Noname manuscript No.
(will be inserted by the editor)

A Functional Approach to Generic Programming using Adaptive Traversals

Bryan Chadwick · Karl Lieberherr

the date of receipt and acceptance should be inserted later

Abstract Writing functions over complex user-defined datatypes can be tedious and error prone. Generic (or polytypic) programming and higher order functions like `foldr` have resolved some of these issues, but can be too general to be practically useful for larger collections of data types. In this paper we present a traversal-based approach to generic programming using function sets. Our traversal is an adaptive, higher-order function that employs an asymmetric type-based multiple dispatch to fold over arbitrarily complex structures. We introduce our approach in the context of our Scheme library implementation, present a typed model of our system, and provide a proof of type soundness, showing that our flexible, adaptive approach is both useful and safe.

Keywords Traversals · Functional Programming · Data Structures · Generic Programming

1 Introduction

Writing functions over complex user-defined data structures is tedious and error prone, but often unavoidable. Using built-in, higher order, structurally recursive functions such as `map` and `foldr` can alleviate some of this tedium for commonly used data structures, but these are of little use for more complex datatypes. Generalized folds [28,31] were introduced to provide a blueprint for fold abstractions over user-defined datatypes by describing how such functions can be written (or generated) from data definitions. Polytypic programming [16,27,14,3] provides a similar service, allowing programmers to define functions that are applicable to *all* datatypes. Neither of these approaches is a perfect solution for user-defined structures and functions.

Generalized folds provide recursion operators that replace value constructors, like `cons`, with user provided functions. With more complex datatypes containing multiple

Bryan Chadwick
Northeastern University
E-mail: chadwick@ccs.neu.edu

Karl Lieberherr
Northeastern University
E-mail: lieber@ccs.neu.edu

or mutually recursive types, the number and order of functions that must be passed can quickly become overwhelming. On the other hand, polytypic functions work for all datatypes, usually by defining the function over a universal recursive datatype consisting of binary sum and product types. The universal nature of these definitions makes writing functions that operate on high level datatype notions impossible.

As an example, consider a typical implementation of `foldr` over *proper* lists, written in Scheme [18][1]:

```
;; Fold list elements, to the right
(define (foldr func base lst)
  (if (null? lst) base
      (func (car lst)
            (foldr func base (cdr lst)))))
```

One obvious use of this function might be to `sum` a list of numbers:

```
;; Sum elements in a list-of numbers
(define (sum lon)
  (foldr + 0 lon))
```

The function contains no explicit recursion, but as we add new datatypes or attempt to implement more complex functions `foldr` quickly becomes difficult to use or obsolete. Polytypic programming provides a means to implement fold-like functions over more classes of data with a single definition. Consider the implementation of a sum function over arbitrary structures in Generic Haskell [27]:

```
-- Sum integers in any data structure
deepsum {| t |} :: t -> Int
deepsum {|Int|} i = i
deepsum {|Unit|} u = 0
deepsum {|Sum a b|} (Inl aa) = deepsum {|a|} aa
deepsum {|Sum a b|} (Inr bb) = deepsum {|b|} bb
deepsum {|Prod a b|} (aa :*: bb) = ((deepsum {|a|} aa) +
                                    (deepsum {|b|} bb))
```

The generic function `deepsum` is defined by cases on a universal datatype: base types (*e.g.*, `Int` and `Unit`) and binary `Sum` and `Prod` types. Since any Haskell datatype can be encoded as binary sums and products, `deepsum` can be called on any datatype. However, unlike our use of `foldr`, structural recursion is no longer implicit and programmers must reason about datatypes at a much lower level. This low level reasoning limits the functions that can be written and the types of results they can return.

In this paper we present a traversal-based approach to generic programming that uses sets of functions to fold over a data structure. We propose the use of a generic, adaptive traversal function that walks a structure guided by control declarations. The traversal folds recursive results by selecting from a *set* of user-defined functions using a type-based multiple dispatch. Our traversal provides a flexible, adaptive form of generic programming that can be specialized using programmer-defined functions, control, and contexts.

Getting back to our example, using the Scheme implementation of our approach we can define a function to sum all the numbers in a possibly nested and possibly improper (but non-circular) list:

[1] In later examples we will use both parentheses `()` and square brackets `[]` within Scheme code for readability.

```
;; Sum the elements of nested lists of numbers
(define (deep-sum lst)
  (traverse lst
      (funcset [(number) (n) n]
               [(empty) (e) 0]
               [(cons number number) (c n m) (+ n m)])))
```

The `funcset` form builds a set from a list of typed functions. Each declares its formal argument types, followed by argument names and a body expression. The symbols `number`, `empty`, and `cons` represent the types of Scheme numbers, the empty list, and non-empty lists respectively. The `traverse` form is used to recursively traverse the structure of a given value. During the walk it combines recursive results by selecting and applying a matching function from the given set based on the formal and actual argument types. In this case, when a `number` or the `empty` list is reached, the traversal selects and applies the first or second function from the set. When a `cons` is reached with `number`s as recursive results from its `car` and `cdr`, then the third function is applied. Any other case, *e.g.*, a `symbol`, causes a runtime/dispatch error.

Our `deep-sum` function is considered *generic* because the traversal automatically adapts to different data structures (here different list depths) and applies elements from the given set of functions. Is this function safe, in the sense that the *traversal* will never induce a runtime error? The answer is yes, as long as the given value is constructed only of lists (*i.e.*, `cons` or `()`) and numbers. More precisely, the function set safely handles Scheme values that can be finitely derived by the following grammar:

List ::= `(cons` Any Any`)` | `()`
Any ::= List | Number

Supporting this style of traversal-based generic programming while providing guarantees of dispatch safety is the topic of this paper. Our contributions can be summarized as follows:

- We present a new, flexible approach to traversal-based generic programming that uses sets of functions and an adaptive traversal to recursively fold over structures (Section 3). Our approach is called *functional adaptive programming* (or AP-F), and is implemented as a library and collection of macros in PLT Scheme [1]. It is useful for implementing generic functions over complex data hierarchies and uses an asymmetric multiple dispatch that avoids ambiguities and supports function overloading. AP-F improves on other generic approaches like generalized folds [31, 28] and Scrap Your Boilerplate (SYB) [21, 22] by supporting extensible functions, traversal control, contexts, and more flexible traversal return types.
- We introduce a corresponding model, semantics, and type system (Sections 4 and 5) that describe the essential features of AP-F: data structures, traversals, and dispatch using functions sets. Our model datatypes are an extension of sum-of-products and function sets are like multi-entry closures, similar to function objects in object-oriented languages. Our type system is more flexible than other approaches and confirms the safe application of a function set over the traversal of a data structure.
- We give a proof of type soundness for our model (Section 6), showing that type correct traversals and function sets can be executed without runtime dispatch errors. Soundness relies on a special case of multi-method checking, that we refer to as the *leaf-covering problem*, allowing function overloading while maintaining safe and complete dispatch. Proving soundness may seem superfluous for a library in

a dynamically typed language like Scheme, but the results presented here are also applicable to our other implementations of traversal-based generic programming in Java and C#.

Our goal is to provide a safe form of extensible functions over traversals while maintaining the flexibility and safety of separate, hand-coded functions. We take strides in this direction by demonstrating a useful implementation of our approach, modeling its essential features, giving it a type system, and proving it sound. The main benefit of our approach can be characterized as *separation of concerns*, but additional benefits can be seen in our other AP-F implementations, including implicit parallelism, domain specific control strategies, and extensive support for parametrized types and data structures.

2 Background

Before going into the details of our approach, we begin by describing some programming problems with functions and data structures in more detail. To illustrate a common situation, consider data structures representing a simple boolean expression language with literals, negation, conjunction, and disjunction. Simple Scheme structures with comments describing their intended uses are shown below.

```
;; A BExp is one of Lit, Neg, ...
;; A Lit is one of True or False
;; (make-True)
(define-struct True ())
;; (make-False)
(define-struct False ())
```

```
;; (make-Neg BExp)
(define-struct Neg (inner))
;; (make-And BExp BExp)
(define-struct And (left right))
;; (make-Or BExp BExp)
(define-struct Or (left right))
```

The structures themselves are only useful for creating and describing data, but we can implement a typical evaluation function, `eval-bexp`, that reduces the representation of a boolean expression to a Scheme value, `#t` or `#f`.

```
;; eval-bexp: BExp -> boolean
(define (eval-bexp e)
  (cond [(True? e) #t]
        [(False? e) #f]
        [(Neg? e) (not (eval-bexp (Neg-inner e)))]
        [(And? e) (and (eval-bexp (And-left e))
                       (eval-bexp (And-right e)))]
        [(Or? e) (or (eval-bexp (Or-left e))
                     (eval-bexp (Or-right e)))]))
```

As a convention, we precede all function definitions with a comment that describes the classes of values that the function expects as parameters and returns as a result. In this case `eval-bexp` accepts a `BExp` and returns a `boolean`. Our implementation is about as concise as it can be in Scheme when written as a single function. The function's recursion is explicit and, apart from the short-cutting evaluation of `and` and `or` in Scheme, it mimics our structural definitions exactly.

Polytypic programming cannot directly help us define this particular function, since the idea of evaluation does not make sense for *all* possible types. Instead, there has been much work on abstracting these kinds of functions by creating structural recursion operators, typically called *folds* [31,28]. Using our data definitions as a guide, we can refactor the implementation of `eval-bexp` by creating a higher-order function for folding `BExp`s into a different structure.

```
;; fold-bexp: BExp A A (A -> A) (A A -> A) (A A -> A) -> A
(define (fold-bexp e tru fals nott andd orr)
  (cond [(True? e) tru]
        [(False? e) fals]
        [(Neg? e) (nott (fold-bexp (Neg-inner e)
                                   tru fals nott andd orr))]
        [(And? e) (andd (fold-bexp (And-left e)
                                   tru fals nott andd orr)
                        (fold-bexp (And-right e)
                                   tru fals nott andd orr))]
        ...))
```

The comment preceding `fold-bexp` describes its signature. The function accepts five arguments, one for each structure definition (*concrete* variants of `BExp`). We use `A` as a place-holder for the return type of our function, since it should be the same throughout. The individual functions passed to `fold-bexp` match the arity of the corresponding constructors, using values instead of zero-argument functions. For each case we replace the original constructor by calling the corresponding function with the results of recursively folding the immediate fields of the structure.

Because `fold-bexp` works at the level of constructors, we can use it to give a more succinct definition of `eval-bexp`, without mentioning any structural recursion.

```
;; Wrappers for Scheme and/or
(define (and-f a b) (and a b))
(define (or-f a b) (or a b))

;; eval-bexp: BExp -> boolean
(define (eval-bexp e)
  (fold-bexp e #t #f not and-f or-f))
```

We have reduced evaluation to a one line function and can now use our fold to write other functions that match this structural pattern. Essentially, we have abstracted the *traversal* of a data structure from the most interesting parts of our function.

The general fold approach is manageable for smaller data structures, but a number of questions/concerns remain:

1. Who writes the fold function? Hand writing folds for different data structure is still tedious and difficult. We could probably implement it as a specialized macro, but mutually recursive structures can complicate things.
2. What about parametrized structures like lists? Should we accept a function to recur into the parameter, or just do a shallow fold? These interpretations correspond to *Church* and *Scott* encodings [33] of datatypes, respectively.
3. What about mutually recursive structures? Should the fold implement the recursion, or just treat them as parametrized structures?
4. Do we really have to pass all those functions? With many constructors performance can suffer, and the order and number of functions quickly becomes overwhelming. There are techniques to eliminate parameter passing for internal recursion [12], but the interface of the original fold function must still accept all necessary parameters.
5. Can we abstract over multiple constructor cases with a single function?
6. Are we limited to folding to a single class of values? Is it safe to do otherwise?

In answering these questions we have completely reformulated the notion of Adaptive Programming (AP) [26] to provide a safe, flexible approach to traversal-based generic programming in a functional setting, we call it *functional* adaptive programming (AP-F). Our approach groups functions into *sets*, which are applied over a data

structure by a generic traversal. The traversal uses a type-based multiple dispatch to support case abstraction (allowing a single function to be called in-place of multiple constructors) and overloading (allowing a more specific function to override a more general one in special cases). Because dispatch is separate, function sets are easy to combine/extend. The separate traversal allows us to easily control/limit our recursion, for efficiency or algorithm correctness. Our approach can emulate generalized folds by providing a function for each (concrete) constructor, and polytypic programming by generating extensible function sets.

In the rest of this paper we discuss our new style of traversal-based programming. In Section 3 we describe our library implementation with a number of increasingly complex examples. We then step back and model its essential features in Section 4 by providing minimal syntax and semantics. In order to perform meaningful type checking of AP-F programs in both dynamically typed (like Scheme) and statically typed languages (like Java), we provide a type system (Section 5) give a proof of type soundness (Section 6), that shows that type correct traversals do not induce runtime dispatch errors. We discuss related work in Section 7 and conclude in Section 8 with a mention of future work.

3 Traversal-Based Generic Programming

Our AP-F Scheme library provides macros and functions for defining and using structures, unions/variants, functions sets, and different forms of traversal. In this section we thoroughly introduce its main features and syntax. It is worth noting that in this paper we focus on immutable data structures and will not be concerned with the redefinition of structures and/or functions. In particular, we will consider only non-cyclic structures and will silently allow overridden definitions.

3.1 Data Structures

Traditional traversals are based solely on structural recursion, and AP-F is not much different. In order to traverse a structure we need a description to guide our recursion. Our system supports basic Scheme datatypes, namely `booleans`, `numbers`, `symbols`, `strings`, `chars`, and `lists`. While others are primitive, `boolean` and `list` types are unions of `true` and `false` (the types of `#t` and `#f`), and `empty` and `cons` (the types of empty and non-empty lists), respectively. Using these built-in datatypes as a base, AP-F allows programmers to define more complex data structures and unions.

Our library's syntax for data definitions is described below in EBNF notation. Concrete syntax is surrounded in double quotes and ID is used to represent Scheme identifiers. We view data structures as either `concrete`, describing the structure of values, or `abstract`, describing named unions that provide groupings and subtypes, allowing programmers to abstract over multiple types.

Definition ::= Concrete | Abstract
Concrete ::= `(concrete` ID `[` FieldDef* `])`
FieldDef ::= `(` ID ID `)`
Abstract ::= `(abstract` ID `[` ID$^+$ `])`

Programmers can use `concrete` definitions to introduce new concrete structures (similar to Scheme's `define-struct`), which are considered distinct types. A concrete type

is defined as a list of fields, each with a name and a type. Similarly abstract types are introduced by **abstract** definitions, which declare subtype relationships. For example, a typical definition of *binary-trees* can be described as follows:

```
(abstract Tree [Node Leaf])
(concrete Node [(left Tree) (right Tree)])
(concrete Leaf [(item number)])
```

The first line defines the union `Tree`, with elements `Node` and `Leaf`, and the second and third lines define the concrete structure of `Node` and `Leaf` instances. Based on these definitions, we refer to `Node` and `Leaf` as *subtypes* of `Tree`.

Abstract definitions are viewed as true unions in the sense that they can be defined over otherwise unrelated types (including other abstract types). We can, for example, define the union of all atomic datatypes:

```
(abstract atom [boolean symbol number string char])
(atom? #\space) ;; -> #t
(atom? '(5 7))  ;; -> #f
```

Or the union of all built-in datatypes:

```
(abstract built-in [atom list])
(built-in? #\space) ;; -> #t
(built-in? '(5 7))  ;; -> #f
```

Our library uses these definition forms to introduce structures, traversals, and other useful functions. In particular, for each abstract type our library constructs the obvious predicate (*e.g.*, `atom?`) and for each concrete type it defines field accessors (*e.g.*, `Node-left`), a predicate (*e.g.*, `Node?`), and a short-hand constructor (*e.g.*, `Node` rather than `make-Node`, which is introduced by **`define-struct`**).

Returning to our example from Section 2, equivalent `BExp` structures can be written in AP-F as follows:

```
(abstract BExp [Lit Neg And Or])

(abstract Lit [True False])
(concrete True [])
(concrete False [])

(concrete Neg [(inner BExp)])
(concrete And [(left BExp) (right BExp)])
(concrete Or [(left BExp) (right BExp)])
```

The first line defines `BExp` as the abstract union of four types: `Lit`, `Neg`, `And`, and `Or`. `Lit` is also abstract, with `True` and `False` as concrete variants. Others are defined as concrete types with field name/type pairs, *e.g.*, `(inner BExp)`. We will use these structures throughout the rest of this section, and extend them when needed to demonstrate different aspects of our library.

3.2 Traversals and Functions

In order to write traversal-based generic functions, AP-F introduces two new forms of Scheme expressions (SExps). The syntax of our new **`traverse`** and **`funcset`** expressions is defined below.

```
   SExp ::= ... | Traverse | FuncSet
Traverse ::= (traverse SExp SExp)
FuncSet ::= (funcset Func*)
   Func ::= [(ID*)(ID*) SExp]
```

A FuncSet represents a set of functions, each with argument types, argument names, and a body expression, similar to a list of typed `lambda` expressions. We will refer to a function in the set as a *case* and to its argument types as its *signature*. A traverse expression traverses its first argument, using elements from its second argument, a function set, to fold together recursive results.

As a first example, below we define a simple function to convert a `BExp` into a `string`. For brevity we rename `string-append` to `++`. Our function set, `tostring`, is defined first:

```
(define ++ string-append)
;; tostring: a set of functions
(define tostring
   (funcset [(True) (t) "true"]
            [(False) (f) "false"]
            [(Neg string) (n i) (++ "(not " i ")")]
            [(And string string) (a l r) (++ "(and " l " " r ")")]
            [(Or string string) (o l r) (++ "(or " l " " r ")")]))
```

Each function in the `funcset` handles one of our boolean `BExp` subtype constructors, identified by its first argument type. Using `tostring` we can define a top-level function, `BExp->string`, that converts the given `BExp` into a `string` using a traversal:

```
;; BExp->string: BExp -> string
(define (BExp->string e)
   (traverse e tostring))

;; Test/Example
(BExp->string (And (Neg (True))
                   (False)))   ;; -> "(and (not true) false)"
```

The `traverse` form proceeds with a depth-first walk of the given `BExp` instance. After recursively traversing the fields of the current node, `traverse` selects a function from the given set that best matches: (1) the type of the current node, and (2) the result types of traversing each of the fields. The selected function is then applied to the original node (as its first argument) and the traversal results of its fields. Our asymmetric ordering gives preference to earlier arguments and ensures that there is a *unique* best matching function signature.

For instances of `True` or `False` selecting a function is simple. Since there are no fields, the traversal selects the first or second function in `tostring` based on the type of the node itself. When applied to a `Neg` instance, `traverse` first processes its `inner` field. If the result is a `string`, then the third function is selected and applied. Similarly for `And` and `Or`, with both fields (`left` and `right`) being traversed before selecting a function. Any other case, *e.g.*, `(Neg number)`, would result in a runtime/dispatch error.

3.3 Traversal Control

Returning to our boolean expression example, we originally used `fold-exp` to implement `eval-bexp`, but our fold was not capable of a *short-cutting* traversal. AP-F provides another `traverse` form that takes a third argument. This new argument represents a *control* that guides the traversal through a structure. When a control is not given the traversal proceeds *everywhere*. The syntax additions are described below:

$$
\begin{aligned}
\text{SExp} &::= \ldots \mid \text{Control} \\
\text{Traverse} &::= \ldots \mid \texttt{(traverse}\ \text{SExp}\ \text{SExp}\ \text{SExp}\ \texttt{)} \\
\text{Control} &::= \texttt{(make-bypass}\ \text{FieldUse}^{+}\ \texttt{)} \\
\text{FieldUse} &::= \texttt{(}\ \textsc{Id}\ \textsc{Id}\ \texttt{)}
\end{aligned}
$$

A control is created using the `make-bypass` form that instructs the traversal to *bypass* (or skip over) the given fields, passed as `(type name)` pairs. We differentiate between field definitions and uses in our grammar because of their alternate meanings: `(name type)` and `(type name)` respectively. To make the evaluation of `And` and `Or` short-cutting, we specify that their `right` field should be bypassed:

```
(define eval-ctrl (make-bypass (And right)
                               (Or right)))
```

The function set that will be used to implement short-cutting evaluation is shown below:

```
;; evaluate: A function set
(define evaluate
   (funcset [(True) (t) #t]
            [(False) (f) #f]
            [(Neg true) (n t) #f]
            [(Neg false) (n t) #t]
            ;; The right side will not be traversed
            [(And false BExp) (a l r) #f]
            [(Or true BExp) (o l r) #t]
            [(And true BExp) (a l r) (BExp-eval r)]
            [(Or false BExp) (o l r) (BExp-eval r)]))
```

Our set, `evaluate`, is a bit more complex than `tostring`. For `True` and `False` instances, the function selection is as before, but for the other constructors there is more than one function to choose from.

After traversing the `inner` field of a `Neg` instance, a result of type `true` or `false` (*i.e.*, a `#t` or `#f` value) matches the third or fourth case, respectively. Before describing the rest of the function set, it is important to see how the top-level function, `BExp-eval`, is defined:

```
;; BExp-eval: BExp -> boolean
(define (BExp-eval e)
  (traverse e evaluate eval-ctrl))
```

We use `traverse`, passing the given `BExp`, our function set, and the previously defined control. When the current node is an `And` or `Or`, `eval-ctrl` tells the traversal to skip its `right` field. After the traversal of their `left` field is complete, a function is selected based on the type of the current node, the result type of the `left` traversal, and the type of the unchanged `right` field. The intent to bypass is reflected in the type of the third argument of the last four function cases. We use `BExp`, instead of `true` or `false`, which matches the original type of the field. In the first two of these cases we can immediately return `#t` or `#f`. In the final two cases we make a recursive call to evaluate the `right` side of the expression. Since the right side of the expression is only traversed when necessary, we achieve our short-cutting evaluation strategy.

3.4 Traversal Contexts

There are times when purely compositional functions do not suffice. In cases where information about the ancestors of a sub-structure is important to a function's result, programmers typically add an argument to the function definition, and pass information to recursive invocations, updating the argument when appropriate. Our traversal library supports this style of function using a notion of *traversal contexts*. The original two argument traversal form is extended by adding an additional function set and a root

context. The new `traverse` syntax is shown below including syntax for representing fields as types.

```
  Traverse ::= ... | (traverse SExp SExp SExp SExp)
FieldType ::= any-field | Id . Id
```

The first function set passed to `traverse` is still responsible for combining recursive traversal results, but the second is responsible for updating the context at interesting points during traversal. The context is available to each function case as its last argument. However, functions can ignore the context (or other later arguments) simply by using a shorter signature. Functions responsible for context updating can accept up to three arguments that represent the current node of the structure, the next field to be traversed, and the parent's (previous) context. The field to be traversed is encoded as a FieldType, shown in the grammar as either the special identifier `any-field`, or of the form `type.field`, *e.g.*, `And.left`. The field type represents the pending traversal of the named `field` of the given `type`. AP-F defines corresponding field-types for each field of a `concrete` definition, making them subtypes of `any-field`.

Getting back to our example, we extend `BExp` structures with variable expressions and implement a function that transforms a `BExp` into *negation normal* form. The updated structure definitions are shown below; for brevity we elide our unchanged structures.

```
;; Add Var to BExp definition
(abstract BExp [Lit Neg And Or Var])
(concrete Var [(id symbol)])
```

The new `Var` structure contains a `symbol` representing an identifier, and is added as a variant to our `abstract` type `BExp`.

Our strategy for implementing this transformation is to keep track of the number of nested outer `Neg` expressions during the traversal. We can then change the signs of variables and literals accordingly, following the usual rules with `And` and `Or` under negation. The structure definitions and a function set for tracking nested negations as a context are shown below.

```
(abstract Sign [Even Odd])
(concrete Even [])
(concrete Odd [])

(define sign-updt
  (funcset [(BExp any-field Sign) (e f s) s]
           [(Neg Neg.inner Even) (n f s) (Odd)]
           [(Neg Neg.inner Odd) (n f s) (Even)]))
```

We represent our context by a `Sign`: either positive, `Even`, or negative, `Odd`. We also define a function set, `sign-updt`, for managing the `Sign` context with three cases. The first is a default case: for any `BExp`, before traversing `any-field` with a context of `Sign` we pass the sign unchanged to sub-expressions. The other cases flip the current `Sign` from `Even` to `Odd` (or vice versa) when traversal enters the `inner` field of a `Neg`. The traversal takes care of propagating and passing the updated context when functions are applied.

With our traversal context sorted out, we can write a function set, `neg-normal`, that recursively normalizes a `BExp`. The code is shown below.

```
(define neg-normal
  (funcset [(Lit Even) (l s) l]
           [(True Odd) (t o) (False)]
           [(False Odd) (f o) (True)]
```

```
[(Neg BExp) (n e) e]
[(Var symbol Even) (v id s) v]
[(Var symbol Odd) (v id s) (Neg v)]
[(And BExp BExp Even) (a l r s) (And l r)]
[(And BExp BExp Odd) (a l r s) (Or l r)]
[(Or BExp BExp Even) (o l r s) (Or l r)]
[(Or BExp BExp Odd) (o l r s) (And l r)]))
```

The function set is best explained case by case. The first matches `Lit` instances within an `Even` context, simply returning the original literal. The next two cases match `True` and `False` instances within an `Odd` context, returning their negation. After normalization, only variables are negated, so the case for `Neg` accepts just two arguments. The function ignores its context and returns the recursively normalized `inner BExp`.

Cases for `Var` return the original variable within an `Even` context, and its negation within an `Odd` context. The final four function cases rebuild or convert `And`/`Or` instances under `Even` or `Odd` contexts respectively. The cases follow the De Morgan conversion rules for conjunction/disjunction, *e.g.*, $\neg(a \wedge b) \equiv (\neg a \vee \neg b)$, with the recursive traversal having already propagated negations.

A traversal expression completes the definition of our function, `BExp-normalize`, shown below.

```
;; BExp-normalize: BExp -> BExp
(define (BExp-normalize e)
  (traverse e neg-normal sign-updt (Even)))
```

We pass four arguments to `traverse`: the given `BExp`, our normalizing function set, the `sign-updt` functions, and a root context. Since we begin with no outer `Neg`, our initial context is `Even`.

3.5 Extensible Functions

Separating function sets from traversal also allows us to independently extend/override function sets. AP-F supports such extension using a `merge-func` form. The additional syntax of FuncSet is shown below.

FuncSet ::= ... | **`(merge-func`** SExp SExp **`)`**

Given two function sets, `merge-func` intuitively extends the first by adding all function cases from the second. Any duplicate signatures will be overridden, giving preference to the second function set.

A typical use of traversals where function extension is convenient is when performing *functional updates* over a particular structure, similar to `map` over lists. The foundation of such a transformation, named `copy`, is shown below.

```
;; Rebuild/Copy BExp structures
(define copy
  (funcset [(True) (e) (True)]
           [(False) (e) (False)]
           [(Neg BExp) (e in) (Neg in)]
           [(And BExp BExp) (e l r) (And l r)]
           [(Or BExp BExp) (e l r) (Or l r)]
           [(Var symbol) (e s) (Var s)]))
```

Each function case in `copy` rebuilds our `BExp` structures during traversal by calling the individual constructors on recursive results.

As an example, we can extend `copy` with specialized functions that will simplify constant (non-variable) expressions to `True` or `False` literals and eliminate nested negations. Our additional function set, `simplify`, is shown below.

```
(define simplify
  (funcset [(Neg True) (n t) (False)]
           [(Neg False) (n t) (True)]
           [(Neg Neg) (n e) (Neg-inner e)]
           [(And False) (a l) l]
           [(And BExp False) (a l r) r]
           [(And True BExp) (a l r) r]
           [(And BExp True) (a l r) l]
           [(Or True) (o l) l]
           [(Or BExp True) (o l r) r]
           [(Or False BExp) (o l r) r]
           [(Or BExp False) (o l r) l]))
```

Our functions handle specific cases where the current `BExp` can be simplified based on recursive results. A `Neg` instance can be simplified when its recursive result is a `Lit` by returning its negation, or when its recursive result is a `Neg` by returning the `inner BExp`. Instances of `And` and `Or` have a number of cases that can be simplified, when at least one of the recursive results is a `Lit`. The first case uses a shorter signature, ignoring the recursive result from its `right` field, since it is not needed. In each case, the original `BExp` can be replaced by the simplified results from its `left` or `right` field.

We can create the top-level function, `BExp-simplify`, as shown below, using the extended function set including `copy` and `simplify`.

```
;; BExp-simplify: BExp -> BExp
(define (BExp-simplify e)
  (traverse e (merge-func copy simplify)))
```

For cases where `simplify` does not match, functions from `copy` are used to rebuild the structure. Because the function selected during traversal is unique, the application of functions is well ordered and depth-first, so confluence and critical pairs are not a problem. The traversal gives us the added benefit of implicit recursion, so our transformation applies to the entire data structure. This kind of transformation is so common that AP-F provides a function set, named `TP` for *type-preserving* [24,21], that implements the `copy` functionality for all defined structures.

3.6 Mutual Recursion

Up till now our data structures have only been *self* recursive, where recursive occurrences within `BExp` subtypes are only of type `BExp`. Mutually recursive types can sometimes make processing instances more complicated, particularly in object-oriented languages where visitors take the place of functions [20,9]. However, AP-F handles mutual recursion just like self recursion, with the traversal selecting matching functions from the given set.

As an example, we can extend our `BExp` structures to represent variable binding. We add a new `BExp` variant, `Let`, that contains a `Bind` and a body `BExp`. A binding is represented with a symbol and a `BExp`. The structures are shown below.

```
;; Add Let to BExp definition
(abstract BExp [Lit Neg And Or Var Let])

(concrete Let [(bind Bind) (body BExp)])
(concrete Bind [(id symbol) (e BExp)])
```

The types `BExp` and `Bind` are considered mutually recursive, since a `Let` is a `BExp` and has a `Bind`, which has a `BExp`. We can extend our previous example, `BExp-simplify`, to handle our new structures by extending our function sets and redefining the top-level function as follows:

```
;; Extend copy for Let and Bind
(define copy-w/let
  (merge-func copy (funcset
                    [(Let Bind BExp) (l b e) (Let b e)]
                    [(Bind symbol BExp) (b id e) (Bind id e)])))

;; Extend simplify for Let and Bind
(define simplify-w/let
  (merge-func simplify (funcset [(Let Bind Lit) (l b e) e])))

;; BExp-simplify: BExp -> BExp
(define (BExp-simplify e)
  (traverse e (merge-func copy-w/let simplify-w/let)))
```

Our new function sets `copy-w/let` and `simplify-w/let` add support for the new `Let` and `Bind` structures. We add construction cases for each to `copy`, and simplify a `Let` with a `Lit` body, since the binding is unnecessary in this pure an total setting. `BExp-simplify` is redefined by merging our new function sets. Because each case is handled separately, the presence of mutual recursion does not affect our traversal: functions are still applied as usual.

3.7 Performance

In order to gauge the performance of our initial AP-F implementation we performed several experiments using PLT/DrScheme version 4.1.5 on a Dell Inspiron laptop with a 1.5 Ghz Pentium M processor running Linux. AP-F is provided as a module, which exports syntax for `abstract` and `concrete` definitions and other expressions like `funcset` and `traverse`. Our tests were run in the DrScheme language "*Pretty Big*".[2]

Figure 1 contains the results of comparing various implementations of `deep-sum` over lists, and each of the `BExp` functions presented in this section. The first row is hand-written Scheme, the next is a fold-based implementation, followed by Generic Haskell (for DeepSum), and finally AP-F.

	DeepSum	ToString	Eval	Norm	Simp	Simp/Let
Hand	1.0	1.0	1.0	1.0	1.0	1.0
Fold	1.2	1.0	6.0	—	1.1	0.9
GenHaskell	7.0	—	—	—	—	—
AP-F	126.0	13.2	148.0	232.0	195.8	146.9

Fig. 1 Performance of `dep-sum` and `BExp` functions.

The number shown for each implementation is the time taken on a large data structure instance divided by the time taken by a hand-written version of the same function. Missing numbers mark where an approach is not applicable. For DeepSum we use the same list of numbers in both Scheme and Haskell. For the others we generated a `BExp` of depth 10. For the final case (Simp/Let) we included `Let` and `Bind` structures.

As with the hand-written code, the performance of `fold` functions is heavily dependent on the order of the structure predicate tests, but the implementations otherwise

[2] The performance of hand-written and `fold`-based functions was inexplicably poor when using DrScheme's "*Module*" language.

perform very well. Generic Haskell uses a compilation step to produce type-class definitions and specific instances for deeply nested lists. The slowdown is likely due to dictionary passing and lazy evaluation.

The slowdown of AP-F is due to our use of PLT Scheme's structural reflection, our own ad-hoc subtype testing, and function set dispatch. The slowdown is mostly proportional to the number of function cases, though the length and order of signatures allows some functions to be eliminated more quickly during selection. ToString is the exception, since the real work of appending strings is almost identical for all three implementations. Overall our performance is adequate for a prototype implementation, but improving the execution times of AP-F programs is a priority for future work. We have experimented with different traversal and dispatch strategies within our Java and C# implementations [7], but have yet to apply these ideas to Scheme.

3.8 Errors and Assumptions

Having seen several examples of our AP-F library and functions, it is worth going over the assumptions that AP-F makes and the different errors that can occur when using and writing traversal-based functions. As with any Scheme-based library, programmers can raise a traditional `error` during the execution of a function. AP-F does not attempt to interact with Scheme exception mechanisms, so programmer-raised errors behave as expected.

AP-F assumes a bit more about the structures that will be traversed. While structures defined with `abstract` and `concrete` do not support mutation, most Scheme implementations allow mutation of `cons` lists and hand-defined structures, which allows programmers to construct cyclic instances. AP-F assumes that traversed structures are acyclic, but traversal-based functions can be written for cyclic structures by using control to avoid infinite recursion.

All the function sets presented thus far have been *type-correct* and *complete* with respect to the structures being traversed. However, when this is not the case AP-F raises an error during function selection. A simple example is shown below. We traverse a `list`, but only handle the `empty` case:

```
(traverse '(1 2) (funcset [(empty) (e) 0]))
```

Running this expression results in the following AP-F runtime error:

```
No applicable function found for arguments:
  (cons number number)
```

The error states that a matching function for the signature `(cons number number)` was not found in the given function set. In this case the problem is easy to fix by adding a new case, but often we want to be certain that a traversal expression will *never* raise such an error. Being able to statically eliminate such dispatch errors from traversals is the main topic of the rest of the paper. While externally verifying Scheme programs is a partial goal, the model and soundness of AP-F are directly applicable to our other implementations in statically typed languages, namely Java and C#.

4 A Model of AP-F

Now that we have discussed the features of our AP-F implementation, in this section we formally describe syntax and semantics of a simplified model of the key aspects of

our library. The model captures AP-F's structure definitions, adaptive traversal, and type-based dispatch, allowing us to define a type system that verifies that traversals are free from dispatch errors.

4.1 Syntax

We begin by giving a concise description of our minimal syntax, which embodies most of our implementation; it is described in Figure 2. The only notable features missing are base types (like `number`) and field names.

$$
\begin{array}{rll}
x & ::= & \text{variable names} \\
C & ::= & \text{concrete type names} \\
A & ::= & \text{abstract type names} \\
T & ::= & C \mid A \\
P & ::= & D_1 \ldots D_n \;\; e
\end{array}
\qquad
\begin{array}{rll}
D & ::= & \texttt{(concrete } C \texttt{ [} T_1 \ldots T_n \texttt{])} \\
 & \mid & \texttt{(abstract } A \texttt{ [} T_0 \ldots T_n \texttt{])} \\
e & ::= & x \mid \texttt{(} C\ e_1 \ldots e_n \texttt{)} \mid \texttt{(traverse } e_0\ F \texttt{)} \\
F & ::= & \texttt{(funcset } f_1 \ldots f_n \texttt{)} \\
f & ::= & \texttt{[(} T_0 \ldots T_n \texttt{) (} x_0 \ldots x_n \texttt{) } e \texttt{]}
\end{array}
$$

Fig. 2 AP-F Model Language Syntax

A simplified AP-F program, P, is a sequence of data structure definitions (`abstract` and `concrete` types) followed by an expression. Concrete type definitions only mention the types of their fields, as names will not be important. Expressions, e, are either variable references, value constructions, or traversals. We model the simplest form of traversal expressions from our library, representing the traversal of a structure using a given function set, F. Functions and function sets are the same as in our library. A function set is a sequence of functions, each of which is a sequence of type names, followed by parameter names and a body expression.

Based on the definitions in a program, we define a subtype relation, $\leq$, in Figure 3, as the reflexive, transitive closure of the immediate subtype relationship from abstract definitions.

$$
\text{[S-Refl]}\ \frac{}{T \leq T}
\qquad
\text{[S-Def]}\ \frac{\texttt{(abstract } A \texttt{ [} T_0 \ldots T_n \texttt{])} \in P}{T_i \leq A}
\qquad
\text{[S-Trans]}\ \frac{T \leq T'' \qquad T'' \leq T'}{T \leq T'}
$$

Fig. 3 Subtyping Rules

Our model does not include base types, but our basic boolean expression structures (from Sections 2 and 3) can still be defined. The original `BExp` definitions without field names are shown in Figure 4. To complete the program definition we construct a simple `BExp` in the program's body.

4.2 Well-Formedness Rules

In order to avoid purely syntactic problems in our semantics, we restrict syntactically valid programs with a few *well-formedness* rules. They check the sanity of a program's definitions and allow us to focus on the key issues of our semantics.

```
;; ASTs for boolean expressions
(abstract BExp [Lit Neg And Or])
(abstract Lit [True False])
(concrete True [])
(concrete False [])
(concrete Neg [BExp])
(concrete And [BExp BExp])
(concrete Or [BExp BExp])

;; Simple program body
(Or (And (True) (False)) (Neg (False)))
```

Fig. 4 Model Example: Boolean expression structures

TypesOnce(P): Each type must only be defined once.

CompleteTypes(P): Each type used in the right-hand side of a definition must itself be defined.

NoSelfSuper(P): Each `abstract` type must not occur in the right-hand side of its own definition.

SingleSuper(P): Each type should occur in the right-hand side of at most one `abstract` definition.

The first two rules check for the existence and completeness of a program's definitions. TypesOnce ensures that each type is *defined* only once, and CompleteTypes makes sure each type *use* corresponds to a defined type. The rules do not restrict recursion in the data structures or the shapes that can be defined, since they only require that a definition exists and is unique.

SingleSuper enforces a simplifying assumption on our type hierarchies, restricting types to a form of single inheritance. Together with NoSelfSuper the rules ensure a linear supertype relation, which gives us a total ordering on function signatures: each `abstract` type can have multiple subtypes, but only one supertype. Requiring a total order on function signatures simplifies our dispatch semantics by avoiding the usual *diamond problem* when multiple inheritance and multiple dispatch interact [29, 8].

4.3 Semantics

We use a (small-step) reduction semantics to model AP-F traversals. We begin with a description of values, v, runtime expressions, e, and evaluation contexts, E, described in Figure 5.

$$
\begin{array}{l}
v ::= \texttt{(}C\ v_1\ \ldots\ v_n\texttt{)} \\
e ::= \cdots \\
\quad \mid \texttt{(dispatch}\ F\ v_0\ e_1\ \ldots\ e_n\texttt{)} \\
\quad \mid \texttt{(apply}\ f\ v_0\ v_1\ \ldots\ v_n\texttt{)}
\end{array}
\qquad
\begin{array}{l}
E ::= [\,] \\
\quad \mid \texttt{(}C\ v\ \ldots\ E\ e\ \ldots\texttt{)} \\
\quad \mid \texttt{(traverse}\ E\ F\texttt{)} \\
\quad \mid \texttt{(dispatch}\ F\ v_0\ v\ \ldots\ E\ e\ \ldots\texttt{)}
\end{array}
$$

Fig. 5 Values, runtime expressions, and evaluation contexts

Values are simply constructions in which all sub-expressions are also values. Runtime expressions (`dispatch` and `apply`) are not in our surface syntax, but are used to

model structural recursion and function application respectively. The use of `apply` is mainly cosmetic, in order to avoid over complicating rules involving `dispatch`. Evaluation contexts account for the reduction strategy, which is deterministic and left-most/inner-most.

Figure 6 contains definitions of our reflective meta-functions and substitution. The function *types* is used to return the concrete types of a list of values, others functions are simply convenient accessors for portions of abstract syntax.

$$
\begin{array}{lcl}
\mathit{types}(\texttt{(}C_0\ \ldots\texttt{)}\ \ldots\ \texttt{(}C_n\ \ldots\texttt{)}) & = & (C_0\ \ldots\ C_n) \\
\mathit{argtypes}(\texttt{[(}T_0\ \ldots\ T_n\texttt{) (}x_0\ \ldots\ x_n\texttt{) }e\texttt{]}) & = & (T_0\ \ldots\ T_n) \\
\mathit{functions}(\texttt{(funcset }f_1\ \ldots\ f_n\texttt{)}) & = & (f_1\ \ldots\ f_n)
\end{array}
$$

$$
\begin{array}{lcl}
x[v/x] & = & v \\
x'[v/x] & = & x' \quad \text{if } x' \neq x \\
\texttt{(}C\ e_1\ \ldots\ e_n\texttt{)}[v/x] & = & \texttt{(}C\ e_1[v/x]\ \ldots\ e_n[v/x]\texttt{)} \\
\texttt{(traverse }e_0\ F\texttt{)}[v/x] & = & \texttt{(traverse }e_0[v/x]\ F[v/x]\texttt{)} \\
\texttt{(dispatch }F\ v_0\ e_1\ \ldots\ e_n\texttt{)}[v/x] & = & \texttt{(dispatch }F[v/x]\ v_0\ e_1[v/x]\ \ldots\ e_n[v/x]\texttt{)} \\
\texttt{(apply }f\ v_0\ v_1\ \ldots\ v_n\texttt{)}[v/x] & = & \texttt{(apply }f[v/x]\ v_0\ v_1\ \ldots\ v_n\texttt{)} \\
\texttt{(funcset }f_1\ \ldots\ f_n\texttt{)}[v/x] & = & \texttt{(funcset }f_1[v/x]\ \ldots\ f_n[v/x]\texttt{)} \\
\texttt{[(}T_0\ \ldots\ T_n\texttt{) (}x_0\ \ldots\ x_n\texttt{) }e\texttt{]}[v/x] & = & \texttt{[(}T_0\ \ldots\ T_n\texttt{) (}x_0\ \ldots\ x_n\texttt{) }e[v/x]\texttt{]} \quad \text{if } x \notin \overline{x_i} \\
\texttt{[(}T_0\ \ldots\ T_n\texttt{) (}x_0\ \ldots\ x_n\texttt{) }e\texttt{]}[v/x] & = & \texttt{[(}T_0\ \ldots\ T_n\texttt{) (}x_0\ \ldots\ x_n\texttt{) }e\texttt{]} \quad \text{if } x \in \overline{x_i}
\end{array}
$$

Fig. 6 Reflection and Substitution Definitions

The substitution of a value for a variable within an expression, denoted $e[v/x]$, is defined over all terms, including function sets, F. Substitution within function definitions only occurs when the variable is free in the function body. Since only values can be substituted, and functions are not first-class, α-conversion or renaming is not necessary to avoid capture.

Figure 7 completes our meta-functions with signature comparison and type-based function selection implemented by *choose*. The helper function *chooseOne* selects the most specific applicable function in a set, given the actual argument types. *possibleFs* filters the function set, returning only the functions that are *possible* to apply to the given types. *possible* returns true if all arguments are elementwise related, since a function may be applied to subtypes of its argument types or when actual arguments are refined from supertypes. At runtime however, the actual argument types will always be `concrete` and without subtypes, so the second check, $T_0 \leq T_0'$, is irrelevant. The check becomes important when we use *possibleFs* with approximate types, as is necessary during type checking. *chooseOne* uses *best* to select the most specific function in the filtered set, using *better* to compare function signatures. For simplicity we compare only functions with the same number of arguments, though our library implementation is more flexible.

Finally, Figure 8 gives a relation, $\rightarrow$, which completes our small-step semantics with a *notion of reduction*, *i.e.*, with axioms or again, with contraction rules. The left-hand side of each unconditional rule represents a potential reducible expression, or *potential redex*. If a potential redex can be contracted then it is considered an actual redex, *i.e.*, no longer potential.

A `traverse` expression with a constructed value as its first argument can be contracted (R-Trav) producing a `dispatch` expression by including the function set, the original value, and wrapping each field in a `traverse` expression. A `dispatch` expres-

$$
\begin{aligned}
\mathit{choose}(F, (C_0 \ldots C_n)) &= \mathit{chooseOne}(\mathit{possibleFs}(F, (C_0 \ldots C_n)), (C_0 \ldots C_n)) \\
\mathit{chooseOne}((), (T_0 \ldots T_m)) &= \texttt{error} \\
\mathit{chooseOne}((f_0\ f_1 \ldots f_n), (T_0 \ldots T_m)) &= \mathit{best}(f_0, (f_1 \ldots f_n), (T_0 \ldots T_m)) \\
\\
\mathit{best}(f, (), (T_0 \ldots T_m)) &= f \\
\mathit{best}(f, (f_0\ f_1 \ldots f_n), (T_0 \ldots T_m)) &= \texttt{if}\ \mathit{better}(\mathit{argtypes}(f_0), \mathit{argtypes}(f)) \\
&\quad\ \texttt{then}\ \mathit{best}(f_0, (f_1 \ldots f_n), (T_0 \ldots T_m)) \\
&\quad\ \texttt{else}\ \mathit{best}(f, (f_1 \ldots f_n), (T_0 \ldots T_m)) \\
\\
\mathit{better}((), ()) &= \texttt{false} \\
\mathit{better}((T_0\ T_1 \ldots T_n), (T'_0\ T'_1 \ldots T'_n)) &= ((T_0 \not\equiv T'_0\ \wedge\ T_0 \leq T'_0)\ \vee \\
&\quad\ (T_0 \equiv T'_0\ \wedge\ \mathit{better}((T_1 \ldots T_n), (T'_1 \ldots T'_n)))) \\
\\
\mathit{possibleFs}(F, (T_0 \ldots T_n)) &= \mathit{filter}(\lambda f.\, \mathit{possible}(\mathit{argtypes}(f), (T_0 \ldots T_n)), \mathit{functions}(F)) \\
\\
\mathit{possible}((), ()) &= \texttt{true} \\
\mathit{possible}((), (T'_0 \ldots T'_m)) &= \texttt{false} \\
\mathit{possible}((T_0 \ldots T_n), ()) &= \texttt{false} \\
\mathit{possible}((T_0\ T_1 \ldots T_n), (T'_0\ T'_1 \ldots T'_m)) &= (T'_0 \leq T_0\ \vee\ T_0 \leq T'_0)\ \wedge \\
&\quad\ \mathit{possible}((T_1 \ldots T_n), (T'_1 \ldots T'_m))
\end{aligned}
$$

Fig. 7 Function Selection Meta-functions

[R-Trav]

$$\texttt{(traverse (}C\ v_1 \ldots v_n\texttt{)}\ F\texttt{)} \rightarrow \texttt{(dispatch}\ F\ \texttt{(}C\ v_1 \ldots v_n\texttt{) (traverse}\ v_1\ F\texttt{)} \ldots \texttt{(traverse}\ v_n\ F\texttt{))}$$

[R-Dispatch]

$$\texttt{(dispatch}\ F\ v_0\ v_1 \ldots v_n\texttt{)} \rightarrow \texttt{(apply}\ f\ v_0\ v_1 \ldots v_n\ \texttt{)} \quad \text{if } f \neq \texttt{error}$$

where $f = \mathit{choose}(F, \mathit{types}(v_0\ v_1 \ldots v_n))$

[R-Apply]

$$\texttt{(apply [(}T_0 \ldots T_n\texttt{) (}x_0 \ldots x_n\texttt{)}\ e\texttt{]}\ v_0\ v_1 \ldots v_n\ \texttt{)} \rightarrow e[\overline{v_i/x_i}]$$

Fig. 8 Reduction Rules

sion containing only values can be contracted (R-Dispatch), when the result of *choose* is not `error`, to an `apply` expression. A `dispatch` expression that violates this side condition is considered *stuck*, *i.e.*, a potential but not actual redex, representing a run-time error. Our last rule (R-Apply) substitutes the values for the formal parameters of the selected function.

4.4 From Reduction to Evaluation

Following Danvy's lecture notes at AFP'08 [10], a one-step reduction function can be defined using our reduction relation that decomposes a non-value expression into an evaluation context, E, and a potential redex. If the potential redex can be contracted, then the contractum can be recomposed with (plugged into) the evaluation context resulting in a reduced program. Figure 9 gives sketches of the functions *decompose*, *recompose*, and *reduce* that implement the one-step reduction function of our semantics.

We define *reduce* as decomposition followed by contraction and recomposition when one of our reduction rules applies. The function *decompose* traverses an expression while accumulating an evaluation context. Expression cases that match evaluation contexts are handled explicitly by recurring on the inner expression. Other expressions, *e.g.*,

$$
\begin{aligned}
\mathit{decompose}(\texttt{(}C\ v\ \ldots\ e_0\ e\ \ldots\ \texttt{)},\ E) &= \mathit{decompose}(e_0,\ \texttt{(}C\ v\ \ldots\ E\ e\ \ldots\ \texttt{)})\\
\mathit{decompose}(\texttt{(traverse }e_0\ F\texttt{)},\ E) &= \mathit{decompose}(e_0,\ \texttt{(traverse }E\ F\texttt{)})\\
\mathit{decompose}(\texttt{(dispatch }F\ v\ \ldots\ e_0\ e\ \ldots\ \texttt{)},\ E) &= \mathit{decompose}(e_0,\ \texttt{(dispatch }F\ v\ \ldots\ E\ e\ \ldots\ \texttt{)})\\
\mathit{decompose}(e,\ E) &= \langle e,\ E\rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathit{recompose}(e, [\,]) &= e\\
\mathit{recompose}(e_0,\ \texttt{(}C\ v\ \ldots\ E\ e\ \ldots\ \texttt{)}) &= \mathit{recompose}(\texttt{(}C\ v\ \ldots\ e_0\ e\ \ldots\ \texttt{)},\ E)\\
\mathit{recompose}(e_0,\ \texttt{(traverse }E\ F\texttt{)}) &= \mathit{recompose}(\texttt{(traverse }e_0\ F\texttt{)},\ E)\\
\mathit{recompose}(e_0,\ \texttt{(dispatch }F\ v\ \ldots\ E\ e\ \ldots\ \texttt{)}) &= \mathit{recompose}(\texttt{(dispatch }F\ v\ \ldots\ e_0\ e\ \ldots\ \texttt{)},\ E)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{reduce}(v) &= v\\
\mathit{reduce}(e) &= \texttt{let }\langle e',\ E\rangle = \mathit{decompose}(e, [\,])\\
&\quad\ \texttt{in }\mathit{recompose}(e'', E)\\
&\quad \text{if } e' \rightarrow e''
\end{aligned}
$$

Fig. 9 One-step Reduction Function

`apply`, match the final case returning a pair of the expression and context. *recompose* does the reverse, building an expression until the empty context is reached.

Our one-step reduction function can be used to iteratively define an evaluation function, as shown in Figure 10. The function *evaluate* implements the iteration of the

$$
\begin{aligned}
\mathit{evaluate}(v) &= v\\
\mathit{evaluate}(e) &= \mathit{evaluate}(\,\mathit{reduce}(e)\,)\\
&\quad \text{if } e \text{ is not } \mathit{stuck}
\end{aligned}
$$

Fig. 10 Reduction-based Evaluation Function

one-step reduction function from Figure 9. This definition can be 'refocused' into an abstract machine and further transformed resulting in a more typical big-step evaluation function [10,11], but the version of Figure 10 is sufficient for our purposes here. For efficiency our actual implementation is, of course, based on a big-step evaluation function.

4.5 Example

With our example definitions (Figure 4) we can add a simple traversal and function set that implements `BExp` evaluation, shown in Figure 11. Without base types, we traverse the expression producing a `Lit`, representing a result of `True` or `False`. Similar to the Scheme example (Section 3), multiple dispatch is used to match the interesting cases during traversal. For `Neg` this means matching `True` or `False` and returning its negation; for `And` or `Or` this means capturing the all-true and all-false cases respectively. The other two cases for `And` and `Or` are handled by more general signatures using `Lit`.

5 A Type System for AP-F

Programs written using our AP-F library can raise many different kinds of unrelated errors. Our model has been specifically created to eliminate all but those relating to

```
;; ... Definitions from Figure 4 ...

(traverse (Or (And (True) (False))
              (Neg (False)))
          (funcset [(Lit) (l) l]
                   [(Neg True) (n t) (False)]
                   [(Neg False) (n f) (True)]
                   [(And True True) (a l r) r]
                   [(And Lit Lit) (a l r) (False)]
                   [(Or False False) (o l r) r]
                   [(Or Lit Lit) (o l r) (True)]))
```

Fig. 11 Model Example: Boolean expression evaluation

traversals, function sets, and dispatch. In order to rule out runtime errors and predict the class of values a program may return, we impose a type system on our model. Though our type system rules out standard errors like unbound variable uses, we are mostly interested in eliminating errors resulting from function selection (*choose* and *chooseOne* in Figure 7).

For any type-correct program we obtain a typing derivation that constrains the return values of traversals and function sets based on the shape of datatypes. Our judgment (*well-typed*) is separated into three mutually recursive relations; one for each of expressions, functions, and traversals. We use two type environments: Γ for variables, and $\mathcal{X}$ to capture the return types of recursive datatype traversals. We represent environments as a list of pairs, with syntax shown in Figure 12.

$$\Gamma ::= \emptyset \mid \Gamma, x{:}T$$
$$\mathcal{X} ::= \emptyset \mid \mathcal{X}, T{:}T'$$

Fig. 12 Variable and Traversal Environments

In certain typing rules we will denote the set of the left-hand sides of pairs from Γ (also $\mathcal{X}$) by $dom\ \Gamma$. New pairs will be appended to environments, and lookup, denoted $\Gamma(x)$, will occur from right to left, selecting the latest binding if duplicate names exist.

5.1 Functions

We begin with the simplest of our typing rules. Since functions are not first-class, type-checking a function depends only on the type of its body expression when parameters are bound to the types given in its signature. Our single rule for $\vdash_F$ is shown in Figure 13.

$$[\text{T-FUNC}]\quad \frac{(\Gamma, x_0{:}T_0, \ldots, x_n{:}T_n) \vdash_e e_0 : T}{\Gamma \vdash_F \texttt{[(}T_0 \ldots T_n\texttt{) (}x_0 \ldots x_n\texttt{)}\ e_0\texttt{]} : T}$$

Fig. 13 Function Typing Rule

5.2 Expressions

Figure 14 shows our typing rules for expressions ($\vdash_e$). Variables must be bound to a

$$\text{[T-Var]}\quad \frac{x \in \mathit{dom}\ \Gamma}{\Gamma \vdash_e x : \Gamma(x)} \qquad \text{[T-New]}\quad \frac{\texttt{(concrete}\ C\ \texttt{[}T_1\ \ldots\ T_n\texttt{])} \in P \quad \text{for } i \in 1..n \quad \Gamma \vdash_e e_i : T_i' \quad T_i' \leq T_i}{\Gamma \vdash_e \texttt{(}C\ e_1\ \ldots\ e_n\texttt{)} : C}$$

$$\text{[T-Trav]}\quad \frac{\Gamma \vdash_e e_0 : T_0 \quad \Gamma; \emptyset \vdash_{\mathcal{T}} \langle T_0, F\rangle : T}{\Gamma \vdash_e \texttt{(traverse}\ e_0\ F\texttt{)} : T}$$

$$\text{[T-Dispatch]}\quad \frac{\begin{array}{c}\emptyset \vdash_e v_0 : C \\ \text{for } i \in 1..n \quad \Gamma \vdash_e e_i : T_i' \\ \text{for } f \in \mathit{possibleFs}(F, (C\ T_1'\ \ldots\ T_n')) \quad \Gamma \vdash_F f : T_f \quad T_f \leq T \\ \mathit{covers}(F, (C\ T_1'\ \ldots\ T_n'))\end{array}}{\Gamma \vdash_e \texttt{(dispatch}\ F\ v_0\ e_1\ \ldots\ e_n\texttt{)} : T}$$

$$\text{[T-Apply]}\quad \frac{f = \texttt{[(}T_0\ \ldots\ T_n\texttt{) (}x_0\ \ldots\ x_n\texttt{)}\ e\texttt{]} \quad \Gamma \vdash_F f : T \quad \text{for } i \in 0..n \quad \emptyset \vdash_e v_i : T_i' \quad T_i' \leq T_i}{\Gamma \vdash_e \texttt{(apply}\ f\ v_0\ v_1\ \ldots\ v_n\texttt{)} : T}$$

Fig. 14 Expression Typing Rules

type in the environment (T-Var) and value construction requires subtypes (T-New) for each expression (*i.e.*, *field*) of a concrete structure. Traversal expressions (T-Trav) delegate to a more specialized judgment, $\vdash_{\mathcal{T}}$ (Section 5.3), passing the variable environment and an empty traversal environment, $\mathcal{X} = \emptyset$. For `dispatch` expressions (T-Dispatch) we use *possibleFs* to be sure all *possible* functions unify to a common supertype. Function application (T-Apply) requires subtypes of a function's formal parameter types.

One subtle (but key) aspect of the T-Dispatch rule is the use of the meta-function, *covers*. Its properties will be discussed in Section 5.4, but the main idea of *covers* is to verify that a function set, F, contains a possible function for each sequence of `concrete` types that are subtypes of the given sequence. In this case, it means that F has at least one function that can be applied to *values* of the given types. The use of *covers* here corresponds to our typing rules for concrete traversals, which is discussed in the next section.

5.3 Traversals

Traversal expressions are typed using a specialized judgment, $\vdash_{\mathcal{T}}$, that takes data structure definitions and the function set into account. The two rules, one for each of concrete and abstract types, are shown in Figure 15.
We read $\Gamma; \mathcal{X} \vdash_{\mathcal{T}} \langle T, F\rangle : T'$ as follows:

> *In type environment Γ with traversal types $\mathcal{X}$ the traversal of a value of type T with function set F returns a value of type T'.*

[T-ATrav]

$$\frac{\texttt{(abstract}\ A\ \texttt{[}T_0\ \ldots\ T_n\texttt{])} \in P \quad \text{for } i \in 1..n \quad T_i \in \mathit{dom}\,\mathcal{X} \Rightarrow T_i' = \mathcal{X}(T_i) \quad \text{for } i \in 1..n \quad T_i \notin \mathit{dom}\,\mathcal{X} \Rightarrow \Gamma; \mathcal{X}, A{:}T \vdash_{\mathcal{T}} \langle T_i, F\rangle : T_i' \quad \text{for } i \in 1..n \quad T_i' \leq T}{\Gamma; \mathcal{X} \vdash_{\mathcal{T}} \langle A, F\rangle : T}$$

[T-CTrav]

$$\frac{\texttt{(concrete}\ C\ \texttt{[}T_1\ \ldots\ T_n\texttt{])} \in P \quad \text{for } i \in 1..n \quad T_i \in \mathit{dom}\,\mathcal{X} \Rightarrow T_i' = \mathcal{X}(T_i) \quad \text{for } i \in 1..n \quad T_i \notin \mathit{dom}\,\mathcal{X} \Rightarrow \Gamma; \mathcal{X}, C{:}T \vdash_{\mathcal{T}} \langle T_i, F\rangle : T_i' \quad \text{for } f \in \mathit{possibleFs}(F, (C\ T_1'\ \ldots\ T_n')) \quad \Gamma \vdash_F f : T_f \quad T_f \leq T \quad \mathit{covers}(F, (C\ T_1'\ \ldots\ T_n'))}{\Gamma; \mathcal{X} \vdash_{\mathcal{T}} \langle C, F\rangle : T}$$

Fig. 15 Traversal Typing Rules

Γ is the standard variable type environment. $\mathcal{X}$ is an environment of traversal return types for (possibly recursive) types that may depend on the traversal return of T. The function set F is constant for a given expression, and is passed throughout a derivation.

The typing of the traversal of an **`abstract`** type proceeds by typing each of the elements T_i separately. If a binding for T_i exists in $\mathcal{X}$ ($T_i \in \mathit{dom}\,\mathcal{X}$) then the result, T_i', must be the same as the bound result type, which we denote $\mathcal{X}(T_i)$. Otherwise, we calculate the result type with $A : T$ added to $\mathcal{X}$ using the same function set, F. The final line of the premise constrains the result type for the union to be a common supertype of the traversal the individual elements.

The rule for **`concrete`** types is more involved due to function selection. For *field* types bound in $\mathcal{X}$ the result, T_i', must be the same as the bound result type. For unbound field types we calculate the result type of a traversal with $C : T$ added to $\mathcal{X}$ using the same function set F. Using the return types, T_i', of field traversals we can approximate the *possible* functions from F that can be called after traversing an instance of C. The final return type, T, is the common supertype of the *possibleFs* given the field return types. On the last line of the premise the meta-function $\mathit{covers}(\cdot)$ is used to determine whether or not the function set is *complete* with respect to all possible value sequences corresponding to subtypes of the given types. The attributes of *covers* are important to the type soundness of our model and deserve special discussion.

5.4 Function Set Coverage

Type checking AP-F programs infers the return types of traversal expressions, but being sure that function selection always succeeds requires an analysis of function set signatures. In particular, our asymmetric multiple dispatch implemented by *choose* means that after traversing a concrete value, any of the *possible* functions may be called based on the types of sub-traversal return values. In general, we cannot know (until runtime) which concrete subtypes will be returned, so we require that all cases be handled by the function set.

In order to guarantee successful dispatch, $\mathit{covers}(\cdot)$ must check all concrete subtypes of the possible argument types and ensure that a *possible* function exists. Because our

type hierarchies and function signatures can be arranged into trees (or at least *directed acyclic graphs*), we call the problem *leaf-covering*. The solution involves the Cartesian product of the sequence of type hierarchies [5], but the actual implementation of *covers* is not important to our soundness, only that each concrete sequence of subtypes has a *possible* function:

$$\begin{array}{l} \mathit{covers}(F,\ (T_0\ T_1\ \ldots\ T_n)) \Leftrightarrow \\ \qquad \forall\, C_0,\ C_1,\ \ldots,\ C_n \text{ with } C_i \leq T_i\ .\ \mathit{possibleFs}(F,\ (C_0\ C_1\ \ldots\ C_n)) \neq () \end{array}$$

As a consequence, *covers* is preserved by subtyping. If $\forall i \in 1..n\,.\,T'_i \leq T_i$, then:

$$\mathit{covers}(F,\ (T_0\ T_1\ \ldots\ T_n)) \Rightarrow \mathit{covers}(F,\ (T'_0\ T'_1\ \ldots\ T'_n))$$

Because runtime values are made only of concrete types, *e.g.*, `(Neg (True))`, then function selection cannot fail as long as sub-traversals (at runtime) return subtypes of their expected types. The implementation of *covers* is beyond the scope of this paper, though we have experimented with several different approaches. The abstract problem of leaf-covering is *coNP-complete* [5], however, the number of function arguments (*i.e.*, structure fields) tend to be small, and individual type hierarchies are usually tractable. In our Java implementation, called DemeterF, the largest number of arguments is 13. With approximately 90 classes in all, the deepest subtype chain is 4 classes, *i.e.*, $C \leq A_1 \leq A_2 \leq A_3$.

5.5 Typing Example

Returning to our model example in Figure 11, we can assign a type to the body of our program using the T-TRAV rule. The first argument to `traverse` is given the type `Or` by successive applications of T-NEW. Since `True` and `False` have no fields, their constructions become axioms for the derivation. The second part of T-TRAV requires the use of our traversal judgment:

$$\emptyset; \emptyset \vdash_{\mathcal{T}} \langle\, \texttt{Or}, F \rangle : T$$

From the definitions in Figure 4 `Or` is a **`concrete`** type, so a derivation requires the use of T-CTRAV:

$$\frac{\begin{array}{c} \texttt{(concrete Or [BExp BExp])} \in P \quad \emptyset; (\emptyset,\ \texttt{Or}{:}T_{\texttt{or}}) \vdash_{\mathcal{T}} \langle\, \texttt{BExp}, F \rangle : T_{\texttt{bexp}} \\ \text{for } f \in \mathit{possibleFs}(F,\ (\texttt{Or}\ T_{\texttt{bexp}}\ T_{\texttt{bexp}})) \quad \emptyset \vdash_F f : T_f \quad T_f \leq T_{\texttt{or}} \\ \mathit{covers}(F,\ (\texttt{Or}\ T_{\texttt{bexp}}\ T_{\texttt{bexp}})) \end{array}}{\emptyset; \emptyset \vdash_{\mathcal{T}} \langle \texttt{Or}, F \rangle : T_{\texttt{or}}}$$

The traversal type derivation recursively continues to the abstract types `BExp` and `Lit`, eventually coming to the applications of T-CTRAV for `True` and `False` that do not require recursion. For these types there is only one *possible* function, which simplifies the rule further. An instance for the type `True` is shown below.

$$\frac{\begin{array}{c} \texttt{(concrete True [])} \in P \\ \emptyset \vdash_F \texttt{[(Lit) (l) l]} : \texttt{Lit} \quad \texttt{Lit} \leq T_{\texttt{true}} \\ \mathit{covers}(F,\ (\texttt{True})) \end{array}}{\emptyset; \mathcal{X} \vdash_{\mathcal{T}} \langle \texttt{True}, F \rangle : T_{\texttt{true}}}$$

Assigning a type to the single function and checking function set coverage is then trivial. The constraints build up as we come back through the abstract definitions of `Lit` and `BExp`. Ignoring other variants of `BExp` for simplicity, we have the constraints:

$$\texttt{Lit} \leq T_{\texttt{true}} \quad \texttt{Lit} \leq T_{\texttt{false}} \quad T_{\texttt{true}} \leq T_{\texttt{lit}} \quad T_{\texttt{false}} \leq T_{\texttt{lit}} \quad T_{\texttt{lit}} \leq T_{\texttt{bexp}}$$

We can make these true by setting each of the return types to `Lit`. Other `BExp` variants (`Neg`, `And`, and `Or`) are recursive, which causes equality constraints to be generated instead.

6 Soundness

In order to prove our AP-F model sound, we construct a Wright-Felleisen [37] style proof of type-soundness, by way of *progress* and *preservation*. Our proof ultimately shows that the reduction of a well-typed AP-F program will not get *stuck*, and will result in a value of the expected type. An expression e is considered *stuck* if there does not exist an expression e' such that $e \rightarrow e'$. In particular, an expression is stuck if it is of the form:

$$E[\,(\texttt{dispatch}\ F\ v_0\ v_1\ \ldots\ v_n)\,]$$

and *choose* (Figure 7) results in an error:

$$\mathit{choose}(F,\ \mathit{types}(\,v_0\ v_1\ \ldots\ v_n)) = \texttt{error}$$

We note that *choose* returns `error` precisely when:

$$\mathit{possibleFs}(F,\ \mathit{types}(\,v_0\ v_1\ \ldots\ v_n)) = ()$$

Meaning that F does not contain a function applicable to the given arguments.

Our proof begins with a few AP-F specific lemmas (function and traversal specialization) then moves on to more standard soundness lemmas such as substitution and well-typed contexts. In order to prove that reduction preserves the type of a program, it is necessary to start at the dispatch level and work up to expressions. We begin by proving that *possibleFs* applied to a sequence of subtypes returns a *subset* of the functions returned by *possibleFs* applied to supertypes.

Lemma 1 (Function Specialization) *As a sequence of argument types is specialized through subtyping, the set of possible functions does not increase.*

If $\forall i \in 1..n\ \ T'_i \leq T_i$ *then*
$\mathit{possibleFs}(F,\ (T'_1\ \ldots\ T'_n)) \subseteq \mathit{possibleFs}(F,\ (T_1\ \ldots\ T_n))$

Proof: We argue using induction on the type sequences by case analysis of the definition of *possible* (Figure 7), used to filter the functions of F. Consider a single function $f \in F$ with formal argument types, $(T^f_0\ \ldots\ T^f_m)$. Our lemma depends on a single implication that must hold of *possible* (given our subtype sequence assumption):

$$\mathit{possible}((T^f_0\ \ldots\ T^f_m),\ (T'_0\ \ldots\ T'_n)) \ \Rightarrow\ \mathit{possible}((T^f_0\ \ldots\ T^f_m),\ (T_0\ \ldots\ T_n))$$

The three base cases of *possible* (Figure 7) are simple, so we consider them together. If the first case applies, then our implication follows trivially, while the two `false` cases are not relevant, since they may only decrease the set of selected functions. Proof of the lemma then hinges on showing our implication holds for the inductive case of the

definition, particularly the first component of the conjunction. In our case this reduces to:

$$(T_0^f \leq T_0') \vee (T_0' \leq T_0^f) \;\Rightarrow\; (T_0^f \leq T_0) \vee (T_0 \leq T_0^f)$$

which follows from transitivity (and reflexivity) of the program's subtype relation, $\leq$, as both disjunction components of the implication are immediate:

$$(T_0^f \leq T_0') \;\Rightarrow\; (T_0^f \leq T_0) \text{ and } (T_0' \leq T_0^f) \;\Rightarrow\; (T_0 \leq T_0^f)$$

□

In order to complete the dispatch portion of preservation we must also show that well-typed traversal expressions preserve types, the subject of lemma 2.

Lemma 2 (Traversal Specialization, or Subtype Traversals Return Subtypes) *As the type of an expression that is the argument of a traversal is refined, the return type of the traversal expression itself remains a subtype of its original type.*

For any well-typed traversal of a type T_0 with $\Gamma; \emptyset \vdash_{\mathcal{T}} \langle T_0, F \rangle : T$. The traversal of a type $T_0' \leq T_0$ satisfies $\Gamma; \emptyset \vdash_{\mathcal{T}} \langle T_0', F \rangle : T'$ for some $T' \leq T$

Proof: By induction on the traversal type derivation of $\Gamma; \emptyset \vdash_{\mathcal{T}} \langle T_0, F \rangle : T$, we proceed on the last rule of the derivation, which must be one of T-CTrav or T-ATrav, from Figure 15.

If T-ATrav applies ($\texttt{(abstract}\ T_0\ \texttt{[}T_1\ \ldots\ T_n\texttt{])} \in P$) then the rule requires that a traversal of an immediate subtype of T_0 return a subtype of the final result type, which applies inductively to all transitive subtypes of T_0, including T_0'.

If T-CTrav applies ($\texttt{(concrete}\ T_0\ \texttt{[}T_1\ \ldots\ T_n\texttt{])} \in P$) then T_0 can only have itself as a subtype ($T_0 \equiv T_0'$). Regardless of which function in F is actually applied at runtime, we know by the T-CTrav derivation that each function returns a subtype, from the premises of the rule.

□

The final lemmas for preservation are value substitution and well-typed contexts. Substitution proves that function application preserves the type of a traversal expression:

Lemma 3 (Substitution Preserves Type) *Substituting a value of a subtype for a free variable in any expression results in a subtype of the original expression's type.*

Suppose $\Gamma \equiv (\Gamma', x{:}T_x)$. If $\Gamma \vdash_e e : T$, $\emptyset \vdash_e v : T_x'$, with $T_x' \leq T_x$ then $\Gamma' \vdash_e e[v/x] : T'$ and $T' \leq T$.

Proof: By induction on the derivation of $(\Gamma, x : T_x) \vdash_e e : T$. Traversal expressions require lemma 2, and dispatch expressions require lemma 1. We proceed by cases on the last rule used:

Case T-Var $e = x'$. If $x' \neq x$ then $x' : T \in \Gamma'$ and $\Gamma' \vdash_e x' : T$. If $x' = x$ then $e[v/x] = v$ and $T_x' \leq T_x$ by our assumptions.

Case T-New $e = \texttt{(}C\ e_1\ \ldots\ e_n\texttt{)}$ with $T = C$. By the induction hypothesis, for all $i \in 1..n$ $\Gamma \vdash_e e_i[v/x] : T_i''$ for some $T_i'' \leq T_i'$ with $T_i'' \leq T_i$ by transitivity of $\leq$. So $\Gamma \vdash_e \texttt{(}C\ e_1[v/x]\ \ldots\ e_n[v/x]\texttt{)} : C$.

Case T-TRAV $e = \texttt{(traverse}\ e_0\ F\texttt{)}$. By the induction hypothesis, $\Gamma' \vdash_e e_0[v/x] : T_0'$ for some $T_0' \leq T_0$. By lemma 2 the traversal result is $\Gamma; \emptyset \vdash_{\mathcal{T}} \langle T_0', F \rangle : T'$ for some $T' \leq T$, so $\Gamma' \vdash_e \texttt{(traverse}\ e_0[v/x]\ \ F[v/x]\texttt{)} : T'$ and $T' \leq T$.

Case T-APPLY $e = \texttt{(apply}\ f\ v_0\ v_1\ \ldots\ v_n\texttt{)}$ with $f = \texttt{[(}T_0\ \ldots\ T_n\texttt{)}\ \texttt{(}x_0\ \ldots\ x_n\texttt{)}\ e_0\texttt{]}$. If $x \in \overline{x_i}$ then substitution has no effect and the result is T. If $x \notin \overline{x_i}$ then by the induction hypothesis, $(\Gamma', x_0 : T_0, \ldots, x_n : T_n) \vdash_e e_0[v/x] : T'$ for some $T' \leq T$.

Case T-DISPATCH $e = \texttt{(dispatch}\ F\ v_0\ \ e_1\ \ldots\ e_n\texttt{)}$. By the induction hypothesis, for all $i \in 1..n \quad \Gamma \vdash_e e_i[v/x] : T_i''$ and $T_i'' \leq T_i'$. By lemma 1 we know that $\mathit{possibleFs}(F, (C\ T_1''\ \ldots\ T_n'')) \subseteq \mathit{possibleFs}(F, (C\ T_1'\ \ldots\ T_n'))$, so there exists a type $T' \leq T$ such that for all $f \in \mathit{possibleFs}(F, (C\ T_1''\ \ldots\ T_n'')) \quad \Gamma \vdash_F f : T_f$ with $T_f \leq T'$. The result is $\Gamma \vdash_e \texttt{(dispatch}\ F\ v_0\ \ e_1[v/x]\ \ \ldots\ \ e_n[v/x]\texttt{)} : T'$. By the implication property of *covers*:

$$\mathit{covers}(F, (C\ T_1'\ \ldots\ T_n')) \Rightarrow \mathit{covers}(F, (C\ T_1''\ \ldots\ T_n''))$$

So our *covers* premise still holds.

Cases of substitution within functions/sets follow directly from our induction hypothesis.

□

Well-typed contexts means that recomposition of an expression and a context also preserves the type of the outer context. The lemma is similar to substitution.

Lemma 4 (Well-Typed Contexts) *Substituting a closed, well-typed expression, which is a subtype of the original, into the hole of a context preserves the outer context's type.*

For any closed expressions e, e', and context E, if $\emptyset \vdash_e e : T$, $\emptyset \vdash_e e' : T'$ with $T' \leq T$, and $\Gamma \vdash_e E[e] : T_0$, then $\Gamma \vdash_e E[e'] : T_0'$ for some $T_0' \leq T_0$.

Proof: By induction on the structure of the outermost context E and the typing derivation of $E[e]$.

Case $E = [\,]$. Follows from our assumptions, since $\emptyset \vdash_e e : T$, $\emptyset \vdash_e e' : T'$ and $T' \leq T$.

Case $E = \texttt{(}C\ v\ \ldots\ E'\ e_i\ \ldots\ \texttt{)}$. By the induction hypothesis, replacing e with e' in E' maintains the premises of T-NEW. The result type remains C.

Case $E = \texttt{(traverse}\ E'\ F\texttt{)}$. In T-TRAV, by the induction hypothesis and lemma 2, the traversal of $E'[e']$ with the same function set, F, must return a subtype of the traversal result type of $E'[e]$.

Case $E = \texttt{(dispatch}\ F\ v_0\ v\ \ldots\ E'\ e_i\ \ldots\ \texttt{)}$. In T-DISPATCH, by the induction hypothesis and lemma 1, the *possible* functions with $E'[e']$ instead of $E'[e]$ remains a subset, and must unify to a common supertype, which is a subtype of that obtained with $E'[e]$. The premise of *covers* also holds, with proof similar to substitution.

□

We can now state the first half of our soundness theorem: *preservation.*

Theorem 1 (Preservation) *Reduction preserves an expression's type.*

If $\Gamma \vdash_e E[e] : T$ and $E[e] \rightarrow E[e']$ then $\Gamma \vdash_e E[e'] : T'$ with $T' \leq T$.

Proof: Using lemma 4, our proof reduces to showing that our individual reductions preserve type. That is, we must show that $\emptyset \vdash_e e : T_e$ and $e \rightarrow e'$ *implies* $\emptyset \vdash_e e' : T'_e$ and $T'_e \leq T_e$. If we prove this implication, then by lemma 4, it is true that $\Gamma \vdash_e E[e'] : T'$ for some $T' \leq T$. We proceed by showing the implication holds for each of our reduction rules.

Case If R-Apply applies . Follows from substitution, lemma 3.

Case If R-Dispatch applies . Since the function selected, f, is one of the *possible* functions ($choose(F, (T_0 \ldots T_n)) \in possibleFs(F, (T_0 \ldots T_n))$), f is used in the premise of our typing rule (T-Dispatch). Proof follows immediately, as the rule requires that the return types of all *possible* functions be a subtype of the assigned type.

Case If R-Trav applies . The typing derivation of the traversal expression includes both a sub-derivation for the value to be traversed, $e_0 = (C\ v_1 \ldots v_n)$, and a traversal judgment based on the definition of C. By the first sub-derivation, we know that $\emptyset \vdash_e v_i : C_i$ for some $C_i \leq T_i$ where T_i is from the definition of C. The traversal typing for each field type, T_i, contains as a sub-derivation a typing rule for C_i, which can be used to construct a traversal derivation for the expanded `traverse` term.

By lemma 1 the *possible* functions to be used in the typing derivation of the dispatch expression are a subset of those used in the traversal rule for C, and likewise unify to a common supertype (T'_e), which is a subtype of the original, T_e. The use of *covers* in the traversal rule (T-CTrav) for C remains the same for `dispatch`.

□

While preservation itself is interesting, as important is the preservation of function set *completeness*: if a traversal expression is well typed, then *covers* holds after traversal reduction, R-Trav.

Soundness now rests on progress, which in turn relies on function selection succeeding. While preservation says that our *possible* functions return the right types, progress requires that there exists a possible function for well-typed traversals.

Theorem 2 (Progress) *A closed, well-typed expression is either a value, or can be reduced, i.e., is never stuck.*

For any expression e such that $\emptyset \vdash_e e : T$*, then either e is a value, or* $e = E[e']$ *and* $E[e'] \rightarrow E[e'']$.

Proof: By induction on the structure e.

Case $e = x$. This case is impossible since e is closed.

Case $e = (C\ e_1 \ldots e_n)$. If all e_i are values, then e is also a value. Otherwise, by the induction hypothesis, we can decompose e into $E[\,e'\,]$ with $E = (C\ v \ldots E'\ e_i \ldots)$, for for the first non-value and some E', and e' can be reduced.

Case $e =$ `(traverse` $e_0\ F$`)` . If e_0 is a value, then R-Trav applies. Otherwise, by the induction hypothesis we can decompose e into $E[\,e'\,]$ with $E =$ `(traverse` $E'\ F$`)`, for some E', and e' can be reduced.

Case $e =$ `(apply` $f\ v_0\ v_1 \ldots v_n$`)` with $f =$ `[(`$T_0 \ldots T_n$`) (`$x_0 \ldots x_n$`)` e_0 `]` . R-Apply is immediately applicable.

Case $e =$ `(dispatch` $F\ v_0\ e_1\ \ldots\ e_n$`)` . If not all e_i are values, then by the induction hypothesis we can decompose e into $E[\,e'\,]$ with $E =$ `(dispatch` $F\ v_0\ v\ \ldots\ E'\ e_i\ \ldots$ `)`, for some E', and e' can be reduced.

If all e_i are values, then R-DISPATCH applies. Because e is well-typed, it must be the case that $\emptyset \vdash_e v_0 : C_0$ and for all $i \in 1..n\ \emptyset \vdash_e e_i : C_i$. Our premises require that *covers*$(F, (C_0\ C_1\ \ldots\ C_n))$, which matches our necessary property of *covers*: *possibleFs*$(F, (C_0\ C_1\ \ldots\ C_n)) \neq ()$.

□

With preservation and progress we can now state and prove our soundness theorem.

Theorem 3 (Type Soundness) *A closed, well-typed expression e is either a value, or can be reduced to another well-typed expression.*

For any expression e such that $\emptyset \vdash_e e : T$, then e is either a value of type T, or $e \rightarrow e'$ and $\emptyset \vdash_e e' : T'$, with $T' \leq T$.

Proof: By PROGRESS, e is either a value or can be reduced. By PRESERVATION, if e reduces to e', then $\emptyset \vdash_e e' : T'$ and $T' \leq T$.
□

Wright and Felleisen [37] refer to this theorem as *strong* soundness, since reduction is never stuck and the type of the result is correctly predicted. The standard form of type soundness is what they call *weak* soundness:

For any well-typed expression, e, if $e \rightarrow e'$, then e' is not *stuck*.

Proof is immediate from Theorem 3, since a stuck `dispatch` expression is not a value.

7 Related Work

Our view of generic programming is influenced by many different projects ranging from generalized folds [31,28], light-weight functional approaches [24,25,21], and visitors [20, 9] to full-fledged generic programming [16,15], attribute grammars [19], and multi-methods [8,2].

The notion of traversals that we use is closest to Sheard and Fegaras' work on generalized folds [31], drawing inspiration from Meijer *et al.* [28]. Our traversal function is similar to Sheard's general functor, E, which he uses to implement fold, though we group functions in a set, rather than passing them as arguments. Our single `traverse` function takes the place of a number of very complex functions, one for each value constructor. The benefits of a single traversal function become more pronounced when dealing with mutually recursive types, where fold functions can become difficult to manage. Rather than fixing calls to a particular function argument, our type-based dispatch allows function sets to abstract multiple cases into one, or overload a case based on argument types. Our traversal also goes a bit further by supporting function set extension, contexts, and control.

Library and combinator approaches by Lämmel *et al.* [24,25] and the *Scrap Your Boilerplate* series of papers [21–23] support solutions to similar problems using traversal combinators and Haskell's type classes [17]. When the typical *everywhere* traversal is not sufficient, these solutions control recursion using a one-step traversal. Type safety is provided by definition within their implementation language. Our external library

approach provides significantly more flexibility but requires us to formulate soundness separately. Work on more heavy-weight generic programming [27,16] can be used to write traversal functions based on the shape of data constructors, but provide only limited support for function specialization and control.

Our traversals and contexts are similar to an implementation of *attribute grammars* [19]. In Knuth's original description, each attribute is defined by functions over the productions of a context free grammar. In AP-F, `abstract` and `concrete` definitions are similar to non-terminals of a context free grammar.[3] In AP-F, traversing a data structure instance using a function set corresponds to the evaluation of an attribute's functions over a derivation of the grammar. The first function set passed to the extended `traverse` form corresponds to a *synthesized* attribute, with contexts corresponding to an *inherited* attribute. Knuth mentions that attribute grammars can be used to compute arbitrary functions over a derivation of a grammar, and later papers discuss the complexity of checking attribute dependencies and evaluating functions [13]. In AP-F Scheme functions can can be arbitrarily complex, but function sets without hand-coded recursion correspond to *one-pass* (or *one-visit*) attribute grammars, that can be evaluated left-to-right in a single traversal [4]. Our traversal control also allows the application of functions to be limited to a particular portion of the data structure, though it may be possible to encode similar ideas within attribute functions.

AP-F's multiple-dispatch and checking of function sets and structures is related to work on static checking of multi-methods [29]. Though Millstien and Chambers are more concerned with balancing modularity and expressiveness, they do focus on eliminating problems associated with multi-method overloading. Agrawal *et al.* [8] focus on a simple model of dynamic dispatch and reduce the type checking problem to (1) checking the consistency of overlapping signatures, and (2) confirming that call sites are correct. Chambers and Leavens [2] eliminate overloading ambiguities by requiring that every combination of argument types have a *most specific* method signature to dispatch to. Their goal is to catch such errors at compile-time, rather than raising a runtime *method ambiguous* exception. AP-F dispatch is more like CLOS [32], in that we have an implicit total ordering of applicable method signatures (including shorter signatures), which avoids ambiguities. We are more interested in the possible return types during traversal when using a given function set, and making sure that every case has an applicable function.

Our model, type system, and soundness builds on simpler ideas from an earlier paper [6] and has been influenced by work on aspect-oriented semantics [36]. Though we maintain a functional approach, our original motivations for separating traversal from other concerns stems from adaptive programming [26] and other visitor-based approaches [20,34,35]. More recent functional visitor approaches [9,30] have focused on safety and modularization, but can be mainly categorized as design patterns whereas our aim is to provide a useful library for writing flexible and generic traversal-based functions.

8 Conclusion

We have introduced an approach to traversal-based generic programming, AP-F, and a library implementation in Scheme. Instead of requiring programmers to hand write

[3] AP-F actually uses the definitions to automatically construct a parser.

structural recursion our `traverse` form adapts to datatypes. Our approach uses a depth-first traversal that handles mutually recursive structures without programmer effort, supports non-compositional functions using traversal *contexts*, can be guided/limited by *control* expressions. The traversal uses a set of functions to fold recursive results and to update context, with functions selected by a type-based multiple dispatch. Our multiple dispatch provides programmers with much of the flexibility of hand-written functions while also supporting extension, abstraction, and overloading of functions. In order to show that this flexibility is sound and verifiable, we introduced a simplified model of our essential features: traversal, function sets, and dispatch. We presented a type system, and a proof of type soundness, showing that type-correct programs are free from runtime dispatch errors. This allows us to verify that particular traversals, data structures, and function sets are safe, not only for our dynamic Scheme implementation without redefinitions, but also for our other AP-F implementations in statically typed languages.

Acknowledgements This work has been supported in part by a grant from Novartis. We would like to thank Riccardo Puccella for comments on an earlier version of this paper, and Jesse Tov for giving our presentation at Mitch-Fest and for his helpful type system suggestions. We would also like to thank the editors and anonymous referees for their valuable feedback and advice.

References

1. PLT Scheme. Website, 2009. `http://www.plt-scheme.org/`.
2. R. Agrawal, L. G. Demichiel, and B. G. Lindsay. Static type checking of multi-methods. In *OOPSLA '91*, pages 113–128, New York, NY, USA, 1991. ACM.
3. R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming – an introduction. In S. Swierstra, J. Oliveira, and P. Henriques, editors, *Advanced Functional Programming*, number 1608 in LNCS, pages 28–115. Springer, 1999.
4. G. V. Bochmann. Semantic evaluation from left to right. *Commun. ACM*, 19(2):55–62, 1976.
5. B. Chadwick. Algorithms in DemeterF. `http://www.ccs.neu.edu/home/chadwick/files/algo.pdf`, May 2009.
6. B. Chadwick and K. Lieberherr. A type system for functional traversal-based aspects. In *AOSD 2009, FOAL Workshop*, New York, NY, USA, 2009. ACM.
7. B. Chadwick and K. J. Lieberherr. Weaving generic programming and traversal performance. In J.-M. Jézéquel and M. Südholt, editors, *AOSD '10*, pages 61–72, New York, NY, USA, 2010. ACM.
8. C. Chambers and G. T. Leavens. Typechecking and modules for multimethods. *TOPLAS*, 17(6):805–843, November 1995.
9. 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.
10. O. Danvy. From reduction-based to reduction-free normalization. In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, *Advanced Functional Programming, Sixth International School*, number 5382 in LNCS, pages 66–164, Nijmegen, The Netherlands, May 2008. Springer.
11. O. Danvy and K. Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Info. Proc. Let.*, 106(3):100–109, 2008.
12. O. Danvy and U. P. Schultz. Lambda-dropping: transforming recursive equations into programs with block structure. *Theor. Comput. Sci.*, 248(1-2):243–287, 2000.
13. J. Engelfriet and G. Filé. Passes, sweeps, and visits in attribute grammars. *J. ACM*, 36(4):841–869, 1989.
14. J. Gibbons. Datatype-generic programming. In R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *LNCS*. Springer-Verlag, 2007.

15. R. Hinze. A new approach to generic functional programming. In *POPL '99*, pages 119–132, New York, NY, USA, 1999. ACM.
16. P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL'97*, pages 470–482. ACM, 1997.
17. S. P. Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003.
18. R. Kelsey, W. Clinger, and J. Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.
19. D. E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.
20. S. Krishnamurthi, M. Felleisen, and D. P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *ECOOP '98*, pages 91–113, London, UK, 1998. Springer-Verlag.
21. R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI '03*, pages 26–37, New York, NY, USA, 2003. ACM.
22. R. Lämmel and S. Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *ICFP '04*, pages 244–255. ACM, 2004.
23. R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ICFP '05*, pages 204–215. ACM, Sept. 2005.
24. R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *PADL '02*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, Jan. 2002.
25. R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In J. Jeuring, editor, *WGP '00*, pages 46–59, July 2000.
26. K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.
27. 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.
28. E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *FPCA '91*, volume 523, pages 124–144. Springer-Verlag, Berlin, 1991.
29. T. D. Millstein and C. Chambers. Modular statically typed multimethods. In *ECOOP '99*, pages 279–303, London, UK, 1999. Springer-Verlag.
30. B. C. Oliveira. Modular visitor components. In *ECOOP '09*, pages 269–293, Berlin, Heidelberg, 2009. Springer-Verlag.
31. T. Sheard and L. Fegaras. A fold for all seasons. In *FPCA '93*, pages 233–242. ACM, New York, NY, USA, 1993.
32. G. L. Steele, Jr. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990.
33. A. Stump. Directly reflective meta-programming. *Higher Order Symbol. Comput.*, 22(2):115–144, 2009.
34. T. VanDrunen and J. Palsberg. Visitor-oriented programming. *FOOL '04*, January 2004.
35. J. Visser. Visitor combination and traversal control. In *OOPSLA '01*, pages 270–282, New York, NY, USA, 2001. ACM.
36. M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS*, 26(5):890–910, 2004.
37. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.