EXTENDING RESOURCE-BOUNDED FUNCTIONAL PROGRAMMING LANGUAGES WITH MUTABLE STATE AND CONCURRENCY

STEPHEN GILMORE, KENNETH MACKENZIE AND NICHOLAS WOLVERSON*

Abstract. Camelot is a resource-bounded functional programming language which compiles to Java byte code to run on the Java Virtual Machine. We extend Camelot to include language support for Camelot-level threads which are compiled to native Java threads. We extend the existing Camelot resource-bounded type system to provide safety guarantees about the heap usage of Camelot threads. We demonstrate the usefulness of our concurrency extensions to the language by implementing a multi-threaded graphical network chat application which could not have been expressed as naturally in the sequential, object-free sublanguage of Camelot which was previously available.

1. Introduction. Functional programming languages allow programmers to express algorithms concisely using high-level language constructs operating over structured data, secured by strong type-systems. Together these properties support the production of high-quality software for complex application problems. Functional programs in strongly-typed languages typically have relatively few programming errors when compared to similar applications implemented in languages without these beneficial features.

These desirable language properties mean that developers shed the burdens of explicit memory management, but this has the associated cost that they typically lose all control over the allocation and deallocation of memory. The Camelot language provides an intermediate way between completely automatic memory management and unassisted allocation and deallocation in that it provides type-safe storage management by re-binding of addresses. The address of a datum can be obtained in a pattern match and used in an expression (to store a different data value at that address), overwriting the currently-held value.

The Camelot compiler targets the Java Virtual Machine but the JVM does not provide an instruction to free memory, consigning this to the garbage collector, a generational collector with three generations and implementations of stop-and-copy and mark-sweep collections. Camelot allows more precise control of memory allocation, allowing in-place modification of user-defined data structures. The Camelot compiler supports various resource-aware type systems which ensure that memory re-use takes place in a safe manner and also allow static prediction of heap-space usage. Camelot uses a uniform representation for types which are generated by the compiler, allowing data types to exchange storage cells. This uniform representation is called the *diamond type* [1, 2], implemented by a `Diamond` class in the Camelot run-time. The Camelot language implements a type system which assigns types to functions which record the number of parameters which they consume, and their types; the type of the result; and the number of diamonds consumed or freed. The outcome is that the storage consumption requirements of a function are statically computed at compile-time along with the traditional Hindley-Milner type inference procedure.

The novel contribution of the present paper is to explain how such an unusually rich programming model can be extended to incorporate object-oriented and concurrent programming idioms. This contribution is not just a design: it has been realised in the latest release of the Camelot compiler.

*Laboratory for Foundations of Computer Science, The University of Edinburgh, King's Buildings, Edinburgh, EH9 3JZ, Scotland

Structure of this paper. In Section 2 we present the Camelot language in order that the reader may understand the operational context of the work. We follow this in Section 3 with a discussion of our object-oriented extensions to Camelot. This leads on to a presentation of the use of threads in Section 4 followed by an analysis of the management of threads by the run-time system in Section 5. Section 6 explains the relationship between threads in Camelot and threads as traditionally implemented in concurrent functional languages using first-class continuations. Section 7 details the implications for verification of Camelot programs. Related work is surveyed in Section 8 and conclusions follow after that.

2. The Camelot language. The core of Camelot is a standard polymorphic ML-like functional language whose syntax is based upon that of O'Caml; the main novelty lies in extensions which allow the programmer to perform in-place modifications to heap-allocated data-structures. These features are similar to those described in by Hofmann in [3], but include some extra extensions for free list management. To retain a purely functional semantics for the language in the presence of these extensions a linear type system can be employed: in the present implementation, linearity can be enforced via a compiler switch. We are in the process of enhancing the compiler by the addition of other, less restrictive type systems which still allow safe in-place modifications: more details will be given below.

Crucial design choices for the compilation are transparency and an exact specification of the compilation process. The former ensures that the compilation does not modify the resource consumption in an unpredictable way. The latter provides a formal basis for using resource information inferred for the high-level language in proofs on the intermediate language.

In the following sections we will give a brief description of the structure of the language. We will then outline how the language is compiled, and in particular how the memory-management extensions are implemented.

2.1. The structure of Camelot. We will give some examples to indicate the basic structure of Camelot; full details can be found in [4].

Datatypes are defined in the normal way:

```
type intlist = Nil | Cons of int * intlist
type 'a polylist = NIL | CONS of 'a * 'a polylist
type ('a, 'b) pair = Pair of 'a *'b
```

Values belonging to user-defined types are created by applying constructors and are deconstructed using the `match` statement:

```
let rec length l = match l with
       Nil -> 0
     | Cons (h,t) -> 1+length t

let test () = let l = Cons(2, Cons(7,Nil))
    in length l
```

As can be seen from this example, constructor arguments are enclosed in parentheses and are separated by commas. In contrast, function definitions and applications which require multiple arguments are written in a "curried" style:

```
let add a b = a+b
let f x y z = add x (add y z)
```

Despite this notation, the present version of Camelot does *not* support higher-order functions; any application of a function must involve exactly the same number of arguments as are specified in the definition of the function.

2.2. Diamonds and Resource Control. The Camelot compiler targets the Java Virtual Machine, and values from user-defined datatypes are represented by heap-allocated objects from a certain JVM class. Details of this representation will be given in Section 2.4.

Consider the following function which uses an accumulator to reverse a list of integers (as defined by the `intlist` type above).

```
let rec rev l acc = match l with
    Nil -> acc
  | Cons (h,t) -> rev t (Cons (h,acc))
let reverse l = rev l Nil
```

This function allocates an amount of memory equal to the amount occupied by the input list. If no further reference is made to the input list then the heap space which it occupies may eventually be reclaimed by the JVM garbage collector.

In order to allow more precise control of heap usage, Camelot includes constructs allowing re-use of heap cells. There is a special type known as the *diamond type* (denoted by `<>`) whose values represent blocks of heap-allocated memory, and Camelot allows explicit manipulation of diamond objects. This is achieved by equipping constructors and match rules with special annotations referring to diamond values. Here is the `reverse` function rewritten using diamonds so that it performs in-place reversal:

```
let rec rev l acc = match l with
    Nil -> acc
  | Cons (h,t)@d -> rev t (Cons (h,acc)@d)
let reverse l = rev l Nil
```

The annotation "`@d`" on the first occurrence of `Cons` tells the compiler that the diamond value `d` is to be bound to a reference to the space used by the list cell. The annotation on the second occurrence of `Cons` specifies that the list cell `Cons(h,acc)` should be constructed in the diamond object referred to by `d`, and no new space should be allocated on the heap.

One might not always wish to re-use a diamond value immediately. This can sometimes cause difficulty since such diamonds might then have to be returned as part of a function result so that they can be recycled by other parts of the program. For example, the alert reader may have noticed that the list reversal function above does not in fact reverse lists entirely in place. When the user calls `reverse`, the invocation of the `Nil` constructor in the call to `rev` will cause a new list cell to be allocated. Also, the `Nil` value at the end of the input list occupies a diamond, and this is simply discarded in the second line of the `rev` function (and will be subject to garbage collection if there are no other references to it).

The overall effect is that we create a new diamond before calling the `rev` function and are left with an extra diamond after the call had completed. We could recover the extra diamond by making the `reverse` function return a pair consisting of the reversed list and the spare diamond, but this is rather clumsy and programs quickly become very complex when using this kind of technique.

To avoid this kind of problem, unwanted diamonds can be stored on a *free list* for later use. This is done by using the annotation "`@_`" as in the following example which returns the sum of the entries in an integer list, destroying the list in the process:

```
let rec sum l acc = match l with
    Nil@_ -> acc
  | Cons (h,t)@_ -> sum t (acc+h)
```

The question now is how the user retrieves a diamond from the free list. In fact, this happens automatically during constructor invocation. If a program uses an undecorated constructor such as `Nil` or `Cons(4,Nil)` then if the free list is empty the JVM `new` instruction is used to allocate memory for a new diamond object on the heap; otherwise, a diamond is removed from the head of the free list and is used to construct the value. It may occasionally be useful to explicitly return a diamond to the free list and an operator `free: <> -> unit` is provided for this purpose.

There is one final notational refinement. The in-place list reversal function above is still not entirely satisfactory since the `Nil` value carries no data but is nonetheless allocated on the heap. We can overcome this by redefining the `intlist` type as

```
type intlist = !Nil | Cons of int * intlist
```

The exclamation mark directs the compiler to represent the `Nil` constructor by the JVM `null` reference. With the new definition of `intlist` the original list-reversal function performs true in-place reversal: no heap space is consumed or destroyed when the `reverse` function is applied. The `!` annotation can be used for a single zero-argument constructor in any datatype definition. In addition, if every constructor for a particular datatype is nullary then they may all be preceded by`!`, in which case they will be represented by integer values at runtime. We have deliberately chosen to expose this choice to the programmer (rather than allowing the compiler to automatically choose the most efficient representation) in keeping with our policy of not allowing the compiler to perform optimisations which have unexpected results on resource consumption.

The features described above are very powerful and can lead to many kinds of program error. For example, if one applied the `reverse` function to a sublist of some larger list then the small list would be reversed properly, but the larger list could become partially reversed. Perhaps worse, a diamond object might be used in several different data structures of different types simultaneously. Thus a list cell might also be used as a tree node, and any modification of one structure might lead to modifications of the other. The simplest way of preventing this kind of problem is to require linear usage of heap-allocated objects, which means that variables bound to such objects may be used at most once after they are bound. Details of this approach can be found in Hofmann's paper [3]. Strict linearity would require one to write the list length function as something like

```
let rec length l = match l with
      Nil -> Pair (0, Nil)
    | Cons(h,t)@d ->
      let p = length t
      in match p with
           Pair(n, t1)@d1 -> Pair(n+1, Cons(h,t1)@d)@d1
```

It is necessary to return a new copy of the list since it is illegal to refer to `l` after calling `length l`.

Our compiler has a switch to enforce linearity, but the example demonstrates that the restrictive nature of linear typing can lead to unnecessary complications. Aspinall and Hofmann [5] give a type system which relaxes the linearity condition while still allowing safe in-place updates, and Michal Konečný generalises this still further in [6, 7]. As part of the MRG project, Konečný has implemented a typechecker for a variant of the type system of [6] adapted to Camelot.

A different approach to providing heap-usage guarantees is given by Hofmann and Jost in [8], where an algorithm is presented which can be used to statically infer heap-usage bounds for functional programs of a suitable form. In collaboration with the MRG project, Steffen Jost has implemented a variant of this inference algorithm for Camelot: the implementation is described in [9]. Both of these implementations are currently stand-alone programs, but we are in the process of integrating them with the Camelot compiler.

One of our goals in the design of Camelot was to define a language which could be used as a testbed for different heap-usage analysis methods. The inclusion of explicit diamonds fits the type systems of [5, 6, 7], and the inclusion of the free list facilitates the Hofmann-Jost inference algorithm, which requires that all memory management takes place via a free list.

2.3. Compilation of expressions. Camelot is initially compiled into the Grail intermediate language[10, 11] which is essentially a functional form of Java bytecode. This process is facilitated by an initial phase in which several transformations are applied to the abstract syntax tree.

2.3.1. Monomorphisation. Firstly, all polymorphism is removed from the program. For polymorphic types $(\alpha_n, \ldots, \alpha_1)\ t$ such as α `list` we examine the entire program to determine all instantiations of the type variables, and compile a separate datatype for each distinct instantiation. Similarly, whenever a polymorphic function is defined the program is examined to find all uses of the function and a monomorphic function of the appropriate type is generated for each distinct instantiation of types.

2.3.2. Normalisation. After monomorphisation there is a phase referred to as *normalisation* which transforms the Camelot program into a form which closely resembles Grail.

Firstly the compiler ensures that all variables have unique names. Any duplications are resolved by generating new names. This allows us to map Camelot variable names directly onto Grail variable names (which in turn map onto JVM local variable locations) with no danger of clashes arising.

Next, we give names to intermediate results in many contexts by replacing complex expressions with variables. For example, the expression $f(a+b+c)$ would be replaced by an expression of the form `let` $t_1 = a + b$ `in let` $t_2 = t_1 + c$ `in` $f(t_2)$. The introduction of names for intermediate results can produce a large number of Grail (and hence JVM) variables. After the source code has been compiled to Grail the number of local variables is minimised by applying a standard register allocation algorithm (see [12]).

A final transformation ensures that `let`-expressions are in a "straight-line" form. After all of these transformations have been performed expressions have been reduced to a form which we refer to as *normalised Camelot*

The structure of normalised Camelot (which is in fact in a type of A-normal form [13]) is sufficiently close to that of Grail that it is fairly straightforward to translate from the former to the latter. Another benefit of normalisation is that it is easier to write and implement type systems for normalised Camelot. The fact that the components of many expressions are atoms rather than complex subexpressions means that typing rules can have very simple premisses.

2.4. Compilation of values. Camelot has various primitive types (`int`, `float`, etc.) which can be translated directly into corresponding JVM types. The compilation of user-defined datatypes, however, is rather more complicated. Objects belonging to datatypes are represented by members of a single JVM class which we will refer to as the *diamond class*. Objects of the diamond class contain enough fields to represent any member of *any* datatype defined in the program. Each instance X of the diamond class contains an integer tag field which identifies the constructor with which X is associated. The diamond class also contains a static field pointing to the free list. The free list is managed via the static methods `alloc` (which returns the diamond at the head of the free list, or creates a new diamond by calling `new` if the free list is empty), and `free` which places a diamond object on the free list. The diamond class also has overloaded static methods called `make` and `fill`, one instance of each for every sequence of types appearing in a constructor. The `make` methods are used to implement ordinary constructor application; each takes an integer tag value and a sequence of argument values and calls `alloc` to obtain an instance of the diamond class, and then calls a corresponding `fill` method to fill in the appropriate fields with the tag and the arguments. The `fill` methods are also used when the programmer reuses an existing diamond to construct a datatype value.

It can be argued that this representation is inefficient in that datatype values are often represented by JVM objects which are larger than they need to be. This is true, but is difficult to avoid due to the type-safe nature of JVM memory management which prevents one from re-using the heap space occupied by a value of one type to store a value of a different type. We wish to be able to reuse heap space, but this can be impossible if objects can contain only one type of data. With the current scheme one can easily write a heapsort program which operates entirely in-place. List cells are large enough to be reused as heap nodes and this allows a heap to be built using cells obtained by destroying the input list. Once the heap has been built it can in turn be destroyed and the space reused to build the output list. In this case, the amount of memory occupied by a list cell is larger than it needs to be, but the overall amount of store required is less than would be the case if separate classes were used to contain list cells and heap nodes.

In the current context it can be claimed that it is better to have an inefficient representation about which we can give concrete guarantees than an efficient one which about we can say nothing. Most of the programs which we have written so far use a limited number of datatypes so that the overhead introduced by the monolithic representation for diamonds is not too severe. However, it is likely that for very large programs this overhead would become unacceptably large. One possibility which we have not yet explored is that it might be possible to achieve more efficient heap usage by using dataflow techniques to follow the flow of diamonds through the program and detect datatypes which are never used in an overlapping way. One could then equip a program with several smaller diamond classes which would represent such non-overlapping types.

These problems could be avoided by compiling to some platform other than the JVM (for example to C or to a specialised virtual machine) where compaction of heap regions would be possible. The Hofmann-Jost algorithm is still applicable in this situation, so it would still be feasible to produce resource guarantees. However, it was a fundamental decision of the MRG project to use the JVM, based on the facts that the JVM is widely deployed and very well-known, and that resource usage is a genuine concern in many contexts where the JVM is used. Our present approach allows us to produce concrete guarantees at the cost of some overhead; we hope that at a later stage a more sophisticated approach (such as the one suggested above) might allow us to reduce the overheads while still obtaining guaranteed resource bounds.

2.5. Remarks. There are various ways in which Camelot could be extended. The lack of higher-order functions is inconvenient, but the resource-aware type systems which we use are presently unable to deal with higher-order functions, partly because of the fact that these are normally implemented using heap-allocated closures whose size may be difficult to predict. A possible strategy for dealing with this which we are currently investigating is Reynolds' technique of *defunctionalization* [14] which transforms higher-order programs into first-order ones, essentially by performing a transformation of the source code which replaces closures with members of datatypes. This has the advantage that extra space required by closures is exposed at the source level, where it is amenable to analysis by the heap-usage inference techniques mentioned earlier.

3. Object-oriented extensions. The core Camelot language as described in Section 2 above enables the programmer to write a program with a predictable resource usage; however, only primitive interaction with the outside world is possible, through command line arguments, file input and printed output. To be able to write a full interface for a game or utility to be run on a mobile device, Camelot programs must be able to interface with external Java libraries. Similarly, the programmer may wish to utilise device-specific libraries, or Java's extensive class library.

This section describes our object-oriented extension to Camelot. This is primarily intended to allow Camelot programs to access Java libraries. It would also be possible to write resource-certified libraries in Camelot for consumption by standard Java programs, or indeed use the object system for OO programming for its own sake, but giving Camelot programs access to the outside world is the main objective.

In designing an object system for Camelot, many choices are made for us, or at least tightly constrained. We wish to create a system allowing inter-operation with Java, and we wish to compile an object system to JVML. So we are almost forced into drawing the object system of the JVM up to the Camelot level, and cannot seriously consider a fundamentally different system.

On the other hand, the type system is strongly influenced by the existing Camelot type system. There is more scope for choice, but implementation can become complex, and an overly complex type system is undesirable from a programmer's point of view. We also do not want to interfere with type systems for resources as mentioned above.

We shall first attempt to make the essential features of Java objects visible in Camelot in a simple form, with the view that a simple abbreviation or module system can be added at a later date to make things more palatable if desired.

3.1. Basic Features. We shall view objects as records of possibly mutable fields together with related methods, although Camelot has no existing record system. We define the usual operations on these objects, namely object creation, method invocation, field access and update, and casting and matching. As one might expect we choose a class-based system closely modelling the Java object system. Consequently we must acknowledge Java's uses of classes for encapsulation, and associate static methods and fields with classes also.

We now consider these features. The examples below illustrate the new classes of expressions we add to Camelot.

Static method calls There is no conceptual difference between static methods and functions, ignoring the use of classes for encapsulation, so we can treat static method calls just like function calls.

```
java.lang.Math.max a b
```

Static field access Some libraries require the use of static fields. We should only need to provide access to constant static fields, so they correspond to simple values.

`java.math.BigInteger.ONE`

Object creation We clearly need a way to create objects, and there is no need to deviate from the `new` operator. By analogy with standard Camelot function application syntax (i.e. curried form) we have:

`new java.math.BigInteger "101010" 2`

Instance field access To retrieve the value of an instance variable, we write

`object#field`

whereas to update that value we use the syntax

`object#field <- value`

assuming that `field` is declared to be a *mutable* field.

It could be argued that allowing unfettered external access to an object's variables is against the spirit of OO, and more to the point inappropriate for our small language extension, but we wish to allow easy interoperability with any external Java code.

Method invocation Drawing inspiration from the O'Caml syntax, and again using a curried form, we have instance method invocation:

`myMap#put key value`

Null values In Java, any method with object return type may return the `null` object. For this reason we add a construct

`isnull` e

which tests if the expression e is a `null` value.

Casts and typecase It may be occasionally be necessary to cast objects up to superclasses, for example to force the intended choice between overloaded methods. We will also want to recover subclasses, such as when removing an object from a collection. Here we propose a simple notation for up-casting:

`obj :> Class`

This notation is that of O'Caml, also borrowed by MLj (described in [15]). To handle down-casting we shall extend patterns in the manner of `typecase` (again like MLj) as follows:

```
match obj with o :> C1 -> o.a()
             | o :> C2 -> o.b()
             | _ -> obj.c()
```

Here `o` is bound in the appropriate subexpressions to the object `obj` viewed as an object of type `C1` or `C2` respectively. As in datatype matches we require that every possible case is covered; here this means that the default case is mandatory. We also require that each class is a subclass of the type of `obj`, and suggest that a compiler warning should be given for any redundant matches.

Unlike MLj we choose not to allow downcasting outside of the new form of `match` statement, partly because at present Camelot has no exception support to handle invalid down-casts.

As usual, the arguments of a (static or instance) method invocation may be subclasses of the method's argument types, or classes implementing the specified interfaces.

The following example demonstrates some of the above features, and illustrates the ease of interoperability. Note that the type of the parameter l is specified by a constraint here. Type inference does not cross class boundaries in Camelot.

```
let convert (l: string list) =
  match l with [] -> new java.util.LinkedList ()
             | h::t ->
```

```
let ll = convert t
in let _ = ll#addFirst h
in ll
```

3.2. Defining classes. Once we have the ability to write and compile programs using objects, we may as well start writing classes in Camelot. We must be able to create classes to implement callbacks, such as in the Swing GUI system which requires us to write stateful adaptor classes. Otherwise, as mentioned previously, we may wish to write Camelot code to be called from Java, for example to create a resource-certified library for use in a Java program, and defining a class is a natural way to do this. Implementation of these classes will obviously be tied to the JVM, but the form these take in Camelot has more scope for variation.

We allow the programmer to define a class which may explicitly subclass another class, and implement a number of interfaces. We also allow the programmer to define (possibly mutable) fields and methods, as well as static methods and fields for the purpose of creating a specific class for interfacing with Java. We naturally allow reference to `this`.

The form of a class declaration is given below. Items within angular brackets $\langle \ldots \rangle$ are optional.

$$\begin{aligned}
\mathit{classdecl} &::= \texttt{class}\ \mathit{cname}\ \texttt{=}\ \langle \mathit{scname}\ \texttt{with} \rangle\ \mathit{body}\ \texttt{end} \\
\mathit{body} &::= \langle interfaces \rangle\ \langle fields \rangle\ \langle methods \rangle \\
interfaces &::= \texttt{implement}\ \mathit{iname}\ \langle interfaces \rangle \\
fields &::= \mathit{field}\ \langle fields \rangle \\
methods &::= \mathit{method}\ \langle methods \rangle
\end{aligned}$$

This defines a class called *cname*, implementing the specified interfaces. The optional *scname* gives the name of the direct superclass; if it is not present, the superclass is taken to be the root of the class hierarchy, namely `java.lang.Object`. The class *cname* inherits the methods and values present in its superclass, and these may be referred to in its definition.

As well as a superclass, a class can declare that it implements one or more interfaces. These correspond directly to the Java notion of an interface. Java libraries often require the creation of a class implementing a particular interface—for example, to use a Swing GUI one must create classes implementing various interfaces to be used as callbacks. Note that at the current time it is not possible to define interfaces in Camelot, they are provided purely for the purpose of interoperability.

Now we describe field declarations.

$$\mathit{field} ::= \texttt{field}\ x\ \texttt{:}\ \tau \ \mid\ \texttt{field}\ \texttt{mutable}\ x\ \texttt{:}\ \tau \ \mid\ \texttt{val}\ x\ \texttt{:}\ \tau$$

Instance fields are defined using the keyword `field`, and can optionally be declared to be mutable. Static fields are defined using `val`, and are non-mutable. In a sense these mutable fields are the first introduction of side-effects into Camelot. While the Camelot language is defined to have an array type, this has largely been ignored in our more formal treatments as it is not fundamental to the language. Mutable fields, on the other hand, are fundamental to our notion of object orientation, so we expect any extension of Camelot resource-control features to object-oriented Camelot to have to deal with this properly.

Methods are defined as follows, where $1 \leq i_1, \ldots, i_m \leq n$.

$$
\begin{aligned}
\mathit{method} ::= {} & \texttt{maker(}x_1\texttt{:}\tau_1\texttt{)} \ldots \texttt{(}x_n\texttt{:}\tau_n\texttt{)}\ \langle\texttt{: super}\ x_{i_1} \ldots x_{i_m}\rangle\ \texttt{=}\ \mathit{exp} \\
& \mid \texttt{method}\ m\texttt{(}x_1\texttt{:}\tau_1\texttt{)} \ldots \texttt{(}x_n\texttt{:}\tau_n\texttt{)}\ \texttt{:}\ \tau\ \texttt{=}\ \mathit{exp} \\
& \mid \texttt{method}\ m\texttt{()}\ \texttt{:}\ \tau\ \texttt{=}\ \mathit{exp} \\
& \mid \texttt{let}\ m\texttt{(}x_1\texttt{:}\tau_1\texttt{)} \ldots \texttt{(}x_n\texttt{:}\tau_n\texttt{)}\ \texttt{:}\ \tau\ \texttt{=}\ \mathit{exp} \\
& \mid \texttt{let}\ m\texttt{()}\ \texttt{:}\ \tau\ \texttt{=}\ \mathit{exp}
\end{aligned}
$$

Again, we use the usual `let` syntax to declare what Java would call static methods. Static methods are simply *monomorphic* Camelot functions which happen to be defined within a class, although they are invoked using the syntax described earlier. Instance methods, on the other hand, are actually a fundamentally new addition to the language. We consider the instance methods of a class to be a set of mutually recursive monomorphic functions, in which the special variable `this` is bound to the current object of that class.

We can consider the methods as mutually recursive without using any additional syntax (such as `and` blocks) since they are monomorphic. ML uses `and` blocks to group mutually recursive functions because its *let-polymorphism* prevents any of these functions being used polymorphically in the body of the others, but this is not an issue here. In any case this implicit mutual recursion feels appropriate when we are compiling to the Java Virtual Machine, and have to come to terms with open recursion.

In addition to static and instance methods, we also allow a special kind of method called a *maker*. This is just what would be called a constructor in the Java world, but as in [16] we use the term maker in order to avoid confusion between object and datatype constructors. The `maker` term above defines a maker of the containing class C such that if `new` C is invoked with arguments of type $\tau_1 \ldots \tau_n$, an object of class C is created, the superclass maker is executed (this is the zero-argument maker of the superclass if none is explicitly specified), expression *exp* (of `unit` type) is executed, and the object is returned as the result of the `new` expression. Every class has at least one maker; a class with no explicit maker is taken to have the maker with no arguments which invokes the superclass zero-argument maker and does nothing. This implicit maker is inserted by the compiler.

3.3. Polymorphism. We remarked earlier that static methods are basically monomorphic Camelot functions together with a form of encapsulation, but it is worth considering polymorphism more explicitly. object-oriented Camelot methods, whether static or instance methods, are not polymorphic. That is, they have subtype polymorphism but not parametric polymorphism (genericity), unlike Camelot functions which have parametric but not subtype polymorphism. This is not generally a problem, as most polymorphic functions will involve manipulation of polymorphic datatypes, and can be placed in the main program, whereas most methods will be interfacing with the Java world and thus should conform to Java's subtyping polymorphism.

3.4. Translation. As mentioned earlier, the present Camelot compiler targets the JVM, via the intermediate language Grail. Translating the object-oriented features which have just been described is relatively straightforward, as the JVM (and Grail) provide what we need. A detailed formal description of the translation process can be found in [17]

3.5. Objects and Resource Types. As described earlier, the use of diamond annotations on Camelot programs in combination with certain resource-aware type systems allows the heap usage of those programs to be inferred, as well as allowing some in-place update to occur. Clearly the presence of mutable objects in object-oriented Camelot also provides for in-place update. However by allowing arbitrary object creation we also replicate the unbounded heap-usage problem solved for datatypes. Perhaps more seriously, we are allowing Camelot programs to invoke arbitrary Java code, which may use an unlimited amount of heap space.

Firstly consider the second problem. Even if we have some way to place a bound on the heap space used by our new OO features within a Camelot program, external Java code may use arbitrary amounts of heap. There seem to be a few possible approaches to this problem, none of which are particularly satisfactory. We could decide to only allow the use of external classes if they came with a proof of bounded heap usage. Constructing a resource-bounded Java class library or inferring resource bounds for an existing library would be a massive undertaking, although perhaps less problematic with the smaller class libraries used with mobile devices. This suggestion seems somewhat unrealistic.

Alternatively, we could simply allow the resource usage of external methods to be stated by the programmer or library creator. This extends the trusted computing base in the sense of resources, but seems a more reasonable solution. The other alternative—considering resource-bound proofs to only refer to the resources directly consumed by the Camelot code—seems unrealistic, as one could easily (and even accidentally) cheat by using Java libraries to do some memory-consuming "dirty work".

The issue of heap-usage *internal* to object-oriented Camelot programs seems more tractable, although we do not propose a solution here. A first attempt might mimic the techniques used earlier for datatypes; perhaps we can adapt the use of diamonds and linear type systems? The use of diamonds for in-place update is irrelevant here, and indeed relies on the uniform representation of datatypes by objects of a particular Java class. Since we are hardly going to represent every Java object by an object of one class we could not hope to have such a direct correlation between diamonds and chunks of storage.

However, we could imagine an abstract diamond which represents the heap storage used by an arbitrary object, and require any instance of `new` to supply one of these diamonds, in order that the total number of objects created is limited. Unfortunately reclamation of such an abstract diamond would only correspond to making an object available to garbage collection, rather than definitely being able to re-use the storage. Even so, such a system might be able to give a measure of the total number of objects created and the maximum number in active use simultaneously.

4. Using threads in Camelot. Previously the JVM had been used simply as a convenient run-time for the Camelot language but the object-oriented extensions described above allow the Java namespace to be accessed from a Camelot application. Thus a Camelot application can now create Java objects and invoke Java methods. Figure 4.1 shows the implementation of a remote input reader in RoundTable, a networked chat application written in Camelot. This example class streams input from a network connection and renders it in a display area in the graphical user interface of the application.

This example shows the Camelot syntax for method invocation (`obj#meth()`), field access (`obj#field`) and mutable field update (`f <- exp`). Both of these are familiar from Objective Caml.

```
(* Thread to read from the network, passing data to a display object *)
class remote = java.lang.Thread
with
  field input : java.io.BufferedReader
  field disp : display
  maker (i : java.io.BufferedReader)(d : display) =
    let _ = input <- i in disp <- d
  method run() : unit =
    let line = this#input#readLine()
    in if isnullobj line then () else
    let _ = this#disp#append line
    in this#run()
end
```

FIG. 4.1. *An extract from the RoundTable chat application showing the OO extensions to Camelot*

This example also shows that even in the object-oriented fragment of the Camelot language that the natural definition style for unbounded repetition is to write recursive method calls. The Camelot compiler converts tail-calls of instance methods (such as **this**#*run*) into while-loops so that methods implemented as in Figure 4.1 run in constant space and do not overflow the Java run-time stack. In contrast recursive method calls in Java are not optimised in this way and would lead to the program overflowing the stack.

A screenshot of a window from the RoundTable application is shown in Figure 4.2. This shows date-and-time-stamped messages arriving spontaneously in the window. The application offers the ability to thread messages by content or to sort them by time. The sorting routine is guaranteed by typechecking to run in constant space because addresses of cons cells in the list of messages are re-cycled using the free list as described in Section 2.2.

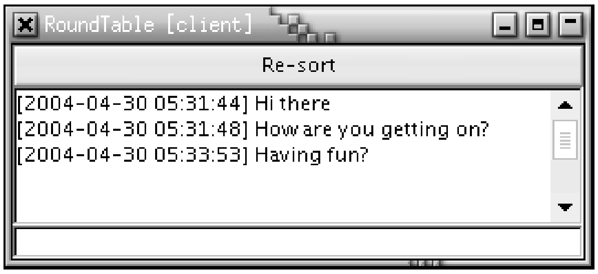

FIG. 4.2. *Screenshot of the Camelot RoundTable application*

The extension of the Camelot compiler to support interoperation with Java facilitates the implementation of graphical applications such as these. The Java APIs used by this application include the Swing graphical user interface components, networking, threads and pluggable look-and-feel components such as the Skin look-and-feel shown above.

5. Management of threads. In designing a thread management system for Camelot our strongest requirement was to have a system which works harmoniously with the storage management system already in place for Camelot. One aspect of this is that the resource consumption of a single-threaded Camelot program can be computed in line with the reasoning explained in Section 1.

In moving from one to multiple threads the most important question with respect to memory usage is the following. Should the free list of storage which can be reused be a single static instance shared across all threads; or should each thread separately maintain its own local instance of the free list?

In the former case the accessor methods for the free list must be synchronised in order for data structures not to become disordered by concurrent write operations. Synchronisation incurs an overhead of locking and unlocking the parent of the field when entering and leaving a critical region. This imposes a run-time penalty.

In the latter case there is no requirement for access to the free list to be synchronised; each thread has its own free list. In this case, though, the free memory on each free list is private, and not shared. This means that there will be times when one thread allocates memory (with a Java **new** instruction) while another thread has unused memory on its local free list. This imposes a penalty on the program memory usage, and this form of thread management would lead to programs typically using more memory overall.

We have chosen the former scheme; we have a single static instance of a free list shared across all threads. Our programs will take longer than their optimum runtime but memory performance will be improved. Crucially, predictability of memory consumption is retained.

There are several possible variants on this second scheme which we considered. They were not right for our purposes but might be right for others. One interesting alternative is a hybrid of the two approaches is where each thread had a bounded (small) local free list and flushes this to the global free list when it becomes full. This would reduce the overhead of calls to access the synchronised global free list, while preventing threads from keeping too many unused memory cells locally. This could be a suitable compromise between the two extremes but the analysis of this approach would inevitably be more complicated than the approach which we adopted (a single static free list).

A second alternative would be to implement weak local free lists. In this construction each thread would have its own private free list implemented using *weak references* which are references that are not strong enough by themselves to keep an object alive if no genuine references to it are retained. Weak references are typically used to implement caches and secondary indexes for data structures. Other high-level garbage-collected languages such as O'Caml implement weak references also. This scheme was not usable by us because the Camelot compiler also targets small JVMs on handheld devices and the J2ME does not provide the necessary class (`java.lang.ref.WeakReference`).

The analysis of memory consumption of Camelot programs is based on the consumption of memory by heap-allocated data structures. The present analysis of Camelot programs is based on a single-threaded architecture. To assist with the development of an analysis method for multi-threaded Camelot programs we require that data structures in a multi-threaded Camelot program are not shared across threads. For example, it is not possible to hold part of a list in one thread and the remainder in another. This requirement means that the space consumption of a multi-threaded

Camelot program is obtained as the sum of per-thread space allocation plus the space requirements of the threads themselves.

At present our type system takes account of heap allocations but does not take account of stack growth. Thus Camelot programs can potentially (and sometimes do in practice) fail at runtime with a `java.lang.StackOverflowError` exception if the programmer overuses the idiom of working with families of mutually-recursive functions and methods which compute with deeply-nested recursion.

Even sophisticated functional language compilers for the JVM suffer from this problem and some, such as MLj [18, 15], do not even implement tail-call elimination in cases where the Camelot compiler does. Several authors consider the absence of support for tail call elimination to be a failing of the JVM [19, 20]. An approach to eliminating tail calls such as that used by Funnel [21] would be a useful next improvement to the Camelot compiler. Techniques such as *trampolining* have also been shown to work for the JVM [22]. The principal reason why the JVM does not automatically perform tail-call optimisation is that the Java security model may require inspection of the stack to ensure that a particular method has sufficient privileges to execute another method; eliminating tail-calls would lead to the discarding of stack frames which contain the necessary security information. However, Clements and Felleisen have recently proposed another security model which allows safe tail-call optimisation [23]; they claim that this requires only a minor change to the mechanism currently used by the JVM (and other platforms), so there may be some hope that future JVM implementations will support proper tail-call optimisation and thus simplify the process of implementing functional languages for the JVM.

6. A simple thread model for Camelot. To retain predictability of memory behaviour in Camelot we restrict the programming model offered by Java's threads.

Firstly, we disallow use of the `stop` and `suspend` methods from Java's threads API. These are deprecated methods which have been shown to have poor programming properties in any case. Use of the `stop` method allows objects to be exposed in a damaged state, part-way through an update by a thread. Use of `suspend` freezes threads but these do not release the objects which they are holding locks on, thereby often leading to deadlocks. Dispensing with pre-emptive thread interruption means that there is a correspondence between Camelot threads and lightweight threads implemented using first-class continuations, `call/cc` and `throw`, as are usually to be found in multi-threaded functional programming languages [24, 25].

Secondly, we require that all threads are run, again for the purposes of supporting predictability of memory usage. In the Java language thread allocation (using **new**) is separated from thread initiation (using the `start` method in the `java.lang.Thread` class) and there is no guarantee that allocated threads will ever be run at all. In multi-threaded Camelot programs we require that all threads are started at the point where they are constructed.

Finally, we have a single constructor for classes in Camelot because our type system does not support overloading. This must be passed initial values for all the fields of the class (because the thread will initiate automatically). All Camelot threads except the main thread of control are daemon threads, which means that the Java Virtual Machine will not keep running if the main thread exits.

This simplified idiom of thread use in Camelot allows us to define *derived forms* for Camelot threads which abbreviate the use of threads in the language. These derived forms can be implemented by *class hoisting*, moving a generated class definition to the top level of the program. This translation is outlined in Figure 6.1.

$$
\begin{array}{ll}
 & \textbf{let rec } \mathit{threadname}(\mathit{args}) = \\
 & \quad \textbf{let } \mathit{locals} = \mathit{subexps} \textbf{ in } \mathit{threadname}(\mathit{args}) \\
 & \textbf{let } \mathit{threadInstance} = \\
 & \quad \textbf{new } \mathit{threadname}(\mathit{actuals}) \textbf{ in } \ldots \\
\rightsquigarrow & \\
 & \textbf{class } \mathit{threadnameHolder}(\mathit{args}) = \mathit{java.lang.Thread} \\
 & \textbf{with} \\
 & \quad \textbf{let rec } \mathit{threadname}() = \\
 & \quad\quad \textbf{let } \mathit{locals} = \mathit{subexps} \textbf{ in } \mathit{threadname}() \\
 & \quad \textbf{method } \mathit{run}() : \mathit{unit} = \\
 & \quad\quad \textbf{let } \_ = \textbf{this}\#\mathit{setDaemon}(\textbf{true}) \\
 & \quad\quad \textbf{in } \mathit{threadname}() \\
 & \textbf{end} \\
 & \textbf{let } \mathit{threadInstance} = \\
 & \quad \textbf{new } \mathit{threadnameHolder}(\mathit{actuals}) \textbf{ in} \\
 & \textbf{let } \_ \ = \ \mathit{threadInstance}\#\mathit{start}() \textbf{ in } \ldots
\end{array}
$$

FIG. 6.1. *Derived forms for thread creation and use in Camelot*

7. Threads and (non-)termination. The Camelot programming language is supported not only by a strong, expressive type system but also by a program logic which supports reasoning about the time and space usage of programs in the language. However, the logic is a logic of partial correctness, which is to say that the correctness of the program is guaranteed only under the assumption that the program terminates. It would be possible to convert this logic into a logic of total correctness which would guarantee termination instead of assuming it but proofs in such a logic would be more difficult to produce than proofs in the partial correctness logic.

It might seem nonsensical to have a logic of partial correctness to guarantee execution times of programs ("this program either terminates in 20 seconds or it never does") but even these proofs about execution times have their use. They are used to provide a bound on the running time of a program so that if this time is exceeded the program may be terminated forcibly by the user or the operating system because after this point it seems that the program will not terminate. Such *a priori* information about execution times would be useful for scheduling purposes. In Grid-based computing environments Grid service providers schedule incoming jobs on the basis of estimated execution times supplied by Grid users. These estimates are sometimes significantly wrong, leading the scheduler either to forcibly terminate an over-running job due to an under-estimated execution time or to schedule other jobs poorly on the basis of an over-estimated execution time.

Because of the presence of threads in the language we now have meaningful (impure, side-effecting) functions which do not terminate so a strong functional programming approach [26] requiring proofs of termination for every function would be inappropriate for our purposes.

8. Related work. The core of the Camelot programming language is a strict, call-by-value first-order functional programming language in the ML family extended with explicit memory deallocation commands and an extended type system which expresses the cost of function application in terms of an increase in the size of the allocated memory on the heap. Other authors have addressed a similar programming model with some variations. Lee, Yang and Yi [27] present a static analysis approach which is used in applying a source-level transformation to insert explicit **free** commands into the program text. Their analysis allows uses of explicit memory deallocation which are not expressible in Camelot due to the linearity requirement of the Camelot type system. Vasconcelos and Hammond [28] present a type system which is superior to ours in applying to higher-order functional programs. Our primary cost computation is memory allocation whereas their primary focus is on run-time abstracted as the number of beta-reductions in the abstract semantic interpretation of the function term against the operational semantics of the language. Our work differs from both of these in considering multi-threaded, not only single-threaded programs.

We have made reference to MLj, the aspects of which related to Java interoperability are described in [15]. MLj is a fully formed implementation of Standard ML, and as such is a much larger language than we consider here. In particular, MLj can draw upon features from SML such as modules and functors, for example, allowing the creation of classes parameterised on types. Such flexibility comes with a price, and we hope that the restrictions of our system will make the certification of the resource usage of object-oriented Camelot programs more feasible.

By virtue of compiling an ML-like language to the JVM, we have made many of the same choices that have been made with MLj. In many cases there is one obvious translation from high level concept to implementation, and in others the appropriate language construct is suggested by the Java object system. However we have also made different choices more appropriate to our purpose, in terms of transparency of resource usage and the desire for a smaller language. For example, we represent objects as records of mutable fields whereas MLj uses immutable fields holding references.

There have been various other attempts to add object oriented features to ML and ML-like languages. O'Caml provides a clean, flexible object system with many features and impressive type inference—a formalised subset is described in [29]. As in object-oriented Camelot, objects are modelled as records of mutable fields plus a collection of methods. Many of the additional features of O'Caml could be added to object-oriented Camelot if desired, but there are some complications caused when we consider Java compatibility. For example, there are various ways to compile parameterised classes and polymorphic methods for the JVM, but making these features interact cleanly with the Java world is more subtle.

The power of the O'Caml object system seems to come more from the distinctive type system employed. O'Caml uses the notion of a *row variable*, a type variable standing for the types of a number of methods. This makes it possible to express "a class with these methods, and possibly more" as a type. Where we would have a method parameter taking a particular object type and by subsumption any subtype, in O'Caml the type of that parameter would include a row variable, so that any object with the appropriate methods and fields could be used. This allows O'Caml to preserve type inference, but this is less important for our application, and does not map cleanly to the JVM.

A class mechanism for Moby is defined in [16] with the principle that classes and modules should be orthogonal concepts. Lacking a module system, Camelot is unable to take such an approach, but both Moby and O'Caml have been a guide to concrete representation. Many other relevant issues are discussed in [30], but again lack of a module system—and our desire to avoid this to keep the language small—gives us a different perspective on the issues.

9. Conclusions and further work. Our ongoing programme of research on the Camelot functional programming language has been investigating resource consumption and providing static guarantees of resource consumption at the time of program compilation. Our thread management system provides a layer of abstraction over Java threads. This could allow us to modify the present implementation to multi-task several Camelot threads onto a single Java thread. The reason to do this would be to circumvent the ungenerous thread limit on some JVMs. This extension remains as future work but our present design strongly supports such an extension.

We have discussed a very simple thread package for Camelot. A more sophisticated one, perhaps based on Thimble [31], would provide a much more powerful programming model.

A possibly profitable extension of Camelot would be to use defunctionalization [14] to eliminate mutual tail-recursion. Given a set of mutually recursive functions $\mathcal{F}$ whose results are of type `t`, we define a datatype `s` which has for each of the functions in $\mathcal{F}$ a constructor with arguments corresponding to the function's arguments. The collection of functions $\mathcal{F}$ is then replaced by a single function `f: s -> t` whose body is a `match` statement which carries out the computations required by the individual functions in $\mathcal{F}$. In this way the mutually recursive functions can be replaced by a single tail-recursive function, and we already have an optimisation which eliminates recursion for such functions. This technique is somewhat clumsy, and care is required in recycling the diamonds which are required to contain members of the datatypes required by `s`. Another potential problem is that several small functions are effectively combined into one large one, and there is thus a danger that that 64k limit for JVM methods might be exceeded. Nevertheless, this technique does overcome the problems related to mutual recursion without affecting the transparency of the compilation process unduly, and it might be possible for the compiler to perform the appropriate transformations automatically. We intend to investigate this in more detail.

Acknowledgements. The authors are supported by the Mobile Resource Guarantees project (MRG, project IST-2001-33149). The MRG project is funded under the Global Computing pro-active initiative of the Future and Emerging Technologies part of the Information Society Technologies programme of the European Commission's Fifth Framework Programme. The other members of the MRG project provided helpful comments on an earlier presentation of this work. Java is a trademark of SUN Microsystems.

REFERENCES

[1] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.

[2] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. 30th ACM Symp. on Principles of Programming Languages*, 2003.

[3] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.

[4] K. MacKenzie and N. Wolverson. Camelot and Grail: Resource-aware functional programming for the JVM. In *Trends in Functional Programming*, pages 29–46. Intellect, 2004.

[5] David Aspinall and Martin Hofmann. Another type system for in-place update. In *Proc. 11th European Symposium on Programming, Grenoble*, volume 2305 of *Lecture Notes in Computer Science*. Springer, 2002.

[6] Michal Konečný. Functional in-place update with layered datatype sharing. In *TLCA 2003, Valencia, Spain, Proceedings*, pages 195–210. Springer-Verlag, 2003. Lecture Notes in Computer Science 2701.

[7] Michal Konečný. Typing with conditions and guarantees for functional in-place update. In

TYPES 2002 Workshop, Nijmegen, Proceedings, pages 182–199. Springer-Verlag, 2003. Lecture Notes in Computer Science 2646.

[8] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. 30th ACM Symp. on Principles of Programming Languages, New Orleans*, 2003.

[9] S. Jost. `lfd_infer`: an implementation of a static inference on heap-space usage. In *Proceedings of SPACE'04, Venice*, 2004. To appear.

[10] Lennart Beringer, Kenneth MacKenzie, and Ian Stark. Grail: a functional form for imperative mobile code. In Vladimiro Sassone, editor, *Electronic Notes in Theoretical Computer Science*, volume 85. Elsevier, 2003.

[11] K. MacKenzie. Grail: a functional intermediate language for resource-bounded computation. LFCS, University of Edinburgh, 2002. Available at `http://groups.inf.ed.ac.uk/mrg/publications/`.

[12] N. Wolverson. Optimisation and resource bounds in Camelot compilation. Final-year project report, University of Edinburgh, 2003. Available at `http://groups.inf.ed.ac.uk/mrg/publications/wolverson.ps`.

[13] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993*, volume 28(6), pages 237–247. ACM Press, New York, 1993.

[14] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, December 1998.

[15] N Benton and A Kennedy. Interlanguage working without tears: Blending SML with Java. In *Proceedings of the 4th ACM SIGPLAN Conference on Functional Programming*, Paris, September 1999. ACM Press.

[16] K. Fisher and J. Reppy. Moby objects and classes, 1998. Unpublished manuscript.

[17] N. Wolverson and K. MacKenzie. O'Camelot: adding objects to a resource-aware functional language. In *Proceedings of TFP2003*, pages 47–62. Intellect, 2004.

[18] N Benton, A Kennedy, and G Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the 3rd ACM SIGPLAN Conference on Functional Programming*, Baltimore, sep 1998. ACM Press.

[19] Nick Benton. Some shortcomings of, and possible improvements to, the Java Virtual Machine. This is an unpublished note which is available on-line at `http://research.microsoft.com/~nick/jvmcritique.pdf`, June 1999.

[20] Erik Meijer and Jim Miller. Technical Overview of the Common Language Runtime (or why the JVM is not my favourite execution environment). URL: `http://docs.msdnaa.net/ark/Webfiles/whitepapers.htm`, 2001.

[21] Michel Schinz and Martin Odersky. Tail call elimination on the Java Virtual Machine. In *Proceedings of Babel'01*, volume 59 of *Electronic Notes in Theoretical Computer Science*, 2001.

[22] David Wakeling. Compiling lazy functional programs for the Java Virtual Machine. *Journal of Functional Programming*, 9(6):579–603, November 1999.

[23] J. Clements and M. Felleisen. A tail-recursive machine with stack inspection. *ACM Transactions on Programming Languages and Systems*. To appear.

[24] Edoardo Biagioni, Ken Cline, Peter Lee, Chris Okasaki, and Chris Stone. Safe-for-space threads in Standard ML. *Higher-Order and Symbolic Computation*, 11(2):209–225, 1998.

[25] Peter Lee. Implementing threads in Standard ML. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming, Second International School, Olympia, WA, USA, August 26-30, 1996, Tutorial Text*, volume 1129 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 1996.

[26] David Turner. Elementary strong functional programming. In R.Plasmeijer and P.Hartel, editors, *Proceedings of the First International Symposium on Functional Programming Languages in Education*, volume LNCS 1022, Nijmegen, Netherlands, December 1995. Springer.

[27] Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. Inserting safe memory reuse commands into ML-like programs. In *Proceedings of the 10th Annual International Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 171–188. Springer-Verlag, 2003.

[28] Pedro B. Vasconcelos and Kevin Hammond. Inferring costs for recursive, polymorphic and higher-order functional programs. In Greg Michaelson and Phil Trinder, editors, *Proceedings of the 15th International Workshop on the Implementation of Functional Languages*, LNCS. Springer-Verlag, 2003. To appear.

[29] Didier Remy and Jerome Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[30] David MacQueen. Should ML be object-oriented? *Formal Aspects of Computing*, 13(3-5), 2002.

[31] Ian Stark. Thimble — Threads for MLj. In *Proceedings of the First Scottish Functional Programming Workshop*, number RM/99/9 in Department of Computing and Electrical Engineering, Heriot-Watt University, Technical Report, pages 337–346, 1999.