Algorithms for Traversal-Based Generic Programming

Bryan Chadwick Karl Lieberherr

College of Computer & Information Science
Northeastern University, 360 Huntington Avenue
Boston, Massachusetts 02115 USA.
{chadwick, lieber} @ ccs.neu.edu

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, which is more object-oriented friendly, that we call *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 its 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 specialized for arbitrary datatypes, discussed in Section 3. We use this technique to generate traditional polytypic functions like `Show`, as well as useful extensible function 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 entirely 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. We present an implementation of our custom unification algorithm that computes the return types of recursive traversals given a list of function signatures in Section 6.

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-*

WGP'10, September 26, 2010, Baltimore, Maryland, USA.

covering, which is *coNP-complete*. We demonstrate two different algorithms that compute a missing signature with running times that depend on the size of different parameters in Section 7. Bounding one of these parameters makes the corresponding solution tractable.

Our contributions are both practical and theoretical. On the practical side, we provide thorough descriptions of our algorithms and executable solutions written in Haskell. We describe 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 matching [10, 13] and unification [36]. On the theoretical side, we introduce 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 in Section 3. We present a more formal notation and Haskell definitions in Section 4. We then present the details of each algorithm/solution (Sections 5, 6, and 7); and give implementations in Haskell. We discuss related work in Section 8, and conclude in Section 9.

2. Background

What is traversal-based generic programming (TBGP)? The basic view is that it is a separation of structural recursion and functionality. There are several other techniques for doing this (*e.g.*, generalized folds [24, 37], Scrap Your Boilerplate (SYB) [16, 17], and traversal combinators [19, 39]), but we build on ideas from adaptive programming [22, 31]. Our approach is completely functional (*i.e.*, side-effect free[1]) and is conceptually similar to Lämmel's ideas of *updatable fold algebras* [20]. Our Java and C# implementations (collectively called DemeterF) include a class generator, data structures, 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 subclasses. Our CD definitions look similar to EBNF (including concrete syntax) with abstract and concrete classes differing only by the existence of subclasses. An example CD that represents simple expression structures is shown in Figure 1.

A class is defined by an identifier, *e.g.*, `Exp`, followed by an equal sign and a body, terminated by a period ("."). The body of a definition consists of a possibly empty, bar-separated ("|") sequence of subclass names, followed by a possibly empty sequence of field and syntax definitions. A field definition has a name enclosed in angle brackets ("<>") followed by its type, and concrete syntax is enclosed in double quotes. If the subclass list is non-empty then the definition is abstract. Subclass lists are placed in parentheses to distinguish them from shared fields. If the definition has no subclasses, then it is concrete. Parametric polymorphism (*i.e.*, generics) is supported, but is beyond the scope of this paper.

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

Figure 1. CD Example: expression structures

In Figure 1, we define an abstract class `Exp` with four immediate subclasses (`Int`, `Var`, *etc*.). `Int` is a concrete class with a single field named `v`, of type `int`. `Var` and `Def`, also concrete classes, represent variable uses and definitions, respectively. They make use of a DemeterF library class `ident` to represent identifiers. `Bin` is defined as the abstract superclass of `Plus` and `Pow`, which have common fields `l` and `r` of type `Exp`. `Plus` and `Pow` are defined only as concrete (prefix) syntax, inheriting common fields from `Bin`. Using DemeterF's class generator we can create the C# classes that this CD represents, and begin to write functions over them.

2.2 Function-Classes and Traversals

DemeterF includes a `Traversal` class that implements a generic walk over a data structure. The class is parametrized by a function-object that is responsible for folding together recursive traversal results. A function-object is an instance of a function-class (*i.e.*, a subclass of the DemeterF class `FC`) that contains specially named methods. In our Java and C# implementations we chose the name `combine`, to give an intuition of its intended use. As a first example, Figure 2 shows a function-class, `ToString`, that contains 6 combine methods: one for the library class `ident`, and one for each concrete subclass of `Exp`. The implicitly recursive function-class computes a `string` representation for an `Exp`, using prefix notation for binary expressions.

```
class ToString : FC{
  string combine(ident id){ return ""+id; }
  string combine(Int i, int v){ return ""+v; }
  string combine(Var v, string id){ return id; }
  string combine(Def d, string id, string e, string b)
  { return id+" = "+e+"; "+b; }
  string combine(Plus p, string l, string r)
  { return "+ "+l+" "+r; }
  string combine(Pow p, string l, string r)
  { return "^ "+l+" "+r; }

  string toStr(Exp e)
  { return new Traversal(this).traverse(e); }
}
```

Figure 2. Example: `Exp` to string conversion

The final method, named `toStr`, 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 `ToString` as fold functions over the structure. We can use our function-object by creating a new instance and calling `toStr`:

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

When the `traverse` method is called (inside `toStr`), the `Traversal` proceeds with a depth-first walk of the given instance, in this case an `Exp`.[2] Once the walk of an object's fields are com-

[1] This is not strictly enforced, but we do generate data structures with `readonly` (or `final`) fields.

[2] This version of the `Traversal` class uses C# reflection to both inspect the function-object for `combine` methods and `traverse` structures.

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 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.[3] 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.[4]

For example, we may want to write a function to count all the `int` values in a given `Exp`. We can extend the parametrized 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 1; }
}
```

Figure 3. Example: Counting the `int`s 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 `Exp`s.

```
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`. Since the `Def` is no longer needed, 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` objects and 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. 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 and efficient traversals from CDs, including `Show`, `TU`, and `TP`. In Section 4 we introduce notation, datatypes, headers, and functions necessary to describe our more detailed algorithms and their implementations in Haskell. 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.

[3] The library class `ident` has no visible fields.

[4] See Sections 3.2 and 3.3 for `TU` and `TP` implementation details.

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:

$$A = T_1 \mid \cdots \mid T_n$$

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 value constructors). We define the generation of the function-class `Show` (abstractly) as a function from concrete definitions to `combine` methods, using a template to describe the format of our resulting function-class. The template for `Show` is rather simple:

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

   // Generate the rest with GenShow
   ∀C ∈ CD . GenShow(C)
}
```

The template provides a class definition and `combine` methods for primitives that convert each into a **`string`**. The rest of the body for `Show` is generated by GenShow, using a simple rewrite rule mapped to each concrete definition from the CD:

$$\text{GenShow}(\ C = \langle f_1 \rangle\ T_1 \cdots \langle f_n \rangle\ T_n\ ) \rightsquigarrow$$
```
   string combine(C _h, string f1, ···, string fn)
   { return "C("+f1+","+···+","+fn+")"; }
```

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

3.2 Type Unifying Functions

`Show` is a special case of a more general function that is commonly associated with *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, `TU`, we provide a class that is parametrized by the eventual return type, `X`. Our template for `TU` follows:

```
class TU<X> : FC{
   // Methods to override
   abstract X fold(X a, X b);
   abstract X combine();

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

   // Generate the body with GenTU
   ∀C ∈ CD . GenTU(C)
}
```

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

$$\text{GenTU}(\ C = \langle f_1 \rangle\ T_1 \cdots \langle f_n \rangle\ T_n\ ) \rightsquigarrow$$
```
   X combine(C _h, X f1, ···, X fn)
   { return fold(f1, fold(f2, ···)); }
```

Each generated `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 `fold`. Figure 5 shows the resulting `TU` class, specialized for our `Exp` CD. The generated version of `TU` 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 inline traversals and dispatch [6].

```
class TU<X> : FC{
   /* ... */
   X combine(Def _h, X id, X e, X b)
   { return fold(id,fold(e,b)); }
   X combine(Plus _h, X l, X r)
   { return fold(l,r); }
   /* ... */
}
```

Figure 5. `TU` generated for `Exp`s

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.

```
class TP : FC{
   // Generate the body with GenTP
   ∀C ∈ CD . GenTP(C)
}
```

Our generation rule creates a `combine` method that simply reconstructs a new C instance from the recursive traversal results.

$$\text{GenTP}(\ C = \langle f_1 \rangle\ T_1 \cdots \langle f_n \rangle\ T_n\ ) \rightsquigarrow$$
```
   C combine(C _h, T1 f1, ···, Tn fn)
   { return new C(f1, ···, fn); }
```

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 *nearsighted*, since they do not look past the types of the recursive traversal results. They

```
class TP : FC{
  /* ... */
  Def combine(Def _h, ident id, Exp e, Exp b)
  { return new Def(id, e, b); }
  Plus combine(Plus _h, Exp l, Exp r)
  { return new Plus(l, r); }
  /* ... */
}
```

Figure 6. TP generated for Exps

do, however, rely on a generic traversal to adapt their `combine` methods to different structures. When we have a specific CD, the generic/reflective `Traversal` used in Section 2 can be replaced with a generated version that performs *much* better.

Our template for traversal generation is shown below.

```
class Traversal{
   FC fobj;
   Traversal(FC f){ fobj = f; }

   // Generate traversal methods
   ∀A ∈ CD . GenTrav(A)
   ∀C ∈ CD . GenTrav(C)
}
```

The generated `Traversal` class accepts a function-object, which will be used to fold together recursive results as before. Though only concrete classes exist at runtime, the body of the traversal uses the CD's `abstract` definitions to decide between subclasses. Our traversal generation rule, GenTrav, is shown below. First for abstract, then concrete definitions.

GenTrav($A = T_1 \mid \cdots \mid T_n$) $\rightsquigarrow$

```
R traverse<R>(A _h){
  if (_h is T1) return traverse<R>((T1)_h);
     ...
  if (_h is Tn) return traverse<R>((Tn)_h);
  throw new Exception("Unknown A Subtype");
}
```

For abstract classes we create a simple chain of `if` statements that selects the appropriate recursive `traverse` method for the given instance.[5] In order for the `Traversal` to work with different function-classes/objects, we parametrize each traversal method with the return type, R. For abstract types the parameter is carried through to recursive calls.

The generation rule for concrete definitions is bit more complex:

GenTrav($C = \langle f_1 \rangle\ T_1 \cdots \langle f_n \rangle\ T_n$) $\rightsquigarrow$

```
R traverse<R>(C _h){
  object f1 = traverse<object>(_h.f1);
     ...
  object fn = traverse<object>(_h.fn);
  return apply(fobj, new object[]{_h, f1, ..., fn});
}
```

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, 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` method determines the types of the arguments and dynamically dispatches to `fobj`'s most specific `combine` method.

[5] Like Java, C# will statically resolve the overloaded `traverse` calls because of casting.

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 compete 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 be sure they are understandable, even to non-Haskell wizards.[6]

4.1 Types

We model types as symbols. Again we use a simple 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, $\leq$, is built from the relationship given by abstract classes from a CD, which we write $\prec$ and call *extends*:

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

We write its transitive closure $<$, defined as:

$$B < D \equiv B \prec D \vee \exists C \,.\, C < D \wedge B < C$$

Finally, $\leq$ is defined as the reflexive extension of $<$:

$$C \leq B \equiv C = B \vee C < B$$

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

$$C \bowtie C' \equiv C \leq C' \vee C' \leq C$$

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:

$$\vec{s} = (s_1, \ldots, s_n)$$

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

$$S = \{\, \vec{s}_1, \ldots, \vec{s}_n \}$$

[6] The authors are far from being Haskell wizards, so explanations may be (overly) thorough.

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

We extend our subtype relation, $\leq$, to signatures of equal length, say n:

$$\vec{s} \leq \vec{u} \;\equiv\; |\vec{s}| = |\vec{u}| = n \;\wedge\; \forall i \in [1..n] \;\; s_i \leq u_i$$

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, $\sqsubseteq$:

$$\vec{s} \sqsubseteq \vec{u} \;\equiv \vec{s} \leq \vec{u} \wedge \\ \exists i \in [1..n] \,.\, (s_i < u_i) \;\wedge\; (\forall k \in [1..i\text{-}1] \,.\, s_k = u_k)$$

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 `Typ`s. 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 `Typ`s.

```
-- 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` applies our predicate pair-wise to elements from different lists. The function `asymmet` recursively

```
-- 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

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.

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

```
A = <l> B <r> B
B = (C | D).
C = .
D = .
```

Figure 9. Example CD for Dispatch/Type Checking

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`.

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

   C test(A a)
   { return new Traversal(this).traverse(a); }
}
```

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, when given an approximation of the argument/traversal types. These correspond to our *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

```
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)

best :: Sig -> Sig -> [Sig] -> Sig
best s c [] = s
best s c (a:as) = if (symmet c a) && (asymmet a s)
                  then (best a c as)
                  else (best s 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 `symmet`) is found. Again we use patterns to match empty list, `[]`, and non-empty list, `(a:as)`. 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. Because our method selection is *asymmetric*, traversal results of `["A","C","C"]` would also cause the second `combine` to be selected.

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 which methods might possibly be called *before* the traversal, 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 number of residual "*instance*" checks left for runtime. Figure 12 shows Haskell datatypes and functions that calculate dispatch residue. We do this by constructing a decision tree, `Dec`, of type tests. An `(IF` i t d_1 d_2`)` value represents an instance check: if the i^{th} parameter is an instance (or a subtype) of the type t, then the decision continues with the left decision tree, d_1, otherwise it continues with d_2. A `(CALL` $\bar{s}$`)` 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`.[7] 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`$^{\text{th}}$ parameter `Typs`. The infix list operator `!!` returns the `i`$^{\text{th}}$ 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 `Typs` 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`$^{\text{th}}$ argument type. In the outer comprehension we take the pair of the `Typ`, `t`, and the types that can be ignored, `ignr`, and place `t` as the left element of a pair. In the right of each pair we place the list of method signatures that are still `related`, an whose `i`$^{\text{th}}$ arguments are not subtypes of the ignored `typs`. The intuition is that related types, both subtypes and supertypes, are considered during dispatch: the former for matching more specific types (*e.g.*, `Var` rather than `Exp`), the latter for more abstract cases (*e.g.*, `Exp` when/if no other subtype matches). We will eventually build `If` decisions, so the ignored types represent the `Typ` tests that have failed within a chain of instance checks, so redundant tests can be eliminated.

The groups of methods are passed to `buildDec`, which constructs the nested `IF`s to decide between the groups based on the type of the `i`$^{\text{th}}$ 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.

```
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 (sortBy sub_ord 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, [Sig])] -> 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 i t d (buildDec i s gs))
```

Figure 12. Residual Selection Algorithm

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 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`, since `D` $\not\leq$ `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"]))
         (IF 2 "C" (CALL ["A","B","C"])
                   (CALL ["A","B","B"]))
```

Figure 13. Residual Selection Algorithm

[7] This is a very weak check, since the algorithm only chooses from the given methods. Eliminating '*incompleteness*' errors is discussed in Section 7.

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 can be used with 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 `residue` to compute dispatch decisions. These decisions replace `apply` in the generated traversals for concrete types from Section 3.4.

The decision result from `residue` 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 residue 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 representations 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 (`Abst` and `Concr`) definitions, each with a name and a list of `Typ`s. We will use the function `finddef` to lookup a type's definition in a `CD` and `common_super` to determine the closest common supertype of two types, sometimes referred to as their *least upper bound* (LUB). A `Meth` is a pair[8] of a `Sig` and a return `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 recursive types with related 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 in the given `Subst`. The final

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

```
-- Model Simple Class Dictionaries
type CD = [Def]
data Def = Abst Typ [Typ]
         | Concr Typ [Typ]

finddef :: CD -> Typ -> Def
common_super :: Typ -> Typ -> Typ

-- Methods, Return Types, and Substitutions
type Meth = (Sig,Typ)
type Subst = [(String,Typ)]
data RTyp = TVar String | UTyp Typ

-- Apply a Substitution to an RTyp
subst :: Subst -> RTyp -> Typ

-- Find Common Super and Merge Substitutions
lub_rets :: [(RTyp,Subst)] -> (RTyp,Subst)
```

Figure 14. Setup: Helpers and Datatypes

function `lub_rets` combines a list of `(RTyp,Subst)` pairs, finding the `common_super` for the `RTyp` and any duplicate bindings within the substitutions.

We begin our algorithm by showing the extension of `rel_sig` to `RTyp`s, 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

```
rel_rsig :: [(RTyp,Subst)] -> Sig -> Bool
rel_rsig (r:rs) (t:ts) =
   (rel_rsig rs ts) && case (fst r) of
                         (TVar n) -> True
                         (UTyp n) -> (related n t)
rel_rsig [] [] = True
rel_rsig _ _ = False
```

Figure 15. Related Signatures with `RTyp`

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

```
unify_args :: [(RTyp,Subst)] -> [Typ] -> Subst
unify_args [] [] = []
unify_args ((r,sub):rs) (t:ts) =
  let nsub = (foldr lub_subst sub (unify_args rs ts))
  in case r of
       (TVar n) -> lub_subst (n,t) nsub
       (UTyp n) -> nsub

lub_subst :: (String,Typ) -> Subst -> Subst
lub_subst (n,t) [] = [(n,t)]
lub_subst (n,t) ((np,tp):ss) =
  if (n == np) then (n,(common_super t tp)):ss
  else (np,tp):(lub_subst (n,t) ss)
```

Figure 16. Unification and Substitutions

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 `TVar`s 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).

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 `Typs`, 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.

```
typecheck :: CD-> [Meth]-> [Typ]-> Typ-> (RTyp,Subst)
typecheck cd fc rec start =
  if (elem start rec) then (TVar start,[])
  else (checkdef (finddef cd start)) where
    checkdef :: Def -> (RTyp,Subst)
    checkdef (Abst n sts) =
      lub_rets (map (typecheck cd fc (n:rec)) sts)
    checkdef (Concr n ts) =
      let fldts = (map (typecheck cd fc rec) ts)
          argts = (UTyp n,[]):fldts
          relms = filter ((rel_rsig argts) . fst) fc
          rettyp m = (UTyp (snd m),
                      unify_args argts (fst m))
      in lub_rets (map rettyp 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 of type checking each of the subtypes using similar arguments, but placing the current type at the head of the recursive list. If the `Def` is concrete, we `map` our `typecheck` function to determine the return types (and substitutions) for its field types. We then add the current type as a prefix, and `filter` out unrelated signatures. The function `fst`, which returns the left of a pair, is composed (`.`) with the partial application of `rel_rsig`. We `map` a local function `rettyp` 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.

6.2 Example and Discussion

Our `Test` 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 `D` 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 to 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`s (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 `Int` is easily unified with the constraints of `Exp` from other signatures, resulting in a substitution of (`Exp` $\rightarrow$ `Exp`).[9] Second, the return type of bot methods, `Int`, must each be unified with the method it overloads. This unification means that rather than the traversal of a `Plus` returning `Plus`, our best approximation is `Exp`.[10] 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, which are 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 ever 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 viewing 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 the traversal dispatches to a `combine` method. 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, $\prec$, gives the successors of each type. For example, our expression CD from Figure 1 can be drawn as the tree shown in Figure 18.

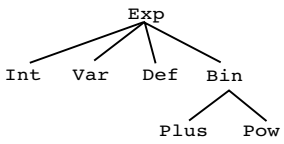

Figure 18. `Exp` Type Tree

We will use the type name, say T, to refer to the tree with T as its root. The function `nodes` returns the set (or a list) of the types in a tree:

$$\texttt{nodes}(T) \equiv \{\, T' \mid T' \leq T \,\}$$

We also define `succs`, which returns the set of a tree's immediate successors:

$$\texttt{succs}(T) \equiv \{\, T' \mid T' \prec T \,\}$$

Finally, we define the function `leaves`, which returns the leaves of a tree:

$$\texttt{leaves}(T) \equiv \{\, T' \in \texttt{nodes}(T) \mid \texttt{succs}(T') = \emptyset \,\}$$

Using our `Exp` hierarchy as an example, `nodes(Exp)` would return the set of all user-defined types, and other function returns would be as follows:

$$\texttt{succs(Exp)} = \{\, \texttt{Int}, \texttt{Var}, \texttt{Def}, \texttt{Bin} \,\}$$
$$\texttt{leaves(Exp)} = \{\, \texttt{Int}, \texttt{Var}, \texttt{Def}, \texttt{Plus}, \texttt{Pow} \,\}$$

[9] In contrast, for our `CountInts` function-class the resulting substitution is (`Exp` $\rightarrow$ `int`)

[10] We could possibly do better if we use a notion of *unions*, resulting in a `Plus` traversal returning the type (`Plus` $\cup$ `Int`).

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:

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

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

$$(a_1, \ldots, a_n) \prec (s_1, \ldots, s_n) \equiv$$
$$\exists i \,.\, a_i \prec s_i \;\wedge\; \forall j \in [1..n] \,.\, j \neq i \implies a_j = s_j$$

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 our $\leq$ 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 root/trees of `(Plus, Exp, Exp)` is shown in Figure 19.

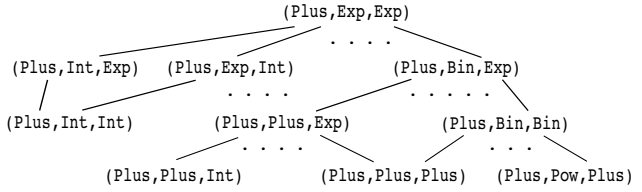

Figure 19. GCP for `(Plus, Exp, Exp)`

The figure 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 in Section **??**.

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:

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

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, D, 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

```
-- Trees of types (hierarchies)
data Tree = T Typ [Tree]

leaves :: Tree -> [Typ]
cross :: [[Typ]] -> [[Typ]]
```

Figure 20. Leaf-Covering Setup

a simple `Tree` datatype for arbitrarily branching trees of `Typ`s. 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, `l`. The body

```
covers :: [Tree] -> [Sig] -> Bool
covers ts ss = let lfs = cross (map leaves ts)
                   one l = (any (symmet l) ss)
               in all one lfs
```

Figure 21. Brute-Force Solution

of our `let` tests if `all` the leaves have `one` signature in the list. If so, then all leaf `Sig`s 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:

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

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 without generating the leaf signatures. We do this by computing the number of leaves in the *intersection* of two or more signatures directly from the trees. The number of intersection leaf signatures can be used to implement `covers` with the help of 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 that they should add to their function-class. We implement this *function* version of the problem with an implementation of the decision procedure using inclusion-exclusion. The key to our algorithm is a search that is best described by analogy to the GCP.

```
roots :: [Tree] -> Sig
succs :: [Tree] -> Sig -> [Sig]
overlap :: [Tree] -> [Sig] -> Int

inclu_exclu :: ([Sig] -> Int) -> [Sig] -> Int
```

Figure 22. Inclusion/Exclusion Solution Helpers

Figure 22 shows the 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 `Tree`s, 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, `inclu_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

```
uncoveredSig :: [Tree] -> [Sig] -> Maybe Sig
uncoveredSig ts ms = down (inex ms) (roots ts) where
    inex :: [Sig] -> Int
    inex = inclu_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, `inclu_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 refer to the GCP representation, *e.g.*, Figure 19. In `down` we prepare to move `toLeaf` one step down the GCP. If there are no successors then we have an uncovered 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`, *i.e.*, `mCov`, and `s` separately, then `s` is the root of a number of 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, `s`, on the path covers more leaves than `ms` alone. We simply follow this path until a leaf is found, or `s` and `ms` cover disjoint leaves, so `s` 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, then our local inclusion-exclusion procedure, `inex`, has the following running time:

$$\texttt{inex}(S) \in O(t \cdot n \cdot 2^{|S|})$$

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

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

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 becomes tractable by fixing the number of trees, and the second by fixing the number of signatures. In practice the number of methods is usually smaller than the number trees, *i.e.*, the length of the 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], but they do not handle classes directly, requiring datatype instances to be encoded as well. 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 multi-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 and traversals; 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, more efficient by supporting static dispatch calculation, and safe by type checking traversals and verifying completeness.

In the future we hope to be able to formalize our other features, including traversal arguments/contexts and control/strategies. Because our traversals are functional, we also hope to be able to increase performance with implicitly parallel traversals.

References

[1] E. Amiel, O. Gruber, and E. Simon. Optimizing multi-method dispatch using compressed dispatch tables. In *OOPSLA '94*, pages 244–258, New York, NY, USA, 1994. ACM.

[2] B. Chadwick. Algorithms in DemeterF. `http://www.ccs.neu.edu/home/chadwick/files/algo.pdf`, May 2009.

[3] B. Chadwick. apf-lib: Traversal-based generic programming in Scheme. Website, 2010. `http://www.ccs.neu.edu/home/chadwick/demeterf/apf-lib/`.

[4] B. Chadwick. DemeterF: Functional adaptive programming tools. Website, 2010. `http://www.ccs.neu.edu/home/chadwick/demeterf/`.

[5] B. Chadwick and K. Lieberherr. Functional Traversal-Based Generic Programming. Submitted to *Higher-Order and Symbolic Computation*, Festschrift for Mitch Wand `http://www.ccs.neu.edu/home/chadwick/files/mitchfest.pdf`.

[6] B. Chadwick and K. Lieberherr. Weaving generic programming and traversal performance. In *AOSD '10*, pages 61–72, New York, NY, USA, 2010. ACM.

[7] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP '92*, pages 33–56. Springer-Verlag, 1992.

[8] C. Chambers and W. Chen. Efficient multiple and predicated dispatching. In *OOPSLA '99*, pages 238–255, New York, NY, USA, 1999. ACM.

[9] C. Chambers and G. T. Leavens. Typechecking and modules for multi-methods. *ACM Ton Programming Languages and Systems*, 17(6):805–843, November 1995.

[10] W. Chen, V. Turau, and W. Klas. Efficient dynamic look-up strategy for multi-methods. In *ECOOP '94*, pages 408–431, London, UK, 1994. Springer-Verlag.

[11] C. Clifton, G. T. Leavens, C. Chambers, and T. D. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA*, pages 130–145, 2000.

[12] B. C. d. S. Oliveira, M. Wang, and J. Gibbons. The visitor pattern as a reusable, generic, type-safe component. In G. Kiczales, editor, *OOPSLA '08*, October 2008.

[13] A. Gräf. Left-to-right tree pattern matching. In *RTA '91: Rewriting Techniques and Applications*, pages 323–334, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[14] R. Hinze. A new approach to generic functional programming. In *POPL '99*, pages 119–132. ACM Press, 1999.

[15] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *ACM Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[16] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI '03*, volume 38, pages 26–37. ACM Press, March 2003.

[17] R. Lämmel and S. Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *ICFP '04*, pages 244–255. ACM Press, 2004.

[18] R. Lämmel, E. Visser, and J. Visser. Strategic programming meets adaptive programming. In *AOSD '03*, pages 168–177, New York, NY, USA, 2003. ACM Press.

[19] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proc. Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, Jan. 2002.

[20] R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In J. Jeuring, editor, *Proceedings of WGP'2000, Technical Report, Universiteit Utrecht*, pages 46–59, July 2000.

[21] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. 616 pages, ISBN 0-534-94602-X.

[22] K. J. Lieberherr, B. Patt-Shamir, and D. Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004.

[23] A. Loeh, J. J. (editors); Dave Clarke, R. Hinze, A. Rodriguez, and J. de Wit. Generic Haskell user's guide – version 1.42 (Coral). Technical Report UU-CS-2005-004, Department of Information and Computing Sciences, Utrecht University, 2005.

[24] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA '91*, pages 124–144, London, UK, 1991. Springer-Verlag.

[25] T. Millstein, C. Frost, J. Ryder, and A. Warth. Expressive and modular predicate dispatch for Java. *ACM Trans. Program. Lang. Syst.*, 31(2):1–54, 2009.

[26] T. D. Millstein and C. Chambers. Modular statically typed multimethods. In *ECOOP '99*, pages 279–303, London, UK, 1999. Springer-Verlag.

[27] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[28] A. Moors, F. Piessens, and W. Joosen. An object-oriented approach to datatype-generic programming. In *WGP '06*, pages 96–106, New York, NY, USA, 2006. ACM.

[29] B. C. Oliveira. Modular visitor components. In *ECOOP '09*, pages 269–293, Berlin, Heidelberg, 2009. Springer-Verlag.

[30] B. C. Oliveira and J. Gibbons. Scala for generic programmers. In *WGP '08*, pages 25–36, New York, NY, USA, 2008. ACM.

[31] D. Orleans and K. J. Lieberherr. DJ: Dynamic adaptive programming in Java. In *Reflection '01*, Kyoto, Japan, September 2001. Springer Verlag. 8 pages.

[32] J. Palsberg and M. I. Schwartzbach. Static typing for object-oriented programming. *Sci. Comput. Program.*, 23(1):19–53, 1994.

[33] C. H. Papadimitriou. *Computational Complexity*, chapter 10, section 10.3. Addison Wesley, December 1993.

[34] S. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003.

[35] B. C. Pierce and D. N. Turner. Local type inference. In *POPL '98*, pages 252–265, New York, NY, USA, 1998. ACM.

[36] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[37] T. Sheard and L. Fegaras. A fold for all seasons. In *FPCA '93*, pages 233–242, New York, NY, USA, 1993. ACM.

[38] G. L. Steele, Jr. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990.

[39] J. Visser. Visitor combination and traversal control. In *OOPSLA '01*, pages 270–282, New York, NY, USA, 2001. ACM.

[40] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *ICFP '03*, pages 127–139, New York, NY, USA, 2003. ACM.