Just when you thought your little language was safe: "Expression Templates" in Java

Todd L. Veldhuizen
Extreme Computing Laboratory
Indiana University Computer Science Department
Bloomington Indiana 47405, USA
tveldhui@acm.org

Abstract

Template techniques in C++ allow a modest degree of *generative programming*: creating specialized code for specialized problems. This use of templates has been controversial; indeed, one of the oft-cited reasons for migrating to Java is that it provides a simpler language, free of the complexities of templates.

The essence of generative programming in C++ is not templates – the language feature – but rather the underlying algorithms in the compiler (template instantiation) which unintentionally resemble an optimization called *partial evaluation* [14, 20]. By devising a partial evaluator for Java, we reproduce some of the generative programming aspects of C++ templates, without extending the Java language. The prototype compiler, called *Lunar*, is capable of doing "expression templates" in Java to optimize numerical array objects.

1 Introduction

Good numeric performance for Java is mostly limited to code written in what might be called *JavaTran* (Java/FORTRAN) style: Fortran 77-like code surrounded by class declarations. A typical example is:

```
public class DoArrayStuff {
  public static apply(float[] w, float[] x,
    float[] y, float[] z)
  {
    for (int i=0; i < w.length; ++i)
      w[i] = x[i] + y[i] * z[i];
  }
}
```

It would be nice if we could use the object-oriented features of Java in performance-critical code. Then we could write high-level code using objects representing arrays, matrices, tensors, and the like. Suppose we had a package which provided efficient numerical array objects; with operator overloading as proposed by the Java Grande committee [7], we could have Fortran-90 style array notation. Using such a package, the `DoArrayStuff` code above could be written as:

```
  public static apply(Array w, Array x, Array y,
    Array z)
  {
    w = x + y * z;
  }
```

Without operator overloading, one would write `w.assign(x.plus(y.times(z)))`.

Can this Array class be implemented efficiently? This question is familiar to designers of C++ numerical libraries. There are four well-known implementation choices for array libraries:

1. **Compiler extension**: Write a compiler or preprocessor which recognizes a particular array class, and optimizes it using special semantics. This is the intent of the `valarray` class in the C++ standard, and of the ROSE preprocessor for C++ [1]. In Java, this approach is used by the Ninja compiler [13], which recognizes a special set of array classes and performs optimizations for them using "semantic inlining".

2. **Pairwise evaluation**: Each subexpression such as `y*z` (or `y.times(z)`) returns an intermediate Array result. The extra loops, memory allocation, and memory movement make this inefficient. Automatically fusing these loops is more difficult than the equivalent Fortran or C loop fusion problem, because the array pointers and loop bounds are fields of array objects. So far, no commercial compiler has been able to eliminate the temporary arrays.

3. **Deferred evaluation**: In this approach, operators such as `y*z` (or `y.times(z)`) construct a parse tree of an expression as a data structure. When a parse tree is assigned to an array, a match can be sought in a library of commonly used array expressions. This requires considerable run-time overhead, and if the expression is not found, one must revert to temporary arrays.

4. **Expression templates** [18]: This C++ technique is similar to deferred evaluation, in that a parse tree of the expression is created. The parsing is done at compile time using C++ templates, by encoding the parse tree as a template type.[1] When a parse tree is assigned to an array, a function like this is called:

```
template<class T>
void assign(Array w, Expr<T> expr)
{
  for (int i=0; i < n; ++i)
    w.data[i] = expr.eval(i);
}
```

[1] For example, `x+y*z` might be parsed as the template type `Expr<Array,Plus,Expr<Array,Times,Array> >`

```
float pow(float x, int n)
{
  if (n == 0)
    return 1.0;
  else
    return x * pow(x,n-1);
}

float a, b;
a = pow(b,3);
```

(a) Some code

```
// pow has been specialized for n=3
float pow__3(float x)
{
  return x * x * x;
}

float a, b;
a = pow__3(b);
```

(b) After partial evaluation

Figure 1: Partial evaluation example

For each array position i, `expr.eval(i)` traverses the parse tree, evaluating the expression. C++ semantics require that this traversal be done at compile time, so the resulting loop is efficient. This technique is the basis of C++ libraries such as Blitz++ [19] and POOMA [9].

Of these approaches, expression template-based array libraries have been popular for C++, since they deliver efficiency without compiler extensions.

It turns out that templates in C++ are strikingly like *partial evaluation*. A partial evaluator takes a program, performs the operations which depend only on known values, and outputs a specialized program [8]. The standard example is shown in Figure 1.

C++ requires that all template parameters be known at compile time. When a template parameter is given by an expression (for example, `Vector<3+8>`), that expression *must* be evaluated at compile time. C++ templates effectively require that a partial evaluator be built into the compiler to evaluate template expressions [14, 20]. It is this partial evaluation which makes possible expression templates and other template-based optimizations.

So here is a thought. Perhaps if we implemented a partial evaluator for Java, we could get the same performance benefits one gets from templates in C++, without templates.

This turns out to be at least partly true. We demonstrate a partial evaluator for Java that allows almost arbitrary computations at compile time, and can be used to implement "expression templates" and similar performance-enhancing techniques from C++. We do this without any language extensions; we do not use GJ or other genericity proposals for Java.

1.1 Structure of this paper

We start by considering an analogue of the C++ "expression templates" technique for Java (Section 2). This gives users array notation, at the cost of very poor performance. When this code is partially evaluated, we obtain performance similar to code written in *JavaTran* style (Section 3). A detailed discussion of how these optimizations occur is deferred to the appendix (Section A).

In Section 4, we overview the Lunar compiler and summarize the optimizations performed. The operation of the partial evaluator is illustrated by some examples using a `Complex` class (Section 5).

We point out some shortcomings of our compiler and some related work in Section 6.

2 A Java version of "Expression templates"

Our goal is to write a class `Array` which provides a 1-D array of floats. Users of this class will write code such as:

```
int n = 1000;
Array w = new Array(n);
Array x = new Array(n);
Array y = new Array(n);
Array z = new Array(n);

w = x + y * z;     // an array expression
```

Since we do not yet have overloaded operators in Java, we will write `w=x+y*z` like this:[2]

```
w.assign(x.plus(y.times(z)));
```

Array expressions will be evaluated for every element in the array; the equivalent *JavaTran* version of `w=x+y*z` is:

```
float[] w, x, y, z;         // ...
for (int i=0; i < n; ++i)
  w[i] = x[i] + y[i] * z[i];
```

We are going to create a highly inefficient mechanism for evaluating array expressions which resembles the "expression templates" technique of C++. Then we will see that a suitable optimizing compiler can automatically turn this into the equivalent *JavaTran* implementation. More generally, such a optimizer has the potential to duplicate many of the generative programming features of C++ templates.

For our pseudo-"expression templates" implementation, we will need objects representing array expressions. We will use the class hierarchy in Figure 2: `Expr` is a common base class of array expressions, and `Array` is itself an expression. To represent an expression such as `y*z` (or `y.times(z)`), we use an instance of a class `BinaryOpExpr` (short for Binary Operator Expression). `BinaryOpExpr` contains two expressions and a pointer to a binary operator object; to create an object representing `y*z`, we could write

```
Expr expr = new BinaryOpExpr(y, z, new Times());
```

where `Times` is an object representing multiplication. The definitions of these classes are shown in Figures 3,4,5.

The expression base class `Expr` defines a single abstract method `float eval(int i)` which evaluates an array expression at a single array index `i`. Hence we can assign an expression to array using this method of class Array:

[2]There is a conflict between the Java semantics of `=` and the semantics desirable for array libraries. If `w` were a subarray, we would want `w=x+y*z` to change the existing array of which `w` is part; making `w` point to a new array created by `x+y*z` would be ineffectual. This problem might be solved by overloading (for example) `w←x+y*z`.

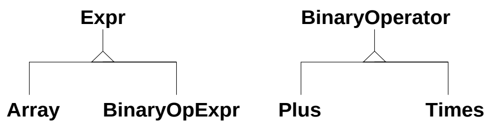

Figure 2: Inheritance diagram for the `Array` classes

```
public class Array extends Expr {

    float data[];
    int length;

    public Array(int n)
    {
        data = new float[n];
        length = n;
    }

    public float eval(int i)
    {
        return data[i];
    }

    public void set(int i, float value)
    {
        data[i] = value;
    }

    public void assign(Expr e)
    {
        int t = length;
        for (int i=0; i < t; i=i+1)
            data[i] = e.eval(i);
    }
}
```

Figure 3: An `Array` class

```
public class Array extends Expr {
 ...
 public void assign(Expr e)
 {
    for (int i=0; i < length; ++i)
      data[i] = e.eval(i);
 }
}
```

Figure 6 shows a test program which exercises the Array class. It initializes some arrays, then evaluates the expression `w=x+y*z`. The method which assigns the expression to `w` is `Array.assign(Expr)`:

```
public void assign(Expr e)
{
    for (int i=0; i < length; i=i+1)
        data[i] = e.eval(i);
}
```

This method loops through the array, evaluating the array expression for each `i`, and storing the result in `w.data[i]`. To evaluate each `e.eval(i)`, six virtual function calls, three bound checks, and numerous pointer indirections are required. Not surprisingly, performance is quite poor with typical Java compilers. This loop has been benchmarked at 0.7 Mflops using the Kaffe JIT compiler on a 300 MHz sparcv9; that is roughly 1 flop every 428 clock cycles.

With partial evaluation, all of this inefficiency can be removed. We have implemented a prototype compiler called

```
public abstract class Expr {
    public abstract float eval(int i);

    public Expr plus(Expr b)
    {
        BinaryOperator plus = new Plus();
        return new BinaryOpExpr(this,b,plus);
    }

    public Expr times(Expr b)
    {
        BinaryOperator times = new Times();
        return new BinaryOpExpr(this,b,times);
    }
}

public class BinaryOpExpr extends Expr {
    Expr a, b;
    BinaryOperator op;

    public BinaryOpExpr(Expr _a, Expr _b,
        BinaryOperator _op)
    {
        a = _a;
        b = _b;
        op = _op;
    }

    public float eval(int i)
    {
        return op.apply(a.eval(i),b.eval(i));
    }
}
```

Figure 4: The parse tree classes `Expr` and `BinaryOpExpr`

```
public abstract class BinaryOperator {
    public abstract float apply(float a, float b);
}

public class Plus extends BinaryOperator {
    public float apply(float a, float b)
    {
        return a+b;
    }
}

public class Times extends BinaryOperator {
    public float apply(float a, float b)
    {
        return a*b;
    }
}
```

Figure 5: The `BinaryOperator` base class and two subclasses

```
public class Test {
    public static void main(java.lang.String[] args)
    {
        // Create some arrays
        int n = 12345;
        Array w = new Array(n);
        Array x = new Array(n);
        Array y = new Array(n);
        Array z = new Array(n);

        // Initialize with data
        for (int i=0; i < n; i=i+1)
        {
            x.set(i,i*0.33f);
            y.set(i,10.0f+i);
            z.set(i,100.0f*i);
        }

        // The java front end does not yet handle
        // implicit casts.  These will be gone in
        // the final paper.
        Expr zexpr = z;
        Expr yexpr = y;

        // With operator overloading, this would be
        // w = x + y * z
        w.assign(x.plus(yexpr.times(zexpr)));
    }
}
```

Figure 6: Test code for the `Array` class

Lunar that compiles Java to an intermediate form, partially evaluates it, and emits C code. This is then compiled to machine code using a C compiler. When applied to the example program (Figure 6), Lunar performs these optimizations:

- Constant and copy propagation through the heap, resolving (when possible) virtual method calls and pointer indirection at compile time.
- Inlining virtual methods.
- Elimination of bound checks when they are provably unnecessary.
- Elimination of unnecessary temporary objects.

The C code generated by Lunar for the Java expression `w.assign(x.plus(yexpr.times(zexpr)))` is shown in Figure 7. (The complete output of the compiler is shown in Appendix B). Lunar is able to eliminate all the virtual functions, remove all bound checks, and even get rid of the temporary `BinaryOpExpr`, `Plus`, and `Times` objects.

We include a detailed discussion of how Lunar transforms the code `w.assign(x.plus(y.times(z)))` into the C code of Figure 7 in the Appendix.

3 Benchmark results

We present preliminary benchmark results on a sparcv9 processor at 300 MHz, using `gcc -O3` as the back end compiler for Lunar. Figure 8 compares the performance of the Lunar compiler to hand-coded C for the array expression $w = x + y * z$. The JavaTran series shows the performance of Lunar on the loop

```
int i = 0;
for (; (i < 12345);)
{
    int __a75 = (i * 4);
    int __a71 = (i * 4);
    float __a31 = *((float *) (array1d__96 + __a71));
    int __a184 = (i * 4);
    float __a171 = *((float *) (array1d__107 + __a184));
    int __a163 = (i * 4);
    float __a176 = *((float *) (array1d__118 + __a163));
    float __a35 = (__a171 * __a176);
    float __a76 = (__a31 + __a35);
    *((float *) (array1d + __a75)) = __a76;
    i = (i + 1);
}
```

Figure 7: C code generated by Lunar for the expression `w=x+y+z` of Figure 6

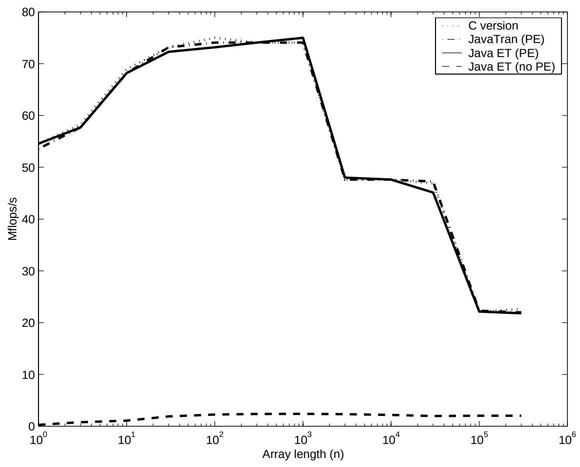

Figure 8: Benchmark results for the array expression `w=x+y*z`, comparing (from top to bottom) hand-coded C, *JavaTran*-style code compiled with Lunar, Java "Expression-templates" version compiled with Lunar (with Partial Evaluation), Java "Expression-templates" compiled with Lunar (no Partial Evaluation)

Phase	Elapsed time (ms)
Parsing	916
Conversion to Lunar IL_2	303
Prepasses	91
Partial evaluation	1569
Unparsing to C	678
Total	3557

Table 2: Time to compile the "expression templates in Java" example of Figure 6, excluding C compilation time.

```
float[] w, x, y, z;   // ...
for (int i=0; i < n; i=i+1)
  w[i] = x[i] + y[i]*z[i];
```

The Java ET series show the performance of `w.assign(x.plus(y.times(z)))` using `Array` objects. When the partial evaluator is disabled (the "no PE" series), performance takes a drastic hit – indicating that Lunar, not `gcc`, is doing the important optimizations.[3]

Table 1 compares performance of Lunar to two JIT compilers (Sun Hotspot and Transvirtual Kaffe).[4] The partial evaluator gives a modest improvement over the *JavaTran*-style code, but the biggest improvement is for the Java version of "expression templates".

Table 2 shows a summary of the time taken by Lunar to compile the "expression templates in Java" example with partial evaluation. The Lunar compiler is written in Java, and was run using Sun Hotspot 1.3beta. Once the compiler is able to compile itself, these times should decrease.

4 Compiler overview

Figure 4 shows an overview of the Lunar compiler. The front end takes Java code and translates it into an intermediate language, called Lunar Intermediate Language 2 (IL_2). Prior to partial evaluation, a prepass puts the code into a form which makes partial evaluation simpler. This code is then passed to the partial evaluator. The output from the partial evaluator is then translated into C and compiled using a C compiler.

Table 3 summarizes the analyses and transformations currently implemented in Lunar. Many of these are similar to those implemented in conventional compilers. However, there are substantial differences between partial evaluation and typical optimizers. Partial evaluators tend to have a flavour of symbolically executing a program, whereas typical imperative compilers are more flow-graph based.

4.1 Lunar Intermediate Language

The Lunar intermediate language (IL) is low level, and bears little resemblance to Java. It knows nothing about object systems, method tables, inheritance, and the like. Figure 10

[3]The performance numbers for Lunar are slower than C for the in-cache region because (a) type and aliasing information is lost en route from the source language (Java) to the back end; and (b) loop invariant hoisting is not yet implemented. Once these problems are corrected, "expression templates" in Java compiled with Lunar should be as fast as low-level C code.

[4]A clear deficiency of these benchmark results is the lack of comparison to a good native Java compiler. The JIT compiler results were a disappointment. There is some possibility that the Hotspot compiler was not actually doing JIT due to some misinstallation problem. This issue will be resolved for the final paper.

```
public class Complex {
  float re, im;

  public Complex(float _re, float _im)
  {
    re = _re;
    im = _im;
  }

  public float magnitudeSquared()
  {
    return re*re + im*im;
  }

  public Complex times(Complex y)
  {
    return new Complex(re*y.re - im*y.im,
        re*y.im + im*y.re);
  }

  public Complex plus(Complex y)
  {
    return new Complex(re + y.re, im + y.im);
  }
}
```

Figure 11: A `Complex` class

shows the subset of IL_2 used in the code excerpts of this paper. A Lunar program is a set of top-level definitions (d), which contain expressions (e). Table 4 shows the primitive operations used in the code excerpts of this paper.

5 Partial Evaluation examples

To illustrate the intermediate language and partial evaluation, we look at two examples using a Complex number class (Figure 11). These examples also illustrate the ability of Lunar to optimize away small objects.

The Java front end converts the `Complex` class of Figure 11 into the Lunar IL2 program shown in Figure 12.[5]

Figure 13 shows the translation of a simple complex number example into Lunar. The method `example(float a)` returns the value $|a + 3i|^2$. The IL_2 code illustrates how object creation and method calls are translated. The partial evaluator is able to propagate the parameter `a` through the heap, inline the object constructor and virtual method call to `magnitudeSquared()`, and calculate part of the magnitude ($3.0 * 3.0 = 9.0$) at compile time. It also eliminates the temporary `Complex` object `q`, which is no longer needed.

Figure 14 shows an example which creates three Complex numbers on the heap. It calculates the squared-magnitude of all three numbers and outputs them to standard output. The partial evaluator resolves the virtual method calls to `times(...)` and `magnitudeSquared()`, inlines and completely evaluates them, then eliminates the temporary `Complex` objects. The C code contains just three print statements.

[5]The front end gives the program to the partial evaluator directly as trees in memory, not via a temporary file. This allows errors discovered during partial evaluation to be reported at their appropriate locations in the Java code.

JVM or compiler	"JavaTran"	"Expression Templates"
Lunar (no PE); gcc -O2 backend	33.1	2.4
Lunar (with PE); gcc -O2 backend	74.6	74.6
Sun Hotspot 1.3beta (JIT)	1.4	0.4
Transvirtual Kaffe (JIT)	4.3	0.7

Table 1: Mflops/s for the array expression $w = x + y * z$, $n = 1000$, single precision. PE=Partial Evaluation

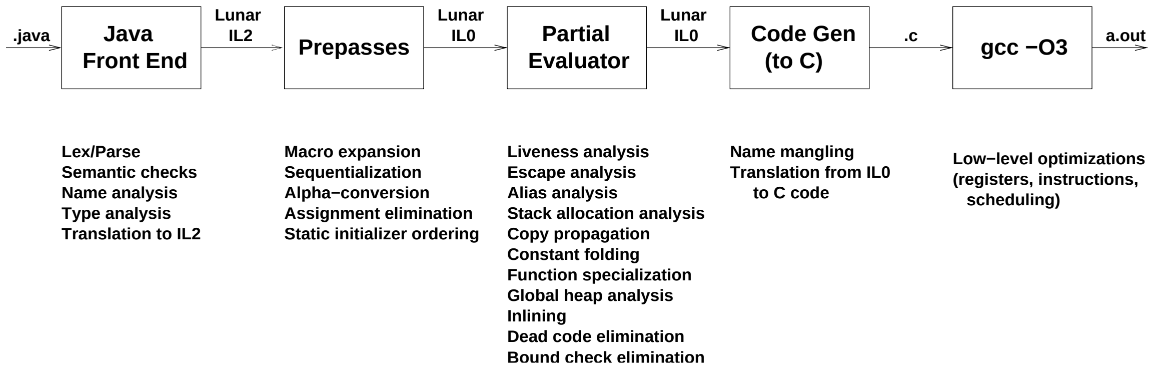

Figure 9: Overview of the Lunar compiler

Prepass transforms	**Purpose**
Macro expansion	Expand macros used to simplify the Java front end
Sequentialization	Name intermediate values, simplify language structure
Assignment elimination	Remove variable assignments; ensure that variable names map uniquely to values within a scope.
α-conversion	Rename variables to ensure unique names
Static initializer ordering	Find initialization order for global variables; delete unused functions and global variables.
Partial evaluator	**Purpose**
Liveness analysis	Decide how function parameters are used.
Escape analysis	Decide if values might escape into the heap.
Alias analysis	Decide if heap objects are alias-free.
Stack allocation analysis	Allocate objects on the stack instead of the heap when possible.
Copy propagation	Find variables that refer to the same value.
Constant folding	Fold constants, eliminate primitive operations
Function specialization	Specialize functions according to known argument values
Global heap analysis	Constant and copy propagation through the heap.
Inlining	Selectively inline functions to improve optimization.
Dead Code Elimination	Eliminate dead variables, code, functions, and global variables.
Bounds check elimination	Remove bounds checks that are provably unnecessary.

Table 3: Analyses and transformations in the Lunar compiler

Primitive	C equivalent	Description
storew(p,i,x)	((word*)p)[i] = x	store word, aligned
storewfinal(p,i,x)	((word*)p)[i] = x	store word, "write-once"
readw(p,i)	((word*)p)[i]	read word, aligned
alloc(i)	malloc(i)	memory allocation
f+(f1,f2)	(f1 + f2)	float addition
f-(f1,f2)	(f1 - f2)	float subtraction
f*(f1,f2)	(f1 * f2)	float multiplication

Table 4: Selected primitive operations in IL_0

d	::=	`function` $f(v_1, \ldots, v_k)$ e	function definition
	|	`global` v `=` e	global variable definition
e	::=	c	literal (decimal, character, string, or float)
	|	v	variable use
	|	`let [` $v_1, \ldots, v_n$ `] =` e_1 `in` e_2	introduce variables
	|	p `(` $e_1, \ldots, e_n$ `)`	primitive
	|	e_0 `(` $e_1, \ldots, e_n$ `)`	function call
	|	`if` e_1 `then` e_2 `else` e_3	functional-style if
	|	`begin` $e_1 \ldots e_n$ `end`	sequence of expressions
	|	`[` $e_1, \ldots, e_n$ `]`	multiple-value construction
	|	`[]`	void expression
	|	`assign` v `:=` e	variable assignment
	|	`loop` e	loop expression
	|	`break` e	break out of loop
	|	`return` e	return value from function
	|	`try` e_1 `catch [` $v_1, \ldots, v_k$ `]` e_2	try/catch statement
	|	`throw` e	throw statement

Figure 10: Definitions (d) and expressions (e) in IL_2

```
// Field layout
global Complex.re = 1
global Complex.im = 2

// Virtual function table
global Complex$vtable =
  let vtable = alloc(16) in
   begin
    storewfinal(vtable,1,Complex.magnitudeSquared$)
    storewfinal(vtable,2,Complex.times$Complex)
    storewfinal(vtable,3,Complex.plus$Complex)
    vtable
   end

// Complex.Complex(float,float)
function Complex.Complex$float;float(this,_re,_im)
  begin
    java.lang.Object.Object$(this)
    storew(this,Complex.re,_re)
    storew(this,Complex.im,_im)
  end

// Complex.magnitudeSquared()
function Complex.magnitudeSquared$(this)
  f+(f*(readw(this,Complex.re),readw(this,Complex.re)),
     f*(readw(this,Complex.im),readw(this,Complex.im)))

// Complex.times(Complex)
function Complex.times$Complex(this,y)
  let $new___a1 = alloc(12) in
   begin
    storewfinal($new___a1,0,Complex$vtable)
    Complex.Complex$float;float($new___a1,
      f-(f*(readw(this,Complex.re),readw(y,Complex.re)),
         f*(readw(this,Complex.im),readw(y,Complex.im))),
      f+(f*(readw(this,Complex.re),readw(y,Complex.im)),
         f*(readw(this,Complex.im),readw(y,Complex.re))))
    $new___a1
   end
```

Figure 12: Translation of `Complex` into Lunar. Names such as `Complex.times$Complex` are a single identifier.

```
public class ComplexTest {
  public static float example(float a)
  {
    Complex q = new Complex(a,3.0);
    float t = q.magnitudeSquared();
    return t;
  }
}
```

(a) Java code

```
function ComplexTest.example$float(a)
  let q =
    let $new___a10 = alloc(12) in
     begin
      storewfinal($new___a10,0,Complex$vtable)
      Complex.Complex$float;float($new___a10,a,3.0f)
      $new___a10
     end
  in
   let t =
    let $callee__a11 = q in
      (readw(readw($callee__a11,0),1))($callee__a11)
   in t
```

(b) Translation into Lunar IL_2

```
function ComplexTest.example$float__2(a__72)
  let a__75 = f*(a__72,a__72) in
   f+(a__75,9.0f)
```

(c) After partial evaluation

Figure 13: Partial evaluation example showing constant folding and copy propagation through the heap.

```
public class ComplexTest2 {
  public static void main(java.lang.String args[])
  {
    Complex x = new Complex(0.866, 0.5);
    Complex y = new Complex(0.70711, 0.70711);
    Complex z = x.times(y);

    java.lang.System.out.println(x.magnitudeSquared());
    java.lang.System.out.println(y.magnitudeSquared());
    java.lang.System.out.println(z.magnitudeSquared());
  }
}
```

(a) Java code which creates 3 temporary Complex objects

```
void sti__dollunar_umain_ureturn_uvalue()
{
    print_ufloat(0.999956f);
    print_ufloat(1.0000091f);
    print_ufloat(0.9999651f);

    // Set global variable indicating program exit status
    __lunar_main_return_value = 0;
}
```

(b) C output, after partial evaluation using the definitions of Figure 12

Figure 14: Partial evaluation example showing aggressive constant folding and elimination of temporary objects

6 Summary

6.1 Caveats

Lunar's Java front end handles a modest subset of Java 1.1. It does not yet support (among other things) interfaces, synchronization, or inner classes. It is possible that use of some Java language features (for example, synchronization) will make it harder to perform the optimizations described in this paper.

Lunar runs in loose numerics mode. It uses the native floating point hardware, without concern for whether this correctly implements Java numeric semantics. This is not a requirement, but a shortcut. Strict numerics could be implemented, although they would likely have performance implications on some platforms.

Lunar does not yet perform null pointer checking. The plan is to follow the example of `gcj`, and trap SIGSEGV signals. This does not require changing any of the translation or code generation; trapping is done by the hardware and handled in a runtime library. Hence, the benchmark results reported here are not compromised by our omission of null pointer checking.

Our Java runtime does not do any garbage collection. In most of the benchmark results presented, the partial evaluator eliminates all memory allocation inside loops. Hence the absence of a garbage collector does not affect our performance in a major way. The exception are the benchmarks of Table 1 which show Lunar results without PE. For these results, not doing garbage collection gives Lunar an unfair advantage over the JIT compilers.

Some of the algorithms in Lunar are $O(n \lg n)$, and a few are quadratic in certain (possibly unlikely) scenarios. Lunar has not yet been tested on large Java programs, so it is possible we will uncover scaling problems. The plan is to provide a set of optimization switches `-O1,-O2,..,-O5` which progressively enable more expensive and expansive forms of partial evaluation.

The algorithms in Lunar do not require closed-program analysis. However, the more of the program Lunar can see, the better it can optimize.

Unlike C++ templates, Lunar offers no guarantee of compile-time evaluation. There is no analogy to type template parameters in C++, although Lunar can specialize functions based on types related by an inheritance hierarchy.

6.2 Related work

There is a wealth of literature about compilers and partial evaluation, both separately and their intersection. The idea of resolving virtual functions through specialization was detailed by Dean et al [4]. Lunar achieves a similar effect "for free" by relying on the heap analyzer to propagate function names through dispatch tables. Khoo [10] used partial evaluation to compile inheritance, doing what would (analogously) in Java be dispatch table (or vtable) layout, but he did not use partial evaluation to resolve and inline virtual functions.

Many of the C++-template-like optimizations performed by Lunar are driven by a heap analyzer, which does constant and copy propagation through the heap. This builds on a tradition of partly-static data structures in the partial evaluation community (e.g. [3]), and of store analyzers in the imperative world (e.g. [15, 16]).

Volanschi et al [21] describe a partial evaluator for Java which relies on user annotations ("specialization classes") to guide specialization of Java programs. This provides finer control of specialization than Lunar, at the cost of requiring language extensions.

The benefits of partial evaluation for scientific programs are well known; see for example [2, 6, 11]. Lunar is not designed to optimize scientific programs per se, but rather to provide reliable partial evaluation semantics which can be used as a driving mechanism for performing domain-specific optimizations (for example, "expression templates" for array classes). In this regard it has similar goals as two- or multi-stage languages (e.g. [17]) which seek to provide a natural framework for writing "program generators".

6.3 Acknowledgments

Lunar uses JavaCC for lexing and parsing, and the GCC compiler for its back end. The term *JavaTran* derives from the similar term *C++Tran* coined by Scott Haney. I am grateful to Kent Dybvig, Ken Chiuk, Jeremy Siek, and Steve Karmesin for comments and discussion.

References

[1] Bassetti, F., Davis, K., and Quinlan, D. Optimizing transformations of stencil operations for parallel object-oriented scientific frameworks on cache-based architectures. *Lecture Notes in Computer Science 1505* (1998), 107–??

[2] Berlin, A., and Weise, D. Compiling scientific code using partial evaluation. *Computer 23*, 12 (Dec 1990), 25–37.

[3] Consel, C. Binding time analysis for higher order untyped functional languages. In *1990 ACM Conference on Lisp and Functional Programming* (June 1990), ACM, ACM Press, pp. 264–272.

[4] Dean, J., Chambers, C., and Grove, D. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)* (La Jolla, California, 18–21 June 1995), pp. 93–102.

[5] Futamura, Y., Nogi, K., and Takano, A. Essence of generalized partial computation. *Theoretical Computer Science 90*, 1 (Nov. 1991), 61–79.

[6] Glück, R., Nakashige, R., and Zöchling, R. Binding-time analysis applied to mathematical algorithms. In *System Modelling and Optimization* (1995), J. Doležal and J. Fidler, Eds., Chapman & Hall, pp. 137–146.

[7] Interim Java Grande report. Tech. Rep. JGF-TR-4, Java Grande Committee, 1999.

[8] Jones, N. D. An introduction to partial evaluation. *ACM Computing Surveys 28*, 3 (Sept. 1996), 480–503.

[9] Karmesin, S., Crotinger, J., Cummings, J., Haney, S., Humphrey, W., Reynders, J., Smith, S., and Williams, T. Array design and expression evaluation in POOMA II. In *ISCOPE'98* (1998), vol. 1505, Springer-Verlag. Lecture Notes in Computer Science.

[10] Khoo, S. C., and Sundaresh, R. S. Compiling inheritance using partial evaluation. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (New Haven, CN, June 1991), vol. 26(9), pp. 211–222.

[11] Kleinrubatscher, P., Kriegshaber, A., Zöchling, R., and Glück, R. Fortran program specialization. *SIGPLAN Notices 30*, 4 (1995), 61–70.

[12] Kolte, P., and Wolfe, M. Elimination of redundant array subscript range checks. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)* (La Jolla, California, 18–21 June 1995), pp. 270–278.

[13] Moreira, J. E., Midkiff, S. P., Gupta, M., Artigas, P. V., Snir, M., and Lawrence, R. D. Java programming for high-performance numerical computing. *IBM Systems Journal 39*, 1 (???? 2000), 21–56.

[14] Salomon, D. J. Using partial evaluation in support of portability, reusability, and maintainability. In *Compiler Construction '96* (Linköping, Sweden, 24–26 Apr. 1996), pp. 208–222.

[15] Sarkar, V., and Knobe, K. Enabling sparse constant propagation of array elements via array SSA form. *Lecture Notes in Computer Science 1503* (1998), 33–??

[16] Steensgaard, B. Sparse functional stores for imperative programs. In *ACM SIGPLAN Workshop on Intermediate Representations (IR'95)* (Jan. 1995), vol. 30 (3) of *SIGPLAN Notices*, ACM Press, pp. 62–70.

[17] Taha, W., and Sheard, T. Multi-stage programming with explicit annotations. *ACM SIGPLAN Notices 32*, 12 (1997), 203–217.

[18] Veldhuizen, T. L. Expression templates. *C++ Report 7*, 5 (June 1995), 26–31. Reprinted in C++ Gems, ed. Stanley Lippman.

[19] Veldhuizen, T. L. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)* (1998), Lecture Notes in Computer Science, Springer-Verlag.

[20] Veldhuizen, T. L. C++ templates as partial evaluation. In *Proceedings of PEPM'99, The ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, ed. O. Danvy, San Antonio, January 1999.* (Jan. 1999), University of Aarhus, Dept. of Computer Science, pp. 13–18.

[21] Volanschi, E.-N., Consel, C., Muller, G., and Cowan, C. Declarative specialization of object-oriented programs. In *ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '97)* (October 1997), pp. 286–300.

A Detailed discussion of "expression templates" optimization

This appendix explains in detail how Lunar turns

```
w.assign(x.plus(y.times(z)))
```

into the loop code shown in Figure 7.

A.1 Eliminating the tree walk

We step through the optimizations performed by the partial evaluator, referring to Figure 15, which shows the state of the heap during this moment.

Lunar's Java front end uses virtual method tables (vtables) to handle method dispatching. Every object has a pointer to its vtable.[6] Shaded cells in Figure 15 indicate "write-once memory" marked by the Java front end. Write-once memory is guaranteed to only be written once, and never read before it is written. This allows the optimizer to make stronger assumptions about the contents of such cells (for example, write-once memory is immune to aliasing effects).

Initially, we have this code:

```
Array w = new Array(n);
Array x = new Array(n);
Array y = new Array(n);
Array z = new Array(n);

// ...

w.assign(x.plus(y.times(z)))
```

[6]The vtable layout is simpler than it would be for a compiler which handled the full Java language. Lunar's front end does not handle interfaces, and Figure 15 omits methods inherited from `java.lang.Object` for simplicity.

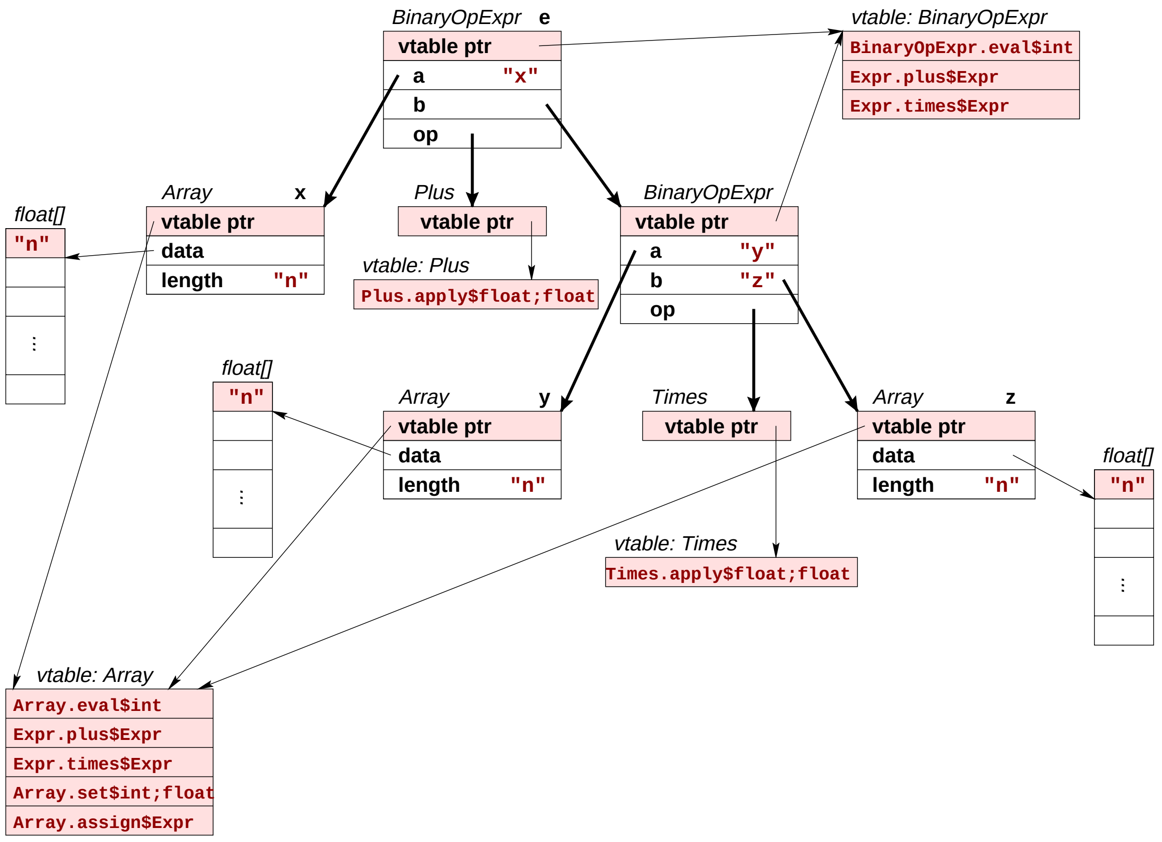

Figure 15: Heap state when the array expression `w.assign(x.plus(y.times(z)))` is evaluated. Shaded heap cells are "write-once" memory created by the Java front end. Names in quotes, such as `"x"`, are variable names used for copy propagation through the heap. Arrows are pointers; bold arrows highlight the parse tree of `w=x+y*z`. Not shown are compiler-generated names for temporary objects.

Since `w` was created locally, Lunar can follow its vtable pointer to find the method `Array.assign(Expr)`. This method can be inlined:

```
// Result of inlining Array.assign(Expr)
Expr e = x.plus(y.times(z));
for (int i=0; i < w.length; i=i+1)
  w.data[i] = e.eval(i);
```

By similar means, it can inline `x.plus(..)` and `y.times(...)`:

```
Expr __a1 = new BinaryExprOp(y,z,new Times());
Expr e = new BinaryExprOp(x,__a1,new Plus());
for (int i=0; i < n; i=i+1)
  w.data[i] = e.eval(i);
```

Since `e` is locally constructed, Lunar knows its vtable pointer, and can resolve `e.eval(i)` to `BinaryOpExpr.eval(..)` and inline it:

```
Expr __a1 = new BinaryExprOp(y,z,new Times());
Expr e = new BinaryExprOp(x,__a1,new Plus());
for (int i=0; i < n; i=i+1)
  data[i] = e.op.apply(e.a.eval(i),e.b.eval(i));
```

The whole `e` object was constructed locally, so the optimizer can follow vtable pointers to determine that:

```
e.op.apply(...)     is     Plus.apply(...)
e.a.eval(...)       is     Array.eval(...)
e.b.eval(...)       is     BinaryOpExpr.eval(...)
```

With these functions inlined, the code becomes:

```
Expr __a1 = new BinaryExprOp(y,z,new Times());
Expr e = new BinaryExprOp(x,__a1,new Plus());
for (int i=0; i < n; i=i+1)
  data[i] = e.a.data[i] + e.b.op.apply(e.b.a.eval(i),
             e.b.b.eval(i));
```

Lunar does copy propagation through the heap. This allows it to determine that `e.a` is the array `x`. After more expansions and inlining, we get:

```
// e.a                   is x
// e.b.op.apply(...)     is Times.apply(...)
// e.b.a.eval(...)       is Array.eval(...)
// e.b.b.eval(...)       is Array.eval(...)
Expr __a1 = new BinaryExprOp(y,z,new Times());
Expr e = new BinaryExprOp(x,__a1,new Plus());
for (int i=0; i < n; i=i+1)
  data[i] = x.data[i] + (e.b.a.data[i] * e.b.b.data[i]);
```

From the heap analysis, the optimizer knows that `e.b.a` is `y` and `e.b.b` is `z`:

```
// e.b.a         is y
// e.b.b         is z
Expr __a1 = new BinaryExprOp(y,z,new Times());
Expr e = new BinaryExprOp(x,__a1,new Plus());
for (int i=0; i < n; i=i+1)
  data[i] = x.data[i] + (y.data[i] * z.data[i]);
```

Presto – the overhead of traversing the parse tree has been eliminated.

At this point, the optimizer sees that `__a1`, `e`, and the `Plus` and `Times` objects are written to, but are never read from nor escape. This allows Lunar to eliminate these objects, resulting in:

```
for (int i=0; i < n; i=i+1)
  data[i] = x.data[i] + (y.data[i] * z.data[i]);
```

A.2 Bound check elimination

The next step is bound check elimination. An expression such as `x.data[i]` expands into C code like this:

```
// Get x.data
char *__a26 = *((char **) (x + Array_dotdata));

// Read array size, stored just before the
// beginning of the array
int __a81 = *((int *) (__a26 + -4));

// Bound check
if (((unsigned) i) >= __a81)
  bound_check_handler(i, __a81);

// Read x.data[i]
int __a82 = (i * 4);
float __a80 = *((float *) (__a26 + __a82));
```

To eliminate these bound checks, Lunar uses a little bit of *generalized partial computation* [5]. In partial evaluation, values are either known or not known. In generalized partial computation, one can know *partial information* about values. Consider the loop

```
for (int i=0; i < n; i=i+1)
  w.data[i] = ....;
```

The loop condition is $i < n$ and the initial value of i is $i = 0$. Lunar eliminates variable assignments prior to partial evaluation, so the loop condition is a loop invariant: $i < n$ is *always* true in the body of the loop, and using the initial binding of i, Lunar can conclude that $0 \leq i < n$.

For the environment of the loop body, Lunar binds i to an *interval*, rather than a value. This indicates that the exact value of i is unknown, but that it lies within some interval:

$$i \mapsto [0, n)$$

For the expression `w.data[i]`, the bound check can be eliminated only if Lunar can prove that $0 \leq i <$ `w.data.length`. Lunar discovers that `w.data.length` is also `n` by doing copy propagation through the heap. Then the bound check is $0 \leq i < n$, which is trivially true since $i \mapsto [0, n)$.

Generalized partial computation is one of several approaches to bound check elimination (see [12] for a summary). It is used in Lunar because it fits nicely into the partial evaluation framework. It can eliminate bound checking for some situations, but not all.

The general problem of bound check elimination (Given a program P, can all bound checks be safely eliminated?) is undecidable, so being able to eliminate checks in common situations is as good as one can hope for.

B Lunar output for the "expression templates" example

```
void sti__dollunar_umain_ureturn_uvalue()
{
    char *__a68 = 0;
    char *array1d = alloc(49384, 4);
    *((int *) (array1d + -4)) = 12345;
    char *array1d__96 = alloc(49384, 4);
    *((int *) (array1d__96 + -4)) = 12345;
    char *array1d__107 = alloc(49384, 4);
    *((int *) (array1d__107 + -4)) = 12345;
    char *array1d__118 = alloc(49384, 4);
    *((int *) (array1d__118 + -4)) = 12345;

    // Initialize the arrays with some data
    for (int i = 0; (i < 12345);)
    {
        float __a43 = ((float) i);
        float __a42 = (__a43 * 0.33f);
        int __a73 = (i * 4);
        *((float *) (array1d__96 + __a73)) = __a42;
        float __a48 = ((float) i);
        float __a47 = (10.0f + __a48);
        int __a128 = (i * 4);
        *((float *) (array1d__107 + __a128)) = __a47;
        float __a53 = ((float) i);
        float __a52 = (100.0f * __a53);
        int __a135 = (i * 4);
        *((float *) (array1d__118 + __a135)) = __a52;
        i = (i + 1);
    }

    // Code generated for
    // w.assign(x.plus(yexpr.times(zexpr)));
    for (int i = 0; (i < 12345);)
    {
        int __a75 = (i * 4);
        int __a71 = (i * 4);
        float __a31 = *((float *) (array1d__96 + __a71));
        int __a184 = (i * 4);
        float __a171 = *((float *) (array1d__107 + __a184));
        int __a163 = (i * 4);
        float __a176 = *((float *) (array1d__118 + __a163));
        float __a35 = (__a171 * __a176);
        float __a76 = (__a31 + __a35);
        *((float *) (array1d + __a75)) = __a76;
        i = (i + 1);
    }
    _dollunar_umain_ureturn_uvalue = 0;
}
```