number of sparks converted varies with $n$, instead of the number of cores seen in previous examples. Converted sparks fluctuate between 100% and not less than 93% (seen on 4 cores), and fluctuations grow smaller with larger $n$. Figs. 17, 19, 18, and 20 give speedups for `matmult` using mutator and total time for both parallel modes. Row-wise parallelisation achieves maximum speedups of 28.26 for $n = 2500$ on 56 hyper-threaded cores for both mutator and total values. Block-wise parallelisation achieves maximum speedups of 32.93 for $n = 1500$ on 52 hyper-threaded cores for both mutator and total values. As with `nbody`, there is little difference between mutator and total reported speedups for `matmult`, and mode 2 proves generally only slightly better than mode 1. The example demonstrates good scalability for both varying $n$ and number of cores. Once again, speedup growth slows after 28 cores due to hyper-threading.

## 5. Related Work

Anti-unification was first described by Plotkin [10] and Reynolds [15] who described the approach using lattices of totally ordered terms. The technique has been explored and expanded upon since, enabling its operation on unranked terms [29], and used in a range of application areas; including generating code from analogy [30], generating invariants for program verification [17], termination checking [31], symbolic mathematical computing [32], and clone detec-

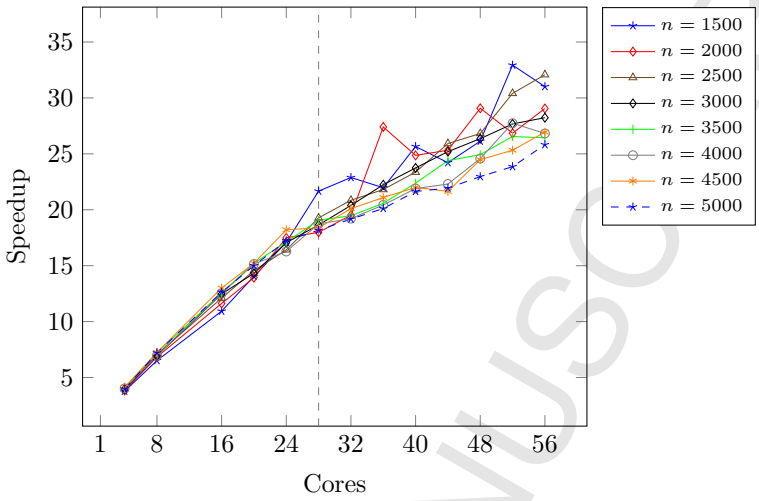


Figure 18: Speedups for `matmult` on *corryvreckan* using the block-wise parallelisation option and reported mutator (`MUT`) time. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores.

tion [33, 18]. Whilst our technique can be considered a form of clone detection, traditional anti-unification approaches to clone detection have focussed on comparing functions within the same program. Instead, we compare functions against potentially arbitrary higher-order functions that can either be from the same program, the Haskell Prelude, or a library.

One closely related technique, *higher-order matching*, was introduced and developed by Huet in 1976 [34] and later by Statman in 1982 [35]. Originally defined for the simply typed lambda calculus, higher-order matching has been adapted to work with Haskell-like languages and as part of program transformation tools [36]. Higher-order matching is a NP-hard problem [36]. Conversely, and by targeting the functions to which it is applied, anti-unification algorithms can be made efficient [17]. One important aspect of our technique is determining whether or not a function *is* an instance of a chosen pattern, a decision that is made by inspecting the structure of the anti-unifier. Higher-order matching algorithms result in a set of possible substitutions from the *language* itself [36], meaning that a decision of whether the examined function is an instance of a given pattern remains unanswered, and potentially unanswerable from the result alone. This collection of inferred substitutions presents another problem: a substitution must be chosen from the overall set, and the substitution must maintain functional equivalence between the examined function and its reconstructed form. Anti-unification derives its substitutions from both examined function and pattern, requiring no selection of inferred substitutions to ultimately derive pattern arguments. Lastly, we envisage that when comparing functions of increasingly different structures reasoning about the anti-unifier and inferred substitutions will play a greater roll in argument inference.

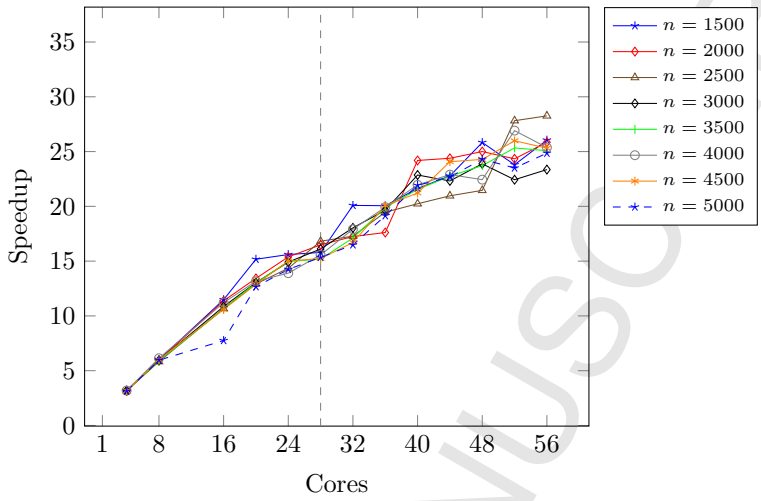


Figure 19: Speedups for `matmult` on *corryvreckan* using the row-wise parallelisation option and reported total (`Total`) time. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores.

Although the refactoring and clone detection fields are not traditionally concerned with the discovery of patterns, it is often a stage in the parallelisation process [8] as patterns as a simplification of parallelisation have been shown to be an effective [4, 5, 37, 38]. Pattern discovery for parallelism has primarily focused on approaches for imperative programs by analysing dependencies and machine learning techniques [39, 40, 8, 41]. Alternative automatic approaches focus on deriving parallel implementations from small programs or specifications, commonly employing the Bird Meertens formalism to calculate parts of the program [42, 43, 7, 44]. Whilst these approaches do not explicitly look for patterns, they use algorithmic structures such as *list homomorphisms* [45] and *hylomorphisms* [4] which are amenable to divide-and-conquer skeletons. These approaches have thus far focussed on translating operations into specific types, e.g. cons- or join-lists, which can then be easily parallelised [44, 7]. The approach by Geser [42] is particularly notable here, as it uses a form of anti-unification (referred to simply as *generalization*). These approaches are all limited by their type translations, and by the specific parallelism mechanics they introduce. In [46], Hu demonstrates how hylomorphisms can be found in arbitrary code, but the approach is limited to hylomorphisms. Our approach and prototype implementation works for Haskell 98, and can in principle discover arbitrary patterns in existing code. These patterns can be parallelised by, e.g., the approaches in [14, 4], but also used as part of code maintenance, and ensuring code reuse. Used for parallelism or performance reasons, our approach also allows the use of a variety of parallelisation or optimisation approaches.

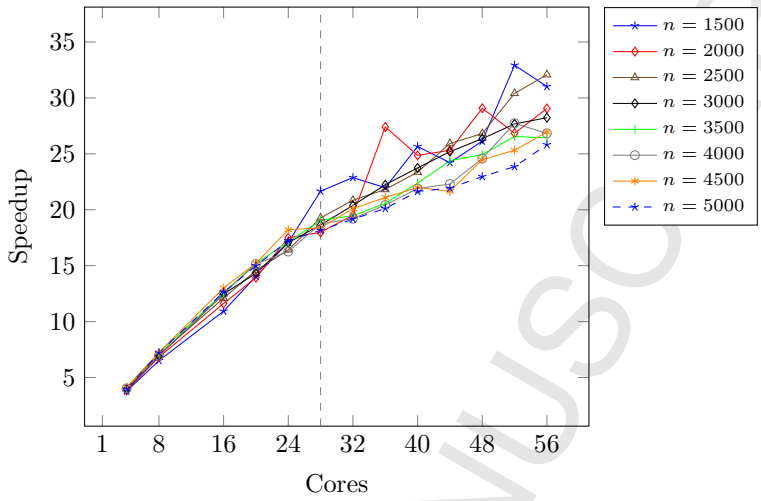


Figure 20: Speedups for `matmult` on *corryvreckan* using the block-wise parallelisation option and reported total (`Total`) time. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores.

## 6. Conclusions and Future Work

In this paper, we have presented a technique to detect instances of recursion schemes in Haskell functions using anti-unification. That is we describe how to detect recursive patterns whose behaviour is well understood. We have described how our approach can be combined with automated or semi-automated refactoring techniques to transform functions into a form that enables the detection of such recursion schemes, and subsequently to rewrite the function as an instance of the pattern that is detected. As part of our approach, we have defined a specialised anti-unification algorithm. We have prototyped our approach by adapting the HaRe refactoring tool for Haskell, and have demonstrated our technique on four benchmarks from the standard *NoFib* suite. We have shown that real performance improvements and speedups can be achieved using parallel versions of the patterns that we find. Whilst our approach shows promising results, our analysis does presently require code to be written in a stylised form. We envisage, however, that further work on argument derivation will allow us to detect recursion schemes in a broader range of structures. We also intend to explore how equational reasoning, in conjunction with standard reduction operations, can be applied to inferred substitutions to successfully derive pattern arguments more flexibly. Additionally, we intend to examine larger, pre-existing examples of Haskell code. Although we have explained the technique with reference to Haskell and the standard *iterN*, *map* and *fold* patterns, none of these is a fundamental limitation. The approach described in this paper is, in fact, completely general. In future, we expect to apply our work to a much broader range of Haskell higher-order functions including many of the libraries that al-

ready exist in e.g. the *hackage* repository. We also intend to apply our approach to more varied patterns, e.g. *unfolds* [47], which have been shown to be useful components of parallel programs [4].

## 7. Acknowledgements

This work has been partially supported by the EU H2020 grant "RePhrase: Refactoring Parallel Heterogeneous Resource-Aware Applications – a Software Engineering Approach" (ICT-644235), by COST Action IC1202 (TACLe), supported by COST (European Cooperation in Science and Technology), by EPSRC grant "Discovery: Pattern Discovery and Program Shaping for Manycore Systems" (EP/P020631/1), and by Scottish Enterprise PS7305CA44.

## References

[1] H. González-Vélez, M. Leyton, A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers, Vol. 40, John Wiley & Sons, Inc., New York, NY, USA, 2010, pp. 1135–1160.

[2] A. Barwell, C. Brown, K. Hammond, W. Turek, A. Byrski, Using "Program Shaping" to Parallelise an Evolutionary Multi-Agent System in Erlang, J. Computing and Informatics (to appear).

[3] M. Cole, Algorithmic Skeletons: Structured Management of Parallel Computation, MIT Press, Cambridge, MA, USA, 1991.

[4] D. Castro, S. Sarkar, K. Hammond, Farms, Pipes, Streams and Reforestation: Reasoning about Structured Parallel Processes using Types and Hylomorphisms, in: Proc. ICFP '16, 2016, pp. 4–17.

[5] V. Janjic, C. Brown, K. Hammond, Lapedo: Hybrid Skeletons for Programming Heterogeneous Multicore Machines in Erlang, Parallel Computing: On the Road to Exascale 27 (2016) 185.

[6] M. Coppola, M. Vanneschi, High-performance data mining with skeleton-based structured parallel programming, Parallel Computing 28 (5) (2002) 793–813.

[7] V. Kannan, G. Hamilton, Program Transformation to Identify Parallel Skeletons, in: Int. Conf. on Parallel, Distributed, and Network-Based Processing (PDP'16), IEEE, 2016, pp. 486–494.

[8] I. Bozó, V. Fordós, Z. Horvath, M. Tóth, D. Horpácsi, T. Kozsik, J. Köszegi, A. Barwell, C. Brown, K. Hammond, Discovering Parallel Pattern Candidates in Erlang, in: Proc. 2014 ACM Erlang Workshop, 2014, pp. 13–23.

[9] W. Partain, The nofib benchmark suite of haskell programs, in: J. Launchbury, P. Sansom (Eds.), Functional Programming, Glasgow 1992: Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, 6–8 July 1992, Springer London, London, 1993, pp. 195–202.

[10] G. D. Plotkin, A Note on Inductive generalization, Machine Intelligence 5 (1970) 153–163.

[11] S. Breitinger, R. Loogen, Y. O. Mallen, R. Pena, The Eden Coordination Model for Distributed Memory Systems, in: Proceedings of the 1997 Workshop on High-Level Programming Models and Supportive Environments (HIPS '97), HIPS '97, 1997, pp. 120–.

[12] S. Marlow, R. Newton, S. Peyton Jones, A Monad for Deterministic Parallelism, in: Proceedings of the 4th ACM Symposium on Haskell, Haskell '11, 2011, pp. 71–82.

[13] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, V. Grover, Accelerating Haskell Array Codes with Multicore GPUs, in: Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming, DAMP '11, 2011, pp. 3–14.

[14] N. Scaife, S. Horiguchi, G. Michaelson, P. Bristow, A Parallel SML Compiler Based on Algorithmic Skeletons, Journal of Functional Programming 15 (4) (2005) 615–650.

[15] J. C. Reynolds, Transformational Systems and the Algebraic Structure of Atomic Formulas, Machine intelligence 5 (1) (1970) 135–151.

[16] C. Brown, Tool Support for Refactoring Haskell Programs.

[17] P. E. Bulychev, E. V. Kostylev, V. A. Zakharov, Anti-unification Algorithms and Their Applications in Program Analysis, in: Proceedings of the 7th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics, PSI'09, 2010, pp. 413–423.

[18] P. Bulychev, M. Minea, An Evaluation of Duplicate Code Detection Using Anti-Unification, in: Proc. 3rd International Workshop on Software Clones, 2009.

[19] C. Brown, H.-W. Loidl, K. Hammond, ParaForming: Forming Parallel Haskell Programs Using Novel Refactoring Techniques, in: Proceedings of the 12th International Conference on Trends in Functional Programming, TFP'11, 2012, pp. 82–97.

[20] K. Hammond, M. Aldinucci, C. Brown, F. Cesarini, M. Danelutto, H. González-Vélez, P. Kilpatrick, R. Keller, M. Rossbory, G. Shainer, The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems, in: Formal Methods for Components and Objects, Vol. 7542 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 218–236. doi:10.1007/978-3-642-35887-6_12.

[21] E. Horowitz, A. Zorat, Divide-and-Conquer for Parallel Processing, IEEE Transactions on Computers C-32 (6) (1983) 582–585.

[22] V. Janjic, C. Brown, K. MacKenzie, K. Hammond, M. Danelutto, M. Aldinucci, J. D. Garcia, RPL: A Domain-Specific Language for Designing and Implementing Parallel C++ Applications, in: Proc. PDP '16: Intl. Conf. on Parallel, Distrib. and Network-Based Proc., 2016, pp. 288–295.

[23] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, P. Trinder, Seq No More: Better Strategies for Parallel Haskell, in: Proceedings of the Third ACM Haskell Symposium on Haskell, Haskell '10, 2010, pp. 91–102.

[24] L. Bergstrom, J. Reppy, Nested Data-parallelism on the Gpu, in: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming, ICFP '12, 2012, pp. 247–258.

[25] C. Brown, M. Danelutto, K. Hammond, P. Kilpatrick, A. Elliott, Cost-Directed Refactoring for Parallel Erlang Programs, International Journal of Parallel Programming (2013) 1–19.

[26] Programatica: Integrating programming, properties and validation.
URL `http://programatica.cs.pdx.edu`

[27] D. A. Plaisted, Handbook of Logic in Artificial Intelligence and Logic Programming (Vol. 1), 1993, Ch. Equational Reasoning and Term Rewriting Systems, pp. 274–364.

[28] R. Hinze, N. Wu, J. Gibbons, Unifying Structured Recursion Schemes, SIGPLAN Not. 48 (9) (2013) 209–220.

[29] T. Kutsia, J. Levy, M. Villaret, Anti-unification for unranked terms and hedges, Journal of Automated Reasoning 52 (2) (2014) 155–190.

[30] H. Gust, K.-U. Kühnberger, U. Schmid, Metaphors and anti-unification, in: Proc. Twenty-First Workshop on Language Technology: Algebraic Methods in Language Processing, Verona, Italy, 2003, pp. 111–123.

[31] J. W. Lloyd, An Algorithm of Generalization in Positive Supercompilation, in: Logic Programming: The 1995 International Symposium, MIT Press, 1995, pp. 465–479.

[32] C. Oancea, C. So, S. M. Watt, Generalisation in Maple 277–328.

[33] C. Brown, S. Thompson, Clone Detection and Elimination for Haskell, in: Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM '10, 2010, pp. 111–120.

[34] G. Huet, B. Lang, Proving and Applying Program Transformations Expressed with Second-order Patterns, Acta Inf. 11 (1) (1978) 31–55.

[35] R. Statman, Completeness, invariance and -definability, Journal of Symbolic Logic 47 (1) (1982) 1726.

[36] O. de Moor, G. Sittampalam, Higher Order Matching for Program Transformation, in: A. Middeldorp, T. Sato (Eds.), Functional and Logic Programming: 4th Fuji International Symposium, FLOPS'99 Tsukuba, Japan, November 11-13, 1999 Proceedings, 1999, pp. 209–224.

[37] L. Gesbert, F. Gava, F. Loulergue, F. Dabrowski, Bulk synchronous parallel {ML} with exceptions, Future Generation Computer Systems 26 (3) (2010) 486 – 490.

[38] F. Gava, F. Loulergue, A static analysis for bulk synchronous parallel {ML} to avoid parallel nesting, Future Generation Computer Systems 21 (5) (2005) 665 – 671.

[39] C. Hammacher, K. Streit, S. Hack, A. Zeller, Profiling Java Programs for Parallelism, in: Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering, IWMSE '09, 2009, pp. 49–55.

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

[41] J. Ahn, T. Han, An Analytical Method for Parallelization of Recursive Functions, Parallel Processing Letters 10 (01) (2000) 87–98.

[42] A. Geser, S. Gorlatch, Parallelizing Functional Programs by Generalization, J. Functional Programming 9 (06) (1999) 649–673.

[43] S. Gorlatch, Extracting and Implementing List Homomorphisms in Parallel Program Development, Science of Computer Programming 33 (1) (1999) 1 – 27.

[44] M. Dever, G. W. Hamilton, AutoPar: Automatic Parallelization of Functional Programs, in: 2014 Fourth International Valentin Turchin Workshop on Metacomputation (META 2014), 2014, pp. 11–25.

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

[46] Z. Hu, H. Iwasaki, M. Takeichi, Deriving Structural Hylomorphisms from Recursive Definitions, SIGPLAN Not. 31 (6) (1996) 73–82.

[47] J. Gibbons, G. Jones, The Under-appreciated Unfold, SIGPLAN Not. 34 (1) (1998) 273–279.

## Appendix A. Proof for Soundness of Anti-Unification Algorithm

**Theorem 1** (Soundness of Anti-Unification Algorithm)**.** *Given the terms $t_1$ and $t_2$, we can find $t$, $\sigma_1$, and $\sigma_2$, such that $t_1 \cong t\,\sigma_1$ and $t_2 \cong t\,\sigma_2$.*

*Proof Sketch.* Given the terms $t$, $t_1$, $t_2$, and substitutions, $\sigma_1$ and $\sigma_2$. The proof is by case analysis on $t_1$ and $t_2$.

**Case $t_1 \cong t_2$:**

By application of EQ with $t = t_2$ and $\sigma_1 = \sigma_2 = \varepsilon$, we must show that $t_1 \cong t_2\,\varepsilon$ and $t_2 \cong t_2\,\varepsilon$.

By Def. 5 and reflexivity, both $t_1 \cong t_2$ and $t_2 \cong t_2$ hold.

**Case $t_2 = (\texttt{Var}\ v) \wedge (v \neq \texttt{p})$:**

By application of VAR when $t = (\texttt{Var}\ v)$, $\sigma_1 = (v \mapsto t_1)$, and $\sigma_2 = \varepsilon$, we must show that $t_1 \cong (\texttt{Var}\ v)\,((v \mapsto t_1))$ and $(\texttt{Var}\ v) \cong (\texttt{Var}\ v)\,\varepsilon$ hold.

By Def. 4 and reflexivity, $t_1 \cong t_1$ holds.

By Def. 5 and reflexivity, $(\texttt{Var}\ v) \cong (\texttt{Var}\ v)$ holds.

**Case $t_1 = (\texttt{Var}\ v^{\texttt{f}}_{a_i}) \wedge t_2 = (\texttt{App}\ (\texttt{Var}\ u)\ (\texttt{Var}\ v^{\texttt{p}}_{a_i})) \wedge (u \neq \texttt{p}) \wedge (v^{\texttt{f}}_{a_i} \equiv v^{\texttt{p}}_{a_i})$:**

By application of ID where

$$\begin{aligned} t &= (\texttt{App}\ (\texttt{Var}\ u)\ (\texttt{Var}\ v^{\texttt{p}}_{a_i})) \\ \sigma_1 &= (\!|\,(v \mapsto id), \varepsilon\,|\!) \\ \sigma_2 &= \varepsilon \end{aligned}$$

we must show that:

$$(\texttt{Var}\ v^{\texttt{f}}_{a_i}) \cong (\texttt{App}\ (\texttt{Var}\ u)\ (\texttt{Var}\ v^{\texttt{p}}_{a_i}))\,(\!|\,(u \mapsto id), \varepsilon\,|\!) \tag{1}$$

$$(\texttt{App}\ (\texttt{Var}\ u)\ (\texttt{Var}\ v^{\texttt{p}}_{a_i})) \cong (\texttt{App}\ (\texttt{Var}\ u)\ (\texttt{Var}\ v^{\texttt{p}}_{a_i}))\,\varepsilon \tag{2}$$

For (1):
By Def. 6, $(\texttt{Var}\ v^{\texttt{f}}_{a_i}) \cong (\texttt{App}\ (\texttt{Var}\ u)\,(u \mapsto id)\ (\texttt{Var}\ v^{\texttt{p}}_{a_i})\,\varepsilon)$.
By Def. 4 and Def. 5, $(\texttt{Var}\ v^{\texttt{f}}_{a_i}) \cong (\texttt{App}\ (\texttt{Var}\ id)\ (\texttt{Var}\ v^{\texttt{p}}_{a_i}))$.
By Def. 8 and reflexivity, $(\texttt{Var}\ v^{\texttt{f}}_{a_i}) \cong (\texttt{Var}\ v^{\texttt{p}}_{a_i})$ holds.

For (2):
By Def. 5 and reflexivity, $(\texttt{App}\ (\texttt{Var}\ u)\ (\texttt{Var}\ v^{\texttt{p}}_{a_i})) \cong (\texttt{App}\ (\texttt{Var}\ u)\ (\texttt{Var}\ v^{\texttt{p}}_{a_i}))$ holds.

**Case** $t_2 \in \mathcal{R}_{\mathtt{p}}$**:**

By application of RP where $t = (\mathtt{Var}\ \alpha)$, $\sigma_1 = (\alpha \mapsto t_1)$, and $\sigma_2 = (\alpha \mapsto t_2)$, we must show that both $t_1 \cong (\mathtt{Var}\ \alpha)\,(\alpha \mapsto t_1)$ and $t_2 \cong (\mathtt{Var}\ \alpha)\,(\alpha \mapsto t_2)$ hold.

By Def. 4 and reflexivity, both $t_1 \cong t_1$ and $t_2 \cong t_2$ hold.

**Case** $t_1 = (C\ t_{11}\ \ldots\ t_{1n}) \wedge t_2 = (C\ t_{21}\ \ldots\ t_{2n}) \wedge$
$\forall i \in [1, n],\ (t_{1i} \cong t_i\,\sigma_{1i}) \wedge (t_{2i} \cong t_i\,\sigma_{2i})$**:**

By application of CONST where

$$\begin{aligned} t &= (C\ t_1\ \ldots\ t_n) \\ \sigma_1 &= (\!|\, \sigma_{11}, \ldots, \sigma_{1n} \,|\!) \\ \sigma_2 &= (\!|\, \sigma_{21}, \ldots, \sigma_{2n} \,|\!) \end{aligned}$$

we must show that:

$$\begin{aligned} (C\ t_{11}\ \ldots\ t_{1n}) &\cong (C\ t_1\ \ldots\ t_n)\,(\!|\, \sigma_{1i}, \ldots, \sigma_{1n} \,|\!) \\ (C\ t_{21}\ \ldots\ t_{2n}) &\cong (C\ t_1\ \ldots\ t_n)\,(\!|\, \sigma_{2i}, \ldots, \sigma_{2n} \,|\!) \end{aligned}$$

By Def. 6, $(C\ t_{11}\ \ldots\ t_{1n}) \cong (C\ (t_1\,\sigma_{11})\ \ldots\ (t_n\,\sigma_{1n}))$
and $(C\ t_{21}\ \ldots\ t_{2n}) \cong (C\ (t_1\,\sigma_{21})\ \ldots\ (t_n\,\sigma_{2n}))$.
By assumption, substitutivity and reflexivity,
both $(C\ t_{11}\ \ldots\ t_{1n}) \cong (C\ t_{11}\ \ldots\ t_{1n})$ and $(C\ t_{21}\ \ldots\ t_{2n}) \cong (C\ t_{21}\ \ldots\ t_{2n})$
hold.

**Case** $t_1 = [t_{11}, \ldots, t_{1n}] \wedge t_2 = [t_{21}, \ldots, t_{2n}] \wedge$
$\forall i \in [1, n],\ t_{1i} \cong t_i\,\sigma_{1i} \wedge t_{2i} \cong t_i\,\sigma_{2i}$**:**

By application of LIST where

$$\begin{aligned} t &= [t_1, \ldots, t_n] \\ \sigma_1 &= (\!|\, \sigma_{11}, \ldots, \sigma_{1n} \,|\!) \\ \sigma_2 &= (\!|\, \sigma_{21}, \ldots, \sigma_{2n} \,|\!) \end{aligned}$$

we must show that:

$$\begin{aligned} ([\ t_{11}\ \ldots\ t_{1n}]) &\cong ([t_1\ \ldots\ t_n])\,(\!|\, \sigma_{1i}, \ldots, \sigma_{1n} \,|\!) \\ ([t_{21}\ \ldots\ t_{2n}]) &\cong ([t_1\ \ldots\ t_n])\,(\!|\, \sigma_{2i}, \ldots, \sigma_{2n} \,|\!) \end{aligned}$$

By Def. 6, $([t_{11}\ \ldots\ t_{1n}]) \cong ([(t_1\,\sigma_{11})\ \ldots\ (t_n\,\sigma_{1n})])$
and $([t_{21}\ \ldots\ t_{2n}]) \cong ([(t_1\,\sigma_{21})\ \ldots\ (t_n\,\sigma_{2n})])$.
By assumption, substitutivity and reflexivity,
both $([t_{11}\ \ldots\ t_{1n}]) \cong ([t_{11}\ \ldots\ t_{1n}])$ and $([t_{21}\ \ldots\ t_{2n}]) \cong ([t_{21}\ \ldots\ t_{2n}])$
hold.

**Otherwise:**

By application of OTHERWISE where $t = (\texttt{Var}\ \alpha)$, $\sigma_1 = (\alpha \mapsto t_1)$, and $\sigma_2 = (\alpha \mapsto t_2)$, we must show that both $t_1 \cong (\texttt{Var}\ \alpha)\,(\alpha \mapsto t_1)$ and $t_2 \cong (\texttt{Var}\ \alpha)\,(\alpha \mapsto t_2)$ hold.

By Def. 4 and reflexivity, both $t_1 \cong t_1$ and $t_2 \cong t_2$ hold. □

Adam Barwell is a PhD student at the University of St Andrews working in the area of static analysis and refactoring. Adam graduated from University College London in 2013 where he developed an interest in programming langauge theory, and is now studying under Kevin Hammond and Chris Brown.

Chris Brown is a Post-Doctoral Research Fellow at the University of St Andrews working in the area of Refactoring and Parallelism. Chris has a Ph.D. from the University of Kent under the supervision of Simon Thompson, where he worked on the implementation of the HaRe refactoring tool for Haskell. Chris now works at St Andrews building radically new refactoring techniques and methodologies to exploit parallel skeleton and pattern applications in both the functional and imperative domains. Chris has published at major conferences in the field of functional programming, refactoring and parallelism.

Kevin Hammond is a Professor of Computer Science at the University of St Andrews working in the area of Parallelism. In the past, Kevin has helped implement Haskell, worked on EU Projects such as ParaPhrase, and is currently working on the RePhrase project.

*Biographies (Photograph)

Click here to download high resolution image

*Biographies (Photograph)

Click here to download high resolution image

*Biographies (Photograph)

Click here to download high resolution image

- We present a technique using anti-unification to discover recursion schemes in Haskell functions
- We present a new, specialised anti-unification algorithm.
- We have implemented our approach using the Haskell refactoring tool HaRe
- We have demonstrated our approach on a range of standard examples
- Our technique is in principle completely general, able to discover arbitrary patterns in code.