Juaskell: Implementing Evaluation Strategies in Java.

Kevin Hammond and David Walsh-Kemmis

Division of Computer Science, University of St Andrews,
North Haugh, St Andrews, Fife, KY16 9SS, Scotland.
Email: {kh,dhw}@dcs.st-and.ac.uk

Abstract

We report on a novel attempt to incorporate functional language technology in the widely used object-oriented language Java. Juaskell takes the concept of evaluation strategies, which are designed to cleanly separate control from algorithm in a parallel setting, from Glasgow Parallel Haskell and re-implements this technology in Java. While Java does have rudimentary support for concurrent threads, experience has shown that the standard mechanism is difficult to exploit correctly, and error-prone in use. It is thus highly desirable to provide high-level behavioural abstractions over threads, such as evaluation strategies.

The implementation of Juaskell involves finding a number of analogues to common functional language concepts and implementation techniques in an object-oriented setting, notably higher-order functions, parametric polymorphism and lazy evaluation. Solutions to some of these problems proved to be more complex than expected, due to deficiencies in the Java language definition. The resulting Juaskell system has been used to implement a multi-threaded version of the Mandelbrot algorithm. For this program, which involves the creation of a large number of threads, performance overhead due to strategic code amounts to 9.6% of the total execution time, an acceptably low figure.

1 Introduction

This paper reports on an experimental implementation that aims to bring functional language techniques in contact with a wider audience, through adapting techniques developed in a functional setting to an object-oriented framework.

Java is a relatively clean and simple object-oriented language that provides direct support for concurrent threads [Sun95]. Experience has shown, however, that this support is difficult to manage, frequently leading to deadlock [Hol98]. It is also very low-level, frequently leading to complicated and perhaps incomprehensible code.

This paper describes the integration of *evaluation strategies* — a high-level approach to controlling parallel evaluation [THL98] — with standard Java threads. Using evaluation strategies, it is possible to program threaded code at a much higher level of abstraction than normal. In particular, code dealing with parallel control can be cleanly separated from that dealing with algorithmic content. By following simple design rules, it is also possible to ensure the absence of deadlock, without needing to resort to complex user-defined locking code, and priority control mechanisms.

A primary goal of our research is that it should be possible to use a completely standard implementation of the Java language. While evaluation strategies rely on concepts taken from functional programming languages such as Haskell [PH+97], it is also vitally important to implement evaluation strategies as naturally as possible in Java. That is, unlike much previous work [e.g. BKR98, Wak97], *we are not trying to implement a functional language in Java, but rather we hope to transfer our technology into a different setting with the expectation that this will provide benefits to another community*.

Although the focus of our research is on parallel evaluation, at this stage we are concerned with demonstrating proof of concept in terms of language rather than performance gains. We hope however that it will ultimately be possible to exploit parallel implementations of Java that have been developed elsewhere, especially if these implementations use the standard Java threading interface that we have also exploited.

The remainder of this paper is structured as follows. Section 2 introduces evaluation strategies, while Section 3 describes their implementation in Java. Section 4 covers issues concerning parallel implementation. Section 5 describes a simple concurrent demonstrator application, Section 6 discusses related work, and Section 7 concludes.

2 Evaluation Strategies

Evaluation strategies exploit higher-order functions and lazy evaluation to allow control behaviour to be separated from algorithmic code. Unlike approaches such as coordination languages, they can be defined entirely in terms of the host computation language, provided this is sufficiently expressive to support the introduction of new control constructs, as is the case for most functional languages. In a functional setting, the use of higher-order functions allows strategies to be composed to give both nested control and pipelined control.

While evaluation strategies were originally intended to be used for parallel control, there is no reason in principle why they cannot be adapted to other settings, such as concurrency. Such an evolution would require some of the standard assumptions to be reconsidered.

In Haskell terms, an evaluation strategy is a function from some type to the trivial type `()`. The purpose of the strategy is to perform some action that has an operational effect. The trivial type is used to indicate that the result of applying the strategy is uninteresting.

```
type Strategy s = s -> ()
```

For example, a strategy to force the evaluation of an integer could be written

```
forceInt :: Strategy Int
forceInt n | n == n = ()
```

This relies on the fact that demanding the value of *n* in the equality test will cause the argument to be evaluated. The result of the evaluation is shared because GPH uses graph reduction. So the following expression

```
let test = fib 25 in
case forceInt test of
   () -> (test,test)
```

will return a pair containing two copies of the value of *test*, which will have been fully evaluated as a result of the call to *forceInt*. Similarly, a strategy to evaluate the spine of a list could be written as

```
spineList :: Strategy [a]
spineList [] = ()
spineList (a:as) = spineList as
```

This strategy recurses over the structure of its argument, returning `()` when the end of the list is reached.

This use of forcing functions is rather clumsy since it intersperses behavioural code (the forcing functions) with functional code (the result of the expression). In the GPH world, basic parallelism and sequencing can be introduced using the `par` and `seq` combinators: `c` `par` e *sparks c* as a new thread and continues evaluating *e*, where e1 `seq` e2 evaluates *e1* then continues by evaluating *e2*. Evaluation strategies exploit these primitive control combinators to allow the separation of behaviour from algorithm.

```
x `using` s = s x `seq` x
```

The using function applies the strategy *s* to the value *x,* and returns *x* as its result. This allows the strategy to be cleanly demarked from the algorithmic code. For example

```
take 5 primes  `using` spineList
```

will evaluate the spine of the list containing the first 5 primes, and return that partially evaluated list as its result.

A more interesting use is for parallel evaluation

```
parList :: Strategy a -> Strategy [a]
parList s [] = ()
parList s (a:as) = s a `par` parList s as
```

The parList strategy will spark a new parallel thread for each element of its second argument, a list of some type. Each of these threads will apply the strategy *s* to the corresponding element of the list.

```
l `using` parList full
```

will spark a thread to fully evaluate each element of *l*,. whereas

```
l `using` parList none
```

is equivalent to a use of spineList it sparks a thread for each element of *l*, but these threads perform no evaluation. Taken in conjunction with lazy evaluation, evaluation strategies can be used to enforce parallel synchronisation and avoid deadlock.

The astute reader will have noticed a difference in terminology from our earlier work. In order to make evaluation strategies more accessible to those without a functional programming background, we have chosen to use the names *full* rather than *rnf*, *first* rather than *rwhnf* and *none* rather than *r0*. The name *first* is intended to evoke the idea of evaluating the first element of a list. The underlying strategies have the same behavioural semantics as before, however.

3 Implementation in Java

In order to implement evaluation strategies in Java, it is necessary to define an equivalent to higher-order functions, and to provide a mechanism that has an equivalent effect to lazy evaluation. It is also desirable to find an equivalent to parametric polymorphism, in order to allow the definition of generic, possibly parameterised, evaluation strategies.

3.1 Closures

Lazy evaluation is implemented in Juaskell by defining closures to be objects which represent suspended computations (i.e. direct analogues to *suspensions* [Pey87]). As a result of evaluation, a closure will become the object that is the result of evaluating the suspension.

Closures are implemented as objects containing three main fields: an evaluation function; the arguments to the function; and concrete data that may be needed by the closure. This representation is shown in Figure 1. Each closure object also supports a number of standard methods, of which the most important are:

```
public Closure using(Strategy s)
public Closure value()
public Closure force()
```

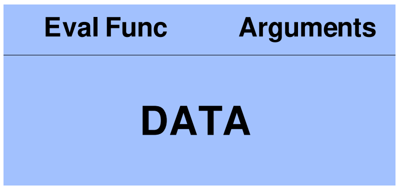

Figure 1: Generic form of a closure

The *force* method evaluates the closure using the default evaluation mechanism (normally to head normal form); *value* returns the value of the closure as an object after forcing evaluation if necessary; and *using* applies the given strategy to the closure. These are all normal Java methods. For example to apply the *full* strategy to closure *c*, we can simply write `c.using(full)`.

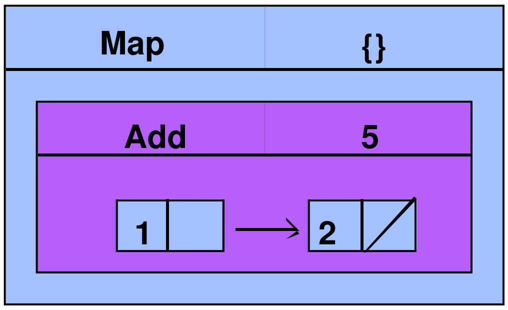

Figure 2: a list closure representing map (+ 5) [1,2]

3.2 Higher-Order Functions

Higher order functions present certain difficulties in Java. The solution we have adopted is to use the standard reflection package of Java 1.1 to construct first order methods from a template and the appropriate concrete arguments. The newly constructed method can then be invoked to yield a closure result. While this is potentially inefficient and somewhat clumsy to construct, there appears to be no simpler solution in standard Java.

Figure 2 shows the representation of a list closure which maps an adding function over a simple list. Here the data part of the closure is itself a closure containing a function. Evaluating the map closure will cause the inner function to be applied to each element of the list in turn, so creating a list of closures, which can then be evaluated on demand to yield the final result. The value of this result can be extracted as a list object using the *value* method.

3.3 Parametric Polymorphism

In order to support polymorphic strategies, it is necessary to treat all values in the same way. In the absence of direct language support for parametric polymorphism in Java (as proposed, for example, by Myers, Bank and Liskov [MBL97]) this must be achieved by using a suitably generic representation for values.

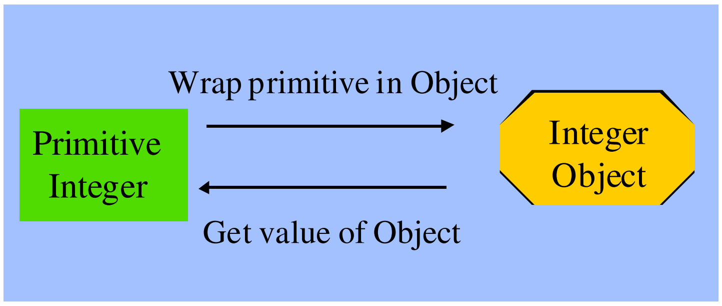

Figure 3: Parametricity

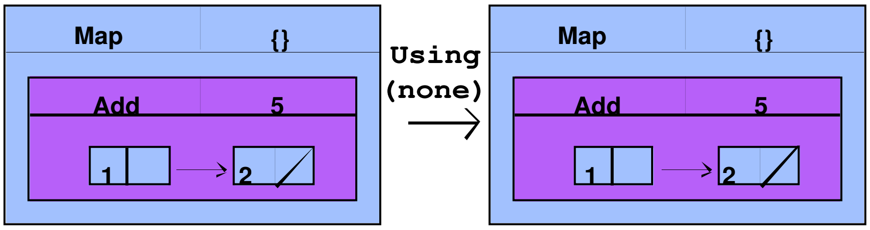

Figure 4: The none Evaluation Strategy

In a functional setting, this is usually done by *boxing* primitive values so that all values have a uniform representation. In an object-oriented setting, the equivalent approach is to treat all values as objects. Unfortunately, Java deviates from the object-oriented ideal in that primitive values such as integers are not objects (it could be argued that this is done for reasons of efficiency, were it not for the fact that Java implementations are notoriously inefficient).

The solution we have adopted is to box primitives, so providing a uniform interface, and to unbox them explicitly when required (using the *intValue* method for *ints* etc.). This is illustrated in Figure 3.

3.5 Basic Sequential Evaluation Strategies

Figures 4–6 illustrates the three basic evaluation strategies: *none, first* and *full*, showing their effect on the *map* closure of Figure 2. The *none* strategy has no effect on the attached closure as shown in Figure 4. Figure 5 shows the effect of the *full* strategy: the closure is transformed into a linked list containing the results of the evaluation. Each of the list objects is boxed in order to support lazy evaluation.

Unfortunately, it proved to be impossible to support lists as a subclass of *Closure* as we had originally intended, so the representation we have used requires the closure's data field to be followed to obtain the value of each cons cell. This is clumsier than we would like, and complicates the provision of list processing code, but appears to be needed to support lazy lists.

Finally, Figure 6 shows the effect of using the *first* evaluation strategy. In this case only the first element of the list is evaluated (and only to a closure that when demanded will actually perform the addition), and the tail of the list becomes a closure mapping the addition function over the tail of the list.

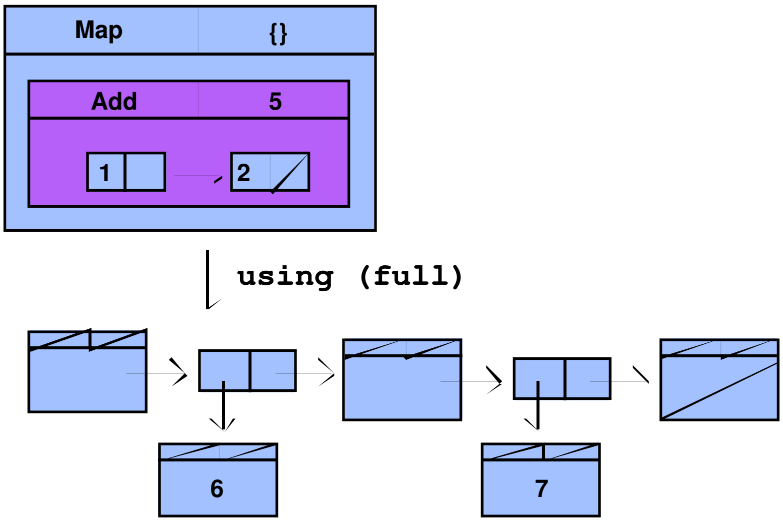

Figure 5: the full Evaluation Strategy

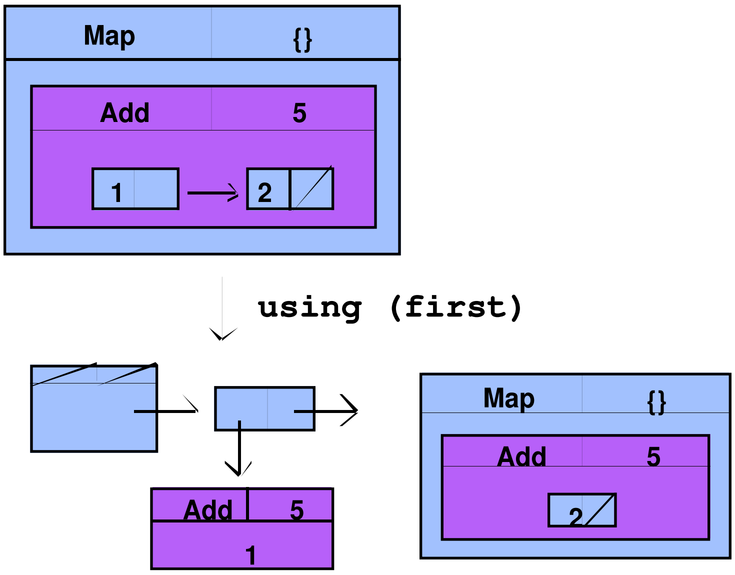

Figure 6: the first Evaluation Strategy

4 Parallelism

Parallelism is currently simulated using Java language threads, though we intend to modify our system to exploit the parallel Java// implementation of Java, which provides a slightly different interface to parallel threads [CaVa97].

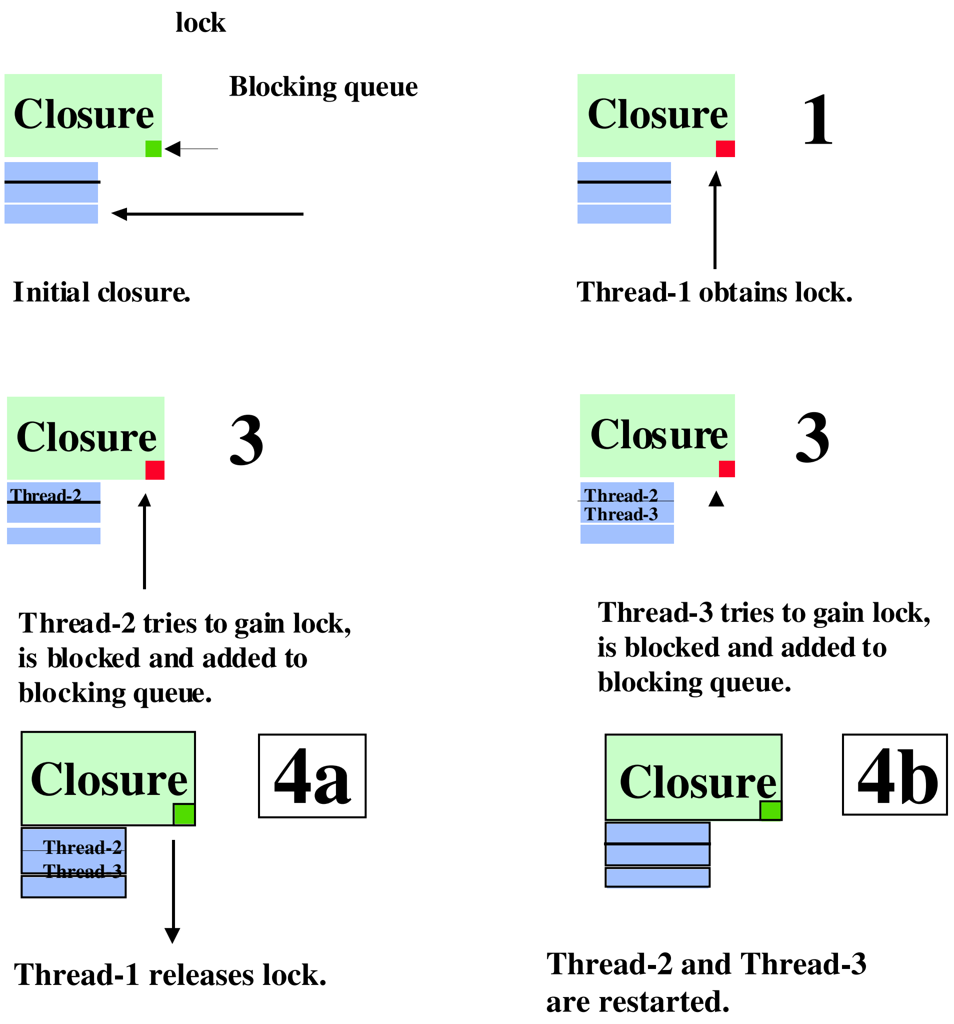

Figure 7: Avoiding Deadlock

In order to support parallelism in Juaskell, we implement two new methods: *spark* creates a thread to evaluate an object using the *first* strategy, *sparker* creates a thread to evaluate an object using any strategy. For example, the *parList* strategy sparks a new thread for each element in a list. Each thread evaluates its list element using the strategy that is passed as the parameter to the *parList* method. The code for *parList is:*

```
parList (Closure c) {
  …
  ((ObjList)c.data).head.sparker(s);
  ((ObjList)c.data).tail.using(new ParList(s));
  …
}
```

4.1 Avoiding Deadlock

In order to avoid deadlock we introduce a locking protocol taken directly from our parallel implementations, based on the well-known sequential technique of *black-holing*. In the Juaskell setting, this technique works as follows (illustrated in Figure 7):

1. The first thread to evaluate a closure sets a lock, to show that the closure is currently under evaluation. This lock is the equivalent of a black hole flag.

2. This thread carries out the evaluation.

3. If another thread tries to access the closure:

   a) It becomes blocked.

   b) It is added to a queue of threads waiting to access the closure: the blocking queue.

4. When the evaluating thread has finished, it must:

   a) Unset the lock: the closure is no longer under evaluation.

   b) Restart all threads in the blocking queue.

5. Restarted threads now access the value of the closure, with no need to lock it.

4.2 Pipelining

Pipelining creates a thread to convert the list to a form where another thread can simply scan down the list and get the values. The second thread (the consumer) follows the first one (the producer) down the list as shown in Figure 8. More sophisticated pipes are possible.

5 Using Strategies in a Mandelbrot program

As a first attempt at exploring the use of evaluation strategies in an object-oriented setting, we have implemented a multi-threaded version of a Mandelbrot set generator. `parList` strategies are used to introduce independent threads to compute sub-components of the final result (in total approximately 4000 threads were introduced for our test program). The resulting program should be easily parallelisable, using e.g. the Java// parallel threads package [CaVa97]. The time taken to adapt the standard version to use strategies was negligible, and the program is not significantly longer than the unthreaded version.

On a single processor machine, measurements show that evaluation strategies bear an overhead of 9.6% of execution time compared with manual introduction of the same threads. This is due mainly to the cost of packaging the set of complex numbers into lists so that they can be manipulated by our standard strategic functions, and then manipulating these lists. The multi-threaded version is, of course, much slower than the non-threaded version, since the standard implementation of Java threads is extremely inefficient. Properly designed parallel thread packages, such as Java//, should, however, bear much lower costs.

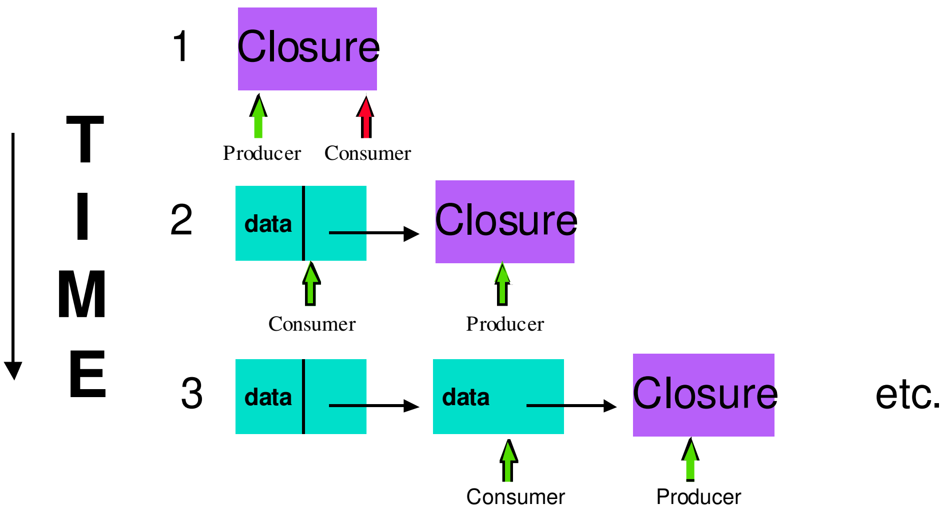

Figure 8 Using Evaluation Strategies to Introduce Parallel Pipelining

6 Related Work

6.1 Functional Language Implementations in Java

While our primary aim is not to provide a new functional language implementation in Java, it may be possible to exploit the results of such implementation work. We are aware of three prototype functional language implementations in Java. Wakeling has implemented a Haskell to Java code compiler called VSD [Wak97] and Benton et al. have implemented a similar code generator for ML [BKR98]. Finally, there is a Java implementation of the lazy parallel functional language Ginger, which exploits Java threads to simulate parallel evaluation [AmJo98].

In common with our approach, Wakeling's closure representation is also based on hierarchically structured classes, supporting an evaluation method plus other methods to provide the basic G-machine operations on objects, such as stack unwinding. Both Wakeling and Benton et al. report poor performance from the current implementation of the Java Virtual Machine, and this accords with our own experience: the JVM runs at approximately the same speed as the HUGS interpreter.

Unlike the first two implementations, the Ginger implementation uses a more interpretive combinator-based approach. Programs are explicitly partitioned into sub-graphs, each of which is executed by a different Java thread. In producing this implementation, Amatya and Joy encountered many of the problems with Java threads that we alluded to earlier, including deadlock avoidance and lack of control over thread scheduling. Their experience adds weight to our contention that Java

6.2 High-Level Control of Parallelism in Object-Oriented Languages

At a more general level, there have been several attempts to provide higher-level control of parallelism in an object-oriented setting, of which the best known is perhaps the Actor model [Agh86]. Rather than abstracting from objects as units of control, however, as we have done, the Actor model treats each object as an independent agent, capable of communicating with any other agent. In constrast with our approach, this does not provide a handle on deadlock avoidance, nor does it allow abstraction of behaviour. The latter is important for the construction of high-level models of parallel behaviour (e.g. [HBT+98], which describes an operational semantics for the Glasgow Parallel Haskell language), as well as making it easier to comprehend parallel programs.

Some more recent parallel object-oriented systems also have some similarity to the ideas introduced here, though they are normally much less general in scope. For example, ICC++ [CRP+96] and pC++ [BBG+93] both allow the user to define data-parallel *collections* that have a similar purpose to our `parList` strategies. The COOL language [CGH94] provides task-level parallelism by supporting parallel functions within classes. Juaskell, in contrast, supports both task- and data-parallelism through different strategies, and allows variable levels of granularity control. Finally, the Cid language [Nik95] exploits ideas from the Id dataflow language to provide a simple extension of C with explicit fork/join primitives, globally synchronised objects and automatic granularity control. Cid therefore focuses on mechanisms that can be used to introduce parallelism rather than focussing on high-level behavioural abstraction as in this paper.

6.3 Parallel Java Implementations

Several parallel implementations of Java have now been produced. The most directly relevant of these is the Java// [CaVa97] system, which supports synchronisation between local and global objects using “wait-by necessity”, a low level version of the blocking mechanism we have used in Section 4.1.

6.4 Pizza: Polymorphism and Higher-Order Functions in Java

Pizza extends Java with parametric polymorphism and higher-order functions [OdWa97]. In contrast to our approach Pizza is a strict superset of Java, which is then translated into standard Java. Using Pizza would certainly have avoided many of the difficulties that we have described above. However, a primary objective of our research was to use *completely standard* Java in order to allow evaluation strategies to be added to existing sequential Java programs. We therefore reluctantly rejected Pizza as our base language of choice.

7 Summary and Conclusions

We have full and working implementations of Juaskell in Java, integrated with the standard Java threads package. Our implementation works with a completely unmodified version of Java version 1.1. Since it is a series of user-space packages, it should be easy to extend or modify our implementation as needed. To date we have only implemented strategies for primitive types and for our own list package. It would not be difficult to provide strategies for other user-defined or library data structures such as trees, however.

We have successfully used Juaskell to implement a concurrent version of a simple Mandelbrot set generator. Compared with a non-strategic version of the program, this carries an overhead of 9.6%, which seems an acceptable tradeoff for the advantages of deadlock avoidance and abstraction that we claim.

In constructing the Juaskell system, a number of unexpected difficulties arose. We had hoped that it would be possible to define strategies directly as Java methods. However, it is neither possible to pass methods as parameters, nor to store them in objects — in Java, methods are definitely not first-class citizens! The solution adopted here (using the reflection package) is both inefficient and rather clumsy to use.

The fact that integers and other primitive values are not objects in Java represents a curious non-orthogonality of design that required us to implement explicit boxing and unboxing operations. This adds a level of obscurity that we had hoped to avoid.

Thirdly, we had hoped that we could use Java's overloading mechanism as a reasonable substitute for Haskell's type class mechanism. It is not possible in Java, however, to inherit methods arbitrarily, and this restricted the construction of generic methods, requiring new definitions for each new class.

Finally, we had hoped to use Java's built-in object locking mechanism to ensure synchronisation without requiring explicit locks. Java's operational semantics for threads do not, however allow such a simple and elegant design, and we have consequently implemented the black hole mechanism described above. It was also necessary to take great care with Java thread priorities in order to enforce correct behaviour. On the basis of our experience, providing simple deadlock-free concurrency control will make programming with threads much more accessible to general programmers, and should also reduce the likelihood of producing programs containing obscure thread-related bugs. We note that our system will not *absolutely prevent* a Java programmer from writing deadlock-inducing code — we have not prohibited side-effects on shared variables — indeed this cannot be done without stepping outside the Java language! Deadlock can, however, be avoided by exclusive use of Juaskell objects to communicate shared values between threads.

The difficulties with polymorphism and higher-order functions would, of course, be overcome by the use of Pizza [OdWa97] rather than Java. However, our objective is to use completely standard Java mechanisms, so as to ensure complete transparency for strategic programmers and also to allow the use of strategies with existing non-strategic programs.

In conclusion, we feel that we have demonstrated the in-principle feasibility of using evaluation strategies for an object-oriented language. While the integration is not as elegant as we would have liked, due to deficiencies of the Java language design, it does appear to be usable and should dramatically reduce the possibility of error when writing multi-threaded programs. While we have not yet demonstrated real speedups using strategic parallelism in this context, we hope to pursue this line of research in the near future, and to report on our findings shortly. Finally, as has also been done with Pizza, we have shown that technology developed in a purely functional context can be usefully applied in other language settings.

References

[Agh86] Agha, G., *ACTORS: A Model of Concurrent Computation in Distributed Systems.* Series in Artificial Intelligence. MIT Press Cambridge, Mass, 1986.

[AmJo98] Amatya, A.M.S, and Joy, M., "Threading Ginger — Version 0.1", Submitted to *1998 Intl. Workshop on Impl. Func. Langs. (IFL '98)*, UCL, London, Sept. 1998.

[BKR98] Benton, N., Kennedy, A., and Russell, G., "Compiling Standard ML to Java Bytecodes", *Proc. ICFP '98*, Baltimore, MD, Sept. 1998.

[CaVa97] D. Caromel, J Vayssiere, "A Java Framework for Seamless, Sequential, Multi-threaded and Distributed Programming" *Proc. ACM Workshop on Java for High-Performance Network Computing,* Palo Alto CA., 1998, pp. 141–150.

[CGH94] R. Chandra, A. Gupta, J. Hennessy, "Integrating Concurrency and Data Abstraction in the COOL Parallel Programming Language", *IEEE Computer*, **27**(8), pp. 14–26, February 1994.

[BBG+93] Bodin, F., Beckman, P., Gannon, D, Yang, S, Kesavan, S., Maloney, A. and Mohr, B., "Implementing a Parallel C++ Runtime System for Scalable Parallel Systems", *Proc. Supercomputing '93,* Portland, OR, 1993, pp. 588–597.

[CRP+96] A.A. Chien, U.S. Reddy, J. Plevyak, J. Dolby, "ICC++ - A C++ Dialect for High Performance Parallel Computing", *Dept of Comp. Sci.., Univ. of Illinois*, 1996.

[HBT+98] Hall, J.G., Baker-Finch, C., Trinder, P.W. and King, D.J., "An Operational Semantics for Parallel Lazy Evaluation", Submitted to *1998 Intl. Workshop on Impl. Func. Langs. (IFL '98)*, UCL, London, Sept. 1998.

[Hol98] Holub, A., "Programming Java threads in the real world", *Javaworld,* Sep. 1998.

[MBL97] Myers, A.C., Bank, J.A., and Liskov, B., "Parameterized Types for Java", *Proc. 1997 ACM Symp. on Principles of Prog. Langs.,* pp. 132–145, Jan. 1997.

[Nik95] Nikhil, R.S., "Parallel Symbolic Computing in Cid", *Proc. Workshop on Parallel Symbolic Comp.*, Springer-Verlag LNCS 1068, pp. 217–242, 1995.

[OdWa97] Odersky, M. and Wadler, P.L., "Pizza into Java: Translating Theory into Practice", *Proc. 1997 ACM Symp. on Principles of Prog. Langs. (POPL '97),* Paris, France, pp. 146–159, Jan. 1997.

[PHA+97] *Report on the Functional Language Haskell,* Version 1.4, Peterson J.W. and Hammond K. (Eds.), L. Augustsson, B. Boutel, F.W. Burton, J. Fairbairn, J.H. Fasel, A.D. Gordon, M.M. Guzmán, P. Hudak, R.J.M. Hughes, T. Johnsson, M.P. Jones, R. Kieburtz, R. Nikhil, W.D. Partain, S.L. Peyton Jones, and P.L. Wadler, Technical Report, Yale University, April 1997.

[Pey87] Peyton Jones, S.L. *The Implementation of Functional Programming Languages*, Prentice Hall International, 1987.

[Sun95] Sun Microsystems, Inc., "The Java Language Environment: A White Paper", Sun Microsystems, Inc., Mountain View, CA, 1995.

[THM+96] Trinder, P.W., Hammond, K., Mattson, J.S. Jr., Partridge, A.S., and Peyton Jones, S.L., "GUM: A Portable Parallel Implementation of Haskell", *1996 ACM Conf. on Prog. Lang. Design and Impl. (PLDI '96)*, pp. 79–88, 1996.

[THL98] Trinder, P.W., Hammond, K., Loidl, H.-W. and Peyton Jones, S.L., "Algorithm + Strategy = Parallelism", *J. Functional Programming,* **8**(1):23—60, Jan. 1998.

[Wak97] Wakeling, D., "VSD: a Haskell to Java Virtual Machine Code Compiler", *1997 Intl. Workshop on Implementing Functional Languages (IFL '97)*, St Andrews, Scotland, Springer-Verlag LNCS 1467, 1997, pp. 39–52.