Functional Extensions to an Object-Oriented Programming Language

Warwick B. Mugridge, John G. Hosking, John Hamer

Report No. 49, November 1990.

Department of Computer Science

University of Auckland, New Zealand

rick@cs.aukuni.ac.nz jham1@cs.aukuni.ac.nz john@cs.aukuni.ac.nz

Kea is a strongly-typed, object-oriented programming language with some functional and procedural features. However, the functional aspects of the language are rather weak. To overcome these limitations, extensions to the language are proposed that are modelled on the capabilities of strongly-typed functional languages, transformed and integrated within the object-oriented framework of *Kea*.

Functions are strongly typed; they may be higher-order and implicitly polymorphic. A function may be *multivariant*, corresponding to a strongly-typed form of the multi-methods of CLOS (Keene, 1989). Multivariant functions permit despatching to be avoided in some cases and may be used to avoid the restrictions of the contravariance rule.

Functional Extensions to an Object-Oriented Programming Language

W. B. Mugridge, J.G. Hosking, J. Hamer

Department of Computer Science

University of Auckland, New Zealand

1. Introduction

Kea[1] is a strongly-typed, object-oriented programming language with some functional and procedural features (Hamer, 1990; Hosking et al, 1990a). However, the functional aspects of the language are rather weak. To overcome these limitations, extensions to the language are proposed that are modelled on the capabilities of strongly-typed functional languages, transformed and integrated within the object-oriented framework of *Kea*.

Functions are strongly typed; they may be higher-order and implicitly polymorphic. The implicit polymorphism is more general than inclusion and bounded parametric polymorphism (Cardelli and Wegner, 1985) and breaks down the distinction between overloading and other forms of polymorphism. A function may be *multivariant*, corresponding to a strongly-typed form of the multi-methods of CLOS (Keene, 1989). Multivariant functions permit despatching to be avoided in some cases and may be used to avoid the restrictions of the contravariance rule.

Other work has addressed related issues. Cardelli and Wegner (1985) consider types in object-oriented frameworks and introduce *Fun*, a functional programming language that provides some object-oriented features. However, *Fun* does not permit inherited functions to be overridden. Because of taking a structural approach to subclassing, it can not distinguish between classes with the same components but different behaviours.

Eiffel (Meyer, 1988) and Trellis/Owl (Schaffert et al, 1985) have some of the facilities introduced in *Kea*, but they do not provide higher-order and polymorphic functions. In addition, despatching is only permitted on the first (implicit) argument of a function or procedure. Several type problems have been found in Eiffel, including violation of the contravariance rule (Cook, 1989). OBJ2 (Goguen and Mesegeur, 1987) integrate some aspects of functional and object-oriented languages in an algebraic approach; higher-order functions are not permitted.

Canning et al (1989) argue for the use of interfaces in object-oriented programming. They also introduce functional programming features in an object-oriented setting. Budd (1989) considers the integration of functional and object-oriented approaches (along with procedural and logic programming).

---

[1] Kea was previously known as *Class Language*.

The next section introduces the object-oriented nature of *Kea*.[2] Section 3 introduces relevant work (issues and features) in functional programming languages. The functional extensions to *Kea* introduced in Section 4 are modelled on functional languages, but with an object-oriented approach. Multivariant functions (typed multi-methods) are introduced in Section 5. The contravariance rule and its consequences for subtyping are raised in Section 6; we give a general approach to avoid the rule's restrictions. The final section concludes the paper and indicates future work.

2. Introduction to Kea

Kea is a single-assignment language. It is intended to model a consistent state of something like a building and its components. The motivation for developing *Kea* was the development of applications for checking conformance of buildings with codes of practice (Mugridge et al, 1987; Hamer et al, 1988; Hamer et al, 1989a, b; Hosking et al, 1990b). *Kea* was originally modelled on backward-chaining systems like Mycin (Buchanan and Shortliffe, 1984) and was influenced by the object-oriented form of Smalltalk (Goldberg and Robson, 1983) and the database ideas of Smith and Smith (1977).

Kea inherits from the object-oriented paradigm the notions of information-hiding, abstract data types, and multiple inheritance within a class generalisation structure. In addition, it introduces the novel notion of multiple, dynamic classification. However, *Kea* is also a single-assignment language and so does not allow for state-change, unlike most object-oriented languages.

A complete *Kea* program consists of the definition of classes, plus the declaration of instances (references to objects) and procedures (methods). Typically, the outermost procedural code calls procedures within (class) instances. Procedures direct the flow of control, causing the evaluation of instances as needed. There is no procedural assignment; procedures do not specify how to evaluate, only what is to be evaluated. Thus the single-assignment aspect of the language remains pure. We ignore procedures in the remainder of this paper.

2.1 Classes

A class consists of a signature and an implementation. The signature consists of the object parameters and public attributes of the class. Object parameters are used to pass information to a newly created object of a class from the context in which it was created.

The outline of an example program is shown in Figure 1. The class *Section* has a signature consisting of two object parameters (*theEnvironment* of class *Environment* and *connectivity* of class *SectionConnectivity*) and a public attribute (*theRoof* of class *Roof*). Object parameters supply to an object of a class all the external information that is needed by the object. Explicit parameters provide for good modularity of classes as no assumptions need to be made about the context in which an object is created.

The implementation of a class consists of expressions for public and private (non-public) attributes. An attribute of a class, like an instance, is typed and has an expression that can be evaluated to provide a value or an object. The expression associated with an instance is

---

[2] Only aspects of *Kea* relevant to this paper are covered; Hamer (1990) provides a complete description of the (unextended) language.

evaluated when a value is required, such as in the evaluation of another expression. A feature (public attribute or parameter) of a component object is referenced using the "^" operator, as illustrated in the expression for the attribute *name* in Figure 1. Private attributes are hidden, providing for encapsulation of a class.

2.2 Objects and Collections

An object is created on demand with the pseudo-function *new*. For example, the instance *section* in Figure 1 is assigned an object of type *Section* on the first reference to the value of *section*. The new object is passed two parameters; these expressions are not evaluated until they are needed. Parameter passing may be expressed either positionally or by explicitly specifying expressions for each of the formal class parameters.

An instance or a parameter may be assigned an existing object of an appropriate type, allowing for arbitrary connections between objects. This is illustrated in Figure 1: the object referred to by the attribute *theRoof* is passed as a parameter to each of the new *storeys*.

```
class Section.
 parameters      theEnvironment: Environment.
                 connectivity: SectionConnectivity.
  public theRoof
     := new Roof(sectionDetail := connectivity,
            windArea := theEnvironment^windArea,
            heightToEaves := sum(collect(s in storeys,
                        s^height))).
  name
     := connectivity^name.
  storeys
     := create bag Storey(
        aStorey in selectedStoreys with parameters(
           theStorey := aStorey,
           numberOfStoreys := numberOfStoreys,
           theRoof := theRoof,
           allStoreys := storeys)).
  selectedStoreys: set storeyLevel := ...
  numberOfStoreys: integer
     := ask('How many storeys are there in', name, ' ? ').
 end Section.
```

```
instance
 section := new Section(theEnvironment := environ,
                        connectivity := connectivity).
 environ := new Environment(...).
 connectivity := new SectionConnectivity (...).
...
```

Figure 1. Outline of a Program

A collection (set or list) of objects is created with the pseudo-function *create*, as illustrated by the attribute *storeys* in Figure 1. For an object of class *Section* the collection of storeys is passed to each of the storeys; this capacity for self-reference is made possible by the lazy evaluation of parameters. Several operators and functions are provided for manipulating collections of values. The function *sum* adds up the elements of a collection. The function *collect* creates a collection

from each of the components of a collection (i.e., like a relational project). The function *select* selects a subset of a collection based on a predicate (i.e., like a relational select).

2.3 Inheritance and Types

A class may have several generalisations (superclasses), providing for multiple inheritance. A class inherits the signatures and implementations of its superclasses; it may add to either. A subclass may override the expression of an inherited attribute. Name clashes between several independent superclasses are not permitted. However, a class inherits an attribute only once from a superclass, even when there are several inheritance paths from that superclass.

Superclass relationships define a (partial) type ordering. In general, an object of type *C* may be assigned to an attribute of type *P* (i.e., *C conforms* to *P*), when *P* is a super*type of *C*. The relation super*type is the transitive closure of the supertype relation. There is no notion of inheritance without a type relationship.

The order of classes is significant when subclasses override an inherited attribute expression. However, multiple inheritance leads to a partial order of classes, and so the order of execution is not completely defined in *Kea*. Snyder (1986) discusses various approaches to turning this partial order into a total order.

2.4 Dynamic Classification

Dynamic classification of objects permits the type of an object to be elaborated at runtime. As information is gathered about an object of some general class, it may be explicitly classified as also belonging to other classes. For example, the classification attribute *elementType* in Figure 2 specifies several possible classes that an object of class *BracingElement* may be classified to. Note that if an instance of class *DiagonalBrace* is created directly, the classification attribute *elementType* is assigned the appropriate value automatically.

The process of classification is carried out on demand, whenever a possible classification may lead to code that can affect the current evaluation of an expression. Classification is lazy in that it is only carried out to the extent that is necessary for the current evaluation.

```
class BracingElement.
 classification elementType:
     [DiagonalBrace, DiagonalBoarding, SheetBracing,
      SheetMaterial2, ReinforcedConcrete, OtherBracing]
     := DiagonalBrace if standardBraceUsed or
                         selectedType = 1
     |  DiagonalBoarding if selectedType = 7
     |  ...
 end BracingElement.
```

Figure 2. Classification

The class structure diagram for *BracingElement* and its subclasses is shown in Figure 3. Classification is shown by the downward arrows coming from a single point, corresponding to a single classification attribute. In this case all the subclasses also inherit from *bracingElement*, as represented by the upward arrow.

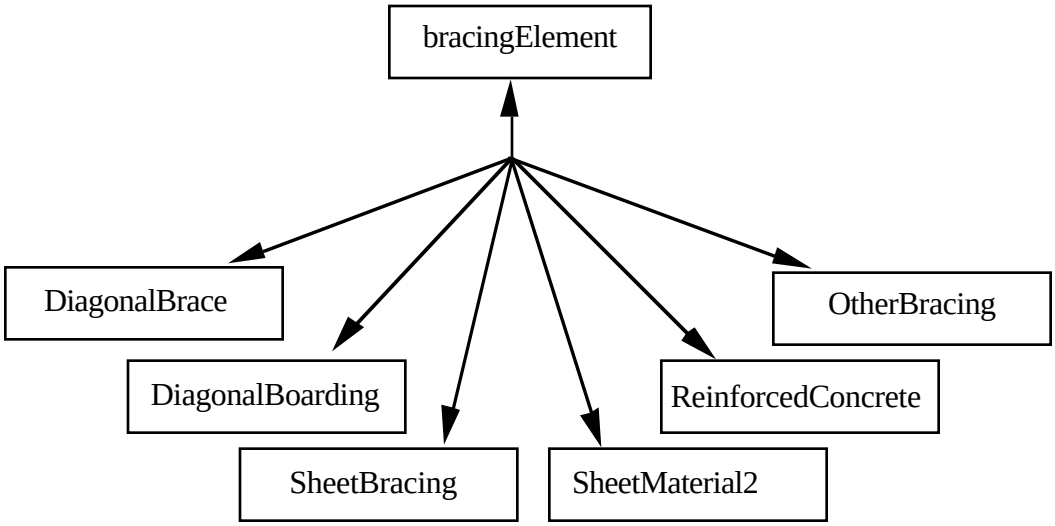

Figure 3. Classification

Multiple classification is achieved with several classification attributes, so that an object can be classified to several independent subclasses. For example, the class *BraceWall* in Figure 4 has two classification attributes. An object of class *BraceWall* could, for example, be elaborated to be also of classes *InternalWall* and *BraceNovice*.

Several advantages of multiple classification are discussed in Hamer et al (1989b). Without classification, all valid combinations of categories have to be encoded outside the class. This leads to redundancy in the code, is non-lazy because the selection between the valid possiblities has to be made all at once, and it means that attributes used in the selection have to be encoded outside the class.

A classification attribute specifies a "cluster": a set of mutually-exclusive subclasses (Smith and Smith, 1977). The definition of clusters means that invalid multiple inheritance can be rejected. For example, in Figure 4 the classes *InternalWall* and *ExternalWall* are mutually-exclusive subclasses of *BraceWall*, so it is invalid for a class to have both these classes as superclasses.

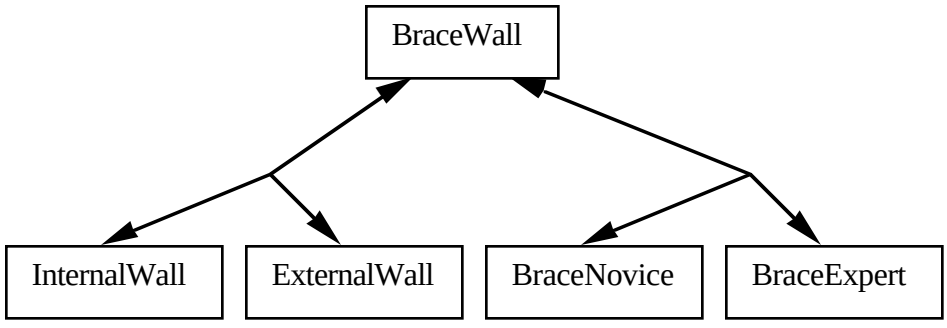

Figure 4. Multiple Classification

3. Functional Programming Languages

Several relevant ideas and techniques are associated with functional languages, including the provision of higher-order and polymorphic functions that are strongly typed. In the following, Hope (Field and Harrison, 1988) is used to illustrate aspects of functional languages that are relevant to the *Kea* extensions.

3.1 Pattern Matching

The Hope function *doubleList* in Figure 5 illustrates the use of "pattern matching" on the *list* data type that appears as the argument of the function. Two cases are given in the function: one for when the argument is an empty list and the other for a non-empty list. In the latter case, the pattern "head :: tail" is used to isolate the two parts of the data structure, the head and the tail of the list, to be used as separate values in the body of the function. The type of *doubleList* is specified as "list(num) → list(num)"; the parameter and result type are both "list(num)".

```
dec doubleList: list(num) -> list(num);
--- doubleList(nil) <= nil;
--- doubleList(head :: tail) <= (2*head) :: doubleList(tail);
```

Figure 5. Pattern Matching

One of the cases is selected at runtime. Several approaches to executing pattern matching are possible: Hope uses "best-fit" matching, while Standard ML tests the patterns in order (Wikstrom, 1987). A compiler can ensure that patterns are exhaustive (all the cases are considered) and not redundant.

3.2 Higher-Order Functions

A function in Hope may be higher-order, in that it can be passed as a parameter and returned as the result of a function. For example, the function *numMap* in Figure 6 takes a function as argument and applies it to each of the elements of a list, returning a list of the resulting values. The function *doubleList* of Figure 5 may be rewritten using *numMap* in Figure 6. The anonymous function provided as argument to *numMap* is written as a lambda expression; it returns a number double the number provided as argument.

```
dec numMap: (num -> num) # list(num) -> list(num);
--- numMap(f, nil) <= nil;
--- numMap(f, head :: tail) <= f(head) :: numMap(f, tail);
```

```
dec doubleList: list(num) -> list(num);
--- doubleList(aList) <= numMap(lambda x => 2 * x, aList);
```

Figure 6. Higher-Order Function

3.3 Polymorphic Functions

A polymorphic higher-order function that applies a "folding" function to each element of a list is shown in Figure 7. The type parameters[3] *alpha* and *beta* may take any type consistent with the declaration. The first argument is a function of type "(alpha # beta → beta)"; it takes two parameters with the result being of the same type as the second parameter.

```
dec fold : (alpha # beta -> beta) # beta # list(alpha) -> beta;
--- fold(fn, identity, nil) = identity;
--- fold(fn, identity, head :: tail)
          <= fn(head, fold(fn, identity, tail));
```

```
dec sum : list(num) -> num;
--- sum(aList) <= fold(+, 0, aList);
```

Figure 7. *Fold*: A Polymorphic Function

The function *fold* is used in the function *sum* in Figure 7, in which "+" is a function of type "(num # num → num)". In this case the function *fold* is used as a function of type "(num # num → num) # num # list(num) → num".

3.4 Lazy Constructors

Data constructors in Hope are lazy, allowing for infinite structures that are evaluated only as much as is needed. Infinite structures in a lazy language allow the termination conditions to be separated from the generator (Hughs, 1989). For example, the Hope function *from* in Figure 8 is a generator that can produce an infinite list of integers (adapted from Field and Harrison, 1988, p67).

```
dec from : num -> list(num);
--- from(x) <= x :: from(x + 1);
```

```
dec sum : num # list(num) -> num;
--- sum(number, head :: tail) <=
          if number = 0 then 0
          else head + sum(number - 1, tail);
```

Figure 8. Infinite Structures through Laziness

The "::" list constructor is lazily evaluated and so the recursive function call of *from* is not evaluated until the value is needed. The example function *sum* in Figure 8 sums the first *number* elements of a list. The evaluation of *sum* forces the construction of the list element by element as each is needed; no more of the list is constructed than is necessary. Hence the terminating condition is defined within *sum* and not in the *from* generator. For example, the function application "sum(10, from(20))" forces the construction of the first 10 elements of the infinite series 20, 21, 22, ...

---

[3] These are called *polytypes* in Hope.

3.5 Polymorphic Data Types

Polymorphic data types may be defined in Hope, as illustrated in Figure 9 (adapted from Field and Harrison, 1988). The poly-type *alpha* of the data type *tree* allows for trees of different types to be used. Pattern-matching is permitted on more than one function parameter; it is used in the function *isEqualTree* to select between the possible pairs of tree data constructors.

```
data tree(alpha) == empty ++ leaf(alpha) ++
                    node(alpha # tree(alpha) # tree(alpha));

dec toList: tree(alpha) -> list(alpha);
--- toList(empty) <= nil;
--- toList(leaf(value)) <= [value];
--- toList(node(value, left, right)) <=
           toList(left) <> [value] <> toList(right);

dec IsEqualTree : tree # tree -> truval;
--- IsEqualTree(empty, empty) <= true;
--- IsEqualTree(leaf(m), leaf(n)) <= n = m;
--- IsEqualTree(node(l1,v1,r1), node(l2,v2,r2)) <=
     (v1 = v2) and IsEqualTree(l1,l2) and IsEqualTree(r1,r2);
--- IsEqualTree(t1, t2) <= false;
```

Figure 9. User-Defined Data Types

4. Adding Functions in *Kea*

Higher-order and polymorphic functions are to be integrated with the object-oriented aspects of *Kea*. Function arguments are lazily evaluated, as with object parameters. Section 5 introduces multivariant functions.

4.1 Simple Functions

Simple functions provide functional abstraction similar to languages like Pascal. For example, the class *Interval* in Figure 10 contains a lower and upper bound of an interval. The class has three public functions; the function *overlap*, for example, takes an object of class *Interval* as parameter and returns the length of overlap of the two intervals.

```
class Interval.
 parameter minimum, distance: float.
 public  minimum, distance, maximum, overlap, withinInterval.
  maximum := minimum + distance.
  overlap(other: Interval)
     := min(other^distance, maximum - other^minimum)
         if withinInterval(other^minimum)
     |  other^maximum - minimum
         if withinInterval(other^maximum)
     |  distance
         if other^withinInterval(minimum)
     |  0.0.
  withinInterval(point: float)
     := point >= minimum and point =< maximum.
 end Interval.
```

```
class Rectangle.
 parameter xMin, yMin, xDist, yDist: float.
 public xMin, yMin, xDist, yDist, xMax, yMax.
  yMax := xMin + xDist.
  yMax := yMin + yDist.
end Rectangle.
```

Figure 10. Functions in class *Interval*

4.2 Higher-Order and Polymorphic Functions in Class *List*

The use of higher-order and polymorphic functions is illustrated in class *List* and its subclasses in Figure 11. The three classes here together define a data structure for a list of integers. The class *List* specifies the signatures of the public functions available (corresponding to "virtuals" in Simula (Dahl and Nygaard, 1966)), as well as defining the function *cons* (corresponding to the cons function of Lisp).

The class *EmptyList* provides the code for the empty list case while the class *ListNode* defines the non-empty list case. This approach differs from Hope functions, which are defined with patterns for the two cases of a list (“nil” and “head :: tail”). The *Kea* code is organised according to the two cases, with the two classes containing the functions. Despatching a function call according to the class of the object concerned plays a similar role to pattern-matching in Hope.

The classification attribute *defaultEmpty* in the class *List* specifies that there are only two immediate subclasses of *List*, and that they are mutually exclusive. An instance of class *List* will be classified to class *EmptyList*, as defined by the expression for *defaultEmpty*, on the first access to a public function of the object (other than *cons*).

```
class List.
 public    cons, filter, map, fold, append.
 classification defaultEmpty: [EmptyList, ListNode]
     := EmptyList.
  cons(front: integer) := new ListNode(front, self).
  filter(keep: fn(integer) -> boolean): List.
  map(trans: fn(integer) -> integer): List.
  fold(accum: fn(integer, Any) -> Any, identity: Any): Any.
  append(other: List): List.
 end List.
```

```
class EmptyList.
 generalisation List.
  filter(keep) := self.
  map(trans) := self.
  fold(accum, identity) := identity.
  append(other) := other.
 end EmptyList.
```

```
class ListNode.
 generalisation List.
 parameter head: integer.
           tail: List.
 public head, tail.
  filter(keep)
     := tail^filter(keep)^cons(head) if keep(head)
     |  tail^filter(keep).
  map(trans)
     := tail^map(trans)^cons(trans(head)).
  fold(accum, identity)
     := accum(head, tail^fold(accum, identity)).
  append(other)
     := tail^append(other)^cons(head).
 end ListNode.
```

Figure 11. The *List* Classes

4.2.1 Higher-Order Functions

Functions in *Kea* may be higher-order and/or polymorphic, as with Hope. For example, the higher-order function *map* in the class *List* in Figure 11 is of type "List → (integer → integer) → List".[4] This function is similar to the Hope function in Figure 6; it returns the list resulting from applying the function *trans* to each of the elements of the provided list.

For example, *map* is used in Figure 12 to supply the increment of each of the integers in a list. An anonymous function of the form "lambda(i) => i + 1" is provided as an argument to *map*; this is similar to the Hope lambda expression in Figure 6. Similarly, the function *filter* is used in Figure 12 to select the positive integers from a list.

[4] Corresponding to the Hope type "list(num) # (num → num) → list(num)".

```
ints := new List^cons(-1)^cons(2).
positives := ints^filter(lambda(i: integer) => i > 0).
increment := ints^map(lambda(i: integer) => i + 1).
```

Figure 12. Using functions *filter* and *map*

4.2.1 Polymorphic Functions

Parametric polymorphism for functions is implicit, both in bounded and unbounded forms. For example, the function *fold* in Figure 11 has the type *Any* specified for some of the parameters. This function corresponds to the Hope function *fold* in Figure 7 and is inferred to be of type "List → (integer → X → X) → X → X", in which *X* is a type variable.[5] The function *fold* is used in Figure 13 to sum the elements of a *List*, in which the actual type of the function application is "List → (integer → integer → integer) → integer → integer".

```
ints := new List^cons(-1)^cons(2).
sum := ints^fold(
           lambda(i: integer, total: integer) => i + total, 0).
```

Figure 13. Using function *fold*

A function may be both higher-order and polymorphic, as shown in Figure 14. This function is of type "X → (X → X) → (X → X)", where X is an unbounded type variable.

```
twice(f: fn(Any) -> Any):= lambda(x: Any) => f(f(x)).
```

Figure 14. A Polymorphic and Higher-Order Function

4.3 Lazy Functions

The *List* class can be used to produce lazy finite and infinite lists, as shown in Figure 15.

```
fromTo(lower: integer, upper: integer)
     := new List if lower > upper
     |  fromTo(lower+1, upper)^cons(lower).

from(here: integer) := new ListNode(here, from(here+1)).

factorial(n: integer) := fromTo(1, n)^fold(
           lambda(el: integer, tot: integer) => el * tot, 1).
```

Figure 15. Lazy Lists

---

[5] Corresponding to the Hope type "list (alpha) # (num # alpha -> alpha) # alpha -> alpha"

5. Multivariant Functions

As with Hope, *Kea* permits the selection of the appropriate code to be based on more than one parameter of a function. Multivariant functions in *Kea* are a typed form of the multi-methods of CLOS (Keene, 1989). As with multi-methods, the code chosen for execution (during despatching) depends on the type of all the function call arguments, rather than just the type of the primary object. *Kea*, however, is strongly-typed; multivariant functions and their calls are statically checked for type-correctness. The selection of the appropriate function *variant* can be made at compile-time if the types of function call arguments are suitable, as discussed below.

5.1 Overloading

A multivariant function is *overloaded* when its variants have unrelated parameter types. For example, in Figure 16 the function *div* accepts either integers or reals. The applicable variant (and hence the range type) can be determined at compile-time for a call of the function *div*. For example, the expression for *aList* in Figure 16 is incorrectly typed because the *List* function *cons* requires an integer parameter; it is rejected at compile-time. Compile-time selection of the appropriate variant for a function call avoids run-time despatching.

```
div(r1: real, r2: real) := r1 / r2.            % real
div(r: real, i: integer) := r / toReal(i).     % real
div(i: integer, r: real) := toReal(i) / r.     % real
div(i1: integer, i2: integer) := i1 div i2.    % integer
```

```
anInt := div(4,2).                        % integer
aReal := div(4, 2.0).                     % real
aList := new List^cons(div(2.0, 4)).      % Type error
```

Figure 16. Overloaded Function

5.2 Despatching

Type information about the parameters of a function call may not be sufficient to select the appropriate function variant at compile-time. For example, consider the fragment of the class *ListNode* in Figure 17 that is extended to provide a function *last*. An element of a list is the last one only if the following list element is the empty list; hence a distinction has to be made based on the type of the class parameter *tail*.

As the argument *tail* in the function call *lastOne(tail)* is only known to be of class *List*, the selection of the appropriate variant must occur at run-time. There are two variants, both of which are relevant to the function call. As the only immediate subclasses of *List* are *ListNode* and *EmptyList*, one of the *lastOne* variants will be selected at runtime.

At run-time, the function call causes the selection of one of the variants, based on the type of the actual parameters. For example, if the actual parameter to the function *lastOne* is an object of type *ListNode*, the second variant is selected. The process of matching may force classification of the arguments to occur.

```
class ListNode.
 ...
 public last := lastOne(tail).

  lastOne(i: EmptyList) := head.
  lastOne(i: ListNode) := i^last.
 end ListNode.
```

Figure 17. Function *lastOne*

5.3 Overloading and Despatching

The need for despatching depends on the particular function call. For example, consider the function *addList* in Figure 18, which extends the *List* classes. This function takes two lists and adds them element by element; the resulting list is the length of the shortest of the two lists.

```
class List.
 ...
  addList(other: List): List.
 end List.
```

```
class EmptyList.
 ...
  addList(other) := self.                % Variant 1
 end EmptyList.
```

```
class ListNode.
 ...
  addList(other: EmptyList) := other.    % Variant 2
  addList(other: ListNode)               % Variant 3
     := tail^addList(other^tail)^cons(head + other^head).
 end ListNode.
```

Figure 18. Function *addList*

The three variants of *addList* have the following types: (1) "EmptyList → List → EmptyList "; (2) "ListNode → EmptyList → EmptyList "; and (3) "ListNode → ListNode → ListNode".

```
empty := new EmptyList.                          % EmptyList
one := empty^cons(1).                            % ListNode
positives := one^filter(lambda(x) => x > 0).     % List
v1 := empty^addList(one).                        % EmptyList
v2 := one^addList(empty).                        % EmptyList
v3 := one^addList(one).                          % ListNode
v4 := one^addList(positives).                    % List
v5 := positives^addList(positives).              % List
```

Figure 19. Overloaded and Despatching Function Calls

Consider the example function calls in Figure 19 (with result types as comments). Despatching is not needed for the expressions of *v1*, *v2*, and *v3*, as sufficient type information is available to

select the appropriate variant statically. The type of each expression is determined from the selected variant; for example, *v3* is of type *ListNode*, the result type of the third variant.

Despatching is required for the calls to *addList* in the expressions of *v4* (to select between variants 2 and 3) and *v5* (to select between all three variants). The expressions are both of type *List*; based on the result types of the variants involved in the two selections.

6. Avoiding the Contravariance Restriction

The contravariance rule specifies that a function *f* of type "AA → B" is a subtype of function *g* of type "A → BB" (i.e., *f* conforms to *g*) if and only if A conforms to AA and B conforms to BB. That is, the subtype may "narrow" down the result type but can only "widen" the parameter type.

6.1 Eiffel

Cook (1989) points out that the contravariance rule is violated in Eiffel (Meyer, 1988), leading to a type problem. The "contravariance" problem can be expressed in Eiffel as shown in Figure 20, in which the class *B* has "narrowed" the type of the parameter *x* of the inherited function *f*.

```
class A export f
feature
 f(x: A) : Integer is do Result := 0 end;
 end
```

```
class B export g
inherit A redefine f
feature
 g: Integer;
 f(x: B) : Integer is do Result := x.g end;
end
```

```
local
     a : A;
     b : B;
     refA : A;
     i : Integer;
do
     a.Create;
     b.Create;
     refA := b;
     i := refA.f(a);
end
```

Figure 20. Contravariance Violation in Eiffel

Consider the function call at the bottom of the program, which leads to the function *f* inside the class *B* being called. When that function refers to the *g* attribute of the parameter an error occurs because the actual parameter is of type *A* and has no attribute *g*.

Cook (1989) argues that the contravariance rule must be enforced in Eiffel to avoid this problem, hence preventing the "narrowing" of inherited function parameters. This is the approach taken

in Canning et al (1989); a subtype relationship can not be defined if the contravariance rule would be violated. It is unfortunate that contravariance seems to prevent useful subclass relationships from being defined.

6.2 Subtyping in *Kea* Without Contravariance Violation

Multivariant functions in *Kea* can allow the benefits of subclassing to be retained without violating the contravariance rule. The problem of Section 6.1 is simply solved in Figure 21, in which the class *ColorPoint* is a subtype of *Point*. The contravariance rule is satisfied because the function *equal* in class *ColorPoint* only partially overrides the inherited function; instead, it provides code to handle the case when the function parameter is of type *ColorPoint.*

```
class Point.
 public x, y, move, equal.
 parameter x, y.
  x: float.
  y: float.
  equal(p: Point) := x = p^x and y = p^y.
 end Point.
```

```
class ColorPoint.
 generalisation Point.
 public color.
 parameter color: Color.
  equal(p: ColorPoint)
     := x = p^x and y = p^y and color = p^color.
 end ColorPoint.
```

Figure 21. Contravariance and Parameter-Loss Solution

The two uses of *equal* in Figure 22 provide the same result; if either the object or the parameter (or both) is of type *Point*, the function variant in the class *Point* is called. The function variant in class *ColorPoint* is only called if the object and the parameter are both of class *ColorPoint.*

```
a := new Point(x := 0.0, y := 0.0).
b := new ColorPoint (x := 0.0, y := 0.0, colour := red).

consistent := a^equal(b) = b^equal(a).
```

Figure 22. Consistency of result from *equal*

A different notion of equality may be required, for example, in which an object of class *Point* can not be equal to an object of class *ColorPoint*. The function *firmEqual* in Figure 23 implements this equality test.

```
firmEqual(p: ColorPoint, q: ColorPoint)
   p^x = q^x and p^y = q^y and p^colour = q^colour.
firmEqual(p: ColorPoint, q: Point)
   false.
firmEqual(p: Point, q: ColorPoint)
   false.
firmEqual(p: Point, q: Point)
   p^x = q^x and p^y = q^y.
```

Figure 23. A Different Notion of Equality

7. Conclusions and Future Work

Kea has strong object-oriented features but requires improvements in its functional capability. After considering the functional features of strongly-typed functional languages like Hope (Field and Harrison, 1988), functional extensions are proposed for *Kea*. Higher-order, polymorphic, and multivariant functions add considerable functional power to the language.

Multivariant functions generalise the notion of despatching in object-oriented languages. They correspond to pattern-matching over multiple function arguments in functional languages and are a strongly-typed form of the multi-methods of CLOS (Keene, 1989). As despatching can be avoided when there is adequate type information about arguments in a function call, there need be no unnecessary overhead on function calls in an object-oriented language. In addition, multivariant functions avoid the restrictions on subtyping imposed by the contravariance rule.

Cardelli and Wegner (1985) distinguish overloading (ad hoc polymorphism) and universal polymorphism (parameteric and inclusion polymorphism). However, multivariant functions show that the distinction is not so clear; whether overloading or inclusion polymorphism is involved can depend on the function calls concerned.

As with CLOS (Keene, 1989), some of the benefits of pattern-matching are unavailable in the multivariants of *Kea*. While the loss of pattern-matching capability arises in part because of encapsulation, there may be ways of introducing a more flexible form of pattern-matching that is consistent with encapsulation.

Encapsulation of multivariant functions is an open issue. Functions in *Kea* may be organised within classes (i.e. based on the object: the implicit first argument), where access is available to all functions of the class, both public and private. A function that is external to a class may only access public functions of an object of that class. Hence the first argument to a function is still given special status, as in many object-oriented programming languages: Smalltalk (Goldberg and Robson, 1983), Eiffel (Meyer, 1988), and Trellis/Owl (Schaffert et al, 1986). This is in comparison with CLOS, which discards the notion of encapsulation all together in introducing multi-methods (Keene, 1989). Further work is needed in considering other ways to integrate encapsulation and multi-methods.

The typing rules and semantics of multivariant functions must now be provided. The introduction of some form of generic classes also needs to be considered. "Type loss" is an important issue that arises with strongly-typed object-oriented languages. For example, Eiffel (Meyer, 1988) has the notion of "like Current" to allow for the sharing of code through inheritance. This issue is being considered in the context of typing multivariant functions in Kea.

Acknowledgements

The authors gratefully acknowledge the financial assistance provided by the Building Research Association of New Zealand (BRANZ), the University of Auckland Research Committee, and the University Grants Committee.

References

Buchanan B G, Shortliffe E H (Eds), 1984. *Rule-Based Expert Systems: the Mycin Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley.

Budd T A, 1989. *Functional Programming in an Object-Oriented Language*, Report 89-60-16, Department of Computer Science, Oregon State University.

Canning P S, Cook W R, Hill W L, Olthoff W G, 1989. Interfaces for strongly-typed object-oriented programming, OOPSLA'89, ACM SIGPLAN Notices, **24** (10) October, 1989, pp457-467.

Cardelli L, Wegner P, 1985. On understanding types, data abstraction, and polymorphism, *Computing Surveys*, **17**(4), pp471-522.

Cook W, 1989. A proposal for making Eiffel type safe, in Cook S (Ed), *ECOOP 89*, Cambridge University Press, pp57-70.

Dahl O J, Nygaard K, 1966. Simula - an Algol-based simulation language, *CACM* **9** (9), pp671-678.

Field A J, Harrison P G, 1988. *Functional Programming*, Addison-Wesley.

Goguen and Meseguer, 1987. Unifying functional, object-oriented and relational programming with logical semantics, in Shriver B and Wegner P (1987), *Research Directions in Object-Oriented Programming*, Prentice-Hall.

Goldberg A, Robson D, 1983. *Smalltalk 80: The Language and its Implementation*, Addison-Wesley.

Halbert D C, O'Brien P D, 1987. Using types and inheritance in object-oriented programming, *IEEE Software*, September 1987, pp71-79.

Hamer J, Hosking J G, Mugridge W B, 1988. The evolution of Class Language, *Proc. Third New Zealand Conference on Expert Systems*, Wellington, New Zealand.

Hamer J, Hosking J G, Mugridge W B, 1989a. Knowledge-based systems for representing codes of practice, Department of Computer Science Report 48.

Hamer J, Mugridge W B, Hosking J G, 1989b. Object-oriented representation of codes of practice, *Procs. Australasian Conference on Expert Systems in Engineering, Architecture, and Construction*, Sydney Australia, December 1989, pp209-221.

Hamer, J, 1990. Expert Systems for codes of practice, PhD Thesis, Department of Computer Science, University of Auckland, New Zealand.

Henderson P, 1980. *Functional Programming: Application and Implementation*, Prentice-Hall.

Hosking J G, Hamer J, Mugridge W B, 1990a. Integrating Functional and Object-Oriented Programming, Procs. Pacific Tools 80 Conference, Sydney, Australia, November 1990.

Hosking J G, Hamer J, Mugridge W B, 1990b. From FireCode to ThermalDesign: KBS for the building industry, to be presented to the 4th New Zealand Expert Systems Conference, Palmerston North, New Zealand, December 1990.

Hughs J, 1989. Why functional programming matters, *The Computer Journal*, **32**(2), pp98-107.

Keene S E, 1989. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Addison-Wesley, 1989.

Meyer B, 1988. *Object-Oriented Software Construction*, Prentice Hall.

Mugridge W B, Hamer J, Hosking J G, 1987. Class Language - a language for building expert systems, *Proc. Second New Zealand Conference on Expert Systems*, University of Auckland, Auckland, New Zealand, pp173-180, February, 1987.

Schaffert G, Cooper T, Bullis B, Kilian M, Wilpori Co, 1985. An introduction to Trellis/Owl, *OOPLSA 86*, pp9-16.

Smith J M, Smith D C P, 1977. Database abstractions: aggregation and generalization, *ACM Trans. on Database Systems*, **2** (2), 1977, pp105-133.

Snyder A, 1986. Encapsulation and inheritance in object-oriented programming, *OOPSLA 86*, pp38-45.

Wikstrom, A, 1987. *Standard ML*, Prentice Hall, 1987.