What Functional Programmers can Learn from the Visitor Pattern

Technical Report, March 2003

Konstantin Läufer
Department of Computer Science
Loyola University Chicago
6525 North Sheridan Road
Chicago, Illinois 60626, USA
laufer@acm.org

ABSTRACT

This paper explores the practical potential for enhancing code reuse in functional languages by leveraging techniques and experiences from object-oriented programming. Since data types in typed functional languages do not support adding new variants, programming techniques derived from the object-oriented Visitor pattern can be readily applied to recursive functions on trees in functional languages.

Specifically, we demonstrate how to define recursive functions on trees in such a way that they can be extended without modification through an inheritance-like mechanism. Furthermore, we demonstrate how the extensions themselves can be made highly reusable through a mechanism akin to mixin-based inheritance, which allows an extension to be applied to any suitable recursive function. These mechanisms combine well with the usual higher-order approach to reuse in functional languages.

Although general encodings of inheritance in functional languages are not novel, such encodings lead to a particularly simple and elegant presentation when limited to recursive functions on trees. Because of their simplicity, we believe that these techniques have the potential to enhance code reuse in the practice of functional programming without requiring additional tool support.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented Programming*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*

General Terms

Design, Languages

Keywords

Algebraic data types, Composite pattern, code reuse, higher-order functions, inheritance, mixins, recursive functions, reusable components, reuse, traversal, trees, Visitor pattern

1. INTRODUCTION

Trees are widely used to represent hierarchical information such as documents or programs. In typed functional languages such as ML [12] and Haskell [10], trees are directly supported in the form of recursive algebraic data types. Data types representing trees are comprised of at least one *variant* for *(interior) nodes* and at least one variant for *leaf nodes*. Once a tree type has been specified, it is easy to define recursive functions on trees. Typically, such functions have at least one branch for each variant of the data type. They traverse the tree recursively by applying themselves to subtrees until a leaf is reached and compute the result on the way back to the root of the tree. In functional languages, *code reuse* is usually achieved by relying on *higher-order functions* to compose behaviors [1].

In object-oriented languages, trees are typically implemented using the *Composite pattern* [9], in which an interface or abstract superclass represents the tree data type and the subclasses represent the variants. Recursive functions over the resulting tree structures are provided using the *Visitor pattern* [9], which makes it easy to add new functions on trees without changes to the data type but makes it hard to add new variants to the data type. In object-oriented languages, code reuse is usually achieved through *delegation* and *inheritance* [11, 14]. Recently, visitor combinators [17] have emerged as a way to improve reuse of imperative Visitor code.

This paper explores the practical potential for enhancing code reuse in functional languages by leveraging techniques and experiences from object-oriented programming. Since data types in functional languages such as Standard ML and Haskell do not support adding new variants anyway, programming techniques derived from the Visitor pattern can be readily applied to recursive functions on trees in functional languages. Furthermore, because all visitors on a particular tree type usually implement the same interface, the absence of subtype polymorphism in the aforementioned languages is not a problem.

Specifically, we demonstrate how to define recursive functions on trees in such a way that they can be extended without modification through an inheritance-like mechanism. Furthermore, we demonstrate how the extensions themselves can be made highly reusable through a mechanism akin to *mixin-based* inheritance [2, 3, 8, 7, 15] which allows an extension to be applied to any suitable recursive function. These mechanisms combine well with the usual higher-order approach to reuse in functional languages. They also apply to other recursive data structures, such as lists, that can be encoded as trees.

Although general encodings of inheritance in functional languages are not novel [16, 5, 4], such encodings lead to a particularly simple and elegant presentation when limited to functions on recursive data structures. Because of their simplicity, we believe that these techniques have the potential to enhance code reuse in the practice of functional programming without requiring additional tool support.

While the examples in this paper have been implemented in Standard ML, other higher-order typed (HOT) languages could have been used as well. The potential benefit of the techniques presented here is expected to be greater in the context of eager languages rather than lazy languages because recursive functions in lazy languages do not usually require the same enhanced tree traversal control as eager languages.

The rest of this paper is organized as follows. Section 2 presents a simple tree example, and Section 3 explains why ordinary recursive functions on trees are not extensible without modification. Section 4 shows how a simple encoding of inheritance can make recursive functions extensible. Section 5 argues why the higher-order approach of separating tree traversal from operations is not always sufficient. Section 6 demonstrates how reuse can be enhanced by encoding mixins. Section 7 discusses the presence of additional arguments to recursive functions, and Section 8 describes how to handle mutual recursion. Section 9 applies the various techniques to an extended example of abstract syntax trees for a simple programming language.

2. RECURSIVE FUNCTIONS ON TREES

In this section, we first present a simple polymorphic tree type along with a typical recursive function on such trees. In our tree type, the type variable `'a` represents the polymorphic element type. This tree type has two variants, a *leaf* holding a data value, and an (interior) *node* with a left and a right subtree.

```
datatype 'a Tree = Leaf of 'a
                 | Node of 'a Tree * 'a Tree
```

Given this type, we can now define trees such as this one, whose element type is `int`.

```
val t = Node(
          Leaf 2,
          Node(
            Leaf 8,
            Leaf 5))
```

Next, we define a simple recursive function that sums the numeric values at the leaves of a tree of integers.

```
fun csum (Leaf i)     = i
  | csum (Node(l, r)) = csum l + csum r

- csum;
val it = fn : int Tree -> int
```

We can apply this function to any argument of type `int Tree`, for example:

```
- csum (Leaf 3);
val it = 3 : int
- csum t;
val it = 15 : int
```

3. RECURSIVE FUNCTIONS ARE NOT EXTENSIBLE

The problem with ordinary recursive functions on trees such as `csum` above is that they are closed to any further extension without changing their original definition, hence the choice of name `c(losed)sum`. Therefore, such functions are not useful as reusable components.

To illustrate this limitation further, let us attempt to extend `csum` in such a way that it adds only those leaves into the result whose integer data is even. The idea is that odd integer values are replaced by zero. Interior nodes are processed as in the original definition of `csum`.

```
fun csumEven (t as (Leaf i)) =
    let val x = csum t in
        if x mod 2 = 0 then x else 0
    end
  | csumEven t = csum t

- csumEven;
val it = fn : int Tree -> int
```

This extended function works fine on leaves:

```
- csumEven (Leaf 4);
val it = 4 : int
- csumEven (Leaf 5);
val it = 0 : int
```

However, it fails for more complex trees:

```
- csumEven t;
val it = 15 : int
```

The correct value for the sum of the even leaf values of `t` is `10`.

What went wrong? The problem is that `csumEven` passes interior nodes on to the original `csum`, which invokes itself recursively instead of `csumEven`. Consequently, it is `csum` that processes the leaves and counts both even and odd values.

4. INHERITANCE: MAKING RECURSIVE FUNCTIONS EXTENSIBLE

To turn a recursive function such as `csum` into a reusable building block, we can rewrite it with reuse in mind. Specifically, we introduce an additional argument `this` to denote the function that results after applying any extensions. We will apply that function recursively, instead of hard-coding recursive calls to `csum`.

```
fun sum this (Leaf i)     = i
  | sum this (Node(l, r)) = this l + this r

- sum;
val it = fn : (int Tree -> int) -> int Tree -> int
```

This version of `sum` is a higher-order function that we can no longer apply directly to trees. Instead, we must first "tie the knot" by forming the fixpoint of `sum` over the argument `this`. Then we can apply the resulting function to the tree to achieve the expected result:

```
fun nsum t = sum nsum t

- nsum;
val it = fn : int Tree -> int
- nsum t;
val it = 15 : int
```

By defining the following call-by-value fixpoint operator on functions, we no longer need to define an explicit fixpoint version of each function. This operator takes a higher-order function and returns its fixpoint on the first argument.

```
fun new f t = f (new f) t

- new;
val it = fn : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
- new sum;
val it = fn : int Tree -> int
- new sum t;
val it = 15 : int
```

But what do we gain from this extra pain? We are now able to reuse extensible functions unchanged as the basis for extension. Functions such as `sum` can be thought of as visitor classes with a single method whose branches correspond to the per-variant methods in the Visitor pattern; the fixpoint formation required before using the function can be viewed as instantiation.

Extended functions *inherit* the behavior from the original function and need to specify only what is different from the original behavior. When the original function is applied recursively, the extended behavior gets applied; this corresponds to *dynamic method dispatch*. While class-based object-oriented languages support inheritance and dynamic method dispatch directly, allowing Visitor classes to be extended easily, we have to encode this capability in the form of the additional `this` argument.

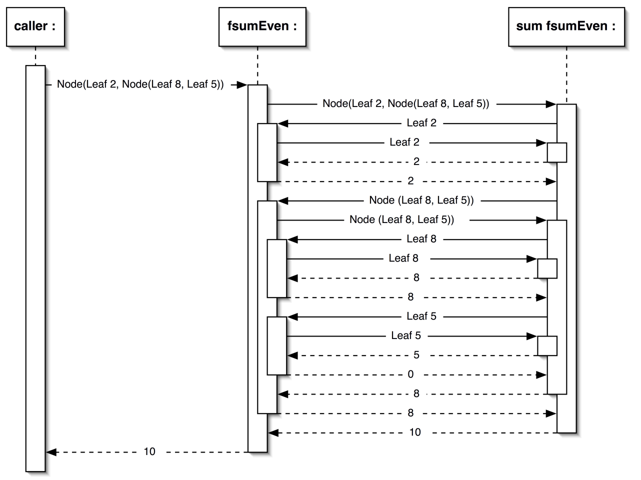

Figure 1: Sequence diagram of the evaluation of `fsumEven t`.

Compared to the general, imperative encoding presented by Thorup and Tofte [16], our restriction to visitor classes leads to this much simpler, applicative encoding of objects and inheritance. It seems practical to use this encoding without additional tool support.

The following extended version of `sum` for adding only the even leaf data values, `fsumEven`, is defined similarly to `csumEven` above. The difference is that `fsumEven` invokes `sum` instead of `csum`, passing itself as the actual argument for `this`, the argument for the resulting extended function.

```
fun fsumEven (t as (Leaf _)) =
    let val x = sum fsumEven t in
        if x mod 2 = 0 then x else 0
    end
  | fsumEven t = sum fsumEven t

- fsumEven;
val it = fn : int Tree -> int
```

This function behaves as expected. When processing an interior node, the inherited behavior from `sum` recurses into the tree and invokes its `this` argument, `fsumEven`, on the subtrees. When reaching a leaf, `fsumEven` thus processes the leaf and replaces any odd value with zero. The UML [13] sequence diagram shown in Figure 4 illustrates this process. Execution time proceeds from top to bottom. Solid arrows represent calls, and their labels indicate actual argument values; dashed arrows represent returns, and their labels indicate return values. The stacked activation boxes represent (directly or, as is the case here, indirectly) recursive function invocations.

```
- fsumEven (Leaf 4);
val it = 4 : int
```

```
- fsumEven (Leaf 5);
val it = 0 : int
- fsumEven t;
val it = 10 : int
```

However, since its only argument is the tree, `f(inal)sumEven` is not extensible any further, as its name indicates.

To allow future extension, an extended function must also accept a `this` argument representing the function that results after applying any further extensions. This requires all extensions to take a `this` argument. An extensible version of `fsumEven` can be written as follows:

```
fun sumEven this (t as (Leaf _)) =
    let val x = sum this t in
        if x mod 2 = 0 then x else 0
    end
  | sumEven this t = sum this t

- sumEven;
val it = fn : (int Tree -> int) -> int Tree -> int
```

As is the case with `sum`, we must first apply the fixpoint operator `new` before passing the tree argument. The function then behaves as desired:

```
- new sumEven;
val it = fn : int Tree -> int
- new sumEven (Leaf 4);
val it = 4 : int
- new sumEven (Leaf 5);
val it = 0 : int
- new sumEven t;
val it = 10 : int
```

Even better, we can apply further extensions to `sumEven` without changing its definition. The following function sums all even leaf data values that are greater than or equal to the threshold `y`.

```
fun sumEvenAtLeast y this (t as (Leaf _)) =
    let val x = sumEven this t in
        if x >= y then x else 0
    end
  | sumEvenAtLeast y this t = sumEven this t

- sumEvenAtLeast;
val it = fn : int -> (int Tree -> int) -> int Tree -> int
- sumEvenAtLeast 5;
val it = fn : (int Tree -> int) -> int Tree -> int
```

Once this function receives its threshold argument, we can form the fixpoint and apply it to trees as usual.

```
- new (sumEvenAtLeast 5);
val it = fn : int Tree -> int
- new (sumEvenAtLeast 5) (Leaf 4);
val it = 0 : int
- new (sumEvenAtLeast 5) (Leaf 5);
val it = 0 : int
- new (sumEvenAtLeast 5) (Leaf 6);
val it = 6 : int
- new (sumEvenAtLeast 5) t;
val it = 8 : int
```

In summary, we can describe the general idiom for converting a closed recursive function `cf : t -> r` to an extensible recursive function `f : (t -> r) -> (t -> r)` as follows. `t` is a polymorphic instance of a recursive data type `Type` type with n variants. `f1` through `fn` are type expressions possibly involving the polymorphic type variables `'a1` through `'ak`. `C1` through `Cn` are contexts not containing `cf`. The fixpoint of `f` over its first argument can be applied to a value of type `Type` (with appropriate type arguments).

```
datatype ('a1...'ak) Type = Variant1 of of f1['a1...'ak]
                          | ...
                          | Variantn of of fn['a1...'ak]

fun cf (Variant1(x1, ..., xm1) = C1[cf]
  | ...
  | cf (Variantn(x1, ..., xmn) = Cn[cf]

fun f this (Variant1(x1, ..., xm1) = C1[this]
  | ...
  | f this (Variantn(x1, ..., xmn)) = Cn[this]
```

In the transformation of `csum` to `sum` above, the general idiom is applied as follows:

```
k = 1
n = 2
f1['a] = 'a
f2['a] = 'a Tree * 'a Tree
m1 = 1
m2 = 2
C1[f] = i
C2[f] = f l + f r
```

5. WHY TREEFOLD IS NOT ENOUGH

At this point or earlier, most functional programmers have wondered why the familiar higher-order `fold` function on trees does not satisfy all of the reuse needs motivated above. In this approach, the `fold` function is responsible for traversing the tree. It takes a tuple of functions, one for each variant of the tree type, which it uses to compute the result for the current node from the partial, recursive results for the subtrees. The functions in the tuple usually know nothing about the tree structure or its traversal.

Specifically, the `fold` method for our simple tree type can be defined as follows:

```
fun fold (ops as (f, _)) (Leaf i)     = f i
  | fold (ops as (_, g)) (Node(l, r)) =
    g(fold ops l, fold ops r)

- fold;
val it = fn :
  ('a -> 'b) * ('b * 'b -> 'b) -> 'a Tree -> 'b
```

A pair of functions to sum the leaf values of a tree is given here. Whenever the `fold` method is processing a leaf, it applies the first, identity function to the leaf data; whenever it is processing a node, it recursively applies itself to the two subtrees and adds up the two partial results.

```
val sumOps = (fn x => x, op +)
```

```
- sumOps;
val it = (fn,fn) : ('a -> 'a) * (int * int -> int)
```

This combination of functions behaves as expected:

```
- fold sumOps;
val it = fn : int Tree -> int
- fold sumOps t;
val it = 15 : int
```

Like the extensible function approach, the treefold approach allows functions to act as reusable components that can be combined without modifying their definition. To take this approach one step further, we can define an point-wise extension operator that allows extensions to be applied to arguments to fold. This operator is right-associative because extensions are applied from right to left.

```
infixr O
fun op O ((f', g'), (f, g)) =
    (fn i      => f' i (f i),
     fn (x, y) => g' (x, y) (g (x, y)))
```

We can now define extensions that result in adding only even leaf data values and only those that exceed a given threshold. The additional arguments `i` and `(x, y)`, respectively, are required for generality in case the extensions want to consider the original partial results as well as those produced by the tuple of functions to be extended.

```
val evenOps =
    (fn i      => fn r => if r mod 2 = 0 then r else 0,
     fn (x, y) => fn r => r)

fun atLeastOps x =
    (fn i      => fn r => if r >= x then r else 0,
     fn (x, y) => fn r => r)

- evenOps;
val it = (fn,fn) :
  ('a -> int -> int) * ('b * 'c -> 'd -> 'd)
- atLeastOps;
val it = fn :
  int -> ('a -> int -> int) * ('b * 'c -> 'd -> 'd)
```

These tuples of functions can be combined to achieve the same effect as the `sumEvenAtLeast` function defined above.

```
- fold ((atLeastOps 5) O evenOps O sumOps);
val it = fn : int Tree -> int
- fold ((atLeastOps 5) O evenOps O sumOps) (Leaf 4);
val it = 0 : int
- fold ((atLeastOps 5) O evenOps O sumOps) (Leaf 5);
val it = 0 : int
- fold ((atLeastOps 5) O evenOps O sumOps) (Leaf 6);
val it = 6 : int
- fold ((atLeastOps 5) O evenOps O sumOps) t;
val it = 8 : int
```

We observe that the treefold approach hard-codes the traversal strategy. This is not a problem as long as a fixed set of traversal strategies is sufficient to achieve the desired result. However, in some cases, traversal decisions depend on the specific structure or data of the tree argument. As we will see in the extended example in Section 9, this is frequently the case with trees representing programs. For example, when processing a node representing a if expression, the "then" branch of the statement should be processed only if the condition evaluates to `true`; otherwise, processing it might lead to an exception.

Although the following example is somewhat contrived, it does illustrate this issue in conjunction with our simple tree type. The purpose of this function is to sum up all leaf data values in the tree in such a way that the right subtree of a node is ignored if the recursive result for the left subtree is even. This cannot be achieved using a fixed traversal strategy such as `fold`.

```
fun sumIgnoreRightIfLeftEven this (Node(l, r)) =
    let val x = sum this l in
        if x mod 2 = 0 then x else x + sum this r
    end
  | sumIgnoreRightIfLeftEven this t = sum this t

- sumIgnoreRightIfLeftEven;
val it = fn : (int Tree -> int) -> int Tree -> int
- new sumIgnoreRightIfLeftEven t;
val it = 2 : int
- new sumIgnoreRightIfLeftEven (Node(Leaf 3, Leaf 5));
val it = 8 : int
```

It should be noted that this problem disappears when lazy evaluation is used. Using `fold` in a lazy language, the right subtree would be processed only when needed anyway, and one could write the following function tuple to achieve the effect of the `sumIgnoreRightIfLeftEven` function:

```
val ignoreRightIfLeftEvenOps =
    (fn i      => fn r => r,
     fn (x, y) => fn r =>
       if x mod 2 = 0 then x else x + y)

- fold (ignoreRightIfLeftEvenOps O sumOps) t;
val it = 2 : int
- fold (ignoreRightIfLeftEvenOps O sumOps)
       (Node(Leaf 3, Leaf 5));
val it = 8 : int
```

6. MIXINS: MAKING EXTENSIONS REUSABLE

In this section, we will explore ways to make extensions such as those presented in Section 4 more reusable. A key limitation of those extensions is that they apply to a fixed, hard-coded original extensible function.

Instead, we would like to define extensions in such a way that they can be applied to any suitable extensible function. We can achieve this effect by abstracting over all recursive invocations of the original function. Revisiting the definition of `sumEven`, we observe that all such invocations are of the form `sum this`, enclosed between pairs of marker comments.

```
fun sumEven this (t as (Leaf _)) =
    let val x = (*-->*) sum this (*<--*) t in
```

```
        if x mod 2 = 0 then x else 0
    end
  | sumEven this t = (*-->*) sum this (*<--*) t
```

We now abstract over recursive invocations of `sum this` by replacing all such invocations with an additional argument `super` to refer to the original function to which this extension applies.

```
fun keepEven this super (t as (Leaf _)) =
    let val x = super t in
        if x mod 2 = 0 then x else 0
    end
  | keepEven this super t = super t

- keepEven;
val it = fn : 'a -> ('b Tree -> int) -> 'b Tree -> int
```

To apply the `keepEven` extension to `sum`, we first abstract over the `this` argument representing the extended function and then form the fixpoint.

```
- fn this => keepEven this (sum this);
val it = fn : (int Tree -> int) -> int Tree -> int
- new (fn this => keepEven this (sum this));
val it = fn : int Tree -> int
- new (fn this => keepEven this (sum this)) t;
val it = 10 : int
```

To provide additional motivation, let us consider another extensible function that multiplies all the leaf data values of a tree, and an extension that increments each leaf data value by a specified value.

```
fun prod this (Leaf i)     = i
  | prod this (Node(l, r)) = this l * this r

fun incBy x this super (t as (Leaf _)) = super t + x
  | incBy x this super t               = super t
```

The following examples show how `incBy` can be applied to different extensible functions.

```
- new prod t;
val it = 80 : int
- new (fn this => incBy 1 this (sum this)) t;
val it = 18 : int
- new (fn this => incBy 1 this (prod this)) t;
val it = 162 : int
```

In object-oriented terminology, abstract extensions that can be applied to different superclasses are called *mixins*. Mixins are not supported directly by most mainstream object-oriented languages but can be encoded using an enhanced version of the Decorator (Wrapper) pattern [9]. This encoding generally works well in conjunction with the Visitor pattern because the interface of the visitors does not change under extension.

Multiple extensions can be combined by chaining them and forming the fixpoint at the end. Since the resulting chains can become unwieldy because of an occurrence of `this` for each extension in the chain, it is convenient to define a function, `extend`, that allows a chain of extensions to be expressed as a list. As expected, `extend [] = new`.

```
fun extend extensions super t =
    let
        fun extend' []            super this t =
            super this t
          | extend' (ext :: exts) super this t =
            ext this (extend' exts super this) t
    in
        new (extend' extensions super) t
    end

- extend;
val it = fn
  : (('a -> 'b) -> ('a -> 'b) -> 'a -> 'b) list
    -> (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
```

The `extend` function takes a list of reusable extensions and an extensible function and returns an extended function that is ready to use on a tree. Multiple extensions can now be applied conveniently.

```
- extend [incBy 1, keepEven] sum t;
val it = 13 : int
- extend [keepEven, incBy 1] sum t;
val it = 6 : int
```

Of course, for additional reusability, the extensions themselves can be higher-order functions, such as the following `keep` extension, which takes a predicate and a replacement value to be used if the predicate is not satisfied.

```
fun keep p z this super (t as (Leaf _)) =
    let val x = super t in
        if p x then x else z
    end
  | keep p z this super t = super t

fun isEven i = i mod 2 = 0

fun isAtLeast x i = i >= x

- extend [keep (isAtLeast 5) 0, keep isEven 0] sum t;
val it = 8 : int
```

In summary, we can describe the general idiom for mixins as follows. Given an extensible function `f : (t -> r) -> (t -> r)`, a mixin on the type of `f` is a function `m : (t -> r) -> (t -> r) -> (t -> r)` (or of a more general type). Here, the first argument of type `t -> r` is the function that results after all extensions are applied to the original function; the second argument is the (possibly already extended) function `f` to be further extended using `m`. The fixpoint of `fn this => m this (f this) : (t -> r) -> (t -> r)` over `this` can be applied to a value of type `t`.

7. DEALING WITH ADDITIONAL ARGUMENTS

It is common for recursive functions to take additional arguments whose values change during the traversal of the tree [6]. The following (extensible) function for prettyprinting a tree illustrates this technique. The `prefix` argument represents the indentation level for the current node. Trees are printed in a LISP-like style.

```
fun printTree this prefix (Leaf i) = Int.toString i
  | printTree this prefix (Node(l, r)) =
    "(\n" ^
    prefix ^ "  " ^ (this (prefix ^ "  ") l) ^ "\n" ^
    prefix ^ "  " ^ (this (prefix ^ "  ") r) ^ "\n" ^
    prefix ^ ")"

- print (new printTree "" t);
(
  2
  (
    8
    5
  )
)
val it = () : unit
```

The `printNicer` extension adds the variant constructor names and commas in the right places to the printed tree so that it prints like its source-level definition. Observe that the extension also has the additional argument before the tree itself.

```
fun printNicer this super prefix (t as (Leaf i)) =
    "Leaf(" ^ (super prefix t) ^ ")"
  | printNicer this super prefix (t as (Node(l, r))) =
    "Node(\n" ^
    prefix ^ "  " ^ (this (prefix ^ "  ") l) ^ ",\n" ^
    prefix ^ "  " ^ (this (prefix ^ "  ") r) ^ "\n" ^
    prefix ^ ")"
```

Fortunately, applying suitable extensions to functions still works using the `extend` function defined in Section 6. The arguments of the extensions to the right of `this` and `super` must match the arguments of the original to the right of `this`.

```
- print (extend [printNicer] printTree "" t);
Node(
  Leaf(2),
  Node(
    Leaf(8),
    Leaf(5)
  )
)
val it = () : unit
```

The difference between the print functions described here and, for example, the `incBy` function from Section 6 is that the argument values of the print functions changes during the recursive traversal of the tree. Therefore, the `prefix` argument to these functions must not be applied until after the fixpoint is formed.

8. DEALING WITH MUTUAL RECURSION

Conceptually, on the basis of Bekic's theorem [18], we do not expect mutual recursion to cause any difficulties. In practice, it turns out that we can handle mutual recursion by introducing an additional argument for each function that participates in the mutual recursion.

The following example illustrates this issue. We first define a version of `csum` that consists of a pair of mutually recursive functions. We then make these functions extensible by introducing two `this` arguments, one for each function in the pair.

```
fun csum1 (Leaf i)      = i
  | csum1 (Node(l, r)) = csum2 l + csum2 r
and csum2 t = csum1 t

fun sum1 this1 this2 (Leaf i)      = i
  | sum1 this1 this2 (Node(l, r)) = this2 l + this2 r
fun sum2 this1 this2 t =  this1 t

- sum1;
val it = fn : 'a -> (int Tree -> int) -> int Tree -> int
- sum2;
val it = fn : ('a -> 'b) -> 'c -> 'a -> 'b
```

As before, we must first "tie the knot" by forming the fixpoint of the two functions over the two `this` arguments, and then we can apply the function `nsum1` to tree arguments.

```
fun nsum1 t = sum1 nsum1 nsum2 t
and nsum2 t = sum2 nsum1 nsum2 t

- nsum1;
val it = fn : int Tree -> int
- nsum1 t;
val it = 15 : int
```

To extend the function, we first apply the extension and then form the fixpoint. For example, we can apply the extension `keepEven` from Section 6 as follows:

```
fun nsumEven1 t =
    (keepEven nsumEven1 (sum1 nsumEven1 nsumEven2)) t
and nsumEven2 t =
    sum2 nsumEven1 nsumEven2 t

- nsumEven1;
val it = fn : int Tree -> int
- nsumEven1 t;
val it = 10 : int
```

9. EXTENDED EXAMPLE: AN INTERPRETER FOR A SIMPLE LANGUAGE

In this section, we present an interpreter for a simple imperative language as an extended example. This language has constants, variables, arithmetic expressions, assignment, sequences of statements, and while loops.

We start by defining the tree data type for abstract syntax trees (ASTs) in this language. Constants carry an integer value; variables have a display name and an integer reference; arithmetic expressions have two subexpressions; assignments have a left-hand side and a right-hand side; sequences have a list of subexpressions; finally, while loops have a condition and a body.

```
datatype Expr = Const of int
```

```
                | Var of string * int ref
                | Plus of Expr * Expr
                | Minus of Expr * Expr
                | Assign of Expr * Expr
                | Sequence of Expr list
                | While of Expr * Expr
```

Next, we provide a non-extensible version of an interpreter for programs represented as ASTs. As usual, the interpreter models constructs in the object language (Expr) using suitable constructs in the metalanguage (appropriately, ML).

```
fun ceval (Const i) = i
  | ceval (Var(_,x)) = !x
  | ceval (Plus(l,r)) = (ceval l) + (ceval r)
  | ceval (Minus(l,r)) = (ceval l) - (ceval r)
  | ceval (Assign(Var(_,x),r)) = ( x := ceval r ; !x )
  | ceval (Sequence(e :: es)) =
    ( ceval e ; ceval (Sequence es) )
  | ceval (Sequence(nil)) = 0
  | ceval (e as (While(c,b))) =
    if ceval c = 0 then 0 else ( ceval b ; ceval e )
```

Now we can define and evaluate programs in this language, such as the following example:

```
val x = Var ("x", ref 2)
val y = Var ("y", ref 3)
val r = Var ("r", ref 0)

val s = While(y,
              Sequence([
                        Assign(r, Plus(r, x)),
                        Assign(y, Minus(y, Const 1))
                        ])
              )
```

In a typical imperative language, the source code for this program would look as follows; the program multiplies `x` and `y`.

```
int x = 2; int y = 3; int r = 0;
while (y > 0) { r = r + x; y = y - 1; }
```

To execute the program `s`, we first evaluate `s` itself and then inspect the result variable `r`.

```
- ceval s;
val it = 0 : int
- ceval r;
val it = 6 : int
```

We now convert `ceval` to an extensible function called `eval` in the usual way, replacing recursive calls with calls to the additional argument `this`.

```
fun eval this (Const i) = i
  | eval this (Var(_,x)) = !x
  | eval this (Plus(l,r)) = (this l) + (this r)
  | eval this (Minus(l,r)) = (this l) - (this r)
  | eval this (Assign(Var(_,x),r)) =
    ( x := eval this r ; !x )
  | eval this (Sequence(e :: es)) =
    ( this e ; this (Sequence es) )
  | eval this (Sequence(nil)) = 0
  | eval this (e as (While(c,b))) =
    if this c = 0 then 0 else ( this b ; this e )
```

Having an extensible version of `eval` enables us to write some extensions to `eval` that instrument the evaluation process. In practice, such extensions may be found in programming environments for debugging purposes.

The first extension, `watch`, prints an informative message each time an assignment is made to the variable specified as an argument. All other language constructs are evaluated normally, that is, by `super`.

```
fun watch (Var(n,y)) this super
                     (e as (Assign(Var(_,x),r))) =
    let val result = super e in
        if y = x then
            print n ^ " = " ^ (Int.toString (!x)) ^ "\n"
        else
            () ;
        result
    end
  | watch _ this super t = super t
```

Having more than two variants in the data type, we can better appreciate the benefits of extending an existing function: the extension only needs to specify the behavior for the cases that are handled differently; all other cases can be passed to the original function.

The second extension, `tracewhile`, prints an informative message each time a while loop is evaluated.

```
fun tracewhile this super (e as (While(c,b))) =
    let val cond = this c in
        if cond = 0 then
            ( print "done\n" ; 0 )
        else
            ( print "repeating\n" ; this b ; this e )
    end
```

We can combine these extensions to evaluate the program in "heavy" debug mode.

```
- extend [watch x, watch y, tracewhile, watch r] eval s;
repeating
r = 2
y = 2
repeating
r = 4
y = 1
repeating
r = 6
y = 0
done
val it = 0 : int
- new eval r;
val it = 6 : int
```

Acknowledgments

The author would like to thank Radhakrishnan Jagadeesan, Michael J. Maher, and Neeraj Mehta for insightful feedback and discussions.

10. REFERENCES

[1] R. Bird and P. Wadler. *An introduction to functional programming.* Prentice Hall International (UK) Ltd., 1988.

[2] G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[3] G. Bracha and D. Griswold. Extending Smalltalk with mixins. In OOPSLA'96 Workshop on Extending the Smalltalk Language, April 1996. Electronic note available at http://www.javasoft.com/people/gbracha/mwp.html.

[4] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software*, pages 415–438, 1997.

[5] J. Eifrig, S. F. Smith, V. Trifonov, and A. E. Zwarico. An interpretation of typed OOP in a language with state. *Lisp and Symbolic Computation*, 8(4):357–397, 1995.

[6] M. Felleisen and D. P. Friedman. *A little Java, a few patterns.* MIT Press, Cambridge, Massachusetts, USA, 1998.

[7] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 94–104, 1998.

[8] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, 1998.

[9] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1995.

[10] K. Hammond, J. Peterson, et al. *Report on the Programming Language Haskell: A Non-strict, Purely Functional Language.* Yale University, New Haven, Connecticut, USA, 1997. Version 1.4.

[11] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 214–223, New York, NY, 1986. ACM Press.

[12] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, USA, 1990.

[13] OMG. Unified modeling language. Specification v1.3, Object Management Group, June 1999. http://www.omg.org/technology/uml/.

[14] K. Ostermann and M. Mezini. Object-oriented composition untangled. In *Proceedings OOPSLA '01, Tampa Bay, FL*, 2001.

[15] Y. Smaragdakis and D. Batory. Mixin-based programming in C++. Technical Report CS-TR-98-27, University of Texas at Austin, Jan. 1998.

[16] L. Thorup and M. Tofte. Object oriented programming and Standard ML. In *ACM SIGPLAN Workshop on Standard ML and its Applications*, June 94.

[17] J. Visser. Visitor combination and traversal control. In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications*, pages 270–282. ACM Press, 2001.

[18] G. Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, Cambridge, Massachusetts, USA, 1993.