Abstract

Polytypic programming is very useful in functional languages to capture generic functionality, but is of little help to programmers in object-oriented languages. We have developed a form of polytypic programming that is more object-oriented friendly, called traversal-based generic programming. The approach involves the use of several algorithms for function set generation, dispatch, and type-checking. In this paper we give an overview of our approach and a detailed account of the various algorithms involved in making traversal-based generic programming useful, safe, and efficient.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features—Data types and structures

General Terms Algorithms, Design

Keywords Traversals, Function-Objects, Type Checking

1. Introduction

Over the years polytypic programming has proven its worth to programmers. It allows library writers to provide more general functions over datatypes, and allows programmers to use these functions on user-defined types with little or no specialization. There are some limitations when the genericity of a function definition does not match the level of abstraction necessary to solve a problem, e.g., higher level notions like evaluation, but when applicable, polytypic functions can eliminate excess boilerplate code. Typical approaches [14, 15] work well in functional languages like Haskell [23, 34], but cannot be applied directly to mainstream object-oriented (OO) languages. This is mainly due to the ad-hoc nature of type hierarchies in class-based OO languages, where each class is considered a type and can be extended/subclassed independently. This makes it difficult to model the generic structure of a hierarchy or translate instances into a universal datatype.

We have previously introduced a more OO friendly notion of generic and polytypic programming that we call traversal-based generic programming (TBGP) [5, 6]. Our approach uses a set of functions (e.g., a multi-entry function-object) to do a deep fold over objects/instances. An adaptive traversal walks an object and folds recursive results by applying functions (or methods) from the set, selected by a type-based multiple dispatch. The traversal selects the function with a signature that best matches the type of the current node and the types of recursive traversal results from fields. The separation of traversal and functions allows sets to be extended by overloading or overriding cases, and provides the necessary flexibility to adapt to the ad-hoc nature of class hierarchies. In addition, our approach supports the emulation of traditional forms of polytypic programming through function set generation.

Implementations of our approach [3, 4] involve a number of algorithms that make function sets more useful, make programs more efficient, and guarantee the safety of traversals and dispatch. In this paper we present some of the more interesting algorithms, in particular:

Generating Polytypic Functions We represent OO type hierarchies with an updated form of class dictionaries [21]. The structure of class dictionaries can in turn be described by a class dictionary. We use this more OO friendly description as a universal datatype over which to write functions that generate function sets that are specialized for arbitrary datatypes (Section 3). We use this technique to generate traditional polytypic functions like show, as well as useful extensible sets for implementing type-unifying and type-preserving [16, 19] style functions.

Type-based Multiple Dispatch Our adaptive traversal selects functions from a set to fold recursive results. We use a type-based asymmetric multiple dispatch, similar to CLOS [38], that determines the most specific matching function. We have an algorithm for signature comparison that operates at runtime, but this can be very inefficient in practice. Most of the function selection process can be done statically using the function signatures and data structures as a guide. We discuss the implementation details of both algorithms in Section 5.

Type Checking Traversals The non-standard nature of our traversal and function sets requires an external type checker. Approximating the traversal and function selection for recursive types can make static type checking difficult, since recursive results affect selection and vice versa. Our idealized model and type system are described elsewhere [5], but here we are concerned with an algorithm for this approximation (Section 6). We present an implementation of a custom unification algorithm that computes the return types of recursive traversals given a list of function signatures.

Function Set Coverage It is not enough to decide the type that a traversal using a function set will return. We must also be sure that at each dispatch point there is at least one function that can be selected for the possible recursive return types, similar to exhaustive pattern match checking. Since our type hierarchies can be described as trees, we call the abstract problem leaf-
expression structures is shown in Figure 1. We view classes as being either abstract, having fields, but no subclass list; or concrete, having fields, but no sub-
classes. Our CD definitions look similar to EBNF (including con-
esting and unification [36]. On the theoretical side, we intro-
duce a new coNP-complete problem that is related to exhaustive
pattern checking. We give two fixed-parameter tractable solutions,
with running times that depend on different parameters. We use a
more efficient solution to the decision version of the problem with
a greedy search technique corresponding to self-reducibility, rather
than the general reduction of search to decision for NP-complete
problems [33].

We begin with a background (Section 2) on our approach to
traversal-based generic programming using our C# implementation
for examples. After discussing the related algorithms in context,
we discuss the generation of polytypic functions that users of our TBGP implementations
unknowingly rely on. Our dispatch and type checking algorithms
are novel applications of older techniques: automata-based match-
ing [10, 13] and unification [36]. On the theoretical side, we intro-
duce a new coNP-complete problem that is related to exhaustive
pattern checking. We give two fixed-parameter tractable solutions,
with running times that depend on different parameters. We use a
more efficient solution to the decision version of the problem with
a greedy search technique corresponding to self-reducibility, rather
than the general reduction of search to decision for NP-complete
problems [33].

2. Background

What is traversal-based generic programming (TBGP)? The basic
view is that it is a separation of structural recursion and function-
ality. There are several other techniques for doing this (e.g., gener-
alized folds [24, 37], Scrap Your Boilerplate (SYB) [16, 17], and
traversal combinators [19, 39]), but we build on ideas from adapt-
ive programming [22, 31]. Our approach is completely functional
(i.e., side-effect free) and is conceptually similar to Lammel’s ideas
of updatable fold algebras [20]. Our Java and C# implementations
(collectively called DemeterF) include a class generator, data struc-
tures, and generic traversal libraries. This section is meant to be a
quick overview of TBGP, but before going into a more thorough
description and showing some function examples, we discuss data
structure descriptions.

2.1 Data Structures: Class Dictionaries

In order to describe object-oriented data structures to be traversed,
we use a convenient, modernized class dictionary (CD) syntax [21].
We view classes as being either abstract, having sub-classes and
possibly common fields, or concrete, having fields, but no sub-
classes. Our CD definitions look similar to EBNF (including con-
crete syntax) but abstract and concrete classes differing only by
the existence of subclasses. An example CD that represents simple
expression structures is shown in Figure 1.

```
Exp = (Int | Var | Def | Bin).
Int = <v> int.
Var = <id> ident.
Def = <id> ident "=" <e> Exp ;" <c> Exp.
Bin = (Plus | Pow) <l> Exp <r> Exp.
Plus = "+".
Pow = "^".
```

Figure 1. CD Example: expression structures

A class is defined as an identifier, e.g., Exp, followed by an
equal sign and a body, terminated by a period (";"). The body of

class ToString : FC{
    string combine(Def d, string id, string e, string b) {
        return "+" +id +" = " +b +"; "
    }
    string combine(Plus p, string l, string r) {
        return "+ " +l +" = " +r +";
    }
    string combine(Pow p, string l, string r) {
        return "^ " +l +" = " +r +";
    }
    string toString(Exp e) {
        return new Traversal(this).traverse(e); } }
```

Figure 2. Example: Exp to string conversion

The final method, named toString, accepts an Exp instance, and
creates a new Traversal passing a function-object, this, an
instance of ToString. The Traversal instance essentially ties the
recursive knot by interpreting the combine methods of the given
Traversal as fold functions over the structure. We can use our
function-object by creating a new instance and calling toString:

```
string s = new ToString().toString(an_exp);
```

When the traverse method is called (inside toString), the
Traversal proceeds with a depth-first walk of the given instance,
in this case an Exp. Once the walk of an object’s fields is com-
plete, the traversal selects the combine method from the function-
object with the most specific argument types. To determine the most
specific matching combine method, the traversal uses the type of

1 This version of the Traversal class uses C# reflection to both inspect the
function-object for combine methods, and to traverse structures.
the current object as its first argument, and the types of the field results as the rest of the arguments to combine, in left-to-right order. For primitive/value types like int the combine method is optional. By default the original value is returned, as illustrated in the combine for int, where the second argument is of type int (i.e., not string).

Taking all arguments into account, our selection is termed multiple dispatch. In the case of ToString, when traversal reaches an ident, the first method matches and can be applied. For a Var, the recursive traversal of the id field (i.e., the first method) produces a string, so the second method, (Var, string), matches and is applied. Other cases follow similarly. If there is no method applicable for the types (current object and recursive results) then the traversal throws an Exception.

2.3 Function Extension

At first glance this may seem like an encoding of generalized folds in C#, but it is actually much more flexible. Because sets of functions are represented as classes, we can override or overload combine methods using inheritance. Our DemeterF library contains several useful generic function-classes, which can be specialized to implement more interesting functions. In particular, we provide two classes, named TU and TP, that can be extended to implement type-unifying and type-preserving functions [16, 19] respectively.

For example, we may want to write a function to count all the int values in a given Exp. We can extend the parameterized class TU by overriding the default combine (with zero arguments) and a fold method, adding a special combine case for the type int. Our class CountInts is shown in Figure 3.

```
class CountInts : TU<int>
    override int combine() { return 0; }
    override int fold(int a, int b) { return a+b; }
    int combine(int i) { return i; }
}
```

Figure 3. Example: Counting the ints with TU

When an int is reached our special combine method is called. When there are no fields (i.e., at the leaves of the structure) the default combine() is called. Otherwise, TU uses the fold method to merge recursive field results (all integers in this case) to a return result, also of type int.

Similarly, we can overload cases of TP (sometimes referred to by OO programmers as copy) to implement transformations or functional updates. The class Simpler in Figure 4 implements part of a simplifying transformation for Exps.

```
class Simpler : TP
    Exp combine(Plus p, Int l, Int r)
    { return new Int(l.v+r.v); }
    Exp combine(Def d, ident id, Exp e, Int b)
    { return b; }
}
```

Figure 4. Example: Exp simplification with TP

Simpler extends TP with combine cases for Plus and Def that match specific types of recursive results. The first method matches a Plus with Int results for its l and r fields, performing the addition and constructing a new Int to return. The second case matches a Def with a body that is an Int, in which case the Def is no longer needed, so we return the inner Int. When our methods are not applicable, TP’s combine methods rebuild the structure automatically. The key is that the functions are applied recursively over an Exp, and only the most specific method will be called. Since TP is more general, our additional methods are called if/when they are applicable.

Of course, the extension of function-classes is not limited to library defined classes. For example, if we change our structures by adding a new subclass of Bin named Times:

```
Bin = (Times | Plus | Pow) ... .
// ...
Times = "*".
```

Then we can also extend our function-classes, e.g., ToString:

```
class ToStrTimes : ToString
    string combine(Times y, string l, string r)
    { return "*" + l + r; }
}
```

We can use instances of ToStrTimes instead of ToString for structures that contain Times instances. The function selection of the traversal adapts our function-objects to the data structure.

2.4 Algorithms

After reading the previous sections there are likely questions that remain. In particular, we foresee the following possibilities:

- The function-class ToString in Figure 2 looks like it follows directly from the CD in Figure 1. Also, what would specific implementations of TU, TP, or Traversal look like for a given CD? Could all of these be automatically generated?
- How is dispatch implemented; how is the most specific method found? What does this mean when function-classes involve overriding and overloading?
- The separate traversal and function-objects are very flexible, but with complex type hierarchies how can we be sure that a traversal will not throw an exception? Given a structure and a function-class, how can this be verified?

In the rest of this paper we provide answers to each of the above questions, in order, by describing the algorithms used in our various TBGP implementations. In the next section we discuss the generation of function-classes from CDs, including Show, TU, and TP, and give the resulting TU and TP for our Exp example. In Section 4 we introduce notation, datatypes, headers, and functions necessary to describe our more detailed algorithms and their implementations. We then give implementations of dynamic and static versions of our multiple dispatch in Section 5. Our algorithm for function-class type checking is presented in Section 6, and an important aspect of function-class checking, method coverage, is discussed in Section 7.

3. Generating Polytypic Functions

One of the benefits of TBGP is the various levels of abstraction at which programmers can write functions. We usually write specific functions for specific datatypes, but there are several useful function-classes that only depend on the data definitions in a given CD. In DemeterF, instead we write functions over the structure of CDs that generate function-classes to be used with a traversal. Though our implementation is complicated by parametrized types, we essentially traverse the abstract syntax tree of the CD to produce specialized combine methods. In this section we give more abstract specifications of our generation (i.e., compilation) of
generic function-classes and traversals from CD definitions by way of rewrite rules.

At runtime our structures are only made up of concrete classes, so generated function-classes depend only on the structure of concrete classes. Before generating function-classes, our implementation transforms more complex CDs into a simpler representation by pushing common fields from abstract classes down into concrete subtypes. For the purpose of generating function-classes it is enough to view a CD as a list of concrete class definitions of the form:

\[ C = \langle f_1 \rangle T_1 \cdots \langle f_n \rangle T_n \]

Where each type, \( T_i \), can be either abstract or concrete. The field names, \( f_i \), are actually not important, but we use them to keep the names of method parameters consistent. When necessary we will view abstract class definitions simply as a list of bar separated subtypes:

\[ A = T_1 | \cdots | T_n \]

Which will be needed for traversal generation, discussed in Section 3.4.

### 3.1 Show

Printing in various forms is traditionally a polytypic function, though it usually needs more than just the structure of types to produce meaningful string representations (e.g., the names of values constructors). We define the generation of the function-class \( \mathit{Show} \) (abstractly) as a function from concrete definitions to methods, using a template to describe the format of our resulting function-class. The template for \( \mathit{Show} \) is rather simple:

```java
class Show : FC{
    // Convert primitives
    string combine(int p) { return **ep; }
    // ... */

    // Generate the rest with GenShow
    \forall C \in CD . GenShow(C)
}
```

The template simply provides a class definition and \( \mathit{combine} \) methods for primitives that convert each into a \texttt{string}. The rest of the body of \( \mathit{Show} \) is generated by \texttt{GenShow}, using a simple rewrite rule mapped to each concrete definition from the CD:

\[
\text{GenShow}(C = \langle f_1 \rangle T_1 \cdots \langle f_n \rangle T_n) \Rightarrow \text{string combine}(C \_h, \text{string } f_1, \ldots, \text{string } f_n)
\]

For each concrete definition with \( n \) fields we create a \texttt{combine} method with \( n+1 \) arguments. The first is of type \( C \), the defined type, and the rest are of type \texttt{string}. During the traversal of an object using an instance of \( \mathit{Show} \), the field traversals will recursively convert the fields into strings before calling the matching \texttt{combine}. Within each method, the return \texttt{string} is constructed by concatenating the separating the recursive field results with commas, wrapping them in parentheses, and prefixing the \texttt{string} with the class name, \( C \).

### 3.2 Type Unifying Functions

\( \mathit{Show} \) is a special case of a more general function that is commonly associated with \texttt{folds}: type-unifying functions, or queries [16]. We can use this kind of function to sum a certain (deep) property over an object, or collect specific objects into a list. In order to generate the equivalent function-class, \( \mathit{TU} \), we provide a class that is parametrized by the eventual return type, \( X \). Our template is shown below:

```java
abstract X fold(X a, X b);
abstract X combine();

// Primitives call default
X combine(int p){ return combine(); }
// ... */

// Generate the body with GenTU
\forall C \in CD . GenTU(C)
```

We define the abstract methods for producing the default result (\texttt{combine()} and folding together two recursive results, respectively. Primitive combine methods can be overridden, but initially return the default result. Our generation rule for concrete definitions is a generalization of that for \texttt{Show}:

\[
\text{GenTU}(C = \langle f_1 \rangle T_1 \cdots \langle f_n \rangle T_n) \Rightarrow X \text{ combine}(C \_h, X f_1, \ldots, X f_n)
\]

Each generated \texttt{combine} method accepts \( n+1 \) parameters: again the first of type \( C \), but the rest are of our type parameter \( X \). If necessary, the return result is computed by nested calls to \texttt{fold}. Figure 5 shows the resulting TU class, specialized for our Exp CD. The generated version of is a direct replacement for the generic/reflective version used in Figure 3. The generated function-class gives us much better performance, especially when we can inline traversals [6].

### 3.3 Type Preserving Functions

Our last function-class generation example is probably the most useful. We use it often to do recursive functional updates and transformations over different types. Since combine methods are optional for primitive types we leave them out of our template, shown below:

```java
class TP : FC{
    // Generate the body with GenTP
    \forall C \in CD . GenTP(C)
}
```

Our generation rule creates a \texttt{combine} method that simply reconstructs a new \( C \) instance from the recursive traversals results.

\[
\text{GenTP}(C = \langle f_1 \rangle T_1 \cdots \langle f_n \rangle T_n) \Rightarrow C \text{ combine}(C \_h, T_1 f_1, \ldots, T_n f_n)
\]

Because the transformation is type preserving, each field result type is the same as its defined type, \( T_i \). The resulting generated TP class for our Exp CD is shown in Figure 6.

### 3.4 Traversal Generation

Our function-classes can be considered \textit{near-sided}, since they do not look past the types of the results of recursive traversal. They do, however, rely on a generic traversal to adapt their \texttt{combine} methods to different structures. When we have a specific CD, the generic/reflective \texttt{Traversal} used in Section 2 can be replaced with a generated version that performs \textit{much} better.

Our template for traversal generation is shown below.
For each of the class’ fields we recursively call traverse and store the result in a local variable. Since our traversal can be used with any function-object, we assume nothing about the return types by using object. Once all the instance’s fields have been traversed we apply our function object, fobj to an array of the results, including the original object as its first element. The elided apply determines the types of the arguments and dynamically dispatches to fobj’s most specific combine method.

3.5 Discussion

The generation of function-classes benefits from the nearsighted nature of our traversals. The combine methods can be generated independently, since they need not look past their argument types.

On the other hand, Traversal generation is lower level, dealing with instance checks and function-object dispatch. By mixing generated function-classes and specialized traversals we can achieve relatively good performance, but in order to achieve performance that competes with hand-written functions, we need to improve our dispatch algorithms, i.e., the implementation of apply, which is the topic of Section 5.

4. Types and Signature Notation

The rest of our algorithms deal with types and method signatures. In this section we define a convenient notation for the descriptions to follow. As we approach the implementation of various algorithms we will also define the Haskell datatypes and function headers that will be needed. We chose Haskell because it allows us to present algorithmic descriptions that are unambiguous, concise, and best of all, executable. All our implementations should be reasonably self explanatory, but we will also walk through the code to make sure they are understandable, even to non-Haskell wizards.

4.1 Types

We model types as symbols. As before we use a slightly simpler model of CDs that does not include syntax strings or common fields. As before, abstract classes are defined by a list of subtypes/variants:

\[ A = T_1 \mid \cdots \mid T_n \]

For the checking of traversals, field names in concrete classes are not important, so we model concrete classes as products of types:

\[ C = T_1 \times \cdots \times T_n \]

Our subtype relation, \( \triangleleft \), is built from the relationship given by abstract classes from a CD, which we write \( \prec \) and call extends:

\[ A = T_1 \mid \cdots \mid T_n \Rightarrow \forall i \in [1..n] \; T_i \prec A \]

We write its transitive closure \( \triangleleft \), defined as:

\[ B \prec D \equiv B \prec D \lor \exists C . C \prec D \land B \prec C \]

Finally, \( \trianglelefteq \) is defined as the reflexive extension of \( \triangleleft \):

\[ C \trianglelefteq B \equiv C = B \lor C \prec B \]

When needed we will also use a relation we call related, or \( \bowtie \), which extends the subtype relation in both directions:

\[ C \bowtie C' \equiv C \trianglelefteq C' \lor C' \trianglelefteq C \]

If two types are related, then we will consider them when simulating multiple dispatch.

4.2 Method Signatures

When discussing combine methods we are mostly interested in their argument types. We model each signature as a vector (or sequence) of symbols. We will use vector (over-arrow) notation for variables that represent signatures, but also refer to them as subscripted elements when convenient:

\[ S = (s_1, \ldots, s_n) \]

Function-classes/objects are represented as sets of signatures; we will use uppercase letters for signature sets:

\[ S = \{ s_1, \ldots, s_n \} \]

Though we will usually use lists, not sets, especially in our Haskell implementations.

4 Like Java, C# will statically resolve the overloaded traverse calls because of casting.

5 The authors are far from being Haskell wizards, so explanations may be overly thorough.
We extend our subtype relation, \( \leq \), to signatures of equal length, say \( n \):

\[
\vec{s} \leq \vec{t} \equiv |\vec{s}| = |\vec{t}| = n \land \forall i \in [1..n] \; s_i \leq t_i
\]

Similarly for our related relation, \( \bowtie \). We refer to the reflexive subtype relation as symmetric, and intuitively call it applicable, stating that a method with the second signature is applicable to runtime argument types described by the first signature.

When two signatures are related, we can order them to model dispatch with an asymmetric relation on signatures, \( \subsetneq \):

\[
\vec{s} \subsetneq \vec{t} \equiv \vec{s} \leq \vec{t} \land \exists i \in [1..n] . (s_i < t_i) \land \forall k \in [1..i-1] . s_k = t_k
\]

This ordering is similar to lexicographic order typically used to sort strings, and is used in our definition of most specific.

4.3 Haskell Headers

Figure 7 shows our initial type definitions and functions for describing our algorithms in Haskell. We model type names, Typ, as strings, and signatures as lists of Typs. We will use the function update to replace the \( i \)th Typ in a Sig with the given Typ. Our reflexive, transitive subtype relation is defined as a binary predicate, which is used to define an ordering function, sub_ord, eventually for sorting lists of Typs.

```haskell
-- Type names
type Typ = String
-- Method argument signatures
type Sig = [Typ]
-- Replace the i-th Typ in a Sig
update :: Sig -> Int -> Typ -> Sig
-- Reflexive/transitive subtype relation
subtype :: Typ -> Typ -> Bool
sub_ord :: Typ -> Typ -> Ordering
-- Related by subtyping/supertyping
related :: Typ -> Typ -> Bool
rel_sig :: Sig -> Sig -> Bool
```

Figure 7. Setup: Datatypes and Relations

We also define our related predicate on types, to represent \( \bowtie \), and extend it to signatures with rel_sig.

Our symmetric and asymmetric signature comparison functions are completely defined in Figure 8. symmet is a mapping of subtype over lists of types: zipWith applied our predicate pair-wise to elements from different lists. The function asymmet recursively looks for the first non-equal type, which must be a subtype. The two cases of asymmet use patterns to match empty lists, [], and non-empty lists, (a:as), where the latter binds a to the head of the list, and as to its tail. We use these two functions in combination to describe our method dispatch in Section 5.

```haskell
-- Symmetric order on signatures (applicable)
symmet :: Sig -> Sig -> Bool
symmet as bs = and (zipWith subtype as bs)

-- Asymmetric order on signatures (more specific)
asymmet :: Sig -> Sig -> Bool
asymmet [] [] = False
asymmet (a:as) (b:bs) = a ++ b = asymmet as bs | otherwise = subtype a b
```

Figure 8. Setup: Signature comparison

4.4 Running Example

In order to demonstrate our signature-based algorithms with a small but meaningful example, we will use the CD definitions shown in Figure 9. The concrete class A has two fields of type B. B is an abstract class with two subtypes, C and D, which have no fields.

For this CD we write a simple function-class, shown in Figure 10. The test method within Test traverses an instance of A.

```haskell
class Test : FC{
    B combine(b){ return b; }
    C combine(a, b l, b r){ return l; }
    C combine(a, b l, C r){ return r; }
    C combine(a, C l, b r){ return new C(); }
}
```

Figure 10. Example for Dispatch/Type Checking

The first combine method simply returns the B when applied. The other combine methods together return the left-most C in the traversed A, or create a new C if none exists. These three methods will demonstrate our method dispatch, type checking, and method coverage in the following three sections.

5. Multiple Dispatch

We now have all the tools to describe the multiple dispatching algorithm used in DemeterF. During traversal, when recursive results have been returned for the fields of an object, the traversal uses the types of the results to determine the most specific method signature to be called. We have devised two different dispatch algorithms: one which assumes nothing about the method signatures to be applied, and a second that calculates a decision tree for a list of method signatures, given an approximation of the argument/traversal types. These correspond to dynamic and static (or mixed) dispatch strategies used for our reflective and static/generated traversal implementations in DemeterF.

5.1 Dynamic Dispatch

Given a signature representing the types of runtime arguments and a list of signatures representing a function-class, our dynamic dispatch returns the selected, most specific signature. An implementation of our dynamic algorithm in Haskell is shown in Figure 11. We divide it into two separate functions that search through the

```haskell
select :: Sig -> [Sig] -> Sig
select c [] = error "No Applicable Sig"
select c (a:as) = if (symmet c a)
   then (best a c as)
   else (select c as)
```

Figure 11. Reflective Selection Algorithm
list of signatures to find the best one. The function select iterates through the list of signatures until an applicable Sig (using asymmet) is found. Again we use patterns to match empty list ([]), and non-empty list ([a:a]). If the list of signatures is empty, then there is no applicable function for the given arguments, and an error is raised. Once an applicable method is found, select then passes it off to the function best, which searches for another applicable signature that is more specific (using asymmet).

5.2 Example and Discussion

Our example function-class Test from Figure 10 consists of four combine methods. If the traversal of the fields of an A returned a D and a C respectively, then the signature passed to select would be ["A", "D", "C"]. For these results, select returns the third combine signature, ["A", "B", "C"]. If the first field of the A was a C instead, then the second combine, ["A", "C", "B"], would be selected. In contrast, because our method selection is asymmetric, traversal results of ["A", "C", "C"] would cause the second signature ["A", "C", "B"] to be selected instead.

The select algorithm suits our purposes for a dynamic adaptive traversal. However, it is not the most efficient. Our inefficiency comes from the fact that all method signatures of the function-class are passed to select, and must be compared to determine which is the most specific. As we saw in our C# examples (e.g., Figure 2), the number of methods applicable at any given concrete type is usually limited. In many of the programs that we have written using DemeterF, our function-classes have less than 5 overloaded methods applicable at each concrete type. If we can figure out before traversal which methods might possibly be called, then we can greatly increase dispatch efficiency.

5.3 Minimal Dispatch (Residue)

By determining which methods might be called at a given point in traversal, we can not only limit our signature search, but we can statically build a decision tree to determine the most specific signature, with a small amount of residual "instance" checks left for runtime. Figure 12 shows Haskell datatypes and functions that calculate the dispatch residue. We do this by constructing a decision tree, Dec, of type tests. An (IF i t d1 d2) value represents an instance check: if the i\textsuperscript{th} parameter is an instance (or a subtype) of the type t, then the decision continues with the left decision tree, d1; otherwise it continues with d2. A (CALL fi) value represents the final selection and dispatch to the given signature.

The top method, residue, accepts a signature, s, that approximates the types to be dispatched during traversal, and the list of method signatures, as. We construct our decision for the parameter number i, which starts at 0. If there are no methods given then we raise an error.\(^6\) If we have tested all arguments, then we can safely (and statically) create a CALL and select the most specific applicable method. Otherwise, we filter the related signatures and collect each of their i\textsuperscript{th} parameter Types. The ifinf list operator !! returns the i\textsuperscript{th} element of the list, and (!!i) is its partial application. We use nub to remove duplicates and the resulting list is used to build a list of pairs, ps, using a custom function accum, in subtype order. accum accumulates a list of pairs of a Typ and a list of Typ that represents an eventual type test, and the prefix of Types representing instance checks that will have failed. The resulting pairs are used to build another list of pairs, with Haskell’s list comprehension notation.

The name gs stands for signature groups: we place signatures into (possibly overlapping) groups by their i\textsuperscript{th} argument type. In the outer comprehension we take the pair of the Typ, t, and the

\(^6\) This is a very weak check, since the algorithm only chooses from the given methods. Eliminating 'incompleteness' errors is discussed in Section 7.

```
data Dec = CALL Sig
  | IF Int Typ Dec Dec
residue :: Sig -> [Sig] -> Dec
residue s as = decision 0 s as

decision :: Int -> Sig -> [Sig] -> Dec
decision i s [] = error "No Signatures Given"
decision i s as =
  if (i >= (length s))
    then (CALL (select s as))
  else
    let ts = nub (map (!!i) (filter (rel_sig s) as))
        ps = accum (sortBySubOrd ts) []
        gs = [(t, [a | a <- as, related (a!!i) t &&
                   not (any (subtype (a!!i)) ignr)])
             | (t, ignr) <- ps]
    in (buildDec i s gs)
    where
      accum :: [Typ] -> [Typ] -> [(Typ, [Typ])]
      accum [] ignr = []
      accum (t:ts) ignr = (t, ignr):accum ts (t:ignr)

buildDec :: Int -> Sig -> [(Typ, [Typ])] -> Dec
buildDec i s [] = error "No Groups"
buildDec i s ((t, as):gs) =
  let d = (decision (i+1) (update s i t) as)
      in if (null gs) then d
       else (IF 1 t d (buildDec i+1 gs))
```

Figure 12. Residual Selection Algorithm

We then pass our method groups to buildDec, which constructs the nested IFs to decide between the groups based on the type of the i\textsuperscript{th} argument. Our list of pairs is deconstructed with a simple pattern match. We compute a decision tree for the next argument (i+1), using update to make the signature more accurate: since d will be under the IF, the type t may be more specific than before. We can then determine whether or not an IF is necessary, if this is the last (or only) signature group.

5.4 Example

With our example function-class Test, we can construct a dispatch for the traversal of an A instance. The traversal of a C or D instance is returned unchanged, so the best bound we can place statically on the dispatch arguments is ["A", "B", "B"]). When residue is called with this and the list of method signatures it returns the Dec instance shown in Figure 13. If we interpret the decision on an argument signature of ["A", "D", "C"], then we select the else branch of the outer IF (D \(\not\subseteq\) C), and the then branch of the inner IF, finally selecting ["A", "B", "C"], the same as our dynamic select algorithm.

```
IF 1 "C" (IF 2 "C" (CALL ["A","C","B"])
  (CALL ["A","C","B"])
  (CALL ["A","B","C"])
)
```

Figure 13. Residual Selection Algorithm
5.5 Implementation Comparison

Each of our dispatch algorithms does its job well. In DemeterF, we use the reflective dispatch (select) to prototype function-classes, or when efficiency is not important. The benefit of select is that it requires no prior knowledge of the method signatures or argument types. So it is applicable to any function-class, over any traversal. Once the development of structures and methods has settled, we typically generate traversal code for a particular function-class and data structures, using reduce to compute dispatch decisions. These decisions replace apply in the generated traversals for concrete types from Section 3.4.

The decision result from reduce is optimal in the sense that we use a minimal amount of decisions to select the appropriate signature. At worst, the path to any CALL decision has a length that is a product of the size of the largest abstract definition and the number of arguments. However, the worst case is almost impossible to recreate, since it requires at least as many methods (e.g., all permutations). Average case runtime is much better, as our example in Figure 13 shows, requiring only 2 tests. In our experience implementing DemeterF, dispatch reduce usually contains less than 3 instance tests, which in certain cases can give traversal-based functions better performance than handwritten Java code [6]. But, building a minimal dispatch decision relies on approximating traversal return types accurately, and assuring that a method exists for each of the approximated traversal result; the topics of the next two sections.

6. Type Checking

Our traversal-based approach using function-objects is very flexible with respect to the different types that can be traversed and the types that can be returned by a given function-class. In particular, the traversal of different (unrelated) types can return completely different results. In order to generate specialized traversals and replace our dynamic dispatch, select, with a more efficient static decision, we calculate the traversal return types of a function-class over a data structure. Type checking a traversal involves solving recursive equations using a simple form of unification [36], inspired by Milner’s original type inference algorithm [27].

Our algorithm accepts a representation of a CD, a function-class, a traversal start type, and a list of types currently being checked. It returns a type paired with a substitution. The type represents the return type of a traversal of an instance of the starting type, using the given function-class. The substitution maps recursive abstract types to the type that the functions actually handle, giving us an upper bound on the possible return type.

6.1 Haskell Setup

The algorithm is best described in Haskell code. Figure 14 shows our datatype and function headers that we will use to model CDs, method types, and substitutions.

A CD is represented by a list of abstract and concrete \((\text{Abst} \text{ and} \ Concr)\) definitions, each with a name and a list of \text{Ty}ps. We will use the function finddef to lookup a type’s definition in a CD and commonSuper to determine the closest common supertype of two types, sometimes referred to as their least upper bound (LUB). A Meth is a pair\(^7\) of a Sig and a return \text{Typ}, and function-classes will be modeled as a list of Meth. Our type checking function returns a pair of a special type, RTyp, and a substitution, Subst, which will be created by unifying the expected and actual method signatures. RTyp represents either a type variable (TVar), for recursive uses, or a normal (user) type (UTyp). The function subst applies a substitution by replacing type variables with their binding

\[ \text{data CD} = [\text{Def}] \]
\[ \text{data Def} = \text{Abst Ty}p \ [\text{Typ}] \]
\[ \text{finddef} :: \text{CD} \rightarrow \text{Typ} \rightarrow \text{Def} \]
\[ \text{commonSuper} :: \text{Typ} \rightarrow \text{Typ} \rightarrow \text{Typ} \]
\[ \text{type Meth} = (\text{Sig}, \text{Typ}) \]
\[ \text{type Subst} = [(\text{String}, \text{Typ})] \]
\[ \text{data RTyp} = \text{TVar String} \mid \text{UTyp Typ} \]
\[ \text{subst} :: \text{Subst} \rightarrow \text{RTyp} \rightarrow \text{Typ} \]
\[ \text{lub_rets} :: [(\text{RTyp}, \text{Subst})] \rightarrow (\text{RTyp}, \text{Subst}) \]

\[ \text{x} \in \text{RSig} \]
\[ \text{data RSig} = [\text{RSig}] \]
\[ \text{data RSig} = [\text{RSig}] \]

Figure 14. Setup: Helpers and Datatypes

in the given Subst. The function lub_rets combines a list of \((\text{RTyp}, \text{Subst})\) pairs, finding the commonSuper for the RTyp and commonSuper for any duplicate bindings within the substitutions.

We begin our algorithm by showing the extension of rel\_sig to RTyp, shown in Figure 15. The function accepts a list of RTyp/Subst pairs and determines whether or not the given Sig is related. What is important here is that we assume that a type variable is always related to a given type (in the case expression). This allows us to approximate the traversal return types and method selection for recursive type uses by overestimating the related signatures.

Figure 16 shows our unification function for method arguments (of related signatures) and least upper bound function for substitutions. We use unify\_args to unify type variables (i.e., recursive

\[ \text{rel_rsig} :: [(\text{RTyp}, \text{Subst})] \rightarrow \text{Sig} \rightarrow \text{Boo}l \]
\[ \text{rel_rsig} (\text{r:rs}) (\text{t:ts}) = \]
\[ \text{rel_rsig} (\text{r:rs} \text{ ts}) \&\& \text{case} (\text{fst} \text{ r}) \text{ of} \]
\[ \text{TVar n} \rightarrow \text{True} \]
\[ \text{UTyp n} \rightarrow \text{rel} \_ \text{ts} \]
\[ \text{rel_rsig} [\text{[]} \text{[]} ] = \text{True} \]
\[ \text{rel_rsig} _ \_ = \text{False} \]

Figure 15. Related Signatures with RTyp

uses) with the methods that might be called at runtime. The call to foldr combines recursive substitutions, then we decide whether or not to add a new binding. For TVars we use lub_subst to merge a new binding that matches the corresponding argument type. The implementation of lub_subst merges bindings of the same name by finding the common supertype of their result type (lub_rets is similar).

\[ \text{rel_rsig} :: [(\text{RTyp}, \text{Subst})] \rightarrow [\text{Typ}] \rightarrow \text{Subst} \]
\[ \text{unify\_args} [\text{[]} \text{[]} ] = \text{[]} \]
\[ \text{unify\_args} (\text{r:rs})(\text{t:ts}) = \]
\[ \text{lub_subst} :: (\text{String}, \text{Typ}) \rightarrow \text{Subst} \rightarrow \text{Subst} \]
\[ \text{lub_subst} (\text{n,t}) [\text{[]} = [\text{[]} \text{[]} ] \]
\[ \text{if} (\text{n} == \text{np}) \text{then} (\text{n,commonSuper t tp}):\text{ss} \]
\[ \text{else} (\text{np,tp}):\text{lub_subst} (\text{n,t}) \text{ss} \]

Figure 16. Unification and Substitutions

\[ (a,b) \] is Haskell syntax for both the type and value constructors for pair.

\[ (a,b) \] is Haskell syntax for both the type and value constructors for pair.
Finally, Figure 17 shows our typecheck function that computes the return type of a traversal given a CD, a list of Meth, a list of recursive abstract Types, and a start Typ that will be traversed. The function computes a return type for a traversal with the given list of methods and a substitution that represents any constraints on uses of recursive abstract types.

\[
\text{typecheck} :: \text{CD} \rightarrow \text{[Meth]} \rightarrow \text{[Typ]} \rightarrow \text{Typ} \rightarrow (\text{RTyp, Subst})
\]

Our algorithm first checks if the start type is an element of the rec list, if so then we return a TVar for the type, which will be unified later. This keeps the algorithm from recurring infinitely. The type checking task is then delegated to a helper function, checkdef, after looking up the type’s definition in the CD. In checkdef we distinguish between abstract and concrete type definitions. For abstract definitions we find the common supertype (\text{Plus ods}, \text{Int}).

% Alternatively, our implementation of type checking handles the possible method, combine(B), which returns the left of a pair, is composed (\text{fst}) with the partial application of \text{rel_rsig}. We map a local function \text{retytp} over the methods to create pairs with the method’s return type, and the unification of any recursive types with the argument types of selected methods. Our last step is to find the least upper bound of the return types and substitutions from the methods.

\[\text{checkdef} :: \text{Def} \rightarrow (\text{RTyp, Subst})\]

\[\text{checkdef} (\text{Abst} n \text{ sts}) =\]
\[\text{lub_rets} (\text{map} (\text{typecheck} \text{ cd fc (n:rec)}) \text{ sts})\]
\[\text{get} \text{ fs} \text{ dts} = (\text{map} (\text{typecheck} \text{ cd fc rec}) \text{ ts})\]
\[\text{argts} = (\text{UTyp n,[]})\]:\text{fidents}\]
\[\text{rels} = \text{filter} ((\text{rel_rsig argts}).\text{fst}) \text{ fc}\]
\[\text{retytp} n = (\text{UTyp} (\text{and} m)\),\]
\[\text{unify_args} \text{ argts} (\text{fst} m)\]
\[\text{in} \text{ lub_rets} (\text{map retytp relms})\]

**Figure 17.** Type Checking Algorithm

Our algorithm first checks if the start type is an element of the rec list, if so then we return a TVar for the type, which will be unified later. This keeps the algorithm from recurring infinitely. The type checking task is then delegated to a helper function, checkdef, after looking up the type’s definition in the CD. In checkdef we distinguish between abstract and concrete type definitions. For abstract definitions we find the common supertype (\text{Plus ods}, \text{Int}).

In checkdef we distinguish between abstract and concrete type definitions. For abstract definitions we find the common supertype (\text{Plus ods}, \text{Int}).

For the abstract function-class example from Figure 10 is rather simple to type check, since the defined classes are not recursive. For a traversal starting at A, we eventually type check C and B to find that the possible method, combine(B), returns a B. The return types trivially unify to B, which is used to create the signature ["A", "B", "B"] and find related methods (argts and relms in Figure 17). All signatures of Test with A as their first argument are related and their return types unify (again, trivially) to C.

For the Simpler function-class over Exp (Figure 4), the methods inherited from TP are equivalent to identity, i.e., traversal of an Int returns an Int, Var returns Var, etc. Our overloaded methods for Plus and Def affect the type checking of Simpler in two ways. First, both method signatures place constraints on recursive uses Exp, though the constraining argument type \text{Int} is easily unified with the constraints of \text{Exp} from other signatures, resulting in a substitution of (\text{Exp} \rightarrow \text{Exp}).\(^4\) Second, the return type of bot methods, \text{Int}, must each be unified with the method it overload. This unification means that rather than the traversal of a Plus returning Plus, our best approximation is Exp.\(^5\) For our starting abstract type Exp, the results of all the subtype traversals unify to a final result of Exp.

For the sake of conciseness we have distilled our algorithm to a minimum, but there are other options when implementing our typing rules as described in another paper [5]. In our Java implementation we use side-effects to reduce duplicate calculations. We could have used monads in Haskell, but we elected for a simple functional approach. Central to handling recursive types is our capture of the argument types for unification. Within unify_args we have chosen to unify to the least upper bound of the argument types. We could also do a bit more work to reduce this restriction on functions and allow the arguments to accept more general types than those returned by recursive traversals. In practice we have not run into any problems where valid traversals fail to type check, so our approximation seems quite reasonable.

### 7. Method Coverage

Our traversal type checking algorithm determines the types that the traversal of a certain structure will return, but we are also interested in whether or not a traversal using a given function-class will raise a dispatch error. In order to verify this we must make sure that all possible concrete types, within our type checking approximation, are handled by the function-class. The two problems are certainly related. The traversal result types discovered in type checking provide an upper bound on the method signatures that must be covered in order to ensure completeness and guarantee a safe traversal for all possible data structure instances.

We refer to the abstract problem as leaf-covering in reference to type hierarchies as trees with abstract nodes and concrete leaves. In DemeterF, method coverage is confirmed after type checking, using the related methods for each concrete type, where dispatch takes place during traversal. After giving some background we describe the leaf-covering problem, present two different solutions, and analyze their running times.

### 7.1 Trees

In order to abstract the leaf-covering problem we view type hierarchies as trees of types where our extends relation, \text{\prec}, gives the successors of each type. For example, our expression CD from Figure 1 can be drawn as the tree shown in Figure 18.

![Fig 18. Exp Type Tree](image-url)
7.2 Graph Cartesian Products

A Graph Cartesian Product (GCP) is a useful metaphor when reasoning about (and visualizing) relationships between method signatures. We define the GCP, $G$, of a sequence of trees, $(T_1, \ldots, T_n)$, as a pair of vertices and edges, $G = (V, E)$ where:

$$V = \text{nodes}(T_1) \times \cdots \times \text{nodes}(T_n)$$
$$E = \{(\vec{s}, \vec{a}) \in V \times V \mid \vec{a} \prec \vec{s}\}$$

Our immediate successor relation, $\prec$, on signatures is defined as follows:

$$(a_1, \ldots, a_n) \prec (s_1, \ldots, s_n) \equiv$$
$$\exists i. a_i \prec s_i \land \forall j. j \neq i \implies a_j = s_j$$

Vertices of the graph are all possible signature permutations made from the types in each of the trees. Edges are formed between signatures that differ by just one element, where the different element in the target signature extends the corresponding element of the source signature. Reachability in the graph is defined by $\leq$ relation, and the leaves of the graph are signatures from the cross product of the leaves of the individual trees.

For example, part of the GCP defined by our Exp CD with roots/trees of $(\text{Plus}, \text{Exp}, \text{Exp})$ is shown in Figure 19.

![Figure 19. GCP for $(\text{Plus}, \text{Exp}, \text{Exp})$.](image)

The graph contains only a sampling of the vertices and edges, since the full graph contains approximately 50 vertices, and 80 edges. We use this more visual analogy to give an alternative definition of leaf-covering.

7.3 Leaf-Covering

Given a sequence of trees, $(T_1, \ldots, T_n)$, we say that a set of signatures, $S$, covers the given trees if $S$ contains an applicable signature for each signature in the cross-product of the leaves of the trees:

$$\text{covers}(S, T_1, \ldots, T_n) \equiv$$
$$\forall \vec{a} \in (\text{leaves}(T_1) \times \cdots \times \text{leaves}(T_n)). \exists \vec{s} \in S. \vec{a} \leq \vec{s}$$

We call this decision procedure the leaf-covering problem, and it shows up in a number situations involving multi-methods. Alternatively, we can define the leaf-covering problem in reference to the GCP. Given a sequence of trees, $(T_1, \ldots, T_n)$, and the implied GCP, we say that a set of selected vertices, $S$, covers the given trees if each leave of the GCP has an ancestor in $S$.

In the case of TBGP, the roots of the trees correspond to the approximate traversal return types, and the cross-product of the leaves corresponds to all possible concrete argument sequences, i.e., possible runtime types for dispatch. The set of signatures, $S$, represents the argument types of combine methods.

7.3.1 Example

For our example CD from Figure 9, our Test function class (Figure 10) fully covers the traversal. From type checking we know that when traversing an $A$ instance, the subtraversals return $B$s, so our signature to cover is $(A, B, B)$. Checking coverage is easy for Test, since this signature, i.e., the root of the GCP, is one of the function-class’ methods. If we remove this signature from Test, then the leaf signature $(A, D, D)$ is left uncovered. Note that if combine$(A, B, D)$ was part of our function-class then there is no need to have combine$(A, B, B)$, since it will never be called.

7.4 Brute-Force

Our task is to implement covers. The definition of the problem admits a straightforward solution: compute all the possible_leaf combinations, and check that each has an applicable signature, i.e., subtype returns True. Datatypes and helper functions are shown in Figure 20. In order to model type hierarchies as trees we define

```haskell
-- Trees of types (hierarchies)
data Tree = T Typ [Tree]
leaves :: Tree -> [Typ]
cross :: [[Typ]] -> [[Typ]]
```

![Figure 20. Leaf-Covering Setup](image)

a simple Tree datatype for arbitrarily branching trees of Typs. The function leaves returns all the leaves of a given Tree, and the function cross returns the cross product of a list of lists.

Figure 21 shows the straightforward encoding of our covers predicate into Haskell. We bind lfs to the cross product of the leaves of all trees, and a local function one that returns True if any of the signatures are applicable to the given leaf, 1. The body covers testss = let lfs = cross (map leaves ts)
            one l = (any (symmet l) ss) in all one lfs

![Figure 21. Brute-Force Solution](image)

of our let tests if all the leaves have one signature in the list. If so, then all leaf Sigs are covered.

7.4.1 Running Time

The brute force solution is easy to understand, but is very inefficient. If we use $t$ as a bound on the size of each tree, then our brute-force solution has running time:

$$\text{covers}(S, T_1, \ldots, T_n) \in O(|S| \cdot n \cdot t^e)$$

This solution is exponential in $n$, the number of trees (i.e., type hierarchies), even when the number of methods, $|S|$ is small. We have proven that the decision problem is coNP-complete by reducing DNF validity to leaf-covering [2], but there is another way to think about the problem.

7.5 Inclusion-Exclusion and Search

If we consider the leaves that each signature covers, then the leaves covered by $S$ is simply their union. We cannot efficiently calculate this union since it involves generating an exponential number of signatures. We can, however, calculate the size of the union efficiently. We do this by computing the number of leaves in the intersection of signatures directly from the trees. The intersection count(s) can be used to implement covers using the well-known inclusion-exclusion principle. Of course, knowing that a function-class does not cover all necessary cases is not as helpful to programmers as returning an uncovered signature.

We implement this function version of the problem with a more efficient implementation of the decision procedure using inclusion-exclusion. The key to our algorithm is a search procedure that is best described by analogy to the GCP. Figure 22 shows their types
of our helper functions. The function `roots` returns the root signature of a list of trees. If we consider the GCP for a list of Trees, then `succs` returns the list of immediate successors of the given signature, and `overlap` returns the number of leaves shared by all the given signatures. The implementation of inclusion/exclusion, `inclus_exclu`, takes a function that computes the number of overlapping leaves of the given list of signatures, and returns the size of their union.

Figure 23 shows our algorithm, `uncoveredSig`. The function is passed trees representing the hierarchies (i.e., $T_1, \ldots, T_n$) of traversal results and a list of signatures representing a function-class (i.e., $S$). If the given signatures cover all leaves, then the function returns `Nothing`, meaning no uncovered leaves. Otherwise it returns a signature wrapped in `Just`, which is the ancestor (in the GCP) of a group of uncovered leaves. The implementation

```haskell
uncoveredSig :: [Tree] -> [Sig] -> Maybe Sig
uncoveredSig ts ms = down (inex ms) (roots ts) where
  inex :: [Sig] -> Int
  inex = inclus_exclu (overlap ts)

down :: Int -> Sig -> Maybe Sig
down mCov toLeaf = let ss = succs ts toLeaf
  in if (null ss) then (Just toLeaf) else (across mCov ss)

across :: Int -> [Sig] -> Maybe Sig
across mCov [] = Nothing
across mCov (s:ss) = let cov = inex (s:ms)
  sCov = inex [s]
  in if (cov > mCov)
    then if (cov == mCov + sCov) then (Just s) else (down mCov s)
    else (across mCov ss)
```

Figure 23. Leaf-Covering Algorithm

is split into three functions. `inex` is a partial application of our helper functions, `inclus_exclu` and `overlap`, to the given trees. The mutually recursive functions `down` and `across` start at the root signature (`given by roots`) and search for a successor `Sig` that, when added to `ms`, covers more leaves than `ms` alone. The names of the functions draw refer to the GCP representation of signatures. In `down` we prepare to move `toLeaf` one step down the GCP. If there are no successors then we have a `Nothing` leaf, otherwise we search `across` the successor signatures. The body of the `let` in `across` compares the number of the covered leaves of various signature lists (`[s:ms]` and `[s]`). If the coverage of `s` with `ms` is the same as the coverage of `ms` (mCov) and `s`, then `s` is the root of completely uncovered leaves.

The algorithm works by starting at the `roots` of the trees, the `top` of the GCP. By definition the root signature together with `ms` must cover all leaves. If `ms` does not cover all the leaves by itself, then there exists a path from the root of the GCP such that every signature, `a`, on the path covers more leaves than `ms` alone. We simply follow this path until a leaf is found, or `a` and `ms` cover disjoint leaves, so `a` is the ancestor of only uncovered leaves.

7.5.1 Running Times
If we again use $t$ as a bound on the size of our trees, the our local inclusion-exclusion procedure, `inex`, has the following running time:

$$\text{inex}(S) \in O(t \cdot n \cdot 2^{\lvert S \rvert})$$

Where $(t \cdot n)$ represents the running time of the `overlap` calculation. Because the height and width of the GCP are bounded by $t \cdot n$
The total running time of our search `uncoveredSig` is:

$$\text{uncoveredSig}(T_1, \ldots, T_n, S) \in O(t^3 \cdot n^3 \cdot 2^{\lvert S \rvert})$$

Our second algorithm still runs in exponential time, but instead of depending on the number of trees, it is exponential in the number of signatures (the length of `ms`). Because both of our algorithms are only exponential in **part** of their input, they can both be termed as fixed parameter tractable. The first by fixing the number of trees, an the second by fixing the number of signatures. In practice the number of methods usually smaller than the number trees (the length of signatures).

8. Related Work
Our notion of generic programming is related to a number of functional programming approaches including generalized folds [24, 37], library and combinator approaches by Lämmel et al. [18, 19] and the *Scrap Your Boilerplate* series of papers [16, 17]. Our use of function-objects over a generic traversal is closest to Lämmel’s updatable fold algebras [20], where a function record is used to fold over datatypes. The use of extensible and generated function-classes allows us to emulate traditional generic programming [15, 23].

Like other generic programming implementations we can write functions over an encoding of a universal datatype that can be used to generate functions (like GenTU and GenTP), although our datatype representation is meant to better represent object-oriented class hierarchies. Updatable fold algebras provide similar extensible function records and some, but not all, of the typing flexibilities of ad hoc function-classes. There have been a few attempts to port datatype generic programming ideas to object oriented languages [28, 30]. The benefit of our approach is that classes do not require datatype encodings. CDs give us a concise way to express the structure of a hierarchy, over which we can write and/or generate functions. In comparison, one drawback is that safety must be checked from outside our implementation language, though it helps us simplify the system and requires a less complicated OO type system.

Our traversal flexibility comes mainly from our multiple dispatch. There has been much work on implementing multiple dispatch in various (usually OO) languages including Cecil [7], MultiJava [11], CLOS [38], and more recently JPred [25]. Making dispatch efficient has also been addressed, [1, 8] are a few approaches. Our implementation was developed independently but ended up being similar to an approach described by Chen et al. [10]. We make fewer assumptions about the efficiency of the individual dispatch steps, relying only on single subtype checks (e.g., `instanceof`), but their efficiency claims still apply.

Our typing rules, type checking, and method coverage algorithms are inspired by several papers ranging from aspect-oriented programming [40] to type polymorphism [27]. Our multiple dispatch and coverage checking is related to several static method type checking approaches [9, 26], and we draw on ideas from local type inference [32, 35]. Our method coverage algorithm is also likely applicable in cases of functional visitor frameworks where traversal is implemented outside the visitor’s control, e.g., [12, 29].
9. Conclusion

We have presented algorithms used in our approach to traversal-based generic programming (TBGP), and demonstrated implementations of each in Haskell. We discussed algorithms for generating polytypic functions; dynamic and static versions of our asymmetric, multiple dispatch; flexible type checking for function-classes over a generic traversal; and function-class coverage checking (leaf-covering). The algorithms themselves are applicable outside of our TBGP implementations, but in our case they help make DemeterF more useful by providing extensible function-classes, efficient by supporting static dispatch calculation, and safe by type checking traversals and verifying completeness.

References


