A Declarative Framework for Analysis and Optimization

Henry Falconer, Paul H J Kelly, David M Ingram, Michael R Mellor,
Tony Field and Olav Beckmann

Department of Computing, Imperial College London
180 Queen's Gate, London SW7 2BZ, U.K.
{p.kelly}@imperial.ac.uk

Abstract. DeepWeaver-1 is a tool supporting cross-cutting program analysis and transformation components, called "weaves". Like an aspect, a DeepWeaver weave consists of a query part, and a part which may modify code. DeepWeaver's query language is based on Prolog, and provides access to data-flow and control-flow reachability analyses. DeepWeaver provides a declarative way to access the internal structure of methods, and supports cross-cutting weaves which operate on code blocks from different parts of the codebase simultaneously. DeepWeaver operates at the level of bytecode, but offers predicates to extract structured control flow constructs. This paper motivates the design, and demonstrates some of its power, using a sequence of examples including performance profiling and domain-specific performance optimisations for database access and remote method invocation.

Introduction Aspect-oriented programming tools, such as AspectJ [12], can be used to implement performance optimizations. However, tools like AspectJ are too weak, both to perform interesting optimizing transformations, and to capture the conditions for their validity. Similar problems arise when using AspectJ for static program analysis, e.g. to check usage rules for library code, or to detect software defects. For many, this is a consequence of deliberate simplicity in the aspect language design. This paper presents a prototype system which is powerful enough to express complex analyses (and transformations) - yet, like an aspect weaver, retains a declarative style by which some simplicity can be retained. The tool is motivated and illustrated using a series of examples, including intra-method performance profiling, and domain-specific optimizations for database access and remote method invocation.

Example We begin with a very simple motivating example, a domain-specific performance optimisation for Java code that uses the JDBC (Java Database Connectivity) library. Consider this Java fragment:

```
ResultSet staff = statement.executeQuery("SELECT * FROM employees");
ResultSet clients = statement.executeQuery("SELECT * FROM customers");
... complex and messy code that uses clients but not staff ...
```

The first "executeQuery" call is redundant since its result set "staff" is never used. With DeepWeaver-1, we can write a weave that eliminates such redundant calls; see Figure 1.

The query part of the weave is a subset of ISO Prolog, with a rich set of built-in predicates to query properties of Java bytecode. When the query succeeds, the value bound to the weave's parameters (in this case "ExecCall") are passed to the Java action.

```
weave removeSelect(CodeBlock ExecCall):
  method("ResultSet Statement.executeQuery(String)", ExecMethod),
  call(ExecMethod, _, ExecCall, _),
  assignment(Result, ExecCall, _),
  \+ reaching_def(Result, _, _).
{
  System.out.print("Removing redundant SELECT at ");
  System.out.println(ExecCall.method);
  ExecCall.remove();
}
```

Fig. 1. A complete DeepWeaver-1 weave to locate and remove "executeQuery" calls whose results could never be used. The Java action is triggered for each instance of the Prolog variable "ExecCall" for which the query succeeds. The \+ operator is Prolog's "not", and in this example it succeeds if no match for the "reaching_def" predicate can be found.

How it works: The first step in Figure 1 is to find "ExecMethod", the body of the "executeQuery" method. Then we find a call to this method, "ExecCall". Then we find the assignment of "ExecCall"s return value to a variable "Result". Finally, we check that no *reaching definition* for "Result" can be found - that is, that the value returned by the "executeQuery" call is not used. Note that the SQL query string (`String`) cannot induce side effects, so the transformation is always valid, provided no exceptions are thrown.

1 Contributions

This paper introduces the DeepWeaver-1 language design, and illustrates its power using selected examples. The key contributions offered include:

– JoinPoints are `CodeBlock`s: the query part of a weave binds the weave parameters to `CodeBlock`s, which are contiguous regions of bytecode. The action part of the weave can then operate on these `CodeBlock`s, removing, modifying or replacing them. We show that this simple idea is remarkably effective.

– Structured control flow is rendered as predicates, which yield `CodeBlock`s. Thus, we operate on a low-level intermediate representation, yet can analyse code in structured, source-code terms - independently of source code details.
– Actions can operate on `CodeBlock`s from different parts of the codebase – thus, weaves can be truly "cross-cutting". The parameters bound by an action's query may refer to `CodeBlock`s gathered from disparate parts of the codebase.
– DeepWeaver-1's "interjections" provide a way for an action to inline advice before, after, around or instead of any `CodeBlock`. Interjections provide typed templates which can be bound to free variables at the context of use. Interjections can, furthermore, be specialised by replacing placeholder method calls.
– We introduce, motivate and demonstrate these concepts by means of a series of example applications. We conclude with a critical evaluation of the success of the work.

2 Background

Our primary motivation has been to build performance optimisation tools, in particular tools which embody domain-specific performance optimisation expertise. We have found AspectJ a remarkably valuable tool for building extensible profilers. We have also explored "domain-specific optimisation components" - which apply performance optimizations automatically. Our experience has been extremely positive: using AOP techniques has the potential to reduce dramatically the complexity of such software. It has also been frustrating; our requirements stretch the capabilities of conventional aspect languages. This paper reports on our first attempt to build a tool that does what we want.

We are not the first to explore these ideas. AspectJ extensions such as tracematch [4] and dataflow pointcut [15] cover some of the same ground. Meanwhile quite a number of tools have used Prolog or Datalog to query the codebase [9, 13, 18]. We review these and other work from the literature, and contrast them to DeepWeaver-1, in Section 5.

3 The design of a deeper weaver

Program representation A key feature of our design is our decision to operate on a low-level intermediate representation (our implementation is built on SOOT [17]). A common alternative is to work with the abstract syntax tree (AST), and many successful tools do this [19, 7, 6, 8]. This supports transformation by pattern-matching and tree rewriting very easily. However, our experience has been that more complex and interesting transformations are not easily expressed this way. Some support for this view will, we believe, be provided by examples presented later in this paper.

```
weave loopWithNoYield(CodeBlock Loop):
  method("* Thread.yield()", YieldMethod),
  loop(Loop),
  searchForCall(Loop, Loop, YieldMethod, TargetIfFound),
  null(TargetIfFound).

{
  System.out.print("Found a loop not broken by a yield() call in ");
  System.out.println(Loop.method);
}
```

Fig. 2. This weave finds loops which are not broken by a call to the `yield` method (as might be required in a non-pre-emptive threading context). The `loop` predicate matches all loops, whether arising from Java's `while`, `do..while` or `for` constructions. The `searchForCall` predicate finds the targets of all calls to the specified method between two points, and yields null if a path with no call exists.

Predicates capture control structure An AST makes it easy to match structured control-flow constructs such as loops. However, you need a rule for each loop type (or perhaps introduce a hierarchy of node subtypes). Using Prolog (or Datalog) predicates to characterise loops accounts for this rather neatly. For example, in Figure 2, the predicate `loop` matches all control-flow cycles.

CodeBlocks DeepWeaver predicates are used to identify program constructs, which can then be operated on in the Java action part of the weave. This is done using `CodeBlock`s. A `CodeBlock` is a contiguous, non-empty sequence of bytecode.

Since Java only has structured control-flow, control flow constructs can be represented faithfully as `CodeBlock`s.

To avoid the confusion caused by stack instructions, `CodeBlock`s are represented using the SOOT "Jimple" intermediate representation [17]. For many common uses of DeepWeaver, the programmer need not be concerned with details of the SOOT representation, and we discourage programmers from operating at that level.

In the Java part of a weave, the programmer has access to the `CodeBlock`s passed in from the Prolog query part. We have imposed no limit to the possible transformations that can be applied. However, there are common operators which are "safe", in that they cannot yield invalid bytecode control flow when applied to well-formed `CodeBlock`s[1]. These include `cb.remove()`, `cb.interjectBefore()` and `cb.interjectAfter()`. As we shall see in the next section, these allow code to be inserted at any specified point.

[1] In the current implementation, we do have some predicates that can create non-well-formed `CodeBlock`s (such as the LHS of an assignment).

```
aspect counters
{ // table of counter variable ids for each method
  static Hashtable counterLocals = new Hashtable();

  // first, a weave to add a counter variable to all methods
  weave addCounterVariable(SootMethod Method):
    method("* *(..)", Method).
  {
    DeepWeaverClass targetClass = DeepWeaverClass.classFromMethod(Method);
    Local myCounter = targetClass.insertLocalIntoMethod(Method,
                                                        IntType.v());
    counterLocals.put(Method, myCounter);
  }
  // now, a template for the code to insert
  interjection incrementCounter(int counter) {
    counter += 1;
  }
  // Insert the code at each access to an object of class C
  weave countAccesses(CodeBlock Access):
    assignment(Access, Rhs, Assignment), type(Access, "C").
  {
    InterjectionPrototype incCode
      = Interjection.getInterjectionByName("counters.incrementCounter")
                    .makePrototype();
    // Create one-element parameter list for interjection
    List params = new ArrayList();
    Local myCounter = (Local)counterLocals.get(Access.method);
    params.add(myCounter);

    // Insert the parameterised interjection
    Access.interjectBefore(incCode, params);
} }
```

Fig. 3. This aspect consists of two weaves, applied one after the other. The first adds a counter variable to each method. The second inserts code to increment the counter wherever an object of class C is updated. The interjection is a template whose bytecode is inserted at each insertion point. It is parameterised with the id of the counter variable.

```
aspect placeholders {
  interjection maybeCall() {
    if (Math.random() < 0.5) {
      Interjection.PLACEHOLDER_METHOD_FT(0);
    } else {
      System.out.println("Call omitted to reduce load");
  } }
  weave chickenOut(SootMethod Method, CodeBlock Target,
                   CodeBlock Location, List Params ):
    method("static void C.*()", Method),
    call(Method, Target, Location, Params).
  {
    InterjectionPrototype ip
      = Interjection.getInterjectionByName("placeholders.maybeCall")
                    .makePrototype();
    ip.replaceMethodCall(0, null, Method, Params);
    Location.replace(ip, Params);
} }
```

Fig. 4. This example shows the use of an interjection placeholder method. This weave rewrites calls to (static, void, parameterless) calls of class C, so they are omitted randomly. The method being called, `Method`, is substituted into the interjection.

Interjections Interjections are templates for code that is to be inserted. A simple example is shown in Figure 3. The body of the interjection is copied directly at the specified location. Thus, in this example, it is the context's copy of the counter variable that is incremented. We discuss type safety of interjection parameters in Section 4.3.

Structure of a DeepWeaver aspect A DeepWeaver aspect consists of a sequence of weaves, together with definitions of interjections, helper predicates, and helper methods in Java. Each weave consists of a Prolog "query" part, followed by a Java "action" part.

The action part of a weave is executed once for each successful match of the query part.

Weave composition Weaves are applied to the target program in the sequence they appear in the aspect. Each weave is applied across the whole codebase, before the next begins.

Actions change the codebase. To avoid chaos, updates to the codebase are not executed immediately, during execution of the action. Instead, the implementation defers changes until all query matching has been completed. Thus, queries always apply to a consistent, static view of the codebase — and `CodeBlock`s always refer to what they were bound to.

Note that interjections provide a mechanism for "code quoting". However, it is substitution *into* quoted code that is often tricky in metaprogramming sys-

```
predicate columnUsed(CodeBlock Target, CodeBlock Column):
 // Find program points where Target is used
 reaching_def(Target, Use, false),

 // Check that use is subject of a "get"
 call("* ResultSet.get*(String)", Use, Location, ColumnArgs),
 encloses(Location, Use),

 // Get parameter value, i.e. column name
 member(ColumnArg, ColumnArgs),
 local_constant_def(ColumnArg, Column).
}
```

Fig. 5. To rewrite the "select" query to specify the columns needed, we need to track down which columns will be accessed. We need to trace all possible control paths, and find where the ResultSet is used. Each use involves a "get" method, whose parameter is the name of the column being accessed. We need to collect all these names. It may be that the ResultSet escapes (by being returned, or passed as a parameter) — in which case we give up. `Reaching_def`'s third parameter reflects whether the definition escapes from the current method; set to "`false`" ensures the predicate fails if we are unable to track down all the uses.

```
weave refineSelectQuery(CodeBlock QueryString, List Columns):
 // Find JDBC execute() call site
 method("ResultSet Statement.executeQuery(String)", ExecMeth),
 call(ExecMeth, _, ExecCall, QueryStringLocal),

 // Find variable to which result is assigned
 assignment(Target, ExecCall, _),

 // Find the query string (from the method's constant pool)
 member(QSLocal, QueryStringLocal),
 local_constant_def(QSLocal, QueryString),

 // Collect set of Column strings used to access the ResultSet
 findall( Column, columnUsed(Target, Column), Columns ).
{
 // (Assume for brevity that it is a SELECT * and Columns is not empty)

 // Create new query String from list of Columns to select
 // (definition omitted for brevity)
 String newQuery = makeNewQuery(QueryString, Columns);

 // Replace existing string constant with new query string
 QueryString.replaceValue(StringConstant.v(newQuery));
}
```

Fig. 6. A weave to implement the "select *" optimisation. We use Prolog's "findall" to collect all the columns accessed, using the predicate in Figure 5.

tems. Figure 4 shows our current solution. The interjection calls a special static method, `Interjection.PLACEHOLDER_METHOD_FT(0)`. We replace this with the method we need to call using "`ip.replaceMethodCall(0, null, Method, Params)`". Each placeholder is numbered, from zero.

4 Evaluation

To evaluate the effectiveness of the DeepWeaver-1 design, we present two example optimisations that we have developed. In section 4.3 we review the results of these case studies.

4.1 The "Select *" optimisation

Many common tutorial introductions to using the Java Database Connectivity (JDBC) library begin with code like this:

```
ResultSet r = s.execute("select * from employee");
while ( r.next() ) {
  String col1 = r.getString("Name");  // Get column 1
}
```

This is inefficient; we found, for example, that with a 10-column table, it is about twice as fast to select just the one column that is being used:

```
ResultSet r = s.execute("select Name from employee");
while ( r.next() ) {
  String col1 = r.getString("Name");  // Get column 1
}
```

To implement this in DeepWeaver-1, we need to find the columns being accessed, as illustrated in Figure 5. The DeepWeaver predicate to do this collects the set of parameters of get methods applied to the result set returned by the call to the execute method. Figure 6 shows how this is used to rewrite the original query to specify the columns needed.

4.2 The RMI Aggregation optimisation

Java's Remote Method Invocation (RMI) mechanism is convenient but if used carelessly results in unnecessary communication. Figure 7 illustrates the potential value of aggregating RMI calls. We have built several implementations of this optimisation [21]. Correctness issues for the optimisation are quite subtle; a formal analysis is given in [2].

Consider an RMI call A, followed by some intervening code, then a second RMI call B. Our approach to RMI aggregation is to consider whether RMI call A can be relocated so that it can be combined with the later call, RMI B.

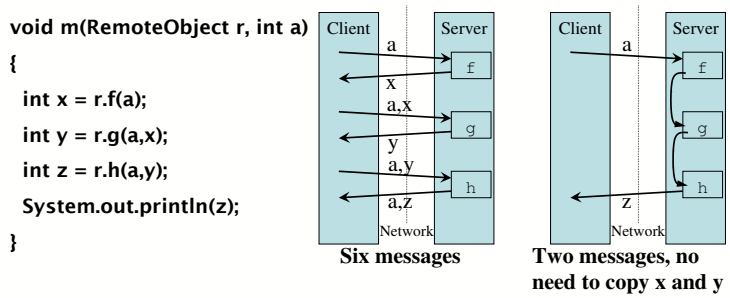

Fig. 7. A sequence of three method calls on a remote object "r" results in three call-return round-trip network transactions. Furthermore, parameter "a" is transferred several times, and also results "x" and "y" are returned unnecessarily. The aggregated implementation suffers fewer network latencies and transfers less data.

```
predicate isAggregatableRMIPair(CodeBlock CallA, CodeBlock CallB,
                          List ParamA, List ParamB, CodeBlock ResultOfA):
  // A and B are distinct RMI calls, and A precedes B:
  call(RemoteMethodA, RemoteObjectA, CallA, ParamA),
  type(RemoteObjectA, "java.rmi.Remote"),
  precedes(CallA, CallB), CallA \= CallB,
  call(RemoteMethodB, RemoteObjectB, CallB, ParamB),
  type(RemoteObjectB, "java.rmi.Remote"),
  dominates(A,B), post_dominates(B,A),

  // If A or B is in a loop they're both in the same one:
  forall ( loop(Loop),
    ( ( encloses(Loop, CallA), encloses(Loop, CallB)
      ); // OR
      ( \+ encloses(Loop, CallA), \+ encloses(Loop, CallB)
    ) ) ),
  // A's result is not used before B
  assignment(ResultOfA, CallA, _),
  \+ ( reaching_def(ResultOfA, UseOfResult, _),
    precedes(UseOfResult, CallB)
  ),
  // Assignments between A and B do not have externally-visible effects:
  \+ ( between(CallA, OnPath, CallB, false),
       assignment(Lhs, Rhs, OnPath),
       externally_visible(Lhs)
  ),
  // No method or constructor calls can occur between call A and call B:
  \+ ( between(CallA, Location, CallB, false),
       call(Method, Target, Location, Params),
       \+ side_effect_free_method(Method)  ).
```

Fig. 8. A predicate to test the validity of RMI aggregation.

RMI Aggregation validity conditions The conditions under which this is valid are encoded in Figure 8. In summary, this states:

- **A and B are distinct RMI calls, and A precedes B.**
- **All paths to B go through A**, that is, "`dominates(A,B)`")
- **All paths from A go through B** ("`post_dominates(B,A)`")
- **If A or B is in a loop they're both in the same one.**
  Note that the `forall` predicate succeeds when all instances of `Loop` satisfy the condition. In Prolog, "`;`" is logical "or"; recall that `\+` is "not").
- **A's result is not used before B**
- **Assignments on paths between A and B do not have externally-visible effects.** This is necessary for two reasons: firstly, another thread might observe changes to externally-visible objects out-of-order relative to the state of the remote server. Secondly, if call A throws an exception, the intervening code will already have been executed — so we must ensure it has no effects visible in or beyond the exception handler.
  The "false" parameter to `between` ensures we find all assignments `OnPath` that *might* be reached. The definition of "`externally_visible`" could be very sophisticated; a simple implementation would check that the LHS is a local variable with no escaping uses prior to call B.
- **No method or constructor calls can occur between call A and call B – apart from those known to be side-effect free**: This is required because DeepWeaver's analysis is currently not inter-procedural. In our prototype implementation we treat simple String operations as side-effect free in order to allow aggregation in the presence of simple code to prepare string parameters.

This example has been simplified for presentation purposes. In particular we have only considered the case where A returns a result but B does not.

RMI Aggregation implementation The conditions for validity of the RMI aggregation optimisation, listed above, can easily be assembled to form a DeepWeaver predicate for the query part of the weave:

```
isAggregatableRMIPair(CallA, CallB, ParamA, ParamB,
                      ResultA, ResultB)
```

The action part of the weave is sketched in Figure 9. The code required to do this is somewhat involved. First we delete call A. We need to identify any parameters common to the two calls, and to check for where the result from call A is used as a parameter to call B. We then need to construct the body and formal parameter list for the aggregate call, which is inserted into the server class (if the target object's type is actually an interface, we need to insert the code into all classes that implement it). Finally, we need to construct the actual parameter list for the call to this new method, which replaces call B.

```
 weave rmiResultForwarding(
                CodeBlock CallA,  CodeBlock CallB,
                List ParamA, List ParamB,
                CodeBlock ResultA,  CodeBlock ResultB )
  isAggregatableRMIPair(CallA, CallB, ParamA, ParamB, ResultA, ResultB).
{
        ...Remove call A.
        ...Create new server method, to implement the aggregated call.
        ...Insert it into each potential callee class.
        ...Replace call B with call to the new aggregated method.
}
```

Fig. 9. Outline of implementation of RMI aggregation. The implementation is too complicated to include here, mainly due to the necessary manipulation of parameter lists for the new aggregated method.

4.3 Evaluation: discussion

Performance: weave-time The time taken to apply a weave to a codebase depends, of course, on the weave itself. To evaluate this, we created an artificial benchmark generator creating simple candidates for the RMI aggregation optimisation. Applying the RMI aggregation weave to 100 such classes takes less than 20 seconds.

It is quite possible to write predicates which are extremely inefficient (for example, queries involving all control flow paths in a method). Although this has not yet proven a serious concern, the worst-case behaviour could be very poor.

Performance: execution time There is no performance overhead for code inserted using interjections (for example in Figure 3 where interjected code increments a counter). The bytecode is inlined directly. The performance imporvement that can be achieved by changing the code base obviously depend on the nature of the transformations specified by the Deepweaver actions. For example, we have found that the "Select *" optimisation can easily yield a factor of two reduction in query execution time (MS SQL Server, using one column from a 10 column table). The value of the RMI aggregation optimisation depends on the network performance, and the amount of redundant data movement that can be avoided. The performance improvements of the optimisation were evaluated in our earlier work, which used a run-time framework with higher overheads [21].

The query language and program model DeepWeaver's design centres on the use of Prolog predicates to identify `CodeBlock`s within a low-level three-address-code IR. Many interesting applications are expressed quite naturally this way. Some are difficult. For example, a transformation that interchanges a pair of nested loops: this is easy to express as rewriting in an abstract syntax tree. With DeepWeaver the loop (excluding its body) is not a contiguous block.

Queries are not always easy to write or to understand. We are developing idioms for common situations, and aim to support them with more built-in operators – for example to express regular expressions over paths [11].

The separation between query and action parts can sometimes be awkward, since some queries need to use Java (for example to query a software configuration description). Also, actions commonly need to issue followup queries.

A key design decision was whether to build our own Prolog engine. The alternative would be to export the code factbase into an external Prolog engine, or to implement the Prolog primitives as calls from Prolog to Java. We made this decision in order to retain control over query execution, and we plan to explore query optimisation. The disadvantage is that our Prolog implementation is partial, and there would be advantages in exploring the use of the full power of Prolog, perhaps including constraint Prolog.

As it stands our Prolog implementation is very restrictive; we lack terms, higher-order and non-declarative features.

The action language DeepWeaver resembles AspectJ (and is built as a pure extension of the AspectBench compiler for AspectJ [3]). In AspectJ, the Java "advice" is inserted at the selected joinpoint. In DeepWeaver, we don't have a joinpoint - we have a set of bindings for the query predicate's parameters. This is passed to the Java action part, which is executed *at weave time*.

This gives us considerable expressive power, which we need for complex examples like RMI aggregation. However it makes some tasks much more difficult and more prone to error. A key area for enhancement is to introduce a more refined type system that distinguishes different kinds of `CodeBlock` – essentially we need to expose SOOT's Jimple IR more explicitly.

The interjection mechanism DeepWeaver's interjections provide a means to create a bytecode template that can be parameterised and then inserted into the codebase. A key weakness is that we cannot check that actual parameter types match formal parameter types. Note also that interjections are naturally used in a polymorphic way – this is what we need, for example, in the RMI aggregation example, where the type of the aggregated method depends on the type of the original methods. We also need to take care with exceptions: if interjected code might throw an exception, the host code must be able to handle it.

Another weakness is that only interjections can be used as prototypes - we commonly wish to move or duplicate `CodeBlock`s. To handle this we need to account for a `CodeBlock`'s free variables and ensure inserted code's variables are bound properly.

5 Related Work

Query languages for code DataLog, a similarly restricted, declarative form of Prolog, has been adopted by several projects. JQuery [20] and CodeQuest [9]

use it to query the Eclipse [16] IDE's abstract syntax tree. CrocoPat [5] uses it for design pattern recovery and metrics. Bddbddb [13] is closer to our work in operating at the bytecode level, and supports intra- and inter-procedural dataflow analyses. CrocoPat and bddbddb include substantial query optimisation.

Transformation and rewriting AspectJ [12] provides a join point model for matching points in Java code, at which new code will be inserted before and/or after. A code matching query is restricted, for comprehensibility reasons, to various predefined code points (e.g. method call) but allows flexibility in identifier and type matching. Aspicere [1] tries to work around the restrictions in the joinpoint model when applied to C using a Prolog-like query language.

Many IDEs support refactoring program transformations initiated by the user. JunGL [18] is an interesting refactoring scripting language, quite comparable to DeepWeaver-1, that uses Datalog queries embedded within an ML-based language.

TXL [6] and Stratego [19] are very general and powerful tools for source-code transformation. They both provide a mechanism to specifying grammars, pattern matching and rewriting. Stratego supports strategies to control rewrite order, and dynamic rules for context sensitive rewrites.

Establishing the soundness of optimizing program transformations is an issue that we have not dealt with formally in this paper. Soundness has been addressed in other contexts, however, for example Rhodium [14], which is a domain-specific language for specifying program analysis and transformation in the context of a C-like intermediate language.

6 Conclusions and Further Work

DeepWeaver-1 is a prototype tool which shows considerable promise:

- It provides a delivery vehicle for domain-specific performance optimisation components. DeepWeaver aspects package together a declarative statement of the conditions for validity of an optimisation (the "query part"), with the implementation of the transformation (the "action part").
- DeepWeaver-1's low-level IR, and the role of `CodeBlock`s as the focus for queries, works remarkably well when combined with the power of a Prolog-like query language. Prolog predicates allow structured control flow to be captured where it is needed, whilst also supporting control-flow and data-flow–based reasoning. These are often useful for characterising the validity of optimisations.
- We have demonstrated the power of the approach using a small number of application examples.

There remain many challenges in developing these ideas. We plan two major directions for further work:

1. Enhancing the query language. We need to clarify the rendering of the IR in the query language, and refine the type system. We also need to support user-definable data-flow analyses in an elegant and expressive way (bddbddb [13] is promising in this regard). Similarly, and likely by this means, we need to handle inter-procedural analysis. Key to these is query optimisation. We may also extend the query language conceptually to handle `CodeBlock`s with holes, free variables, and to represent "slices" of dependent instructions.
2. Enhancing the static safety and expressiveness of the action language. Our goal is to attain a statically-verifiable guarantee that a weave cannot generate a "broken" codebase: basic safety properties such as stack balance, name clashes, initialised and bound variables (see, for example, SafeGen [10]). Central to this is to support type-checked parameterisation of interjections, and to check that the free variables of cloned `CodeBlock`s are bound to variables of the right type when they are interjected.

Acknowledgments This work was supported by the EPSRC SPOCS grant (EP/E002412), EPSRC PhD studentships, and by an IBM Faculty Award.

References

1. Bram Adams and Tom Tourwé. Aspect orientation for C: Express yourself. In *AOSD SPLAT Workshop*, 2005.
2. Alexander Ahern and Nobuko Yoshida. Formalising Java RMI with explicit code mobility. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 403–422, New York, NY, USA, 2005. ACM Press.
3. Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Bruno Dufour, Christopher Goard, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, Julian Tibble, and Clark Verbrugge. ABC the AspectBench compiler for AspectJ: A workbench for aspect-oriented programming language and compilers research. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 88–89, New York, NY, USA, 2005. ACM Press.
4. Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 345–364, New York, NY, USA, 2005. ACM Press.
5. Dirk Beyer and Claus Lewerentz. CrocoPat: Efficient pattern analysis in object-oriented programs. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC 2003, Portland, OR, May 10-11)*, pages 294–295. IEEE Computer Society Press, Los Alamitos (CA), 2003.
6. James R. Cordy, Charles D. Halpern-Hamu, and Eric Promislow. TXL: a rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

7. Kei Davis and Daniel J. Quinlan. Rose: An optimizing transformation system for C++ array-class libraries. In *ECOOP '98: Workshop on Object-Oriented Technology*, pages 452–453, London, UK, 1998. Springer-Verlag.
8. Dawson R. Engler. Incorporating application semantics and control into compilation. In *USENIX Conference on Domain-Specific Languages (DSL'97)*, pages 103–118. USENIX, 1997.
9. Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with datalog. In Dave Thomas, editor, *ECOOP'06: Proceedings of the 20th European Conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, Berlin, Germany, 2006. Springer.
10. Shan Shan Huang, David Zook, and Yannis Smaragdakis. Statically safe program generation with SafeGen. In *GPCE*, pages 309–326, 2005.
11. David Lacey, Neil Jones, Eric Van Wyk, and Carl Christian Frederikson. Proving correctness of compiler optimizations by temporal logic. *Higher-Order and Symbolic Computation*, 17(2), 2004.
12. Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.
13. Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, New York, NY, USA, 2005. ACM Press.
14. Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL 2005: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 364–377, New York, NY, USA, 2005. ACM Press.
15. Hidehiko Masuhara and Kazunori Kawauchi. Dataflow pointcut in aspect-oriented programming. In *The First Asian Symposium on Programming Languages and Systems (APLAS'03)*, volume 2895 of *Lecture Notes in Computer Science*, pages 105–121. Spinger-Verlag, November 2003.
16. The Eclipse Foundation Inc. The Eclipse extensible development platform.
17. Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. SOOT - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.
18. Mathieu Verbaere, Ran Ettinger, and Oege de Moor. JunGL: a scripting language for refactoring. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 172–181, New York, NY, USA, 2006. ACM Press.
19. Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer-Verlag, June 2004.
20. Kris De Volder. Jquery: A generic code browser with a declarative configuration language. In Pascal Van Hentenryck, editor, *PADL '06: Eighth International Symposium on Practical Aspects of Declarative Languages*, volume 3819 of *Lecture Notes in Computer Science*, pages 88–102. Springer, 2006.
21. Kwok Cheung Yeung and Paul H. J. Kelly. Optimising Java RMI programs by communication restructuring. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference 2003, Rio De Janeiro, Brazil, 16–20 June 2003*, LNCS, June 2003.